Hybrid Algorithms for On-Line Search and Combinatorial Optimization Problems.

Yury V. Smirnov

May 28, 1997

CMU-CS-97-171

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored in part by the National Science Foundation under grant number IRI-9502548. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations or the U.S. government.

Keywords: Hybrid Algorithms, Cross-Fertilization, Agent-Centered Search, Limited Lookahead, Pigeonhole Principle, Linear Programming Relaxation.

“ Each problem I solved became a pattern,
that I used later to solve other problems.”
René Descartes

To my wife, Tanya,
and my sons, Nick and Michael

Abstract

By now Artificial Intelligence (AI), Theoretical Computer Science (CS theory) and Operations Research (OR) have investigated a variety of search and optimization problems. However, methods from these scientific areas use different problem descriptions, models, and tools. They also address problems with particular efficiency requirements. For example, approaches from CS theory are mainly concerned with the worst-case scenarios and are not focused on empirical performance. A few efforts have tried to apply methods across areas. Usually a significant amount of work is required to make different approaches "talk the same language," be successfully implemented, and, finally, solve the actual same problem with an overall acceptable efficiency.

This thesis presents a systematic approach that attempts to advance the state of the art in the transfer of knowledge across the above mentioned areas. In this work we investigate a number of problems that belong to or are close to the intersection of areas of interest of AI, OR and CS theory. We illustrate the advantages of considering knowledge available in different scientific areas and of applying algorthms across distinct disciplines through successful applications of novel hybrid algorithms that utilize benefitial features of known efficient approaches. Testbeds for such applications in this thesis work include both open theoretical problems and ones of significant practical importance.

We introduce a representation change that enables us to question the relation between the Pigeonhole Principle and Linear Programming Relaxation. We show that both methods have exactly the same bounding power. Furthermore, even stronger relation appears to be between the two methods: The Pigeonhole Principle is the *Dual* of Linear Programming Relaxation. Such a relation explains the "hidden magic" of the Pigeonhole Principle, namely its power in establishing upper bounds and its effectiveness in constructing optimal solutions.

We also address various groups of problems, that arise in agent-centered search. In particular, we consider goal-directed exploration, in which search by a physical or fictitious agent with limited lookahead occurs in partially or completely unknown domains. The resulting Variable Edge Cost Algorithm (VECA) becomes the first method of solving goal-directed exploration problems that incorporates strong guidance from heuristic knowledge, yet is still capable of providing linear worst-case guarantees, even for complex search domains and misleading heuristics.

This work aims at expanding the handset of AI tools that concern search efficiency and provides the foundation for further development of hybrid methods, cross-fertilization and successful applications across AI, CS theory, OR and other Computational Sciences.

Acknowledgements

I was happy and proud to spent five years of my scientific life at the School of Computer Science, Carnegie Mellon University. Presence of world-class researchers, professors and students, wonderful people, of an outstanding Artificial Intelligence division within the School, and the diversity of interests promoted multiple interactions and had a strong impact on populating my AI knowledge with exciting results and ideas. Without doubts, such a scientific environment promotes creative thinking and analytical reasoning.

No words can express my gratitude to my advisor, Manuela Veloso. She gracefully agreed to supervise my research at the point in time, when I had hard time deciding where to go. She devoted an incredible amount of time and effort to help me with my research. Manuela was always there for me, no matter how many deadlines were marked in her calendar. At the times, when my research efforts would get flooded with the abundance of unrelated ideas, my advisor would spent an extra effort on filtering out redundant ones and concentrating my research on specific issues.

I would like to thank Jaime Carbonell for his time, effort, bright ideas and valuable comments on making my thesis research complete and sound. Jaime's remarks used to give an opportunity to view the same problems from different prospectives. His help in my research and a remote supervision of the Prodigy group greatly enriched my thesis work. It was a great pleasure to work with such a great scientist, whose presence inspired the research enthusiasm.

I am very grateful to Avrim Blum, my thesis committee member, whose deep knowledge and the diversity of interests were always extremely helpful in combining my inter-disciplinary efforts and extracting the most beneficial features from approaches from different scientific areas. I was deeply touched that Avrim could find time and strength to keep an eye on my research at a difficult point of his life.

Special thanks to Bart Selman, my external committee member, who found time to come for both the Thesis Proposal and the Defense, who regularly sent me comments on the thesis work, kept in touch regarding other research papers. His valuable comments on my thesis development served as a good guidance of my research.

I will always remember the Vice Dean Sharon Burks, who spent an enormous effort to make my life, as well as lives of other students, much easier by solving a seemingly infinite sequence of our problems. She is always weeling to go out of her way for her "kids," with every particular student, with our particular problems throughout the timespan of our graduate careers. I will always remember her as a kind, loving and caring person, as the "Mother" of the Department.

My first two years as a graduate student within the Algorithms, Combinatorics and Optimization inter-disciplinary program gave me a multi-dimensional vision on seemingly plain things. I'm very grateful to my first advisor, Gary Miller, whose support and supervision enabled me to take seven doctorate courses in Fall-93 and got me through the toughest years of graduate studying.

I would like to thank my officemate, Sven Koenig, who pointed me to a set of open problems and, thus, opened for me the curtain of the challenging world of Artificial Intelligence. During last several years, Sven acted a senior graduate student and a knowledgefull colleague. Sven's comments were extremely helpful during the hard time of my thesis proposal and the preparation for the thesis defense.

Great thanks to Joseph O'Sullivan who was always ready to come up with help in resolving various technical problems. Special thanks to Prasad Chalasani who accommodated me for the first seven days in the US. His advice helped me a lot in understanding a new culture.

Great thanks to all the members of the Prodigy research group of my time being: Jim Blythe, Eugene Fink, Peter Stone, Karen Haigh, Michael Cox and others. Eugene was always ready to discuss any tough problems, whether it was within his areas of interest or not. The diversity of interests of the *Prodigies* greatly stimulated my inter-disciplinary development, as well as multiple interactions with Astro and members of the Reinforcement Learning group.

My wife Tanya was my greatest supporter. Without her care for our family, two sons, Nick and Michael, and me at times when deadlines seemed to stack like a pile of pancakes on the breakfast table, I would never be able to come even close to the defense. Tanya was always ready to cheer me up, whether the research appeared to be stuck in a hopeless dead-end, I lost in an important road race, or kids got sick. Only because the family bases were covered by Tanya's care, I could concentrate exclusively on research, thesis writing and regular running. Both Nick and Mike went smoothly through a complete switch from one linguistic and cultural environment to another. I am very thankful for their understanding the situation, although Nick used to threaten me that his middle school graduation party will take place before my Commencement. In general, both Nick and Mike were terrific and helped me greatly as much as they could.

Contents

1 Introduction **1**
- 1.1 Retrospectives of Hybrid Approaches . . . 3
- 1.2 The Choice of the Areas . . . 4
- 1.3 The Problems . . . 6
- 1.4 Methodology of Hybrid Approaches . . . 8
- 1.5 Contributions of the Thesis . . . 10

2 Combinatorial Optimization Problems **11**
- 2.1 Pigeonhole Principle . . . 12
- 2.2 Linear Programming Relaxation . . . 15
  - 2.2.1 Important Facts from Linear Programming . . . 15
  - 2.2.2 Definition of Linear Programming Relaxation . . . 17
- 2.3 Converting Problems into the Integer Programming Form . . . 18
- 2.4 PHP is the Dual of LPR . . . 23
- 2.5 More Complicated Applications of PHP . . . 27
- 2.6 HPHP Mimicking LPR with Chvatal-Gomory's Cuts . . . 35
- 2.7 Implications for Resolution-Based Proof Methods . . . 36
- 2.8 Tough Nuts for the Pigeonhole Principle . . . 38
- 2.9 Summary . . . 40

3 On-Line Search **43**
- 3.1 Agent-Centered Technologies for On-Line Search . . . 44
- 3.2 Goal-Directed Exploration Problem . . . 47
- 3.3 CS Theory Approaches . . . 48
- 3.4 Heuristic-Driven Approaches . . . 49
  - 3.4.1 Agent-Centered A* Algorithm . . . 49
  - 3.4.2 AC-A* Can Be Misled by Heuristic Values . . . 51
  - 3.4.3 AC-A* is not Globally Optimal . . . 53
  - 3.4.4 Learning Real-Time A* Algorithm . . . 55
- 3.5 Summary . . . 56

4 Variable Edge Cost Algorithm **57**
4.1 The Drawbacks of Existing Algorithms 57
4.2 Our Approach: The VECA Framework 59
4.3 Implementation 65
4.4 Experimental Results 67
4.5 Multiple Agents 71
4.6 Summary 72

5 Agent-Centered Approach for Sensor-Based Planning **75**
5.1 Sensor-Based Planning with the Freespace Assumption 76
5.2 Problem Description 76
5.3 D* Algorithm 80
5.4 Complexity Analysis 81
5.4.1 Lower Bounds 81
5.4.2 Upper Bounds 84
5.4.3 Applying VECA to Improve Performance Guarantees 86
5.5 Summary 88

6 GenSAT as Agent-Centered Search **91**
6.1 GenSAT 91
6.2 GenSAT as Agent-Centered Search 93
6.3 New Corners or Branching Factor? 95
6.4 Approximate Satisfaction 100
6.5 Navigational Planning Problems 101
6.6 GenSAT Conclusions 103

7 Further Insights into On-Line Search Complexity **105**
7.1 The Oblongness Factor 106
7.2 Complexity of On-Line Search 108
7.2.1 Resistive Networks 111
7.2.2 Simplifying the Estimates 112
7.3 Estimating Complexity of Planning Problems 113
7.3.1 Using Oblongness to Compare Search Complexities 113
7.3.2 Using Oblongness in Agent-Centered Search 115
7.4 Promising Directions for Cross-Fertilization 116
7.4.1 Problem-Driven Cross-Fertilization 116
7.4.2 Hints on Building Hybrid Solutions 116
7.4.3 Method-Driven Hybrid Approach 117
7.5 Summary 120

8 Conclusions **123**

List of Figures

2.1 Illustration of the Pigeonhole Principle 13
2.2 A Particular Instance of a LP Problem 17
2.3 Chess kings on a circular chessboard. 21
2.4 N-Queen Problem Solutions for N = 4, 5, 6 22
2.5 Hamiltonian Tour on a Chessboard for the Knight 25
2.6 The Set of Adjacent Squares for the Knight on a Chessboard 26
2.7 A Mutilated Checkerboard 27
2.8 Firm 17-Tile Solution. 28
2.9 Modeling Domino Overlap as a Set of Inequalities 29
2.10 Modeling a Firm Tiling of a 6x6 Checkerboard. 32
2.11 Re-distributing the Measure of the 2D Circle 39
2.12 Cutting a Parallelogram from a Triangle with N=5 40

3.1 A Worst-Case Example for all Uninformed Algorithms 49
3.2 Average Exploration Time of Rectangular Mazes with Different Density . . . 51
3.3 A Bad State Space for AC-A* (here: $x = 3$) 52
3.4 Another Bad State Space for AC-A* (here: $x = 3$) 54

4.1 Worst-Case and Empirical Performances 58
4.2 The Basic-VECA Framework 60
4.3 A Tree of the Highest VECA Cost 61
4.4 A Simple Example State Space 62
4.5 The VECA Framework 64
4.6 The Graphical User-Interface of the VECA System 67
4.7 Exploration Behavior of AC-A* with Different Heuristics 69
4.8 Empirical Performance of AC-A* with and without VECA 69
4.9 Empirical Performance of LRTA* with and without VECA 70
4.10 A Bad Graph for the "Optimistic" Team 73

5.1 Outdoor robot navigation with NAVLAB II 77

5.2 Initial knowledge of the robot . . . 78
5.3 Initial graph . . . 78
5.4 Indoor robot navigation with a Nomad . . . 78
5.5 Initial graph (1) . . . 79
5.6 Initial graph (2) . . . 79
5.7 Graph G_1 for $n = 3$ . . . 83
5.8 A Tough Graph . . . 84
5.9 Subgraph of a square grid . . . 86
5.10 Basic-VECA . . . 88
5.11 Improving Worst-Case Complexity through a Spanning Tree . . . 89

6.1 The transition phase for random 3SAT problems. . . . 93
6.2 Comparison of *Poss-Flips* for GSAT, HSAT, NavRGSAT and NavFGSAT. . . 98
6.3 Dynamics of *Poss-Flips* for GSAT. . . . 99
6.4 Percentage of *Poss-Flips* for GSAT with HSAT and GSAT with NavRGSAT. . 100
6.5 Initial and final solutions for NavP. . . . 103

7.1 The Efficiency of Agent-Centered Search and Oblongness . . . 108
7.2 Problems and their *Oblongness* Ranges. . . . 109
7.3 Undirected Graphs that Are Easy for Search. . . . 110
7.4 A Lollipop Graph . . . 110
7.5 Examples of Resistive Networks . . . 111
7.6 Projection of the Mass Center on the Face of a Polyhedron . . . 118
7.7 Troubleshooting N registers . . . 121

Chapter 1

Introduction

From the very first days, Artificial Intelligence (AI) experienced rapid growth and development. On its earlier stages, AI heavily relied on ideas and techniques from other areas including Mathematics, Psychology, and Biology. Although AI continued incorporating and interpreting knowledge from other scientific disciplines that were developed in parallel with it, such as Operations Research (OR), Theoretical Computer Science (CS theory), Statistics, etc., AI was especially active and successful in building its own models, tools for attacking and efficient methods of solving its problems.

"Re-utilization" of the existing knowledge, cross-applications of already developed methods to new problems and hybrid solutions have been always considered as the first thing to do for novel problems. It usually takes longer time to develop independent methods with fresh ideas tailored towards solving newly stated problems. It is especially hard to come up with efficient methods, when the problem instances are real-world complex AI domains. Furthermore, AI seems to be concerned with a wide variety of aspects – from employing Linear Programming techniques that is traditionally considered as the territory of Operations Research, to analyzing worst-case scenarios that is usually attributed to CS theory, to acquiring prior knowledge and building heuristic-guided algorithms. Thus, hybrid solutions with a variable mix of algorithms from different scientific areas is a natural approach in solving AI problems.

However, it takes a certain amount of modeling effort to state a realistic problem in the form amenable to specific methods from particular areas of Science. Usually such a reduction implies some sort of simplification, as the result, the derived solutions are only the approximations of the realistic processes. Nonetheless, more and more often such approximations are very close to the processes they model and serve as good indications of the expected results. In this thesis work we discuss a wide variety of traditional and untraditional models and methods from different scientific areas with the emphasize on AI, OR and CS theory. We hope that the adequacy of the problem representation is reflected in the variety of considered areas and methods.

There has been a noticeable raise of interest recently to hybrid solutions and cross-applications of the most efficient methods from various scientific areas to practical AI problems. In particular, several recent efforts have tried to merge methods of OR, CS theory and AI. These efforts indicate that it is challenging to identify and compare the strengths of existing approaches from different areas, to attempt to combine them in a single, hybrid method in order to extract the best from many worlds. However, methods from the above mentioned scientific areas use different problem descriptions, models and tools. They also address problems with particular efficiency requirements. For example, algorithms from CS theory are mainly concerned with the worst-case scenarios and are not focused on empirical performance. Usually a significant amount of work is required to make different approaches "talk the same language," be successfully implemented, and, finally, solve the actual same problem with an overall acceptable efficiency.

This thesis constitutes a systematic approach that attempts to advance the state of the art in the transfer of knowledge across the above mentioned areas. In this work we investigate a number of problems that belong to or are close to the intersection of areas of interest of AI, OR and CS theory. Hybrid approaches developed in the thesis work, are illustrated through successful applications to several groups of problems that include both open theoretical problems and ones of practical importance. In each case, we demonstrate how the representation of each specific group of problems can be changed to create a multi-linguistic environment, so that methods from distinct scientific areas can be applied to the problems under investigation. Throughout the thesis work we emphasize on selecting or designing those kinds of representation changes for every group of problems that could be utilized by automated reasoning, planning or scheduling systems.

The methodology of hybrid approaches introduced later in this chapter, distinguishes two main cases.This stratification is due to the drastic difference in the way of initializing the strategy of research. Although the initial phase is extremely important and probably makes the major contribution to the global success of the approach, in either case it is extremely important to perform thorough analysis along different directions to identify the most effective methods of solving the group of problems under the scope. Finding an appropriate representation that would allow methods from distinct areas to be applied to the same group of problems, is a keystone step that proceeds the analysis. Furthermore, when possible, we recommend to combine the most beneficial features found during the analysis phase, thus, utilizing the advantage of the common representation developed.

Though both cases differ from each other in the problem selection and the performance analysis, they have a lot in common regarding the concluding phase of constructing hybrid solutions. We consider that the global approach succeeds, if we are able to come up at least with the classifications of a certain group of problems and state recommendations on effective usage of particular methods depending on properties of the problem domain. The ultimate goal of the hybrid approach is to combine the best features of the analyzed algorithms in a single framework with an internal, self-regulating process. For example, if a physical or fictitious agent with limited lookahead performs search in an initially unknown domain, the issue of "exploration versus exploitation" becomes a keystone of the efficiency of search. We propose a mechanism of regulating reasonable balance between "rushing to the goal" driven by prior knowledge and "learning more" for reversible[1] domains in Chapter 3.

1.1 Retrospectives of Hybrid Approaches

Hybrid efforts and cross-fertilization between distinct scientific areas have a long-standing history. Even Archimedes was not, probably, the first scientist to apply knowledge across distinct disciplines. The following phrase is the interpretation of the excitement of this great Greek scientist about the power of levers: "Give me a lever long enough, and a place to stand, and I will move the Earth." This scientific idea illustrates a particular application of cross-fertilization between Mechanical Engineering and Astronomy.

In this sneak preview of the thesis, we would like to refer the reader to the main quotation of the thesis due to René Descartes: *"Each problem I solved became a pattern, that I used later to solve other problems."* It brings up the take-home idea that both the re-utilization of already acquired knowledge, and reductions to already solved problems form the basis of applications within a sole discipline, as well as across distinct areas of Computational Sciences.

Speaking about Artificial Intelligence, efforts of bringing techniques and ideas from other areas of Science to AI, in particular, to search and optimization problems, have been attempted multiple times throughout the history of AI. In this thesis work, we are concentrated on hybrid efforts between Computational Sciences, namely, AI, OR and CS theory. Researchers from these three disciplines attempted multiple times to re-apply the most efficient methods to solve problems from other areas, including AI search and optimization problems. Moreover, these efforts have been organized lately in a well-structured form, and were divided in topics corresponding to specific groups of search problems.

The First International Workshop on AI and OR, Portland, OR, June 95, highlighted the achievements and cutting-edge technologies from both disciplines and drew possible successful directions of combining their beneficial features. Job-Shop scheduling with constraints of various types [10, 22, 45] was identified as one of the testbed areas suited for attacking by AI and OR methods or their combinations. The Workshop concluded with a set of open problems that were supposed to shed a light on relations between techniques from AI and OR, such as the bounding power of the Pigeonhole Principle (PHP) and Linear Programming Relaxation (LPR), the complexity of identifying symmetries and its impact on reducing the complexity of

[1] Domain is reversible if it can be represented as an undirected or bi-directed graph. For a detailed discussion, see Chapter 3.

search, among others. In this work we give an answer to the first open problem, namely, we demonstrate the duality relation between PHP and LPR.

Recent surge in interest to methods of solving satisfiability problems was due to the success of local hill-climbing procedures. GSAT [34, 64] and similar procedures – TSAT, CSAT, DSAT, HSAT [27, 29], WSAT [66], WGSAT, UGSAT [18], etc. – have attracted a lot of attention from AI researchers, because they appeared to be capable of finding satisfiable assignments for some large-scale practical problems that cannot be attacked by conventional resolution-based methods. In Chapter 6 we discuss the relations between local hill-climbing procedures and agent-centered search methods.

An effort of bringing graph-theoretical ideas into planning resulted in an attractively simple partial-order planner – `Graphplan` [9]. This planner utilizes models and fundamental methods from the Graph Theory, for example, shortest paths on the directed graph that represents the planning domain, and network broadcasting of the alternative partial plans. Although the current version of `Graphplan` does not incorporate prior knowledge that might effectively lead search towards the acceptable plan, `Graphplan` guarantees at least to construct planning domain graphs in time that is polynomial on the size of the input [9].

The idea of enriching the area of efficient search control (planning) by bringing ideas from related areas of Science to search (planning) problems has been implemented in various modifications. For example, the implementation of the set differencing heuristic from Number Theory implied an elegant polynomial algorithm that approximates the *NP*-hard problem of 2-machine job scheduling and outperforms known greedy polynomial approximations[44].

In the following chapters we present several groups of problems that initiated a strong interest from an interdisciplinary point of view. They are of the particular interest to our current work, because they admit untraditional methods with highly developed techniques from distinct areas of Science. Chapter 2 illustrates a method-driven hybrid approach, where two well-known efficient methods are analyzed and compared along different dimensions. In Chapters 3-6, we apply the problem-driven hybrid approaches, which implies that we need to add the identification of relevant methods prior to the analysis phase. According to the methodology that we develop in this work, in such case, for every group of problems we, first, identify relevant known methods for solving these problems. Then, we analyze their advantages and drawbacks, and, when possible, propose novel hybrid algorithms to solve the considered problems efficiently.

1.2 The Choice of the Areas

The choice of Artificial Intelligence, Theoretical Computer Science and Operations Research has not been accidental. These three areas often attack close problems and are concerned with either the efficiency of deriving solutions or the efficiency of solutions themselves. Nonetheless, even a problem statement may contain a big difference for distinct disciplines. Furthermore, differences in terms, existing techniques and efficiency foci can place the success of a naïve hybrid effort into a doubtful position.

Table 1.1 lists some advantageous features of AI, CS theory and OR. This table does not pretend to be a complete list of all beneficial features, it only highlights some of already identified ones within the considered areas. For example, AI researchers have long realized that prior knowledge can significantly improve empirical performance in practical applications. A solution to the problem, found with the help of the prior knowledge guidance, serves as a benchmark for methods from other areas. Moreover, it can also carry an additional bounding value, for example, any feasible solution establishes an upper bound on the value of the goal function in a minimization problem.

AI	CS Theory	OR
• Preprocessing - Representation Changes - Prior Knowledge • Empirical Performance	• Data Structures • Worst-Case Analysis • Optimal Algorithms • Approximate Algorithms	• Linear Programming • Integer Programming - Modeling - Methods of Solving
• Machine Learning		

Table 1.1: Advantageous Features of AI, CS Theory and OR

Artificial Intelligence has also accumulated an extended library of AI problem-tailored algorithms that are carefully selected from a wide pool of solutions, based on their empirical performances. Since deriving an exact average-case complexity is usually a very hard task that depends both on the domain features and the initial distribution of the problem instances, such an extensive AI algorithmic library is of high value. Empirical performances of its algorithms serve as approximations of otherwise hard-to-derive average-case complexity justifications.

In its turn, CS Theory possesses deep knowledge on data structures that are used in optimal or approximating algorithms. The worst-case analysis concludes with a justification of the optimality of a given algorithm or a guaranteed approximation of the optimal solution that this algorithm provides. From the point of view of hybrid algorithms, the strength of CS theory lies exactly in providing worst-case guarantees. This means that no matter how misleading in specific cases heuristic values may represent actual distances to the goal, the complexity of achieving the goal is not worse than a certain cut-off level established by algorithms from CS theory. Deriving average-case complexity is also often attributed to CS theory, as it is usually built upon and relies on classical CS theory data structures and methods of analysis. Unfortunately, prior knowledge is a rare guest in theoretical algorithms, because it seldomly improves the worse-case complexity or an approximation ratio.

Operations Research is known for the strength of its methods from Linear and Integer Programming. Some researchers believe that methods of solving Linear Programming problems are currently the cutting-edge polynomial methods, i.e. they provide polynomial-time solutions for the most sophisticated problems. OR has also accumulated a broad experience in modeling various problems in a Linear, Non-Linear or Integer Programming form to apply already well-established techniques and solve those problems efficiently. Recent efforts in developing Mixed-Logical Linear Programming [21] constitute another example of a cross-fertilization approach between AI and OR that attempts to capitalize on advantageous features from both areas.

Machine Learning attracted a lot of attention recently from researchers from different disciplines. Sub-areas of Machine Learning, such as Neural Nets, PAC-learning, Reinforcement Learning employ drastically different data structures and techniques varying from decision trees, rules, neurons, perceptrons, ϵ-error, δ-confidence to (Partially Observable) Markov Decision Processes, etc. As such, we placed Machine Learning in between AI and CS theory. Some of the Machine Learning technologies are capable of deriving solutions of significantly better quality than traditional statistical methods in the domains, where accounting all domain states is impossible.

1.3 The Problems

One of the goals of this thesis is to demonstrate how some of the existing methods from AI, CS theory and OR can be applied to solve the same problems along different efficiency dimensions. We consider several groups of problems that include various scenarios arising in agent-centered search [38], namely, the problem of goal-directed exploration, sensor-based planning and search by local hill-climbing procedures. For every problem discussed under this scope, we analyze the efficiency of known algorithms amenable to these problems. Worst-case complexity for some of them has been an open problem before our investigation, for some – the average-case complexity is still an open problem.

Chapter 2 presents a discussion on the bounding power and a close relation between the Pigeonhole Principle and methods of Integer Programming. This discussion is illustrated through a series of combinatorial optimization problems with gradually increasing complexity.

For some problems, the worst-case complexity of a particular algorithm does not predict its average-case (empirical) performance well. On the contrary, some of the algorithms with either worse or still unknown worst-case complexities demonstrate better empirical performances. For example, for randomly generated 2D Traveling Salesman problem, the furthest insertion and nearest neighbor methods construct shorter in average Hamiltonian cycles than the cheapest insertion or spanning tree-based methods [62]. The worst-case complexity of the solution derived by the furthest insertion is still an open problem, the ratio of the length of the solution produced by the nearest neighbor method to the length of the optimal solution is not bounded in the worst-case. Both the cheapest insertion and the spanning tree-based method guarantee a low approximation of two times the length of the optimal solution. Nonetheless, in practical applications both methods with the guaranteed low ratio produce Hamiltonian cycles of longer length.

Another similar example comes from Operations Research. Popular in practical implementations, the Simplex method has an exponential worst-case complexity, whereas the Ellipsoid method [36] provides polynomial worst-case guarantees, but loses to the Simplex method in practice. We explain the above phenomenon the following way: Methods that are concerned with the worst-case complexity are too cautious, they do not follow risky alternatives, but rather search the state space methodically. In their turn, risky methods may lose much in the case when their selected alternatives are misleading and the cost of recovery is high.

Therefore, in this thesis we investigate both the worst-case complexity and the empirical performance of considered algorithms. While developing new hybrid methods, we also cover both performance metrics. In Chapters 2-6 we develop interdisciplinary representations, cross-apply existing problem solving methods, and analyze the performance of efficient solutions to the following set of problems:

1. Combinatorial optimization problems – in this group of problems we are mainly focused on the bounding power competition between the Pigeonhole Principle and methods of Integer Programming, such as Linear Programming Relaxation and Integer cuts. In addition to it, we argue about the duality relation between the Pigeonhole Principle and Linear Programming Relaxation. In Chapter 2 we illustrate this relation through a series of combinatorial optimization problems.

2. Goal-directed exploration – search in partially or completely unknown domains by an agent with limited lookahead. From this large set of problems, we extract several sub-

groups of problems with related problems scenarios that present the major interest to hybrid approaches:

(a) Treasure Hunt - search for a goal state by an agent whose lookahead is limited to the set of available actions at its current state in an unknown, reversible (undirected or bi-directed) state space. We also assume that prior knowledge is provided for every instantiation of an action at a particular state in the form of heuristic values. The initially unknown state space is either static or changes at discrete points in time.

(b) Sensor-Based Planning - search for a goal vertex by an agent whose lookahead is limited to the neighbors of its current vertex. The map of the problem domain is provided to the agent, however, traversability to each vertex is unknown unless the agent senses it from one of the neighboring vertices.

(c) Local Hill-Climbing Procedures - search for a satisfiable assignment on an N-dimensional cube by a fictitious agent whose lookahead is limited to the neighbors of its current vertex. Prior knowledge is provided for every vertex (corner) of the cube, but it may be neither consistent, nor admissible.

First group of problems brings back the discussion on the difference between human-derived and computer-oriented solutions. The "Mutilated Checkerboard" problem introduced in 60's [47, 55] raised a lot of interest in Mathematical, Logical and AI communities. The development of Operations Research and the introduction of Linear Programming methods with polynomial complexity, automatically added OR researchers to the above discussion. Do there exist any complexity barriers? Can a problem solution benefit from a specific presentation and the application of one of the traditional methods from the correspondent area of Science? Is any area of Science more preferable in deriving approximate solutions?

Second group of problems concerns the trade-offs between acquiring more knowledge about the surrounding environment and moving to the goal. The initial uncertainty of an agent about the domain, where one is supposed to live and to act in, is a realistic assumption in many real-world problems. Such an assumption stimulated our efforts in developing goal-directed exploration methods with the emphasis on the empirical performance.

1.4 Methodology of Hybrid Approaches

The development of classical, theory-oriented Computational Sciences and of novel, more application-oriented branches of Science, such as Artificial Intelligence, has arranged a fruitful background for designing new generations of efficient solutions for challenging problems. The whole pool of problems, models and technologies from AI, CS theory and OR seems to promote the idea of hybrid approaches based on achievements of these disciplines.

The methodology of designing hybrid solutions that we follow throughout the thesis work consists of several phases. We distinguish two main types: Problem-driven and method-driven hybrid approaches. In the former case, hybrid efforts start with the identification of efficient methods amenable to the given problem. In the latter case, two or more methods are already provided, hence, the method-driven hybrid approaches skips the first phase. Nonetheless, a certain research often has to be performed in such a case, for example, in finding challenging common applications, where the provided methods can be applied to. Overall, hybrid approaches consist of the following phases:

Phase 1 (Selection). Identification of efficient methods of solving given problems.

Phase 2 (Creating the Environment). Development of an interdisciplinary problem environment, so that methods from different scientific areas can be applied to the same problems.

Phase 3 (Analysis). Analysis of all selected methods along different efficiency dimensions, and the identification of their beneficial features.

Phase 4 (Problem Classification). Classification of the problem instances to be attributed to particular algorithms for the best efficiency.

Phase 5 (Constructing Hybrid Methods). Construction of novel hybrid methods to utilize the identified beneficial features of the selected methods.

Thus, a problem-driven hybrid approach begins with the analysis of a particular problem or group of problems along different directions to identify efficient methods of solving these problems. In parallel, for every group of problems discussed in this work, we design an interdisciplinary problem environment, so that methods from different scientific areas can "talk to each other," be applied to the same problems, and, finally, be compared in the Analysis Phase. Phase 4 (Problem Classification) concludes with recommendations on matching problems with the most efficient methods of solving. When possible, we proceed to the Constructing Hybrid Methods Phase, i.e. we combine the most beneficial features found during the Analysis Phase, thus, utilizing the advantage of the common representation developed.

1.5 Contributions of the Thesis

This thesis expands the handset of AI tools for solving search problems efficiently, and demonstrates the correctness of the methodology of hybrid approaches through successful applications to various groups of problems.

Through bringing problems and methods from different areas of Science to a common testing field, we show that there exists a room for mutual enrichment, that already developed, efficient methods of solving particular problems can be re-applied to new problems in an untraditional fashion, and that the best features of certain solutions can be re-utilized in designing novel efficient methods. In this thesis such testing fields include a) combinatorial optimization problems and b) deterministic on-line search. Already these two fields cover a substantial variety of problems. However, in our extended research we discuss c) the complexity of on-line search and relate it to other known results. The latter development links nicely both directions a) and b).

Concerning combinatorial optimization problems, we consider problems that are traditionally amenable to the Pigeonhole Principle (PHP). We show that if a problem can be transformed in a Integer Programming form, then an application of a Linear Programming Relaxation (LPR) establishes the same bound for the solution value as the PHP. Whereas the PHP often provides the most elegant and efficient solutions, for many combinatorial optimization problems it is hard to come up with a proper heuristic that would hint on finding the desired mapping between the pigeons, the holes, their capacities and the objects of the problem. On the other hand, LPR automatically narrows its consideration to tight constraints that can be viewed as the desired mapping for the PHP. Thus, the PHP and LPR establish the same bounds for the values of combinatorial optimization problem's solutions. Moreover, the relation between the PHP and LPR corresponds to the duality relation in Linear Programming.

Concerning on-line search, we show that the most beneficial features of CS theoretical algorithms - the worst-case guarantees - and of AI algorithms - a heuristic guidance - can be combined in a single hybrid method that preserves both the heuristic guidance and optimal or sub-optimal worst-case guarantees. Such a hybrid approach, thus, inherits strong empirical performance from heuristic-based algorithms and does not lose much to algorithms from CS theory when the heuristic appears to be misleading.

Scalability of methods applicable to on-line search and satisfiability problems, attracted recently a lot of attention. Certain on-line search methods have been successfully applied to challenging large-scale off-line problems and derived efficient solutions. In this thesis we discuss known results about the complexity of some on-line search methods, that include, for example, random walk. Guided by this knowledge, we introduce a novel parameter for search domains – "oblongness" – and argue about the place of successful on-line search methods in the global spectrum of the introduced parameter.

Chapter 2

Combinatorial Optimization Problems

The First International AI and OR Workshop held in Portland, OR, June 95, initiated a dispute on the competitive power of two drastically different methods of deriving upper bounds for combinatorial optimization problems: Pigeonhole Principle (PHP) and Linear Programming Relaxation (LPR).

The Pigeonhole Principle (PHP) has been one of the most appealing methods of solving combinatorial optimization problems. Variations of the Pigeonhole Principle often produce the most elegant solutions to non-trivial problems. However, some Operations Research approaches, such as the Linear Programming Relaxation (LPR), are strong competitors to the PHP: They can also be applied to combinatorial optimization problems to derive upper bounds. Note, that throughout this chapter we use the notion of an upper bound to determine the quality of a solution and not the time complexity of deriving it. For example, an upper bound of value B, where B is a constant, means that the value of an optimal solution is less or equal to B.

It has been an open question whether the PHP or LPR establish tighter upper bounds, when applied to the same problems. Challenged by this open question, we identify that the main reason for the lack of ability to compare the efficiency of the PHP and LPR is the fact that different problem representations are required by the two methods.

We introduce a problem representation change into an Integer Programming form which allows for an alternative way of solving combinatorial problems. We also introduce a series of combinatorial optimization problems, and show how to perform representation changes to convert the original problems into the Integer Programming form. Using the new problem model, we re-define the Pigeonhole Principle as a method of solving Integer Programming problems, introduce the "Hidden" Pigeonhole Principle (HPHP) and determine the difference between PHP and HPHP, show that PHP is the dual of LPR, and demonstrate that HPHP and Integer cuts are actually similar representation changes of the problem domains.

The Pigeonhole Principle is usually attributed to human-derived proofs. Automatic reasoning, planning or scheduling systems seem to have it hard-coded or not implemented, because of the variable nature of the matchings between the objects of the problem and PHP objects (pigeons, holes and their capacities) that is needed for a successful application of the PHP. On the other hand, methods of Linear and Integer Programming look more favorable for the computer-oriented implementations. In this chapter, we consider both approaches, compare their bounding power and propose an alternative way of implementing the Pigeonhole Principle.

2.1 Pigeonhole Principle

Though sounding very simple in its initial form, the PHP is *"one of the most simpleminded ideas imaginable, and yet its generalizations involve some of the most profound and difficult results in all of combinatorics theory"* [78]. It is a simple and an extremely effective method of deriving upper bounds for combinatorial optimization problems. If empowered additionally by appropriate heuristics or mtching between the pigeons, the holes and the objects of the problem, the PHP often provides the easiest way of proving the optimality of a particular feasible solution or impossibility of attaining a certain value. Most commonly the PHP is introduced in the following way:

> **Pigeonhole Principle**: It is impossible to place $N+1$ pigeons into N holes so that there is at most one pigeon in each hole.

In this original formulation the PHP looks like a naïve kindergarten-level rule. It appeared in the literature also as a Dirichlet's Drawer Principle, and was used by Dirichlet in his study of the approximations of irrational numbers by rationals [14] in 1879. However, Gauss used it in 1801 [26], and it is likely that the principle in some form occurred in the literature even earlier.

When taken to a higher level of multiple objects per abstract unit or participating as a part of a multi-step logical proof, the Pigeonhole Principle quickly loses the nuance of obviousness, its applications involve some of the most profound and difficult results in combinatorics. Sometimes, the PHP is stated as providing an answer to the following not-so-trivial question about a finite set of elements and k categories, with every element belonging to one of the categories:

> **Pigeonhole Principle 2**: What is the smallest set of elements such that there is guaranteed to be n_1 elements in the first category, or n_2 elements in the second category, . . . or n_k elements in the k-th category?

In this form the PHP is viewed as the base case of Ramsey's Theorem. Furthermore, under this scope, Ramsey's Theorem itself is often perceived as a vast generalization of the Pigeonhole Principle [32]:

Ramsey's Theorem: Let $q_1, q_2, \ldots, q_n$ and t be positive integers with $q_i \geq t \quad i = 1, \ldots, n$. There exists a least positive integer $R(q_1, q_2, ...q_n; t)$ such that if the t-subsets of a finite set S with cardinality at least $R(q_1, q_2, ...q_n; t)$ are placed into n categories, then for some i there exists a subset $S' \subseteq S$ of size q_i ($|S'| = q_i$) which have all of its t-subsets in the i-th category.

Figure 2.1 shows the reduction of the Pigeonhole Principle 2 to the capacity argument: n_1, n_2, n_3 and n_4 are the thresholds of four shown categories. To stay below the threshold level, each bin i should be filled up by at most $n_i - 1$. Hence, $\sum_{i=1}^{N}(n_i - 1)$ is the maximum capacity of such a multi-bin system. Therefore, any quantity above this capacity will saturate at least one of the categories, and the answer to the Pigeonhole Principle 2 question is $1 + \sum_{i=1}^{N}(n_i - 1)$. When the number of categories is one, the two Pigeonhole Principle definitions coincide. Thus, the two historical definitions of the the PHP are equivalent.

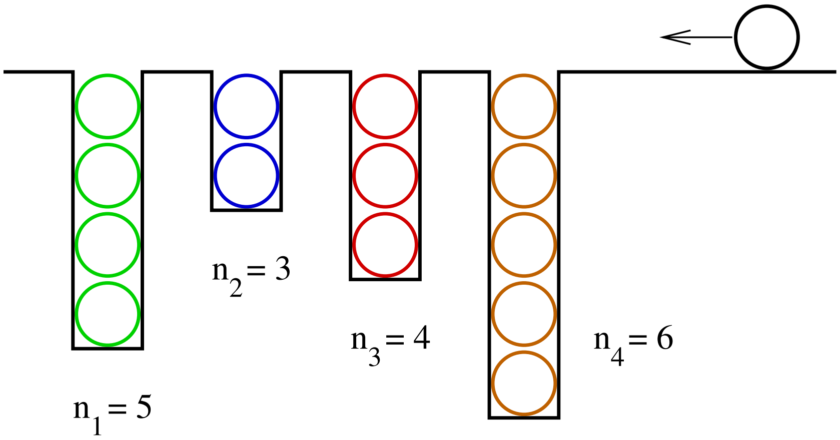

Figure 2.1: Illustration of the Pigeonhole Principle

With respect to a particular problem, we associate the Pigeonhole Principle with labeling some of the original objects of the problem with numbers, called capacities. If an upper bound provided by a capacity argument for such a labeling equals to the value of an optimal solution, we tell that the application of the PHP established the tight upper bound. For some combinatorial optimization problems, none of the labelings of the original objects lead to establishing the tight upper bound. Such problems may require representation changes that would enable labeling complex structures built upon the original objects of the problems. We call such extended applications of the PHP as the "Hidden" Pigeonhole Principle (HPHP). In a certain sense this split between the PHP and HPHP is similar to the difference between resolutions and extended resolutions (see Section 2.7), between solving Integer Programming problems by LPR and by the combinations of integer cuts and LPR (see Section 2.2). We give precise definitions of PHP and HPHP in Section 2.3.

To solve a combinatorial optimization problem by the PHP, one needs to represent a problem in such a way that the principle can be applied, i.e. to identify what in the problem should be mapped into objects (pigeons) and units (holes), and what their capacities are. For simple problems, the applications of the PHP are almost straightforward and are obtained directly from the nature of the problems. However, often a proof by the PHP requires additional heuristic knowledge that would allow to perform this type of mapping effectively. For some combinatorial optimization problems, one has to come up with pigeons and holes that are different from the original objects of the problem. Were this combination of the representation change and the mapping known in advance, the PHP would easily derive the tight upper bound and construct an optimal solution. For many combinatorial optimization problems, however, it is a very challenging task to find such a representation change and a mapping. The difficulty of this task complicates implementations of the PHP in AI reasoning, planning or scheduling systems, and encourages researchers to look for alternative methods.

The introduction of polynomial-time algorithms solving Linear Programming problems allows some Operations Research methods to be applied in an efficient manner to combinatorial optimization problems and establish upper bounds for such problems. For example, Linear Programming Relaxation (LPR) method can be applied effectively to solve some Integer Programming (IP) problems. LPR can be seen as requiring less effort to apply than the PHP, because in general, unlike the PHP, it does not need any additional knowledge or representation changes to derive upper bounds for IP problems.

The two approaches, namely PHP and LPR-based methods can be seen as "competitors" for solving combinatorial problems. It has been an open question[1]: Which method provides tighter upper bounds, when applied to the same combinatorial optimization problems. In this chapter, we report on our work in solving this open question. As hinted so far, our work analyzes carefully the problem representation issues involved in the two approaches. We formally introduce an appropriate representation change that makes the comparison and analysis possible. To clarify the status of the PHP in a non-traditional field of Integer Programming (IP), we re-define it as a particular method of solving IP problems.

Thus, our work includes:

1. Re-defining the PHP as a method of solving Integer Programming problems.

2. Proving that both the PHP and LPR establish the same upper bounds for problems stated in the Integer Programming form.

[1]Identified at the First International Workshop on AI and OR, Portland, OR, June, 1995.

3. Providing an alternative approach to solving combinatorial optimization problems for which the effective representation change required by the PHP is hard to find. The alternative approach consists of the following steps:
   (a) Convert a combinatorial optimization problem into an Integer Programming form.
   (b) Apply LPR to obtain an upper bound B which, as we will prove, is the same as the upper bound obtained by PHP.
   (c) Construct an optimal solution of value B.

2.2 Linear Programming Relaxation

Linear Programming Relaxation (LPR) is an effective Operations Research method of establishing upper bounds for Integer Programming problems [54]. The efficiency of its main calculation engine is supported by the discovery of polynomial methods of solving Linear Programming problems, such as the Ellipsoid algorithm [36]. Actually, LPR is a two-step procedure consisting of relaxing integer requirements and solving the corresponding Linear Programming problem.

Simplisticly, LPR can be viewed as a black-box with a particular instance of an Integer Programming problem as an input and a calculated upper bound as an output. From this point of view, LPR looks preferable to the PHP, because it does not need any heuristic knowledge to derive upper bounds. LPR can be applied to any instance of an Integer Programming problem without additional representation changes. Thus, the main questions of the competitive analysis between the PHP and LPR can be stated as the knowledge representation problem: Which form of presenting combinatorial optimization problems allows to establish tighter bounds?

2.2.1 Important Facts from Linear Programming

In this section we introduce basic definitions and facts from the theory of Linear and Integer Programming. Readers familiar with this subject may skip this section.

A particular instance of a Linear Programming (LP) problem consists of the goal function and the set of inequalities (constraints). Both the goal function and the constraints depend linearly on each variable. Throughout this section we consider variables to be real-valued, and their number is finite, the number of inequalities is also finite. Thus, a typical LP problem is of the following type:

$$\text{goal function}: \quad max(\sum_{i=1}^{N} c_i x_i)$$

$$\text{constraint set } J: \quad \sum_{i=1}^{N} a_{ij} x_i \le b_j \quad j = 1, \dots, M$$
$$x_i \in R \quad i = 1, \dots, N$$

Usually coefficients a_{ij}, b_j and c_i are rational, it allows to limit our consideration to rational-valued variables $x_i \in Q \quad i = 1, \dots, N$. To simplify further discussion, we introduce a particular LP problem which will help to illustrate the discussion:

$$\text{goal function}: \quad max(y) \tag{2.1}$$
$$\text{constraint set } J: \quad -x \le 0 \tag{2.2}$$
$$-y \le 0 \tag{2.3}$$
$$y - x \le 1 \tag{2.4}$$
$$x + y \le 3 \tag{2.5}$$
$$x + 2y \le 8 \tag{2.6}$$
$$x \in Q \tag{2.7}$$
$$y \in Q \tag{2.8}$$

Figure 2.2 shows the feasible region, optimal solution $x^* = (1, 2)$, and the goal vector corresponding to the goal function (2.1). Theory of Linear Programming states that for any optimal solution x^* there exists a subset of constraints, called tight constraints, such that:

- Inequalities representing tight constraints are actually equalities for x^*, or, equivalently, there is no slack for tight constraints with respect to x^*.
- The number of tight constraints can vary from 1 to M.
- Goal function is a positive combination of the left-hand sides of tight constraints.
- It is always possible to represent the goal function as a positive combination of at most N tight constraints.

In our example (2.1-8) inequalities (2.4) and (2.5) form the set of tight constraints with respect to the optimal solution $x^* = (1, 2)$. Thus, the goal function (2.1) can be presented as a positive combination of (2.4) and (2.5): $y = \frac{1}{2}(y - x) + \frac{1}{2}(x + y)$.

If we vary inequality (2.6), it may also become a tight constraint. For example, inequality $x + 2y \le 5$ is a tight constraint with respect to the optimal solution $x^* = (1, 2)$. If added to the existing set of tight constraints (2.4) and (2.5), it would constitute a redundancy: Left-hand

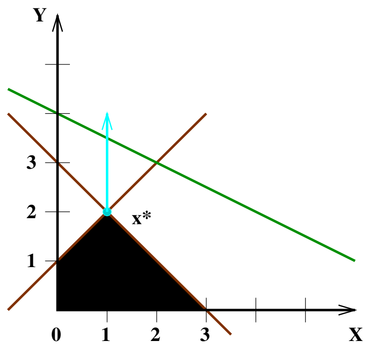

Figure 2.2: A Particular Instance of a LP Problem

sides of any two out of three inequalities can be used in a positive weighted sum to obtain the goal function. On the other hand, the goal function may also vary within a certain range to preserve the same optimal solution and the set of tight constraints. In problem (2.1-8) for any goal function of the form $y + \alpha x$ with $\alpha \in [-1, 1]$, the set of tight constraints consists of (2.4) and (2.5). For the extreme values of $\alpha \in \{-1, 1\}$ one of the tight constraints becomes redundant, since the goal function coincides with the left-hand side of the other tight constraint. Moreover, in both extreme cases there exists an infinite number of feasible solutions attaining the same optimal value. These optimal solutions form a face of a polyhedron of feasible solutions defined by the IP problem's constraints. For example, in problem (2.1-8) the set of feasible solutions forms a 2D tetragon (see Figure 2.2), and the set of optimal solutions is either a zero-dimensional face (a single 2D point) or a one-dimensional face (one of the tetragon's sides).

Thus, on one hand, a positive weighted sum of left-hand sides of tight constraints establishes an upper bound: Since $\sum_j \sum_i \alpha_j a_{ij} x_i = \sum_i c_i x_i$, where $\alpha_j \geq 0$, and $\sum_i a_{ij} x_i \leq b_j$, then $\sum_i c_i x_i \leq \sum_j \alpha_j b_j$. On the other hand, any feasible solution $\bar{x}$ sets a lower bound for the goal function $\sum_i c_i \bar{x}_i$. Furthermore, tight constraints have no slack with respect to optimal solution x^*, therefore, the lower bound provided by x^* coincides with the upper bound established by the positive weighted sum of tight constraints and both are equal to the optimal value $\sum_i c_i x_i^*$.

2.2.2 Definition of Linear Programming Relaxation

Compared with Linear Programming problems, Integer Programming (IP) problems have an additional requirement that some of its variable are integer-valued. Throughout this chapter we consider only those IP problems that have linear goal functions and linear constraints. In general, an addition of integer-valued variables makes an Linear Programming problem NP-hard [25]. However, in some particular cases, it is possible to solve IP problems efficiently, by applying Linear Programming Relaxation (LPR) or a combination of Integer Cuts and LPR

[54]. Since Integer cuts result in adding new feasible constraints that take into account the integrality of variables, we view such a technique as performing a representation change on the problem domain. As we show in Section 2.5, such a perception corresponds completely to traditional representation changes for combinatorial optimization problems that can be stated in the Integer Programming form.

Linear Programming Relaxation is a simple procedure that consists of two steps:

1. Drop (relax) integrality requirements for all variables.

2. Solve the correspondent LP problem.

For some IP problems, like problem (2.1-8) introduced earlier, LPR outputs an integer feasible solution, thus, solving such IP problem. In other cases, when LPR outputs a fractional optimal solution x^*, it establishes an upper bound $\sum c_i x_i^*$ for the goal function. In such cases, LPR is usually not very helpful in indicating the tightness of the upper bound. One has to come up with a feasible integer solution that matches the derived upper bound to prove that the bound is tight.

Integer cuts produce additional constraints that utilize the integral nature of a subset of variables. Additional constraints can be derived in many different ways, Integer cuts produce ones that cannot be obtained by taking positive weighted sums of the existing inequalities. In conjunction with LPR, Integer cuts are capable of solving IP problems, whereas the efficiency of this combination in obtaining tight upper bounds relies on the "quality" of newly created constraints constructed by Integer cuts techniques. Examples of the Hidden Pigeonhole Principle's applications, namely the "Mutilated Checkerboard" problem and the Firm Tiling problem discussed in Section 2.5, illustrate an alternative way of solving these problems through Chvatal-Gomory's (Integer) cuts and LPR.

2.3 Converting Problems into the Integer Programming Form

We identified that the main reason of the lack of ability to compare the efficiency of PHP and LPR is that different problem representations are required by the two methods. Our work consisted of developing appropriate changes of representations that made possible a competitive analysis of the efficiency of these two methods.

To make both PHP and LPR applicable to the same combinatorial optimization problems, we perform representation changes for each problem discussed in this paper to convert them into Integer Programming problems of the following form:

$$\text{goal function}: \qquad max(\sum_{i=1}^{N} c_i x_i)$$

$$\begin{array}{rl} \text{constraint set } J: & \sum_{i=1}^{N} a_{ij} x_i \leq b_j \quad j = 1, \ldots, M \\ \text{integrality requirements}: & x_i \in Z \quad i = 1, \ldots, N \end{array}$$

A combinatorial optimization problem is defined as finding a solution $x^* = (x_1^*, \ldots, x_N^*)$ that is feasible, i.e. satisfies all the constraints from J and the integrality requirements, and also attains the best value of the goal function among all feasible solutions.

We considered a series of known combinatorial optimization problems, to which the Pigeonhole Principle is traditionally applicable. The empirical evidence obtained through this study suggested that the following definition reflects correctly the combinatorial nature of the Pigeonhole Principle.

Definition 2.3.1 *We say that the upper bound for an IP problem is derived by the Pigeonhole Principle, if there exists a subset of constraints $\hat{J} \subseteq J$, such that the sum of the left-hand sides of the inequalities from $\hat{J}$ (repetitions are allowed in $\hat{J}$) is a multiple of the goal function*

$$\sum_{j \in \hat{J}} \sum_{i=1}^{N} a_{ij} x_i = k \sum_{i=1}^{N} c_i x_i \leq \sum_{j \in \hat{J}} b_j$$

and the derived bound is the smallest scaled down sum of right-hand sides $\frac{1}{k} \sum_{j \in \hat{J}} b_j$.

In other words, the PHP establishes an upper bound for an IP problem that corresponds to the smallest value $B = \min_{\hat{J} \subseteq J} \frac{1}{k} \sum_{j \in \hat{J}} b_j$, such that $\sum_{j \in \hat{J}} \sum_{i=1}^{N} a_{ij} x_i = k \sum_{i=1}^{N} c_i x_i$ holds for some positive $k > 0$. If it happens that a feasible solution x^* is known, which attains the derived value B, then we say that the PHP establishes a tight upper bound. Whenever the PHP establishes a tight upper bound, we say that a problem admits a proof by the PHP.

In general, the procedure of considering a subset of constraints and summing them up (possibly with positive weights) results in setting upper bounds for the value of an optimal solution. Any feasible solution provides a lower bound for the optimal value of the problem. We identify applications of PHP with finding a subset of constraints and obtaining the optimal value of the goal function. In this case, the established upper bound matches the lower bound provided by the found optimal solution.

Definition 2.3.2 *If a set of constraints added to an Integer Programming problem converts the original IP problem into another IP problem that admits a proof by the Pigeonhole Principle, we say that the original problem admits a proof by the Hidden Pigeonhole Principle (HPHP).*

Additional constraints can be obtained in many different ways. Weighted positive sums, for example, can simplify some of the existing constraints, but they would not contribute any restrictions on fractional optimal solutions obtained by Linear Programming Relaxation. In Section 2.4 we consider an Integer cuts technique that allows to derive tighter valid inequalities and to keep all integer solutions feasible. Constraints obtained from Branch-and-Bound methods can also be sought as an addition to the existing set of constraints. We consider the way of constructing additional constraints and the sanity check that an optimal integer solution has not been cut off to be the responsibility of the problem solver.

The introduced representation change allows to re-formulate the original PHP statement with 11 pigeons and 10 holes as the following IP problem:

$$
\begin{aligned}
& max(\sum_{i=1}^{11}\sum_{j=1}^{10} x_{ij}) \\
& \sum_{i=1}^{11} x_{ij} \leq 1 \quad j = 1, \dots, 10 \\
& x_{ij} \in \{0, 1\},
\end{aligned}
$$

where x_{ij} represents the amount of the ith pigeon in the jth hole. If we drop integer requirements and sum up all the constraints, we obtain an upper bound: $\sum_{i=1}^{11}\sum_{j=1}^{10} x_{ij} = \sum_{j=1}^{10}\sum_{i=1}^{11} x_{ij} \leq 10$. Since an obvious solution $\{x_{ii} = 1 \ \text{ for } i = 1, \dots, 10; \ x_{ij} = 0 \ \text{ for } i \neq j\}$ provides the same value, we proved it to be optimal. For this simple problem, PHP and LRP approaches are identical.

In most obvious cases, the desired subset of constraints is the whole set of original inequalities. Consider, for example, the problem of placing chess kings on a circular chessboard with even number of squares (see Figure 2.3). One is supposed to find the maximal number of kings that can be placed on such a board so that no king is attacking another. Recall that a chess king attacks all its adjacent squares. For the board presented in Figure 2.3, we get the following IP problem:

$$
\begin{aligned}
& max(\sum_{i=1}^{N} x_i) \\
& x_1 + x_2 \leq 1 \\
& x_2 + x_3 \leq 1 \\
& \dots \\
& x_{N-1} + x_N \leq 1 \\
& x_N + x_1 \leq 1 \\
& x_i \in \{0, 1\}
\end{aligned}
$$

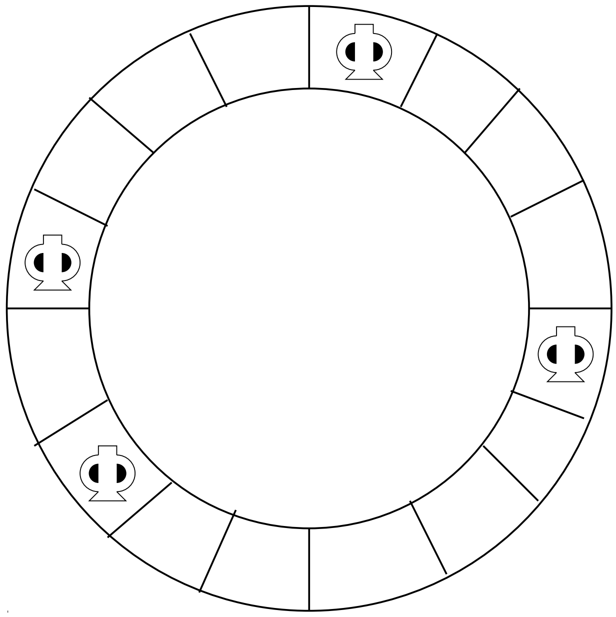

Figure 2.3: Chess kings on a circular chessboard.

If we sum up all the inequalities, we get $2\sum_{i=1}^{N} x_i \leq N$, which is equivalent to $\sum_{i=1}^{N} x_i \leq N/2$. The same upper bound can be obtained from applying a combination of two-coloring and PHP: if we color the circular board with even number of squares in alternating black and white colors, we can place kings on either color attaining the optimum value of $N/2$ derived this way by PHP. Thus, the upper bounds are the same, however, the application of LPR seems to be easier, as it does not need any additional heuristic knowledge.

The case when a proper subset of constraints is involved in obtaining a tight upper bound is more complicated. To apply the Pigeonhole Principle in the original combinatorial form, one has to find which problem's objects should be matched with pigeons and which - with holes. In the Integer Programming form it means that one has to come up with a rule (heuristic) of finding a desired subset of constraints to sum them up. For some problems, it is easy to find such a subset, whereas for others it is not obvious. For example, the popular N-Queen problem [24, 53], which is the problem of placing the maximal number of chess queens on NxN-chessboard, so that no queen is attacking another, can be represented as the following IP problem:

$$max(\sum_{i=1}^{N}\sum_{j=1}^{N} x_{ij})$$

$$\text{for each } i \in \{1,\ldots,N\} \text{ and } j \in \{1,\ldots,N\} \quad \begin{cases} \sum_{k=1}^{N} x_{ik} \le 1 \\ \sum_{k=1}^{N} x_{kj} \le 1 \\ \sum_{(k,l)\in I(i,j)} x_{kl} \le 1 \\ \sum_{(k,l)\in J(i,j)} x_{kl} \le 1 \\ x_{ij} \in \{0,1\} \end{cases}$$

where $I(i,j)$ and $J(i,j)$ are the sets of squares which a chess queen threatens along two diagonals from the square (i,j), including the square (i,j). In this form we have four groups of constraints: row, column, and two diagonal constraints, one of each type per square. For this problem it is easy to find a subset of constraints that provides the tight upper bound. If we sum up N inequalities corresponding only to row constraints for squares from different rows, we get N as the upper bound: $\sum_{i=1}^{N}\sum_{j=1}^{N} x_{ij} \le N$. Beginning with $N = 4$, the problem has an N-Queen solution [24] (see Figure 2.4).

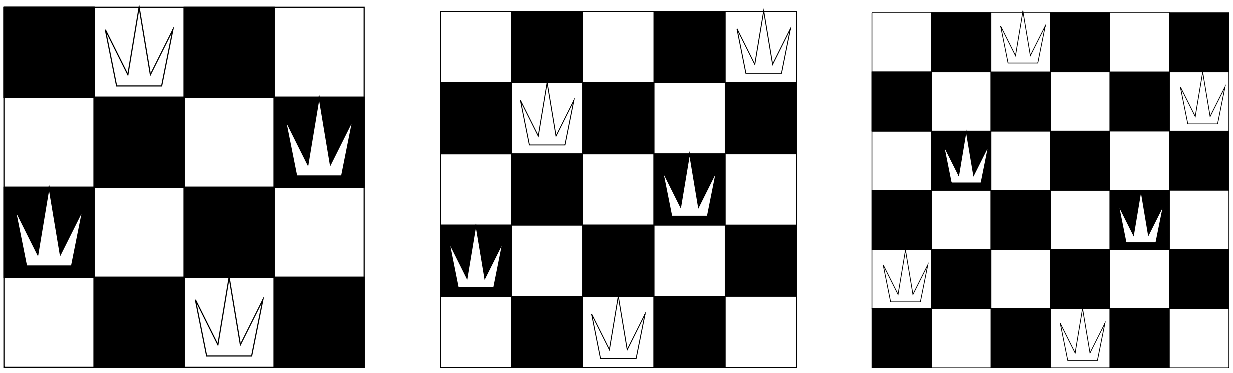

Figure 2.4: N-Queen Problem Solutions for N = 4, 5, 6

Though PHP readily provides the tight upper bound, it is not always immediately clear how to construct an optimal solution that will attain the obtained bound. The N-Queen problem stimulated the development of a generation of backtracking algorithms constructing optimal solutions for the N-Queen problem, which is a hard problem itself. The Chess Knight problem [4] is one of the jewels of PHP applications; it is the one for which the selection of a constraint subset is a challenging task. We discuss this problem in the following section in detail.

2.4 PHP is the Dual of LPR

In this section, we discuss the duality relation between the Pigeonhole Principle and the Linear Programming Relaxation. As we concluded in the previous section, for some problems, it is tempting to apply LPR in a brute-force manner. Instead of requesting additional heuristics that will suggest how to select the desired subset of constraints, one can apply already developed methods of Linear Programming with the hope of deriving the same or even better bound. This brings us back to the main question of this chapter: Is it true that PHP and LPR always provide the same upper bound? The following theorem answers this question:

Theorem 2.1 *If an Integer Programming problem admits a proof by the Pigeonhole Principle, then LPR provides the same optimal value. Conversely, a bound derived by LPR can be matched by PHP.*

Proof: The re-defined in Section 2.3 Pigeonhole Principle is exactly the statement of the non-degenerous *dual* problem. Indeed, if an IP problem admits PHP in deriving the tight upper bound, then there exists a feasible integer solution x^* attaining the derived bound. On the other hand, there exists a subset of constraints $\hat{J}$, sum of the left-hand sides of which is an integer multiple of the goal function: $\sum_{j \in \hat{J}} \sum_i a_{ij} x_i = k \sum_i c_i x_i$, and the value of x^* is the same multiple of bounding constants: $\sum_{j \in \hat{J}} b_j = k \sum_i c_i x_i^*$.

If we apply LPR to the IP problem, it will provide an integer or a fractional optimal solution x^{**} for a relaxed problem. Since PHP has derived the tight upper bound, and the relaxed problem is obtained from the original IP problem by dropping integrality requirements, x^* was one of the candidates for LPR's optimal solution, and the value of LPR's solution x^{**} is equal to PHP's tight upper bound: $\sum_{i=1} c_i x_i^{**} = \sum_{i=1} c_i x_i^*$.

According to the theory of Linear Programming, there exists a subset of the constraint set, called tight constraints, such that the goal function is a positive weighted sum of the left-hand sides of the constraints from this subset $\sum_{j \in \hat{J}} \sum_i \alpha_j a_{ij} x_i = \sum_i c_i x_i$. Moreover, an optimal solution x^{**} has no slack for each of the constraints from $\hat{J}$, that is, satisfies them as equality $\sum_i a_{ij} x_i^{**} = b_j$ for $j \in \hat{J}$. Since all the coefficients in the constraints J and the goal function are integer, all the weights $\alpha_j \geq 0$ (positive coefficients) of the weighted sum are rational. We can find an integer k to scale all rational coefficients up $\beta_j = k\alpha_j$ and make all of β_j integer. In this case, k plays the role of a scaling coefficient, integer β_j tells how many times should the tight constraint $j \in \hat{J}$ be used in an "unweighted" sum of left-hand sides $\sum_{j \in \hat{J}} \beta_j \sum_i a_{ij} x_i = k \sum_i c_i x_i$. Hence, the upper bound $1/k \sum_{j \in \hat{J}} \beta_j b_j = \sum_i c_i x^{**}$ can be matched by PHP.

Therefore, the set of tight constraints forms the desired subset $\hat{J}$ and positive integer weights determine the number of repetitions in $\hat{J}$. Therefore, the value of the solution derived by LPR does not improve the upper bound provided by PHP and, conversely, it can be mimicked by PHP to establish the same upper bound. ■

Theorem 2.1 provides the following answer to the main question of the chapter: *If the optimal value is obtained by PHP for a combinatorial optimization problem stated in the Integer Programming form, LPR provides the same bound as PHP.*

Corollary 1 *The Pigeonhole Principle is the Dual of Linear Programming Relaxation.*

Thus, Definition 2.3.1 from Section 2.3 enabled us to state an interesting relation between two very different approaches. How legitimate is Definition 2.3.1, aren't we pushing the envelope too much? We place this discussion after presenting the results and the proof of Theorem 2.1, because we need to use them in our discussion. We identify the Pigeonhole Principle for IP problems with establishing a set of objects (holes) with associated capacities. From the proof of Theorem 2.1, it immediately implies that such PHP is as powerful in establishing upper bounds as LPR: Tight inequalities constitute holes, their right-hand sides determine holes' capacities, sum of the capacities of holes produce the same upper bound as LPR. On the other hand, the Pigeonhole Principle was defined (see Section 2.1) as the principle dealing with the objects of the problem. Therefore, if the problem is stated in the IP form, PHP cannot produce a tighter upper bound than LPR, other than through changing the problem statement, because none of the combinations of the IP problem objects – variables, inequalities, goal function – can improve LPR's bound. By performing representation changes, for example, Integer cuts, a problem solver changes the problem statement, hence, such an improved bound and the solution itself, should be attributed to the "Hidden" Pigeonhole Principle (see Definition 2.3.2). Such a split between PHP and HPHP is similar to the difference between the techniques of resolutions and extended resolutions (See Section 2.7).

For example, the Firm Tiling problem with the double-firm requirement (see Section 2.5) can be trivially shown to be infeasible by the Pigeonhole Principle. Single-firmness does not imply non-feasibility immediately. One needs to perform a representation change or to come up with a powerful heuristic that would reduce single-firmness to double-firmness, hence, change the problem statement. Furthermore, a simple application of the Pigeonhole Principle is not capable of showing infeasibility in the single-firm problem version. This argument confirms the correctness of our definition that draws a splitting line between PHP and HPHP, with the former one thus becoming the dual of Linear Programming Relaxation.

The discussion on the duality relation became possible after bringing both methods to the common ground of Integer Programming. We demonstrate that the above result is non-trivial by solving the Chess Knight problem [4]: "What is the maximal number of chess knights that can be placed on the 8x8 chessboard in such a way that they do not attack each other?" The classical elegant solution relies on the existence of a Hamiltonian tour of length 64 following the knight moves. One of such tours is presented in Figure 2.5.

8	43	24	53	6	41	22	51
25	54	7	42	23	52	5	40
44	9	62	15	46	31	50	21
55	26	45	32	63	14	39	4
10	33	16	61	30	47	20	49
27	56	29	64	13	60	3	38
34	11	58	17	36	1	48	19
57	28	35	12	59	18	37	2

Figure 2.5: Hamiltonian Tour on a Chessboard for the Knight

One can split the tour into 32 pairs of chess squares that are adjacent in the sense of the knight move, and apply PHP: Each pair can contain at most one knight, otherwise two knights would attack each other. For example, consider pairs of squares $(1, 2)$, $(3, 4)$,...,$(63, 64)$. None of them can accommodate more than one knight (see Figures 2.5 and 2.6). Thus, 32 pairs of adjacent squares can accommodate at most 32 knights. Although this simple proof does not provide us with a solution to the problem, (for example, it allows to place knights on squares 4 and 5 from pairs $(3, 4)$ and $(5, 6)$), it provides a tight upper bound.

Moreover, this proof gives an impression of using a "hidden" application of the Pigeonhole Principle, whereas the Hamiltonian tour is just a heuristic for finding a subset of the constraint set. The Chess Knight problem for the standard chessboard can be presented as the following Integer Programming problem:

$$max(\sum_{i=1}^{8}\sum_{j=1}^{8} x_{ij})$$
$$x_{ij} + x_{kl} \leq 1 \quad i = 1, \ldots, 8 \;\; j = 1, \ldots, 8 \;\; (k, l) \in U(i, j)$$
$$x_{ij} \in \{0, 1\}$$

where x_{ij} represents the amount of knights in the square (i, j); $U(i, j)$ is the set of squares on the chessboard which a chess knight threatens from the cell (i, j) (see Figure 2.6).

If applied to the Chess Knight problem, the Linear Programming Relaxation method provides the same upper bound of 32. Unlike PHP, LPR does not require heuristics to identify any subset of constrains, its calculational routine considers tight constraints inside the solving process. However, LPR is likely to produce fractional solutions, say $x_i = 1/2$ for $i = 1, 2, \ldots, 64$.

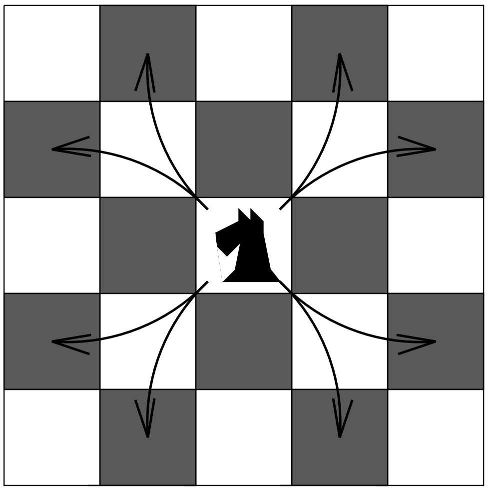

Figure 2.6: The Set of Adjacent Squares for the Knight on a Chessboard

Theorem 2.1 shows that, if applied, the original Pigeonhole Principle will provide the same value. So, if the Hamiltonian tour heuristic were not known, the application of LPR would tell that 32 is the best upper bound that can be obtained by the original PHP. Since a chess knight alternate colors (see Figure 2.6), one can place 32 knights on the chess squares of the same color, thus attaining the optimal value and constructing the optimal solution for the Chess Knight problem.

From the duality relation, it follows that, if an optimal solution of value B for a combinatorial optimization problem is derived by the Pigeonhole Principle, an optimal solution for the last problem in the following chain of representation changes has the same value B:

Original combinatorial problem$\rightarrow$ IP problem $\rightarrow$ Linear Programming problem.

Examples discussed in this section showed that an additional effort is needed to apply the Pigeonhole Principle. For some problems it is easy to match problem's objects with pigeons and holes, in some cases it is a state of the art. Often PHP applications hint on how to construct optimal solutions, though for some problems it is an independent difficult problem.

In its turn, LPR can be applied to any instance of an IP problem without need in additional knowledge. Unfortunately, LPR often outputs a fractional optimal solution, which does not shed any light on how to transform it into an integer one of the same value. We continue the discussion on benefits and disadvantages of PHP and LPR in the next section.

2.5 More Complicated Applications of PHP

In the previous sections, we were able to apply PHP and LPR in a brute-force manner, because each Integer Programming problem contained a subset of constraints, sum of which provided the desired bounds for values of the goal functions. We call such a case a regular application of PHP, as opposed to the "Hidden" Pigeonhole Principle (HPHP) that requires additional representation changes and heuristic knowledge to fulfill a similar task.

This section is devoted to the discussion on HPHP and its relation to the original Pigeonhole Principle. To make it more intuitive, we illustrate the discussion by the classical Mutilated Checkerboard [55] and the Firm Tiling problems:

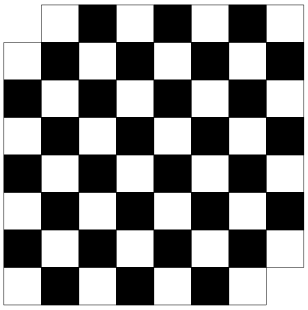

Figure 2.7: A Mutilated Checkerboard

Mutilated Checkerboard Problem: Consider an NxN checkerboard with two opposite corners removed (see Figure 2.7). Can one cover this "mutilated" checkerboard completely by non-overlapping domino pieces, each of the size of two squares of the checkerboard?

Firm Tiling Problem: Consider a checkerboard of size 6x6 made of a soft square cloth and 18 hard tiles of size 1x2. Can one glue all 18 tiles to such a checkerboard, so that the "middle-cut" requirement is satisfied, i.e. each splitting line inside the checkerboard goes through the middle line of at least one tile?

Figure 2.8 shows a firm 17-tile solution. The "middle-cut" requirement for a particular splitting line restricts the cloth to be folded along this splitting line: If the line crosses the middle of at least one tile, the cloth cannot be folded along this line unless the tile is broken. This is what we call a *"single-firm"* tiling. A more restrictive *"double-firm"* tiling requires each splitting line to cross middle lines of at least two tiles. In this section we show that a *"single-firm"* complete tiling implies a *"double-firm"* tiling in the Firm Tiling problem.

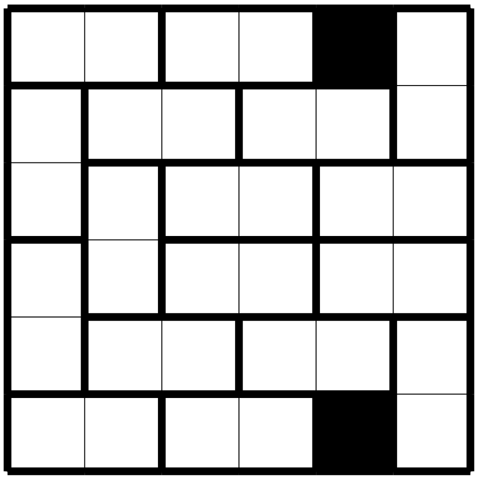

Figure 2.8: Firm 17-Tile Solution.

Following the proposed approach, the first step needed is the representation change converting the above problems into Integer Programming problems. To accomplish this step, one needs to model the fact that domino pieces do not overlap in the Integer Programming form. We assign variables x^v_{ij} to vertical splits, where i is the row number and j is the split in ith row between squares (i, j) and $(i, j+1)$. In the same way, we define variables x^h_{ij} for horizontal splits between squares (i, j) and $(i+1, j)$. One-valued variables $x^v_{ij} = 1$ model the horizontal placement of a domino piece (tile) in such a way so that its middle line is located at ith vertical splitting line and jth row; one-valued variables $x^h_{kl} = 1$ model the vertical placement of a domino piece (tile) with its middle line at kth horizontal split in lth column. In these terms, non-overlapping can be represented by the following set of constraints:

$$x^v_{ij} + x^v_{i-1j} \leq 1 \quad x^v_{ij} + x^v_{i+1j} \leq 1 \quad x^v_{ij} + x^h_{i-1j-1} \leq 1 \tag{2.9}$$

$$x^v_{ij} + x^h_{i-1j} \leq 1 \quad x^v_{ij} + x^h_{ij-1} \leq 1 \quad x^v_{ij} + x^h_{ij} \leq 1 \tag{2.10}$$

Figure 2.9 demonstrates a part of a checkerboard and the correspondence of domino pieces (tiles) placing to 0/1-variables and splits.

In the Mutilated Checkerboard problem the goal function is the unweighted sum of all domino-piece variables: $\sum_{ij} x^v_{ij} + \sum_{ij} x^h_{ij}$. It is easy to guess one of the fractional optimal solutions produced by LPR: $x^*_{**} = 1/2$. It tells that if applied, PHP will provide the same bound of 31 (which is not tight). Furthermore, an optimal fractional solution does not help to find an optimal integer one, even when the upper bound is tight.

If the original checkerboard is of size NxN with N odd, a simple PHP application shows that one can use at most $\frac{N^2-3}{2}$ domino pieces for a non-overlapping covering of the mutilated checkerboard: Each piece contains two squares, and there are $N^2 - 2$ squares in the mutilated

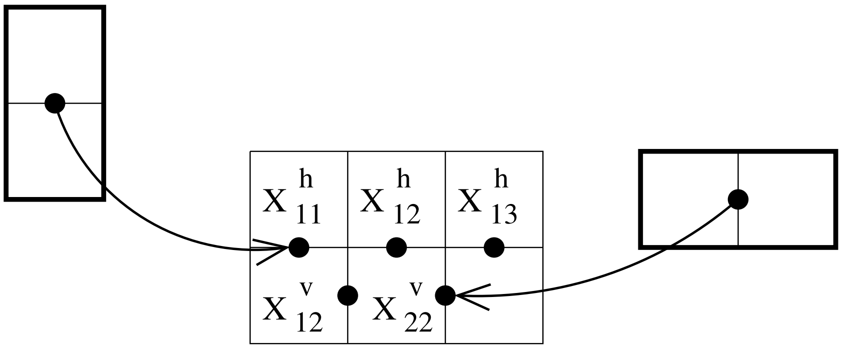

Figure 2.9: Modeling Domino Overlap as a Set of Inequalities

checkerboard. Therefore, one can put at most $\lfloor \frac{N^2-2}{2} \rfloor = \frac{N^2-3}{2}$ non-overlapping domino pieces. Actual attempts to cover the mutilated checkerboard with odd sizes readily give an optimal solution of the above value.

If N is even, the original PHP does not put any additional restrictions on the number of pieces. However, none of the attempts to cover the mutilated checkerboard by domino pieces achieves the desired bound of $\frac{N^2-2}{2}$, all constructed solutions provide at most $\frac{N^2-4}{2}$ pieces. Neither PHP nor LRP provide a tight upper bound for even-sized mutilated checkerboards. Nonetheless, the heuristic of two-coloring the mutilated checkerboard and applying PHP to the monochrome set of squares of smaller size completes the proof of the fact that $\frac{N^2-4}{2}$ is actually the optimal value for covering the mutilated checkerboard with even sizes. If we color the original checkerboard in usual black-and-white chess colors (see Figure 2.7) and then cut off 2 opposite corner squares (of the same color), the remaining mutilated checkerboard contains unevenly colored square sets. The checkerboard presented in Figure 2.7, for example, has 30 black squares and 32 white squares. Since each domino piece contains 2 squares of the opposite colors, applying PHP to a smaller set of black or white squares, we obtain the tight upper bound of $\frac{N^2-4}{2}$. This argument completes the proof of the Mutilated Checkerboard problem by HPHP. Such a modification of PHP is favorable in comparison with LPR and the original PHP, because it is capable of deriving the optimal value of the goal function. As we mentioned before, Linear Programming Relaxation provides $\frac{N^2-2}{2}$ as an upper bound. According to Theorem 1, if applied, PHP outputs exactly the same upper bound.

The idea of two-coloring is a simple elegant heuristic that allows to apply the Hidden Pigeonhole Principle and derive the tight upper bound. After being known for several dozen years, this application of HPHP might seem to be too simple to stimulate the development of alternative methods. We introduce the Firm Tiling problem as an example of a problem which is hard to solve without prior knowledge of the appropriate heuristic. For example, multi-coloring does not help to find the desirable matching between tiles, checkerboard squares, pigeons and holes.

Theory of Integer Programming suggests to apply Integer cuts, for example, Chvatal-Gomory's cuts, to extend the set of valid inequalities. The main idea of Chvatal-Gomory's cut (CG-cut) is to use the integrality of variables in feasible solutions. If a weighted sum of the inequalities derived so far contains the left-hand side with integer coefficients, the right-hand side can be rounded down to the nearest integer: $\sum_{i=1}^{N} a_i x_i \leq b$ implies $\sum_{i=1}^{N} a_i x_i \leq \lfloor b \rfloor$, for integer a_i and $x_i \quad i = 1, \ldots, N$. In a certain sense CG-cuts look as simple as the original Pigeonhole Principle. However, CG-cuts allow to derive constraints that cannot be obtained from the initial set of inequalities by taking weighted positive sums.

We, first, demonstrate the correctness of the Integer Programming formulation of the Mutilated Checkerboard problem and then derive an upper bound for it by means of Chvatal-Gomory's cuts and the Hidden Pigeonhole Principle.

Since we are about to apply some of the techniques of Integer cuts to the Mutilated Checkerboard problem, we first demonstrate simple reductions from Integer Programming theory. For example, Lemma 1 builds a reduction from the set of pairwise constraints with 0/1-vertex variables to the Exclusive Rule for a clique K_N, where a clique is a complete graph with $N \geq 2$ vertices.

Lemma 1 *If a clique K_N admits exclusively 0/1-assignments to its vertices $v_1, \ldots, v_N$ and, for each pair of vertices (v_i, v_j), at most one vertex can be assigned to 1, then there is at most one vertex assigned to 1 in the whole clique K_N.*

Proof: There are $\frac{N(N-1)}{2}$ inequalities of the type $x_i + x_j \leq 1$. We would like to prove that this collection of constraints implies a single clique inequality $\sum_{i=1}^{N} x_i \leq 1$ (Exclusive Rule) for 0/1-variables $x_i \in \{0, 1\} \quad i = 1, \ldots, N$. We prove it by induction on the number of vertices.
Induction Base: If $N = 2$, the given inequality and the clique inequality $x_1 + x_2 \leq 1$ coincide.
Induction Step: Suppose that we can derive all N clique inequalities for each of the N sub-cliques of size $N - 1$. If we sum them up, we get:

$$\sum_{j=1}^{N} \sum_{i \neq j} x_i \leq \sum_{j=1}^{N} 1,$$

which implies that $(N-1) \sum x_i \leq N$ or, equivalently, $\sum x_i \leq N/(N-1)$. Since $N > 2$, $\lfloor \frac{N}{N-1} \rfloor = \lfloor 1 + \frac{1}{N-1} \rfloor = 1$. If we apply CG-cut to the last inequality, we get the desired N-clique inequality:

$$\sum_{i=1}^{N} x_i \leq 1. \blacksquare \tag{2.11}$$

Four (or less) splits that form the border of the square (i, j) correspond to four variables that model placing of domino pieces (tiles). Corner or side squares border with only two or three internal splits. According to Lemma 1, if we consider all six (or less) pairwise inequalities (2.9-10) involving four variables bordering a checkerboard square, CG-cuts provide the Exclusive Rule for the clique associated with the square. This Exclusive Rule corresponds to the requirement of non-overlapping of domino pieces over the square (i, j). The addition of the clique inequalities derived by CG-cuts to the initial set of constraints constitutes a representation change of the IP problem.

Lemma 2 *If clique inequalities for all squares of the NxN mutilated checkerboard (N is even) are added to the set of constraints, the optimal value of the relaxed Linear Programming problem is* $\frac{N^2-4}{2}$.

Proof: Consider the following subset of clique constraints: Pick a monochrome subset of squares of smaller size and sum up all the clique constraints corresponding to these squares (of the same color). Each clique constraint is an unweighted sum of four (or less) clique variables $x_{i_1} + x_{i_2} + x_{i_3} + x_{i_4} \leq 1$. Since squares of the same color do not share sides, the sum of all clique constraints corresponding to squares of the same color is an unweighted sum of variables. On the other hand, all variables are presented in the final sum, because a legitimate placement of a domino piece covers squares of both colors. Since the number of clique constraints corresponds to the number of monochrome squares of smaller size, the sum of the constraints establishes a tighter bound for the goal function $\sum x_i \leq \frac{N^2-4}{2}$. Knowing this bound, it is easy to come up with a solution for an even-sized Mutilated Checkerboard problem that attains this value. ■

Thus, the proof that uses HPHP, relies on additional knowledge that each domino piece covers a bi-chromatic configuration, whereas the combination of LPR with CG-cuts takes into account this argument automatically. CG-cuts expand the set of constraints by adding clique inequalities for all checkerboard squares. After that LPR solves the new Integer Programming problem. We identify the expansion of the constraint set with the representation changes for the original Integer Programming problem. If the two-coloring heuristic were not known, then LPR with CG-cuts could be used to obtain the tight upper bound of $\frac{N^2-4}{2}$. After that, it is relatively easy to construct a solution attaining this value, which is proven to be optimal. The relations between HPHP and LPR with Integer cuts in solving the Mutilated Checkerboard problem are very similar to those between PHP and LPR.

The Firm Tiling problem does not admit the proof by the original PHP, because an obvious fractional solution $x^*_{**} = 1/2$ is feasible for a relaxed LP problem and the "middle-cut" requirement

$$\begin{array}{ll} \sum_{j=1}^{6} x_{ij}^h = \sum_{j=1}^{6} \frac{1}{2} = 3 \geq 1 & i = 1, 2, \ldots, 5 \\ \sum_{i=1}^{6} x_{ij}^v = \sum_{i=1}^{6} \frac{1}{2} = 3 \geq 1 & j = 1, 2, \ldots, 5 \end{array}$$

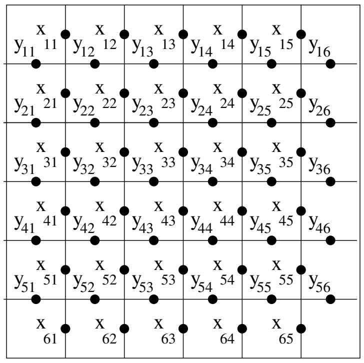

Figure 2.10: Modeling a Firm Tiling of a 6x6 Checkerboard.

is satisfied for all internal splitting lines. However, brute-force attempts to construct a complete firm tiling fail to provide an optimal 18-tile solution. The best firm tiling consists of at most 17 non-overlapping tiles, Figure 2.8 shows one of such tilings.

Nonetheless, the Hidden Pigeonhole Principle is capable of setting the tight upper bound of 17 tiles for this problem. To preserve the beauty of the elegant solution by HPHP for now, we first establish the tight upper bound through the combination of LPR and CG-cuts. We show that if one more constraint

$$\sum x_{ij}^h + \sum x_{ij}^v > 17 \tag{2.12}$$

is added to the original IP problem, the set of feasible integer solutions becomes empty.

Figure 2.10 presents 6x6 checkerboard with 0/1-variables assigned to its internal splits. To avoid superscript notations we denote x_{ij}^v as x_{ij} and x_{ij}^h as y_{ij}. Since x_{ij} and y_{ij} are integer, inequality (2.12) implies

$$\sum x_{ij} + \sum y_{ij} \geq 18. \tag{2.13}$$

In its turn, the latter inequality implies that the whole checkerboard should be covered by tiles. Since clique inequalities prohibit tiles from overlapping, each square is covered exactly by half-a-tile[2]. This is not a surprising conclusion, as we are attempting to cover a 6x6 checkerboard by 18 non-overlapping tiles.

Lemma 3 *Inequality (2.13) and clique inequalities (2.11) for all squares of the 6x6 checkerboard imply* "double-firm" *tiling.*

[2]This fact can be proved by considering a subset of clique constraints corresponding to a monochrome set of checkerboard squares in the way similar to the Mutilated Checkerboard problem.

Proof: We prove the statement of the Lemma by induction. In the induction base, we show it for the leftmost vertical splitting line. Due to the symmetry this proof remains unchanged for horizontal splitting lines.

Induction Base: If we consider the leftmost column of a checkerboard presented in Figure 2.10, inequalities (2.9-10) and (2.13) imply the following six equalities:

$$x_{11} + y_{11} = 1 \quad x_{21} + y_{11} + y_{21} = 1 \quad x_{31} + y_{21} + y_{31} = 1 \tag{2.14}$$

$$x_{61} + y_{51} = 1 \quad x_{41} + y_{31} + y_{41} = 1 \quad x_{51} + y_{41} + y_{51} = 1 \tag{2.15}$$

The "middle-cut" requirement for the leftmost vertical splitting line corresponds to the following inequality:

$$\textstyle\sum_{i=1}^{6} x_{i1} \geq 1 \tag{2.16}$$

Sum of equalities (2.14) and (2.15) produces a new constraint:

$$\textstyle\sum_{i=1}^{6} x_{i1} + 2\sum_{j=1}^{5} y_{j1} = 6 \tag{2.17}$$

Now we can subtract (2.16) from (2.17) and divide it by two:

$$2\sum_{j=1}^{5} y_{j1} \leq 5 \quad implies \quad \sum_{j=1}^{5} y_{j1} \leq 2.5 \tag{2.18}$$

Since all y_{j1} in the left-hand side of inequality (2.18) are integer variables, we can apply CG-cut to obtain a new (tighter) inequality:

$$\textstyle\sum_{j=1}^{5} y_{j1} \leq 2 \tag{2.19}$$

In its turn, constraints (2.17) and (2.19) imply

$$\textstyle\sum_{i=1}^{6} x_{i1} \geq 2 \tag{2.20}$$

Induction Step: Suppose that we have proved that inequalities (2.11) and (2.13) imply the *"double-firm"* tiling for $j_0 - 1$ leftmost columns of the checkerboard ($1 \leq j_0 - 1 \leq 4$). It implies that

$$\textstyle\sum_{i=1}^{6} x_{ij} \geq 2 \quad j = 1, \ldots, j_0 - 1 \tag{2.21}$$

Consider the tiling of left j_0 columns. All tiles $y_{ij} = 1$ with $i \leq j_0$ cover two checkerboard squares in left j_0 columns, the same is true for tiles $x_{ij} = 1$ with $i < j_0$. These tiles contribute two to the amount of covered squares in the discussed portion of the checkerboard. Tiles

$x_{ij_0} = 1$ contribute one to the number of tiled squares in left j_0 columns. If we count the number of tiled squares in left j_0 columns according to the above observation and take into account that each square should be covered by a tile, we get the following equation:

$$\sum_{j=1}^{j_0}\sum_{i=1}^{5} 2y_{ij} + \sum_{j=1}^{j_0-1}\sum_{i=1}^{6} 2x_{ij} + \sum_{i=1}^{6} x_{ij_0} = 6j_0 \tag{2.22}$$

Since a *"single-firm"* tiling requires $\sum_{i=1}^{6} x_{ij_0} \geq 1$, equality (2.22) and "firmness" imply

$$\sum_{j=1}^{j_0}\sum_{i=1}^{5} 2y_{ij} + \sum_{j=1}^{j_0-1}\sum_{i=1}^{6} 2x_{ij} \leq 6j_0 - 1 \tag{2.23}$$

Inequality (2.23) can be divided by two:

$$\sum_{j=1}^{j_0}\sum_{i=1}^{5} y_{ij} + \sum_{j=1}^{j_0-1}\sum_{i=1}^{6} x_{ij} \leq 3j_0 - 1/2 \tag{2.24}$$

If we apply CG-cut to inequality (2.24), we get a tighter constraint:

$$\sum_{j=1}^{j_0}\sum_{i=1}^{5} y_{ij} + \sum_{j=1}^{j_0-1}\sum_{i=1}^{6} x_{ij} \leq 3j_0 - 1 \tag{2.25}$$

which together with equality (2.22) imply the *"double-firm"* tiling of j_0th vertical splitting line:

$$\textstyle\sum_{i=1}^{6} x_{ij_0} \geq 2 \tag{2.26}$$

■

Lemma 3 constitutes the hardest part of the Firm Tiling problem. Were we given the *"double-firm"* tiling requirement as the part of the initial problem, a simple application of the Pigeonhole Principle would provide the negative (infeasible) answer: Since there are ten splitting lines, each passing the middle line of at least two tiles, and none of the tiles can be shared by splitting lines in such counting, one needs at least 20 tiles for a *"double-firm"* tiling, in which case they would overlap. The Integer cuts technique demonstrated in Lemma 3 proves infeasibility through deriving the *"double-firm"* tiling requirement from a required *"single-firmness"* and the completeness of tiling (2.13). Since inequality (2.13) is actually an equality for the 6x6 checkerboard, the *"double-firm"* tiling inequalities make the Firm Tiling problem infeasible.

From a glance, Integer cuts seem to be manipulating with halves and other fractionals in a beneficial manner. However, it is not just a game with fractionals. Integer cuts perform methodological "squeezing" of the polygon of feasible solutions remaining all integer solutions feasible. Moreover, new constraints obtained through Integer cuts can not be derived by taking positive weighted sums of the existing constraints.

Although both problems presented in this section do not admit proofs by the original Pigeonhole Principle, it is possible to perform a representation change of the problem statements and transform both problems into ones that admit proofs by the Pigeonhole Principle. We call such method of solving Integer Programming problems as the Hidden Pigeonhole Principle (HPHP).

Such application of HPHP are useful mainly for deriving the tight upper bound. However, for some problems precise knowledge of the optimal value allows AI planning systems to construct an optimal solution. For the problems presented in this section this can be done, for example, by placing domino pieces randomly or greedily and applying the backtracking techniques when necessary. Success in constructing an optimal solution from the optimal value depends heavily on planning domain properties. Nonetheless, without a HPHP application the optimal value would be unknown, and any attempts to construct a solution attaining non-tight upper bound would fail.

2.6 HPHP Mimicking LPR with Chvatal-Gomory's Cuts

In this section we give an example of a simple problem illustrating how the "Hidden" Pigeonhole Principle can mimic the combination of Chvatal-Gomory's cuts and Linear Programming Relaxation.

> **Air Traffic Problem:** Suppose that there are N airports in the country, each is capable of routing flights to at most M destinations. What is the maximum number of flights throughout the country?

There is no issue in determining the maximal number of flights for the whole country as long as either M or N is even, and the answer is $MN/2$. However, as soon as both M and N are odd, at least one of the airports has to schedule strictly less than M flights.

In Integer Programming, the above argument would be resolved through Chvatal-Gomory's cut. If x_{ij} denotes the presence of a flight between cities i and j, then

$$\sum_{i=1}^{N} \sum_{j=1}^{N} x_{ij} \leq MN \tag{2.27}$$

due to double-counting flights originating from two different cities[3], implies

$$\sum_{i=2}^{N} \sum_{j=1}^{i-1} x_{ij} \leq \frac{MN}{2} \tag{2.28}$$

[3]It would be unwise to route a flight from city i back to city i.

If one applies a CG-cut to inequality (2.28), the resulting inequality

$$\sum_{i=2}^{N}\sum_{j=1}^{i-1} x_{ij} \leq \lfloor \frac{MN}{2} \rfloor$$

would imply that for the case when both M and N are odd, one of the airports is running strictly under the full-scale loading.

The Pigeonhole Principle would imitate Integer cuts by deriving the contradiction. Suppose that all airports are fully loaded, i.e. M flights are scheduled for each of them. We show that in this case, one of the airports is scheduled for at least $M + 1$ flights. Note, that $MN/2$ is not integer. Suppose that every airport is loaded with at least M flights. Then, consider the overall assignments with $\frac{MN-1}{2}$ flights (create the hole of this capacity). Such a loading is impossible according to the Pigeonhole Principle. In this application of PHP, outbound flights are the pigeons, overall number of flights is the hole of capacity $\lfloor \frac{MN}{2} \rfloor$. Thus, the artificially created hole does not have enough capacity, hence, there are routed $\lfloor \frac{MN}{2} \rfloor + 1$ or more flights overall. Now we use PHP again: Since every flight connects two destinations, there will be at least $MN + 1$ destinations accounted from all flights. Now the overall number of flights contains at least $MN + 1$ pigeons, each airport is the hole of capacity M. Contradiction.

Note, that in the above application of PHP, one can notice an elegant swap between the pigeons and the holes. Even for such a simple task as the Air Traffic problem, the application of the Pigeonhole Principle is "hidden" according to our classification, as we had to come with a non-existing hole of capacity $\frac{MN-1}{2}$.

2.7 Implications for Resolution-Based Proof Methods

If we introduce variables x_{ij} indicating that pigeon i is in hole j, then the Pigeonhole Principle with m pigeons and n holes can be modeled by two groups of constraints:

$$x_{i1} \bigvee x_{i2} \bigvee \ldots \bigvee x_{in} \quad for \quad i = 1, \ldots, m \tag{2.29}$$

$$\neg x_{ik} \bigvee \neg x_{jk} \quad for \quad i \neq j, \quad for \quad k = 1, \ldots, n \tag{2.30}$$

First group of constraints forces every pigeon to be placed in at least one hole. Second group of constraints does not allow any pair of pigeons to occupy the same hole. Thus, PHP is re-stated as a conjunction of all disjunctive clauses (2.29-30), i.e. in a standard Conjunctive Normal Form (CNF).

Resolutions of clauses is one of the most common methods of simplifying boolean CNF formulae. It looks for a variable that is shared by two clauses with different polarities and combines these clauses without the shared variable. For example, for the following formula

$$(V_1 \bigvee \neg V_2 \bigvee \neg V_4) \bigwedge (V_2 \bigvee V_3 \bigvee \neg V_5) \tag{2.31}$$

the resolvent is

$$(V_1 \bigvee V_3 \bigvee \neg V_4 \bigvee \neg V_5) \tag{2.32}$$

Statement (2.32) is implied by the original clauses of (2.31). Extended resolution is a generalization of the resolution method that allows the introduction of new variables as the combination of existing ones. It has been shown that if one applies the resolution method to the Pigeonhole Principle stated in (2.29-30), for $m > n$ it would require exponential number of resolutions [1]. This is, the shortest length of the sequence $L_1, L_2, \ldots, L_p, \emptyset$, where each L_i is either the original clause or the resolution of two clauses from the prefix sub-sequence $L_1, L_2, \ldots, L_{i-1}$, is exponential on n.

Some Linear Programming methods, on the other hand, guarantee polynomial worst-case complexity. Furthermore, it has been shown that the constraints of a 0-1 IP problem can be expressed in a logical form [33]. The Pigeonhole Principle stated in (2.29-30) is exactly a 0-1 IP problem. However, a brute-force mimicking of LP constraints possesses an explosive danger, as some of compactly written inequalities from Linear Programming would require much more space, if interpreted in a boolean form. Table 2.1 illustrate the danger of blind copying methods of Linear Programming.

Linear Programming	Logic
$x_1 + x_2 + x_3 + x_4 \leq 3$	$x_1 \wedge x_2 \wedge x_3 \wedge x_4 = False$
$x_1 + x_2 + x_3 + x_4 \leq 2$	$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge$ $(x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) = False$
$x_1 + x_2 + x_3 + x_4 \leq 1$	$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_4) \wedge$ $(x_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge (x_3 \vee x_4) = False$

Table 2.1: LP and Logic Constraints.

McKinnon and Williams suggested a compact way of bookkeeping [48]. The "greater or equal" boolean predicates (possibly nested) $ge(k, \{P_1, P_2, \ldots, P_n\})$ mean "k or more propositions $P_1, P_2, \ldots, P_n$ are true." Using this notation one can re-write Table 2.1 in a compact Logical form, see Table 2.2.

Hooker [20] and Ginsberg [31] considered $ge()$-predicates and showed that extended resolutions can be used effectively in processing logical theories. In particular, Ginsberg showed that

Linear Programming	Logic
$x_1 + x_2 + x_3 + x_4 \leq 3$	$ge(1, \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\})$
$x_1 + x_2 + x_3 + x_4 \leq 2$	$ge(2, \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\})$
$x_1 + x_2 + x_3 + x_4 \leq 1$	$ge(3, \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\})$

Table 2.2: LP and Compact Logic Constraints.

in such case, the Pigeonhole Principle stated in (2.29-30) can be proved to be unsatisfiable in $O(n^5)$ steps [31]. $ge()$-predicates look very much like inequalities from Linear Programming. This fact stimulated our efforts in applying both the Pigeonhole Principle and Linear Programming Relaxation to problems stated in the Integer Programming form. The other stimulus in bringing knowledge from Logic to Integer Programming was to change the direction of the usual research flow, as the vast majority of attempts tries to bring techniques from Integer or Linear Programming into Logic, thus, not so much has been done in the opposite direction [81].

2.8 Tough Nuts for the Pigeonhole Principle

Not every problem that is traditionally attributed to the Pigeonhole Principle can be represented in the Integer Programming form. In this section we introduce two problems that can be solved by the PHP after sophisticated representation changes. However, modeling them as IP problems is a challenge for a problem solver. First problem has a continuous nature that prevents it to be modeled as an IP problem, second problem possesses an explosive combinatorial nature in its description. These two problems are aimed on demonstrating that the length of the description of the problem itself and its solutions might determine the success of applying the Pigeonhole Principle.

> **Circle Covering Problem:** One is given a circle of the diameter 10 and 10 rectangular strips, nine of which have the unit width and tenth has the width of $1 - \epsilon$ with $\epsilon > 0$. Is it possible to cover the entire circle by these ten strips?

The amount of circle covered by a unit-wide strip in the circle covering problem depends on the location of the strip. In particular, the area of the intersection of the circle with the strip is maximal, if the center (symmetry) line of the strip goes through the center of the circle. To apply the Pigeonhole Principle, one needs to apply a representation change to create standard pigeonhole pigeons and holes whose capacity would not depend on the location of the holes (strips) relatively to the center of the circle. The additional complication in this problem is that pigeons (circle area) can contribute simultaneously to several holes (when strips intersect).

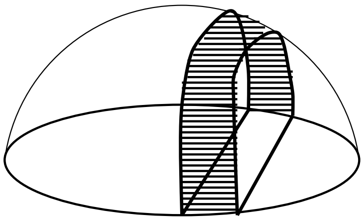

Figure 2.11: Re-distributing the Measure of the 2D Circle

Fortunately, it is possible to perform a simple representation change, so that to create a certain measure that is invariant to the placement of the strip. Figure 2.11 illustrates the idea of re-distributing an initial uniform measure of the circle. If we intersect a 3D sphere with two parallel hyperplanes, unit distance apart, the surface area of the sphere between the hyperplanes is invariant to the actual intersection, as long as the intersection of each hyperplane with the sphere is not empty. This property hints on how we to re-distribute the measure of the circle: We weigh every strip by the surface area of the projection of the strip on the surface of the 3D sphere (or half-sphere for the simplicity of the picture). Thus, each unit strip (hole) has the capacity of at most $1/10$ of the whole circle's amount, no matter how it intersects the circle. The last, tenth strip has the capacity of $1/10 - \delta$, with $\delta > 0$. Hence, the overall capacity that all strips (holes) can hold is at most $1 - \delta < 1$.

> **Triangle Cutting Problem:** One is given an equilateral triangle, which is split into smaller triangles by three sets of N equi-distant lines, parallel to each side of the triangles. Can one cut a set of parallelograms out of such triangle if cutting only along the splitting lines?

The problem of solving the Triangle Cutting problem appears to lie in modeling it in an acceptable way. The enormous amount of possible scenarios in cutting the triangle into parallelograms seems to stop any initial attempt of accounting them all. Nonetheless, the Triangle Cutting problem admits a simple solution by the Pigeonhole Principle after an elegant representation change that identifies the invariant of the cutting procedure. Figure 2.12 shows one of the possible cuttings of a parallelogram off a triangle with $N = 5$. Note, that the described splitting of the triangle produces a family of identical (smaller) triangles with two different orientations. Figure 2.12 also shows smaller set of shaded triangles.

In a certain sense, the proof of the Triangle Cutting problem mimics the proof of the Mutilated Checkerboard problem (see Section 2.5). Each parallelogram contains equal number of triangles of both orientations. Since one of the equi-oriented sets is strictly smaller initially, it is impossible to cut the triangle into parallelograms along the splitting lines. The Triangle Cutting problem hints on how to generalize the Mutilated Checkerboard problem: If one

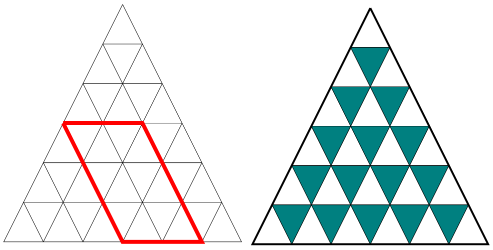

Figure 2.12: Cutting a Parallelogram from a Triangle with N=5

is allowed to cut off rectangles of any finite, connected subset of squares S on the infinite checkerboard with at least one side of each rectangle of even size, such cutting procedure can succeed only if the sizes of monochrome sets of squares in S are the same.

2.9 Summary

In this chapter we considered different methods of attacking combinatorial optimization problems. Although the Pigeonhole Principle and Linear Programming Relaxation seem to live in completely different worlds, we showed that PHP and LPR have the same bounding power. Moreover, one is the dual of another. Such an unpredictable relation became possible after we brought both methods to the "common ground" of Integer Programming and drew a splitting line between the applications of the Pigeonhole Principle that deal with the original objects of the problem and the applications of the Hidden Pigeonhole Principle that deal with various extensions or constructions based upon the original objects of the problem. In its turn, we demonstrated that proofs by the Hidden Pigeonhole Principle are in many ways similar to combinations of Integer cuts and Linear Programming Relaxation.

The results of this chapter enable us to state the following conclusions:

- Resolution-based methods keep pigeons integer, whereas extended resolutions may "cut" pigeons into pieces, manipulate with fractional pigeons and perform a "reconstructive surgery" in the end, if necessary. Such a "cruel" treatment of pigeons sometimes results in a huge performance win, for example, the complexity of the logical proof of the simplest formulation of the Pigeonhole Principle goes down from exponential to polynomial.

- Linear Programming Relaxation applied to a minimization Linear Integer Programming problem is actually a Linear Programming analogue of the Pigeonhole Principle. As a consequence, it is the dual of a correspondent maximization IP problem.

- The "Hidden" Pigeonhole Principle has the same nature as the combinations of Integer cuts and LPR. For example, HPHP can mimic the sequence of Chvatal-Gomory's cuts and LPR.

- Through the set of successes and failures demonstrated in this chapter, one can witness the Occam's Razor Principle in action: In multi-disciplinary approaches, those methods of modeling problems and deriving solutions that require shorter descriptions are more preferable.

Chapter 3

On-Line Search

In this chapter we apply the methodology of hybrid approaches to the on-line search problem. Whereas in Chapter 4 we will be focused on the concluding **Constructing Hybrid Methods** phase, the current chapter contains **Selection**, **Creating the Environment** and **Analysis** phases.

The on-line and off-line search problems are concerned with search in partially or in completely known domains by an agent with limited lookahead. Since problem domains are not known in advance in the on-line version of the problem, the path finding algorithms amenable for this problem need to gather information in the surrounding world to locate a goal state and a path leading to it. Since for this type of problems an agent needs to explore the environment as well as to look for a goal, and agent's knowledge about the surrounding world is limited by a neighborhood centered at the current state of the agent, we call search problems of this kind **goal-directed exploration problems** or, alternatively, **agent-centered search problems**. Examples of such problems include:

- Autonomous mobile robots that have to find the office of a given person in an initially unknown building, or a previously known building which is currently under repairs.

- Software agents that have to find World Wide Web pages containing desired information by following links from their current page. CMU's Web Watcher, for example, complies with this agent-centered strategy [2].

Both AI and Theory researchers have investigated the problem of reaching a goal by an agent with limited lookahead in an initially unknown domain. Difference in terms, the variety of scenarios, different foci make it difficult to extract the most beneficial features from approaches that belong to different areas. Table 3.1 shows a relevant fraction of Table 1.1 regarding the goal-directed exploration problem. In particular, prior knowledge can be a powerful tool in cutting down the search effort, but it usually have little influence on the worst-case complexity. On the

AI	CS Theory
• Preprocessing - Representation Changes - Prior Knowledge • Empirical Performance	• Data Structures • Worst-Case Analysis • Optimal Algorithms • Approximate Algorithms

Table 3.1: Advantageous Features of AI and CS Theory for On-Line Search

other hand, the algorithms that achieve the worst-case complexity seem to be too "cautious" and do not demonstrate strong empirical performance, when applied to on-line search problems.

In Chapter 4 we show how some of CS theory and AI algorithms can be combined to get the best of both worlds: Theoretical component contributes worst-case guarantees, whereas AI component of the hybrid method enables an agent to utilize prior knowledge to guide search until it proves to behave poorly. The current chapter introduces the problem, the terminology, assumptions and overviews existing algorithms from CS theory and AI.

3.1 Agent-Centered Technologies for On-Line Search

A variety of approaches from AI and CS theory are applicable to the problem of **goal-directed exploration** [71]. However, methods from distinct scientific areas often use different languages and are focused on finding answers to different questions even within the same problem framework. Table 3.2 provides a brief comparison of CS Theory and AI terminologies and foci regarding the goal-directed exploration problem.

An abstract version of the goal-directed problem has been known to the theoretical community as Treasure Hunt. Various modifications of the problem of exploring unknown graphs has been also considered in CS theory. Deng and Papadimitriou [13] discovered the dependencies between the worst-case complexity of exploring a directed unknown graph and the

Aspect	**CS Theory**	**AI**
Problem name	Learning Unknown Graphs, On-Line Chinese Postman Problem, Treasure Hunt	Exploration of the Environment, Goal-Directed Exploration
Terms	Vertices, Edges, Untraversed Edges, Paths	States, Actions, Limited Lookahead, Prior Knowledge
Foci	Worst-Case Complexity Average-Case Complexity	Empirical Performance

Table 3.2: CS Theory and AI Terminologies.

deficiency – the measure of how close the unexplored graph is to being Eulerian. Awerbuch et al. [3] investigated the worst-case complexities of "piecemeal" learning and Treasure Hunt problems, where an agent is required to return to the starting position for recharging every so often. However, researchers from CS theory has been mainly concerned with the worst-case complexities of solving static problems of Graph Theory, almost completely ignoring the fact that the empirical performance and the worst-case complexity can differ significantly. Another issue of real-life goal-directed exploration problems is prior knowledge that is often readily available in form of heuristic values. This knowledge can essentially cut down the (empirical) search time, but existing theoretical approaches are often not able to utilize heuristic values, because they were not designed with this thought in mind.

On the other hand, prior knowledge has been known as providing good guidance and cutting down search time in practical AI algorithms. Theoretical analysis of on-line search methods seems to ignore prior knowledge, because it does not improve the worst-case complexity. Furthermore, theoretical analysis of the average-case complexity is known to be a hard task that depends on two factors - the domain instance and the initial distribution. A slight change in any of them can transform the problem from "solvable" into "very hard" and vice versa.

The strength of AI methods comes from their "natural selection" out of the wide pool of heuristic-guided empirical algorithms. However, finding a proper heuristic function is a difficult task for complicated problem domains. In this chapter we show that the domain-heuristic relation is even more sensitive for on-line search than that for off-line search. In Section 3.4.1 we present an example of the problem domain, where a very efficient in general

AI algorithm guided by a consistent, admissible heuristic can lose to uninformed algorithms, including itself[1]. In Chapter 6 we argue why the number of satisfied/unsatisfied clauses is not always efficient in guiding local hill-climbing procedures towards a satisfying assignment, after we interpret such procedures as agent-centered search methods.

We need to note that the goal-directed exploration problem is different from off-line search problems, because off-line search algorithms are concerned with finding action sequences that reach goal states from start states in completely known state spaces. Since real-world AI problems are often too big to fit the memory of the computer, or downloading the problem domain completely may significantly slow down the performance, not all off-line algorithms consider the whole domain as represented in Memory. Moreover, several agent-centered methods has been successfully applied to and solved large off-line search problems, as if they were on-line search problems. This fact encourages us to study properties of on-line search.

As we mentioned before, in the goal-directed exploration problem, the domain is not known in advance, and path finding algorithms need to gather information in the world to locate a goal state and a path leading to it. Besides this difference with off-line search problems, "teleporting" is not allowed in the on-line version of the problem and the complexity is measured as the length of the continuous walk performed by an agent until it reaches the goal state. Already these two features make goal-directed exploration very different from off-line search.

Therefore, in order to solve this problem efficiently, it can be of high interest to combine heuristic-based search approach providing good performance for the cases when heuristics are reliable with exploration that will provide suboptimal performance guarantees, if heuristics are misleading. Thus, we can outline two paradigms of goal-directed exploration: pure exploration and heuristic-driven exploitation. **Pure exploration approaches** explore the state space using only knowledge of the physically visited portion of the domain. **Heuristic-driven exploitation approaches**, on the other hand, totally rely on heuristic knowledge in guiding the search process towards a goal state. Both approaches in their purity have disadvantages in solving goal-directed exploration problems: the first approach does not utilize available knowledge to cut down the search effort, the second approach follows the guidance of prior knowledge, even if it is misleading. In each particular scenario of on-line search we perform careful analysis of heuristic-driven exploitation algorithms in order to find out how much they may lose to pure exploration algorithms and to identify classes of problems, where the latter algorithms would consistently outperform the former ones.

[1]See Section 3.2 for the description of the goal-directed exploration problem and the definition of the heuristic types.

3.2 Goal-Directed Exploration Problem

In this chapter we consider problems with reversible domains, i.e. those problems whose domains can be represented as undirected or bi-directed graphs. The latter means that every directed edge (action) has an opposite directed edge (reverse action), but by traversing an edge the agent does not automatically learn how to traverse the opposite edge unless it has traversed it before. The concept of goal-directed exploration corresponds to the behavior of a new-born, who has no initial knowledge about the surrounding world and explores it through acting. Some of the actions, like setting something on fire, does not immediately imply knowledge on stopping the fire, although we assume in this chapter that every action is reversible.

We use the following notation: $G = (V, E)$ denotes an unknown undirected graph, $v_{start} \in V$ is the start state (vertex), and $G \subseteq V$ is the non-empty set of goal states. $E(v) \subseteq E$ is the set of edges adjacent to vertex v. For simplicity, edges $e \in E$ are assumed to be of unit length $length(e) = 1$, although all the results can be easily extended to graphs with edges of arbitrary non-negative length. The goal distance $h^*(v)$ is the length of the shortest path following which an agent can reach a goal state from v. The weight of the graph is $weight = 1/2 \sum_{v \in V} \sum_{e \in E(v)} length(e)$ – the sum of the lengths of all edges, which in our case coincides with the number of edges $|E|$.

If $e \in E$ has not been learned, then $succ_e(v)$, the successor of v such that $(v, w) = e$, is unknown. To learn the edge, the algorithm has to traverse it. Initially, heuristic knowledge about the effects of traversing edges is available in form of estimates of the goal distances. Classical AI search algorithms attach heuristic values to states. This would force us to evaluate untraversed edges $e \in E(v)$ in v according to the heuristic value of v, since the successor of v is not yet known. We therefore attach heuristic values $h(e)$ to edges instead; they are estimates of $length(e) + h^*(succ_e(v))$, the shortest length of getting from v to a goal state when first traversing e. If all $h(e)$ are zero, we say that the algorithm is uninformed. The algorithm is completely informed, iff $h(e) = length(e) + h^*(succ_e(v))$ for all $v \in V$ and $e \in E(v)$. We say that heuristic values are consistent iff $h(e) \leq length(e) + \min_{e' \in E(succ_e(v))} h(e')$ for all $v \in V$ and $e \in E(v)$. They are admissible iff $h(e) \leq length(e) + h^*(succ_e(v)$ for all $v \in V$ and $e \in E(v)$.

The goal-directed exploration problem can now be stated as follows:

> **The Goal-Directed Exploration Problem**: Get an agent from v_{start} to a vertex in G if all edges are initially unknown, but heuristic estimates $h(e)$ are provided upon request.

We measure the performance of goal-directed exploration algorithms by the length of their paths from the start vertex to a goal vertex. This performance measure is realistic, since usually the time of executing walk dominates significantly the deliberation time of goal-directed exploration algorithms.

In Section 3.3 and Section 3.4 we describe several approaches relevant to goal-directed exploration as examples: pure exploration algorithms and heuristic-based exploitation algorithms.[2] Our selection was based primarily on the efficiency of these algorithms along different dimensions.

3.3 CS Theory Approaches

Theoretical approaches are usually Pure exploration approaches that explore unknown graphs completely. They have no notion of a goal and consequently do not use any prior knowledge to guide search towards a goal location. However, they can be used for goal-directed exploration, since they visit all states during their exploration, including the goal vertices. One can then simply stop the algorithm when it accidentally hits a goal. Conversely, goal-directed exploration approaches explore all edges, if there is no goal. Thus, they can be used as exploration algorithms. Depth-First-Search (DFS) is one of such methods, it can be applied to the problem of exploring unknown, undirected graphs and guarantee that each edge is traversed at most two times. For Eulerian graphs, a simple algorithm of building a Eulerian tour (BETA) has been known since Euler [15] and Hierholzer [23]. Recently several researchers re-considered it as a graph-learning algorithm with linear worst-case complexity [41, 13].

> **Building a Eulerian Tour algorithm (BETA)**: Traverse unexplored edges whenever possible (ties can be broken arbitrarily). If all edges emanating from the current vertex has been explored, re-traverse the initial sequence of edges again, this time stopping at all vertices that have unexplored emanating edges, and applying the algorithm recursively from each such vertex.

Deng and Papadimitriou [13] showed that BETA traverses every directed edge at most twice in Eulerian domains (a superset of undirected domains). This implies the following theorem:

Theorem 3.1 *BETA reaches a goal state of a given goal-directed exploration problem with a cost that is at most* $O(1) \times weight$ *(to be precise: at most* $2 \times weight$*).*

No uninformed goal-directed exploration algorithm can do better than BETA in the worst case. Consider, for example, the graph shown in Figure 3.1, whose directed edges are annotated with their lengths.

Any goal-directed exploration algorithm can first traverse all edges with length c by traversing an edge adjacent to the start vertex and then (by chance) traversing the opposite directed

[2] Other approaches have for example been discussed in CS theory [6, 8, 12], robotics [60, 46, 61] and AI [77, 76, 51].

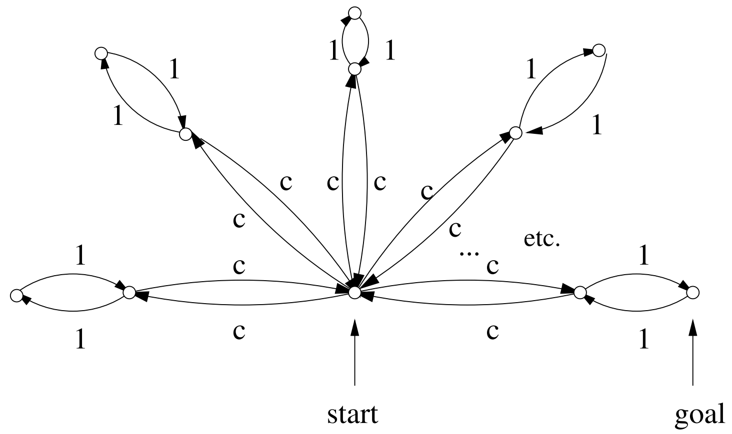

Figure 3.1: A Worst-Case Example for all Uninformed Algorithms

edge back. After that, it is in its starting vertex and cannot traverse unexplored edges any longer. It then has to re-traverse every edge of length c to be able to execute the last two unexplored edges at the end of each "ray," after which it is forced to execute the other edge of length c a second time to return to the starting vertex. Assume that there are k rays and the last ray traversed contains the goal vertex at its end. In this case, the total length of the path is $(4k - 1)(c - 1)$, the weight of the graph is $2kc + 2k$, and the ratio of the two quantities approaches 2 for large c. This implies the following theorem:

Theorem 3.2 *The worst-case complexity of uninformed goal-directed exploration algorithms is* $\Theta(1) \times weight$ *(to be precise:* $2 \times weight$*).*

BETA has the *disadvantage* that it does not make use of any prior knowledge to guide the search towards the goal. Given such knowledge, it is often unnecessary to explore all edges. However, BETA provides a gold standard for other goal-directed exploration algorithms.

3.4 Heuristic-Driven Approaches

AI researchers have long realized that heuristic knowledge can be a powerful tool to cut down search effort – and such knowledge is often readily available. Heuristic-driven exploitation approaches rely on heuristic knowledge to guide the search towards a goal state.

3.4.1 Agent-Centered A* Algorithm

Among heuristic-driven exploitation approaches, A* is one of the most popular off-line search algorithm that exploits heuristic knowledge. However, if applied in its original form to goal-directed exploration problems A* becomes very inefficient as it incurs new costs for moving an agent from one place to another instead of "teleporting". Moreover, since the agent has to perform a continuous travel to a new location, the strategy of A* does not produce the best available action sequence for agent-centered search. Nonetheless, the "greedy" idea of performing the best currently available step appeared to be tempting and was repeatedly utilized in various heuristic-driven exploitation approaches. The list of such algorithms includes Incremental Best-First Search by Pemberton and Korf [58], the Dynamic A* algorithm (D*) by Stentz [72], the Learning Real-Time A* (LRTA*) algorithm by Korf [42], Prioritized Sweeping by Moore and Atkeson [50], the navigation method by Benson and Prieditis [5], etc.

Thus, instead of minimizing at every step the sum of the distance from the starting node to a fringe node $dist(v_s, x)$ and the heuristic value $h(x)$ at that node, as A* does, the agent-centered algorithm can minimize the sum of the distance from the current node to a fringe node $dist(v_c, x)$ and the heuristic value ($h(x)$). This tiny change $dist(v_s, x) + h(x) \rightsquigarrow dist(v_c, x) + h(x)$ produces an extremely efficient agent-centered version of the A* algorithm, but drastically changes the flow of the exploration, thus, leaving the main question of this algorithm's efficiency completely open. Therefore, this agent-centered version of A* uses similar to A* minimization (greedy) step to find a path to the next unexplored edge in the currently known part of the graph, then moves the agent to that edge, traverses it, and repeats the process. We call this algorithm AC-A*:

> **AC-A* (Agent-Centered A* Algorithm)**: Consider all paths that include traversed edges from the current vertex to an untraversed edge e emanating from already visited vertex w. Select a path with minimal expected length from these paths, where the expected length of a path is defined to be the sum of the length of the path from the current vertex to w plus $h(e)$ (ties can be broken arbitrarily). Traverse the chosen path and the unexplored edge e, and repeat the process until a goal state is reached.

AC-A* is very versatile: It can be used to search completely known, partially known, or completely unknown, undirected, Eulerian or non-Eulerian, even dynamically changing graphs and is able to make use of knowledge that it acquires during the search. For example, if it is informed about the effects of some actions, it automatically utilizes this information during the remainder of its search.

AC-A* is known to be efficient under different problem scenarios. If AC-A* is totally informed, for example, it finds a goal state with cost $h^*(v_{start})$ and thus cannot be outperformed by BETA or any other goal-directed exploration algorithm. Figure 3.2 shows the performance of AC-A*, BETA, BETA with short-cuts [71] and 4-VECA[3] in exploring rectangular, bi-directed mazes of size 32×32 and variable density: 0% correspond to a random tree, 100% - to a complete rectangular maze. AC-A* outperforms all other algorithms in average with the only exception: it loses to VECA (see Chapter 4) on "hard-to-explore" sparse tree-like mazes.

[3] See Chapter 4 for the description of VECA.

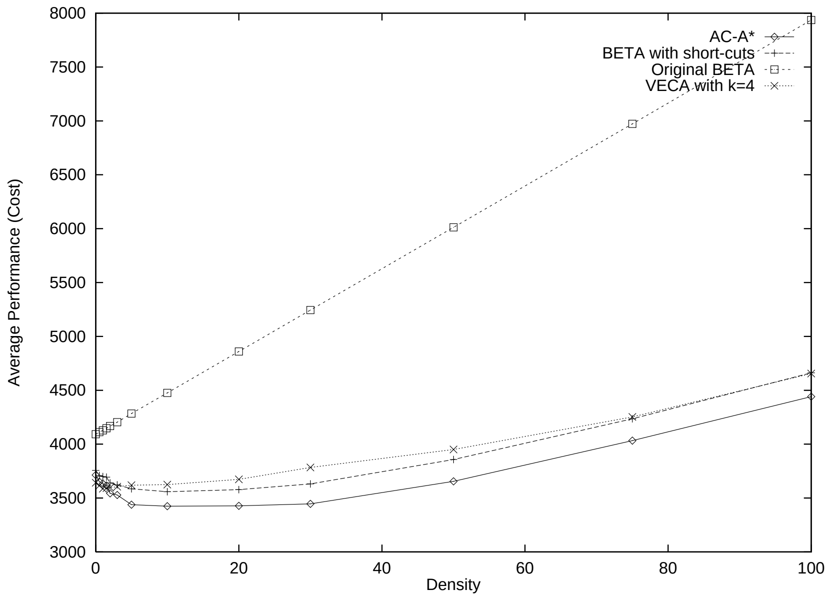

Figure 3.2: Average Exploration Time of Rectangular Mazes with Different Density

In the next two sections we discuss the disadvantages of AC-A* regarding the efficiency of search.

3.4.2 AC-A* Can Be Misled by Heuristic Values

We show that it is possible that consistent, admissible heuristic values can degrade the performance of AC-A* so much that its performance is worse than (a) the one of uninformed AC-A* for the same goal-directed exploration problem and (b) the one of BETA (an uninformed algorithm) for all goal-directed exploration problems of the same size. The goal-directed exploration problem shown in Figure 3.3 is such a scenario.

The problem domain is a bi-directed (Eulerian) graph which has the form of a tree and consists of a "stem" with several "branches." All edges are annotated with their lengths. (For convenience, we replaced each pair of directed twin edges by an undirected edge.) The stem has length x^x for some integer $x \geq 1$ and consists of vertices $v_0, v_1, \ldots, v_{x^x}$. The following table enumerates all branches.

number of branches	cost of each branch	states at which branches attach to the stem
x^{x-1}	1	$v_x, v_{2x}, v_{3x}, \ldots, v_{x^x}$
x^{x-2}	$x+1$	$v_{x^x-x^2}, \ldots, v_{2x^2}, v_{x^2}, v_0$
x^{x-3}	x^2+x+1	$v_{x^3}, v_{2x^3}, v_{3x^3}, \ldots, v_{x^x}$
x^{x-4}	x^3+x^2+x+1	$v_{x^x-x^4}, \ldots, v_{2x^4}, v_{x^4}, v_0$
...	...	...

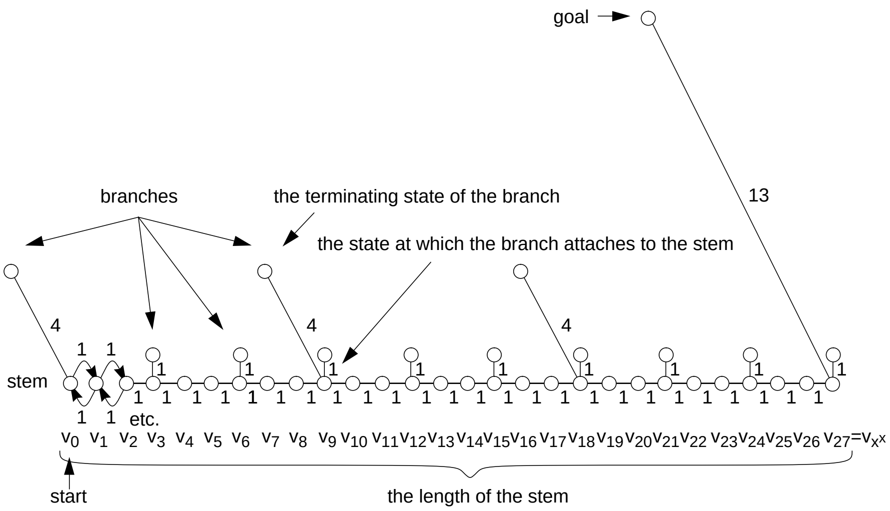

Figure 3.3: A Bad State Space for AC-A* (here: $x = 3$)

In Figure 3.3, for example, the stem has length 27, and there are 9, 3, and 1 branches, respectively, of cost 1, 4, and 13. In general, for each integer i with $1 \leq i \leq x$ there are x^{x-i} branches of length $\sum_{j=0}^{i-1} x^j$ each. These branches attach to the stem at vertices v_{jx^i} for integers j; if i is even, then $0 \leq j \leq x^{x-i} - 1$, otherwise $1 \leq j \leq x^{x-i}$. The starting vertex is v_0 and the goal vertex is the terminating vertex of the longest branch. The weight of this graph is

$$weight \quad = \quad \frac{4x^{x+2} - 6x^{x+1} + 2}{(x-1)^2}$$

This graph is sparsely connected and has a large diameter. Although it has been artificially constructed, our experiments show that similar situations can occur quite frequently in sparsely connected domains with large diameters.

Now assume that $h(e) = length(e)$. These heuristic values are consistent, and therefore admissible. AC-A* can then exhibit the following behavior: It starts at v_0, travels along the stem to its end and then returns along the stem to its starting vertex. Next, it travels along the whole stem again (in the original direction) and visits the terminating vertices of all branches of length 1 on the way (in the order in which they are listed in the table above). It then switches directions again, travels along the whole stem in the opposite direction, and visits the terminating vertices of all branches of cost $x + 1$ on the way (again, in the order in which they are listed in the table above), and so forth. When it visits the terminating vertex of the longest branch, it has found the goal and terminates. Thus, AC-A* traverses the stem $x + 2$ times and each branch twice (once in each direction), except for the longest branch, which it traverses only once. The total length is

$$length = \frac{x^{x+3} + 2x^{x+2} - 6x^{x+1} + x^x + x + 1}{(x-1)^2}$$

The ratio of $length$ and $weight$ approaches $1/4x + 7/8$ for large x. Thus, $length$ and also the worst-case performance of AC-A* grows faster than the weight of the graph.

We have already shown that the worst-case performance of BETA is always linear in the weight of the graph. For this particular example, the worst-case performance of uninformed AC-A* also grows linearly in the weight of the state space, if n is even: In this case, the branch that contains the goal attaches to the stem at the starting vertex. This property ensures that, no matter which edges uninformed AC-A* traverses, it can traverse every edge at most once before it reaches the goal. This is so, because in every undirected (bi-directed) graph the first vertex that BETA encounters in which it can no longer traverse an unexplored edge is the start state. But before this is the case, it must have explored the edge that leads from the starting vertex to the goal. Thus, it cannot traverse any edge more than once before it reaches the goal.

This example shows that the domain-heuristic relation is more sensitive for the goal-directed exploration problem. The fact that a heuristic function majorizes the other one does not necessarily lead to a better performance for a heuristic-driven algorithms that uses the majorizing heuristic. Moreover, the guidance of a consistent, admissible heuristic in some domains can be even worse than the uninformed exploration (that ignores prior knowledge) for agent-centered search. In Chapter 7 we discuss the influence of some domain features on the complexity of search and the sensitivity of the the domain-heuristic relation.

3.4.3 AC-A* is not Globally Optimal

The actions of AC-A* are greedy, every time AC-A* provides the best available action, given the lack of information it has about the domain. Some authors even stated the hypothesis that the behavior of AC-A* optimal. We demonstrate, however, that its behavior is not globally optimal by showing that the worst-case performance of uninformed AC-A* over all goal-directed exploration problems of the same size is worse than that of BETA – a fair comparison, since both algorithms are uninformed. This issue, as well as the average-case complexity of AC-A*, were raised in [13] as an open problem.

Uninformed AC-A* always moves the agent to the unexplored edge that it can reach by a path of the smallest length, traverses that edge, and repeats the process until the goal is reached. We can easily construct a graph on which uninformed AC-A* behaves (almost) identically to the partially informed AC-A* of the previous section. We use the state space from the previous section, but add an additional vertex to the end of every branch, as shown in Figure 3.4.

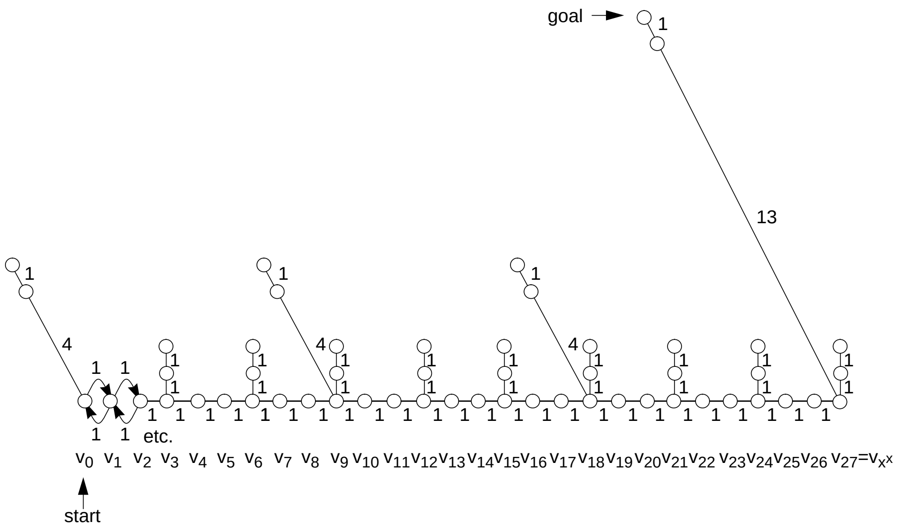

Figure 3.4: Another Bad State Space for AC-A* (here: $x = 3$)

Uninformed AC-A* can then exhibit the following behavior in this graph: It starts at v_0 and travels along the stem to its end. Then, it returns along the stem to its starting vertex and visits the non-terminating vertex of every branch on the way (after each of which it immediately returns to the stem). From then on, it can behave identically to the partially informed AC-A* of the previous section after it had traversed the stem twice: It now knows the lengths of entering the branches whereas the partially informed AC-A* had only heuristic values available, but these coincided with the lengths. For this example,

$$\begin{aligned} weight &= \frac{4x^{x+2} - 4x^{x+1} - 2x^x - 2x + 4}{(x-1)^2} \\ length &= \frac{x^{x+3} + 4x^{x+2} - 6x^{x+1} - 3x^x - x^2 + x + 4}{(x-1)^2} \end{aligned}$$

and the ratio of $length$ to $weight$ approaches $1/4x + 5/4$ for large x. Thus, $length = \Theta(x) \times weight$. This implies the following theorem, since $x = \Omega(\log n / \log\log n)$:

Theorem 3.3 *The worst-case complexity of the uninformed AC-A* for a goal-directed exploration problem is* $\Omega(\frac{\log n}{\log\log n}) \times weight$.

This theorem provides a lower bound on the worst-case performance of uninformed AC-A*. It shows that the worst-case performance increases faster than the weight of the state space. We can also prove an upper bound, using – as part of the proof – a previous result in [62, 39].

Theorem 3.4 *Uninformed AC-A* reaches a goal state of a goal-directed exploration problem with a performance of* $O(\log n) \times weight$.

3.4.4 Learning Real-Time A* Algorithm

Learning Real-Time A* Algorithm (LRTA*) [42] is known as a real-time search method whose efficiency depends on the quality of prior knowledge. It can be applied both to off-line search problems and to goal-directed exploration. LRTA* looks for the most promising vertex among neighbors of the current vertex and updates heuristic values, if necessary. If heuristic values are close to goal distances, or maintain similar quantitative relations, LRTA* (see Table 3.3) may find a goal state after exploring only a tiny fraction of the problem domain.

---

procedure LRTA*(V, E)
Initially, $F(v) := h(v)$ for all $v \in V$.
LRTA* starts at vertex v_{start}:

1. $v :=$ the current vertex.
2. If $v \in Goal$, then STOP successfully.
3. $e := argmin_e F(neighbor(v, e))$.
4. $F(v) := min(1 + F(neighbor(v, e)))$.
5. Traverse edge e, update $v := neighbor(v, e)$.
6. Go to 2.

Table 3.3: Learning Real-Time Algorithm (LRTA*).

LRTA* was designed as a simple reactive algorithm with the guaranteed convergence. Unlike local hill-climbing procedures (see Chapter 6), LRTA* requires memory to keep the updated heuristic values. If heuristic values are close to the goal distance for all domain states, LRTA* may find the optimal path to the goal from any start state. Whereas some problem domains amenable to LRTA* can be often attributed to A* as well, the advantage of LRTA* is that it does not require to search the whole domain in the way A* does. LRTA* may construct a suboptimal solution first, and then improve it through repeated trials, so that the portion of explored state space is still tiny compared with the part of the domain that A* needs to explore to derive a solution [42].

However, there is a price that one can pay for a heedless use of LRTA*: Misleading heuristic values may steer away the search process up to the point, when LRTA* becomes very inefficient. This is especially problematic for the cases when prior knowledge is neither consistent, nor admissible. ϵ-search and δ-search are both modifications of LRTA* [35] that were designed to reduce the inefficiency of LRTA* for problems of this type. In Chapter 4 we compare the behavior of LRTA*, AC-A* and other search algorithms for different heuristic functions. In Chapter 6 we discuss the relations between local hill-climbing methods, LRTA* and one of its versions – ϵ-search.

3.5 Summary

In this chapter we demonstrated that the domain-heuristic relation is more sensitive for on-line search problems than for off-line ones. Even a consistent, admissible heuristic can be more misleading for some on-line search algorithms than no prior knowledge. We also gave a negative answer to the open question on the optimality of AC-A*. Although it is a very efficient empirical algorithm, it loses to other uninformed exploration algorithms over all problems of the same size, and thus is not optimal.

Chapter 4

Variable Edge Cost Algorithm

In the previous chapter we introduced the goal-directed exploration problem and several algorithms from the literature amenable to this problem. We considered two paradigms of the goal-directed exploration problem: Pure exploration and heuristic-driven exploitation. Algorithms attributed to the former paradigm, usually come from CS theory and are focused on the complete exploration of the problem domains. Heuristic-driven exploitation algorithms usually come from AI and establish strong empirical performance. However, in some complicated domains they can lose to pure exploration algorithms.

Chapter 3 finished the first three phases of the methodology of of hybrid approaches applied to on-line search, namely, **Selection**, **Creating the Environment** and **Analysis**. We start this chapter with the conclusion remarks on the analysis of existing algorithms, then we switch to the **Constructing Hybrid Algorithms** phase, leaving the discussion on the **Problem Classification** phase for Chapter 7.

4.1 The Drawbacks of Existing Algorithms

In Chapter 3 we introduced algorithms from CS theory (BETA, Chronological Backtracking) and AI (AC-A*, LRTA*) that can be applied to on-line search problems. Theoretical algorithms, like BETA, do not make use of heuristic values to guide the search towards a goal state. AC-A* does utilize heuristic values, but can be misled by them up to the point where its performance is worse than the performance of BETA (an uninformed algorithm). This does not mean, of course, that one should never use AC-A*. If AC-A* is totally informed, for example, it finds a goal state with the cost that corresponds to the length of the shortest path from the starting location to the goal $h^*(v_{start})$ and thus cannot be outperformed by BETA or any other goal-directed exploration algorithm. The problem with AC-A* is that it takes the heuristic values at face value, even if its experience with them shows that they should not be trusted.

Of course, AC-A* does not know whether it should rely on the given heuristic values before it has gained experience with the state space and the values. We would therefore like to modify AC-A* so that it relies on the heuristic values until they prove to be misleading. It should then gradually rely less and less on the values, by switching from exploitation to exploration. This should be done in a way that would guarantee that the resulting worst-case performance over all goal-directed exploration problems of the same size can never be worse than that of the uninformed goal-directed exploration algorithm with the best possible performance guarantee (BETA). In this case, the misleading heuristic values do not help to find a goal state faster, but they don't hurt either. In the following, we describe how a variety of heuristic-driven exploitation approaches (AC-A* being one of them) can be modified to achieve such a performance guarantee.

The complexity analysis of selected algorithms obtained from the **Analysis** phase of the hybrid approach is likely to produce a picture similar to one shown in Figure 4.1. In the goal-directed exploration problem, the worst-case complexity of BETA or Chronological Backtracking is better than the worst-case complexity of LRTA* or AC-A*. However, for a sequence of experiments performed in unknown domains with some common features like graphs of similar size, density, diameter, etc., the empirical efficiency of heuristic-driven algorithms can compare favorably with one of BETA. Figure 4.1 shows a typical picture that represents both possible range of particular runs, from the shortest path to the worst-case scenario, and the average empirical performances for BETA, LRTA* and AC-A* search for a goal in sparsely connected domains.

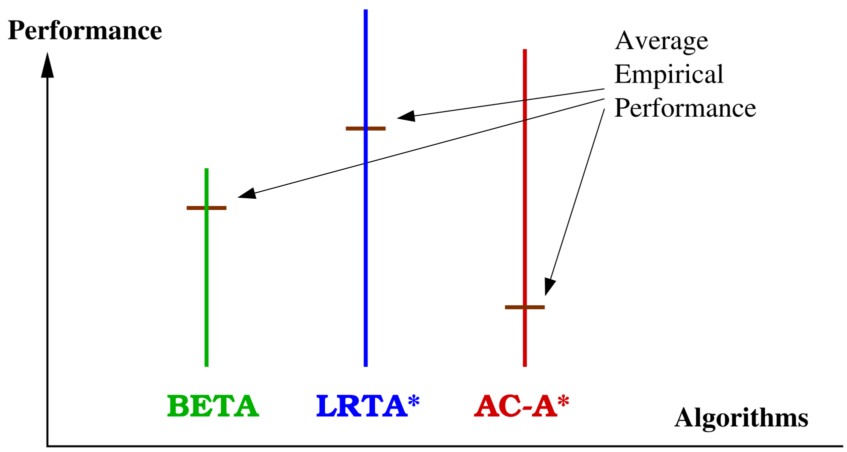

Figure 4.1: Worst-Case and Empirical Performances

The goal of this chapter is to demonstrate how the best features of methods from AI and CS theory can be combined in a hybrid framework.

In Chapter 3, after completing the **Analysis** phase, we identified that pure exploration algorithms do not search for a goal, but they establish strong upper bounds. For the problem of goal-directed exploration such a bound is linear on the weight of the graph, the property that we would like to re-utilize in efficient combinations with heuristic-driven algorithms. On the other hand, heuristic-driven algorithms, for example, LRTA* and AC-A*, establish strong empirical performance unless the combination of the domain and heuristic values appear to mislead search up to the point when the chosen heuristic-driven algorithm may lose noticeably to a pure exploration algorithm. Therefore, we view the behavior of BETA on bi-directed domains (or one of Chronological Backtracking on undirected domains) as the leftmost bar in Figure 4.1, and complete the **Constructing Hybrid Algorithms** phase by combining it with either LRTA* or AC-A*. In the next section we introduce such an algorithmic framework that is built upon a beneficial combination of pure exploration algorithms with heuristic-driven algorithms.

4.2 Our Approach: The VECA Framework

We have developed a framework for goal-directed exploration of undirected or bi-directed domains, called the Variable Edge Cost Algorithm (VECA) [71], that can accommodate a wide variety of heuristic-driven exploitation algorithms (including AC-A* and LRTA*). We first describe a simpler version of VECA that applies to undirected domains, then we show how to generalize it for the bi-directed case. VECA relies on the exploitation algorithm and thus on the heuristic values until they prove to be misleading. VECA monitors the behavior of the exploitation algorithm and uses a pre-set parameter k to determine when the freedom of the exploitation algorithm should get restricted. VECA does it by establishing positive costs for frequently traversed edges. As soon as an undirected edge has been traversed k times or more, VECA restricts further traversals of this edge by the exploitation algorithm through establishing a positive cost for this edge. This action forces VECA to concentrate more on exploring a certain portion of the graph. As a result, VECA switches gradually from exploitation to exploration and relies less and less on misleading heuristic values.

We describe VECA in two stages. We first discuss a simple version of VECA, called Basic-VECA, that applies to undirected domains, i.e. it assumes that an edge traversal identifies the twin of the edge, even if the twin has not been traversed before. Later, we drop this assumption. Throughout this chapter for any edge $(v, w) \in E$ we call the opposite edge $(w, v) \in E$ as the twin, whether the domain is undirected or bi-directed.

Basic-VECA is described in Figure 4.2. It maintains a cost for each edge $e \in E$ that is different from $length(e)$. These VECA costs guide the search. Initially, all of them are zero. Whenever VECA traverses an undirected edge for the first time, it reserves a positive VECA cost for it that will later become its assigned positive cost. First such edge gets a cost

Input:
a goal-directed exploration problem,
the value of Basic-VECA's parameter k (a non-negative, even integer),
and a heuristic-driven exploitation algorithm (to be used in step 3).

Basic-VECA uses three variables for each edge $e \in E$: $count(e)$ keeps track of how many times this edge has been traversed, $reserve(e)$ is its reserved VECA cost, and $cost(e)$ is its actual VECA cost.

1. Set $count(e) := cost(e) := 0$ for all $e \in E$. Set $i := 0$ and $v := v_{start}$.
2. If $v \in G$, then stop successfully.
3. Consider all acyclic paths starting at v and ending at a vertex with an emanating *untraversed edge*. Select a path with minimal VECA cost from these paths, using the heuristic-driven exploitation algorithm to break ties.
4. Consider all edges in the chosen path, one after another. For each edge e in the sequence, do:
   (a) Traverse e from v to $succ_e(v)$.
   (b) Set $count(e) := count(e) + 1$.
   (c) If $count(e) = 1$, then set $i := i + 1$ and afterwards $reserve(e) := 2^{-i}$.
   (d) If $count(e) \geq k$ and $cost(e) = 0$, then set $cost(e) := reserve(e)$.
   (e) If $count(e) > k + 1$, then set $cost(e) := \infty$.
   (f) Set $v := succ_e(v)$.
5. Go to step 2.

Alternatively, step 3 can be replaced by:

3' Consider all acyclic paths starting at v and ending at a vertex with an emanating edge whose VECA cost is zero. Select a path with minimal VECA cost from these paths, using the heuristic-driven exploitation algorithm to break ties.

Figure 4.2: The Basic-VECA Framework

of 1/2 reserved, second - 1/4, third - 1/8, and so on. Figure 4.3 shows the spanning tree of

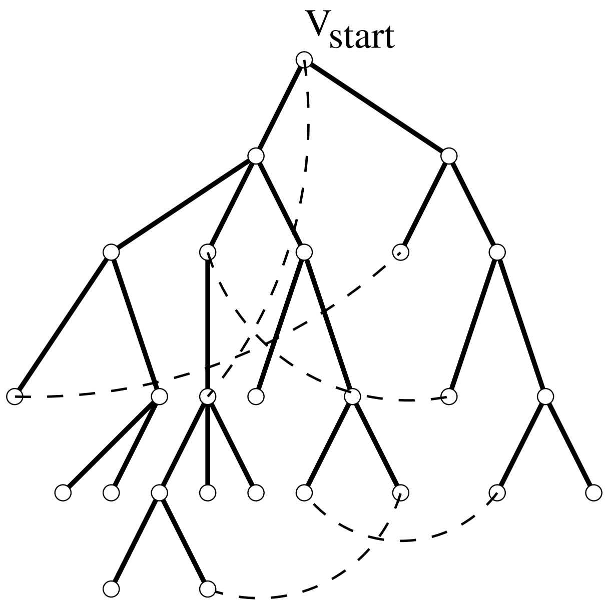

Figure 4.3: A Tree of the Highest VECA Cost

maximum VECA cost. Whenever VECA reaches an unvisited vertex, the just traversed edge should be included in the spanning tree. Inclusion of any other edge of the graph (dashed lines in Figure 4.3) would strictly decrease the VECA cost of the spanning tree.

VECA assigns the reserved cost to an edge when it traverses it for the kth time (or, if $k = 0$, when it traverses an edge for the first time). Whenever an edge is traversed $k + 2$ times, VECA assigns an infinite VECA cost to this edge, which effectively removes it from further consideration. The VECA costs are used as follows: VECA always chooses the least expensive path in terms of VECA cost that leads from its current state to an unexplored edge or, alternatively, to an edge with zero VECA cost. The exploitation algorithm is used to break ties. Initially, all VECA costs are zero and there are lots of ties to break. The more edges are assigned positive VECA costs to, the fewer ties there are and the less freedom the exploitation algorithm has.

To gain an intuitive understanding of the behavior of Basic-VECA, consider a simple undirected graph which is a tree, and assume that Basic-VECA uses step 3. Figure 4.4 shows an undirected edge e that connects two components of the tree, X and Y, with X containing the starting vertex. To reach component Y, Basic-VECA has to traverse edge e first. Thus, e will get a reserved cost 2^{-i} that is strictly bigger than the cost of any edge within component Y and the cost of any acyclic path within Y, because of the *tail majorizing* property: $2^{-i} > \sum_{j>i} 2^{-j}$.

Therefore, the exploitation algorithm can traverse e freely until it has been traversed k times. Then, Basic-VECA assigns this edge some positive VECA cost. At this point in time,

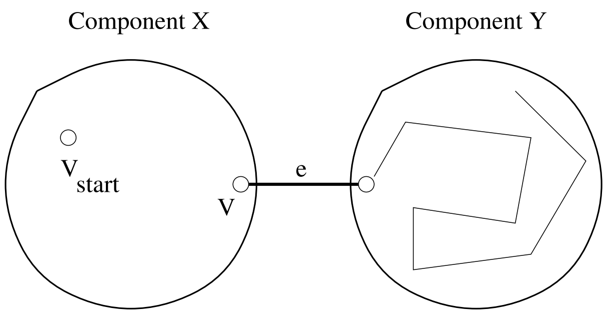

Figure 4.4: A Simple Example State Space

the agent is located in X (the component that contains the start state), since k is even and the agent alternates between both components. If Y does not contain any more untraversed edges, there is no need in traversing e anymore. Otherwise there is a point in time when Basic-VECA traverses e again to reach one of those untraversed edges in Y. When this happens, Basic-VECA prevents the exploitation algorithm from leaving Y until all edges in Y have been learned (this restriction of the freedom of the exploitation algorithm constitutes a switch from exploitation to more exploration): Because Y can only be entered by traversing e, this edge was traversed before any action in Y. Consequently, its positive VECA cost, when assigned from its reserved cost, is larger than the sum of reserved or assigned VECA costs of edges along any acyclic path in Y, because of the *tail majorizing* property. Thus, Basic-VECA cannot leave Y until all of Y's edges have been traversed. When Basic-VECA finally leaves Y, the VECA costs of e is infinite, but there is no need to come back to Y again.

In general, Basic-VECA traverses every undirected edge at most $k + 2$ times before it finds a goal state [71]. This implies the following theorem:

Theorem 4.1 *Basic-VECA with even parameter $k \geq 0$, solves the goal-directed exploration problem for any undirected, unknown graph with the complexity of $O(1) \times weight$ (to be precise: at most $(k + 2) \times weight$).*

A larger k allows the exploitation algorithm to maintain its original behavior longer, whereas a smaller k forces it earlier to explore the problem domain more. The smaller the value of k, the better the performance guarantee of VECA. If $k = 0$, for example, VECA severely restricts the freedom of the exploitation algorithm and behaves like Chronological Backtracking. In this case, it executes every edge at most twice (once in each direction, with the total complexity under $2weight(G)$), no matter how misleading its heuristic knowledge is or how bad the choices of the exploitation algorithm are. No uninformed goal-directed exploration algorithm can do better in the worst case. However, if the heuristic values are not misleading, a small value of k can force the exploitation algorithm to explore the state space unnecessarily. Thus, a stronger performance guarantee might come at the expense of a decrease in the empirical performance. The experiments in Section 4.4 address this issue.

By now, we would like to discuss the effects of dropping the requirement of learning the twin edge after traversing one in bi-directed domains. This type of relaxation might force an algorithm to perform an additional exploration procedure every so often to actually learn the twin before assigning both edges the same positive cost. Thus, we apply almost the same framework to bi-directed domains.

VECA is very similar to Basic-VECA, see Figure 4.5. In contrast to Basic-VECA, however, it does not assume that by traversing a directed edge one identifies its twin. This complicates the algorithm somewhat: First, the twin of an edge might not be known when VECA reserves a VECA cost for the pair. This requires an additional amount of bookkeeping. Second, the twin of an edge might not be known when VECA wants to assign it the positive VECA cost. In this case, VECA is forced to identify the twin: Step 4(e) explores all untraversed edges emanating from the same vertex as the twin edge (thus, including the twin) and returns to that state. This procedure is executed seldomly for larger k, since it is a rare case that a directed edge is traversed k times and the twin of that edge has not yet been learned. Because of this step, though, VECA can potentially execute any directed one more time than Basic-VECA, which implies the following theorem:

Theorem 4.2 *VECA, with even parameter $k \geq 0$, solves any bi-directed goal-directed exploration problem with a cost of $O(1) \times weight$ (to be precise: with a cost of at most $(k/2 + 2) \times weight$).*

For $k = 0$, VECA traverses every directed edge at most twice. Thus, its worst-case performance is at most $2 \times weight$ and equals the worst-case performance of BETA. No uninformed goal-directed exploration algorithm can do better in the worst case if traversing a directed edge does not identify its twin in a bi-directed domain.

Both Basic-VECA and VECA apply to goal-directed exploration in dynamically changing domains that change seldomly at discrete points in time. Basic-VECA provides a variety of scenarios for possible continuation after each domain change. For the case, when the agent is informed about all (visited) vertices with (new and old) unexplored emanating edges, the two main alternatives are:

- Increase the parameter of VECA by two, change all positive VECA costs, continue exploration according to Basic-VECA with parameter $k + 2$.

- Re-start goal-directed exploration from the current vertex by Basic-VECA with the lowest possible parameter 0.

Input:
a goal-directed exploration problem,
the value of VECA's parameter k (a non-negative, even integer),
and a heuristic-driven exploitation algorithm (to be used in step 3).

VECA uses four variables for each directed edge $e \in E$: $count(e)$ keeps track of how many times this edge has been traversed, $reserve(e)$ is its reserved VECA cost, $cost(e)$ is its actual VECA cost, and $euler(e)$ remembers whether it has already been traversed as part of step 4(e).

1. Set $count(e) := cost(e) := euler(e) := 0$ for all $e \in E$. Set $i := 0$ and $v := v_{start}$.
2. If $v \in G$, then stop successfully.
3. Consider all acyclic paths starting starting at v and ending at a vertex with an emanating *untraversed edge*. Select a path with minimal VECA cost from these paths, using the heuristic-driven exploitation algorithm to break ties.
4. Consider all edges in the chosen path, one after another. For each edge e in the sequence, do:
   (a) Traverse e from v to $succ_e(v)$.
   (b) Set $count(e) := count(e) + 1$.
   (c) If $count(e) = 1$ and the twin of e is not yet known, then set $i := i + 1$ and afterwards $reserve(e) := 2^{-i}$.
   (d) If $count(e) = 1$ and the twin of e is known, then let $e' \in E$ be the twin of e. If $count(e') = 0$, then set $i := i + 1$ and afterwards $reserve(e) := 2^{-i}$, else set $reserve(e) := reserve(e')$.
   (e) If $count(e) \geq k$ and the twin of e is not yet known, then do:
      i. Set $v'' := succ(e)$.
      ii. Select an edge $e'' \in E(v'')$ with $euler(e'') = 0$. If there is no such edge, then go to step 4(f) (comment: it holds that $v'' = succ(e)$).
      iii. Traverse e''.
      iv. Set $euler(e'') := 1$.
      v. Set $v'' := succ(e'')$.
      vi. Go to step 4(e)ii.
   (f) If the twin of e is known, then let $e' \in E$ be the twin of e and do:
      i. If $count(e) + count(e') \geq k$ and $cost(e) = 0$, then set $cost(e) := cost(e') := reserve(e)$.
      ii. If $count(e) + count(e') > k$, then set $cost(e) := \infty$.
   (g) Set $v := succ(e)$.
5. Go to step 2.

Alternatively, step 3 can be replaced by:

3' Consider all acyclic paths starting at v and ending at a vertex with an emanating edge whose VECA cost is zero. Select a path with minimal VECA cost from these paths, using the heuristic-driven exploitation algorithm to break ties.

Figure 4.5: The VECA Framework

First alternative shows that Basic-VECA is consistent with respect to exploration in dynamically changing environments. The increase of the parameter's value and, consequently, of the worst-case guarantee is an inevitable price that one have to pay for arbitrary domain changes. Had the change taken part in an unexplored portion of the domain, for example, there would be no need in the increase of VECA's parameter.

If the agent is performing goal-directed exploration in a bi-directed domain and informed about all (visited) vertices with (new and old) unexplored emanating edges, the two main alternatives for VECA are as follows:

- Increase the parameter of VECA by two, change all positive VECA costs, continue exploration according to VECA with parameter $k + 2$.
- Consider a connecting component containing the current vertex. Build a Eulerian tour on the component starting (and finishing) at the current vertex. Assign positive VECA costs according to 0-VECA and the constructed Eulerian tour, continue exploration from the current vertex by VECA with parameter 0.

If the agent is not informed about where new unexplored edges has been added to the domain, one can either re-start exploration from scratch or attempt to re-utilize already acquired map of the explored portion of the domain. In Chapter 5 we discuss the applications of VECA for the goal-directed exploration problems, when the map of the problem domain is provided in advance, but some vertices (edges) from the map can be blocked (untraversable) in the real domain.

Theorem 4.1 and Theorem 4.2 stated the worst-case complexity of Basic-VECA and VECA in terms of the weight of the graph. It is possible to improve these worst-case guarantees through a simple trick: One considers a spanning tree of the explored portion of the domain, sets positive VECA costs and decides where to go next according to the spanning tree VECA costs. Since the weight of a spanning tree is $\Theta(|V|)$, we can state the following corollary:

Corollary 2 *Spanning-Tree-Basic-VECA with even parameter $k \geq 0$, solves the goal-directed exploration problem for any undirected, unknown graph with the complexity of $O(1) \times |V|$ (to be precise: at most $(k+2) \times |V|$).*

Spanning-Tree-VECA, with even parameter $k \geq 0$, solves any bi-directed goal-directed exploration problem with a cost of $O(1) \times |V|$ (to be precise: with a cost of at most $(k+4) \times |V|$).

Chapter D-star contains a detailed discussion on the spanning tree improvement of VECA.

4.3 Implementation

Since the VECA costs are exponentially decreasing and the precision of numbers on a computer is limited, Basic-VECA cannot be implemented exactly as described. Instead, we represent sequences of edges in candidate paths as lists that contain the current VECA costs of the edges in descending order. All paths of minimal VECA cost then have the smallest lexicographic order. Since this relationship continues to hold if we replace the VECA costs of the edges with their exponent (for example, we use -3 if the VECA cost of an action is $1/8 = 2^{-3}$), we can now use small integers instead of exponentially decreasing real values, and steps 3 and 3' can be implemented efficiently using a simple modification of Dijkstra's algorithm in conjunction with priority lists. Table 4.1 presents the description of the modified Dijkstra's algorithm.

---

procedure Modification of Dijkstra's Algorithm, G=(V,E)
Algorithm starts at $v_{current}$:

Initialize Single Source $(G, v_{current})$
$S \leftarrow v_{current}$
$Q \leftarrow V \setminus v_{current}$
Path-cost$[v_{current}] \leftarrow \emptyset$
while $Q \neq \emptyset$
    if (there exists a non-stack edge (v, w) such that $v \in S$,$w \in Q$)
        $S \leftarrow S \bigcup w$
        $Q \leftarrow Q \setminus w$
        parent$[w] \leftarrow v$
        Path-cost$[w] \leftarrow$ Path-cost$[v]$
    else begin pop edge (x,y) from the stack
        while (it is not true that $x \in S$,$y \in Q$)
            pop edge (x,y) from the stack
        $S \leftarrow S \bigcup y$
        $Q \leftarrow Q \setminus y$
        parent$[y] \leftarrow x$
        Path-cost$[w] \leftarrow$ Path-weight$[v]$ $\bigcup$ $\{stack\ \ order((x,y))\}$
        restore stack without (x,y)
    end

Table 4.1: Modification of Dijkstra's Algorithm for Path Costs.

We gave two choices of step 3 to VECA to accommodate a wider variety of exploitation algorithms. For example, when implemented with AC-A*, VECA would use step 3 from Figure 4.2 and Figure 4.5 to search for the least expensive path from the current vertex to an untraversed edge. When implemented with LRTA* or Random Walk, VECA would use step 3' to find the cheapest path to an edge with zero VECA cost. These ways of integrating AC-A* and LRTA* with VECA are natural extensions of the stand-alone behavior of these algorithms.

4.4 Experimental Results

We augment our theoretical worst-case analysis with an experimental average-case analysis, because the worst-case complexity of an algorithm often does not predict its empirical performance well. For the experiments, we use an implementation of the VECA framework that allows for easy selection of setups through a graphical user interface, Figure 4.6 shows a screen dump.

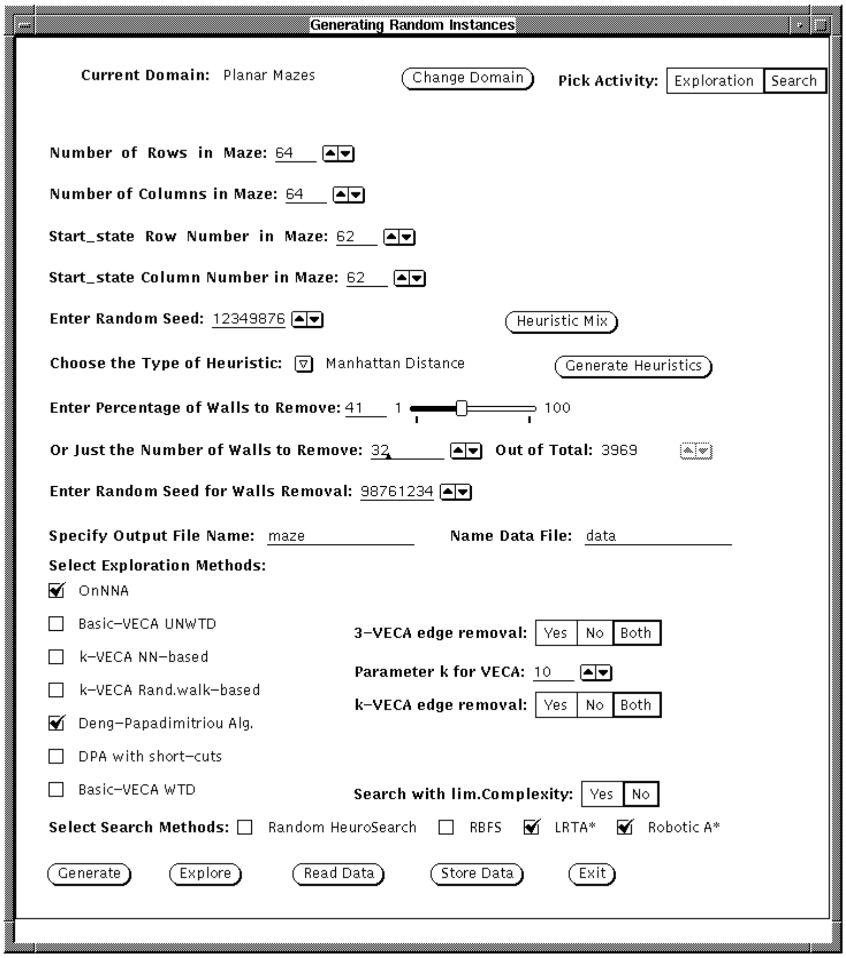

Figure 4.6: The Graphical User-Interface of the VECA System

The task that we study here is finding a goal state in mazes. The mazes were constructed by first generating an acyclic maze of size 64×64 and then randomly removing 32 walls. The edge lengths correspond to the travel distances; the shortest distance between two junctions counts as one unit. We randomly created five mazes with start location (62,62), goal location (0,0), a diameter between 900 and 1000 units, and a goal distance of the start state from 650 to 750 units. For every goal-directed exploration algorithm tested, we performed 10 trials in each of the five mazes (with ties broken randomly). We measure their performance as the total travel distance (cost) from the start state to the goal state averaged over all experiments.

Here, we report our experiments with two heuristic-driven exploitation algorithms, namely AC-A* and Learning Real-Time A* (LRTA*) [42] with lookahead one. These algorithms are integrated into the VECA framework as follows: AC-A* is used with step 3 of VECA. It breaks ties among action sequences according to their cost (see the definition of AC-A*). LRTA* is used with step 3' of VECA. It breaks ties according to how promising the last action of each action sequence is. These ways of integrating AC-A* and LRTA* with VECA are natural extensions of their stand-alone behavior.

In our experiments, we vary both the value of VECA's parameter k and the available heuristic values. We noticed that the assignment of positive VECA costs after odd number of traversals improves the empirical performance of VECA in conjunction with LRTA* or AC-A*. Thus, unless VECA's parameter is zero, we assign a reserved positive cost to a pair of twin edges after it has been traversed $k-1$ (odd) times. The flexibility of VECA allows one to exploit any search rules before the pair of edges is traversed k times, including assigning positive costs one step earlier.

To create heuristic values $h_1(e)$ of different quality for edges $e = (v, w) \in E$, we combine the goal distance $h^*(e)$ with the Manhattan distance $mh(e)$ (the sum of the x and y distance from vertex v to the goal state) using a parameter $t \in [0, 1]$ (t determines how misleading the heuristic values are; the smaller t, the lower their quality):

$$h_1(e) = t \times (length(e) + h^*(succ_e(v))) + (1-t)(length(e) + mh(succ_e(v)))$$

Figure 4.7, for instance, shows two example runs of AC-A* without VECA: The left figure shows which actions AC-A* explored for $t = 1$ until it reached the goal state. A thin line means that an action has been executed at least once; a bold line means that both the action and its twin have been executed at least once. AC-A* moves the agent with minimal cost to the goal. If we increase the contribution of the Manhattan distance to the heuristic values, the total cost of the actions executed from the start state to the goal state increases. The right figure, for example, shows which actions AC-A* explored for $t = 0$. In our experiments, we also use heuristic values $h_2(e)$ that were derived by combining the goal distance with the $Max(X, Y)$ heuristic (the maximum of the x and y distance from state s to the goal state), again using parameter t. Both $h_1(e)$ and $h_2(e)$ are consistent and thus admissible.

Figure 4.8 shows the empirical performance of AC-A* with and without VECA. In the left diagram, the heuristic values $h_1(e)$ were used; the right diagram shows the same results for the heuristic values $h_2(e)$.

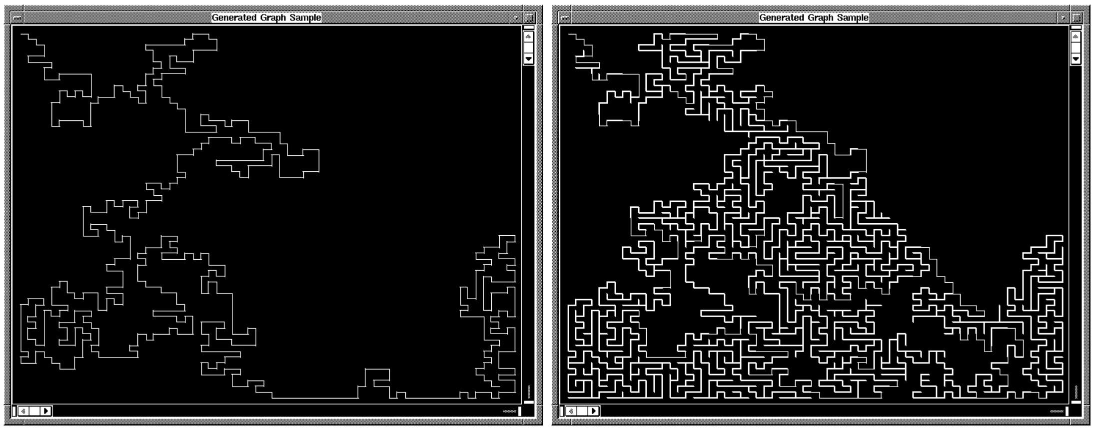

Figure 4.7: Exploration Behavior of AC-A* with Different Heuristics

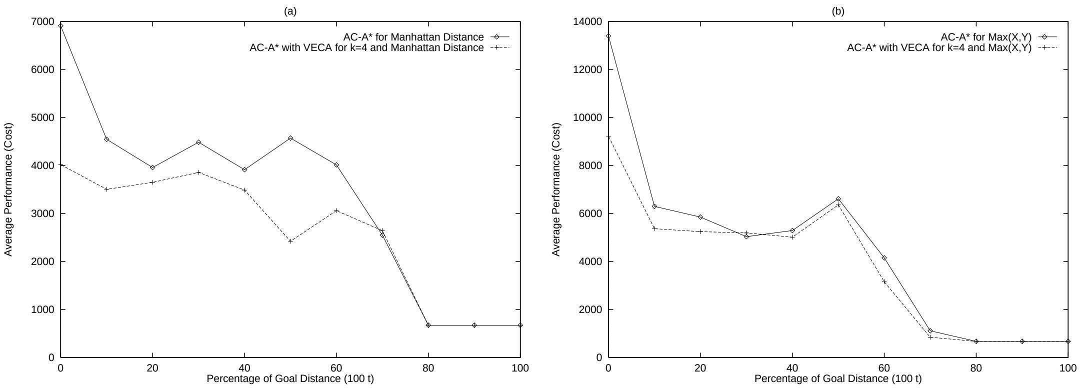

Figure 4.8: Empirical Performance of AC-A* with and without VECA

In both cases, the x axis shows our measure for the quality of the heuristic (100 times the value of t) and the y axis shows the average cost for one run. All graphs tend to decrease for increasing t, showing that the quality of the heuristic values increases, as expected. AC-A* without VECA is already efficient – it does not execute the same action a large number of times. Thus, VECA does not change the behavior of AC-A* if k is large – it turns out that the behavior of AC-A* with VECA for $k = 10$ is already the same as the behavior of AC-A* without VECA. The graphs for $k = 4$ suggest that AC-A* with VECA now outperforms AC-A* without VECA. The abnormal behavior of AC-A* for the Max(X,Y) heuristic around $t = 0.5$ can be explained as follows: Equal amounts of the goal distance and the Max(X,Y) heuristic produce many ties between different directions. AC-A* then does not follow one direction, but tends to alternate between them.

Figure 4.9 shows the empirical performance of LRTA* with and without VECA. The qualitative behavior is the same for $h_1(e)$ and $h_2(e)$. Again, the larger t is, the better is the quality of the heuristics, and all graphs are monotonically decreasing. Only for misleading heuristic values (small t) does LRTA* with VECA outperform LRTA* without VECA. This is so, because VECA forces LRTA* to explore the state space too much if the heuristic values are only moderately misleading. For the same reason, the LRTA* VECA combination with a small k outperforms this combination with a large k only if t is small.

For heuristic values of "high" quality, all considered algorithms establish either optimal or near-optimal behavior. We noticed that LRTA* remains near-optimal the longest, until the contribution of the goal distance is at least 40%. However, when the heuristic became misleading, LRTA* established the worst performance among the above algorithms. The combination of VECA with LRTA* loses to LRTA* for a certain interval of values of t, the range of the interval depends also on the value of VECA's parameter k. For lower k (like 2 or 4), VECA with LRTA* begins losing earlier to stand-alone LRTA*, but gains a win of several multitudes when heuristic values become pure Manhattan distance. VECA with higher values of parameter k deviates later from LRTA*, but the amount of win for misleading heuristics is significantly less. These facts provide an intuition to the dependency of the efficiency of VECA to the value of parameter k.

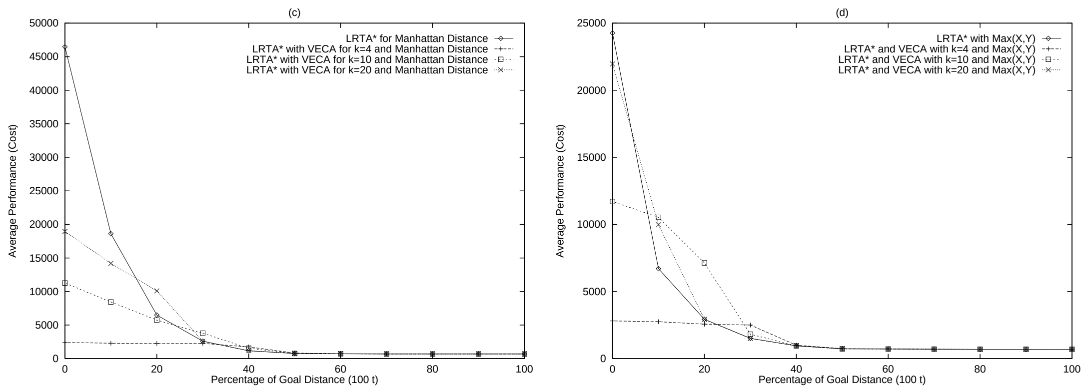

Figure 4.9: Empirical Performance of LRTA* with and without VECA

When comparing the behavior of AC-A* with that of LRTA*, we notice that AC-A* is more efficient. This is to be expected, since LRTA* deliberates much less between action executions – it has been designed for the case that action executions and deliberation are about equally fast, which is not the case here. Thus, VECA is able to improve the empirical performance of LRTA* more than that of AC-A* if the heuristic values are misleading. Also, AC-A* is more brittle than LRTA* towards variations in the quality of heuristic values, since AC-A* makes more global decisions about where to move in the state space and is thus much more affected by a wrong decision.

We also ran experiments with other goal-driven exploitation algorithms, such as biased random walks and an algorithm that moves the agent so that it expands the states in the same order as the A* algorithm does. The results are qualitatively similar, but more impressive: Since both of these goal-driven exploitation algorithms are inefficient to start with, VECA achieves a much larger improvement in the empirical performance if the heuristic values are misleading.

We also performed experiments in different domains, because the performance of goal-directed exploration algorithms does not only depend on the heuristic values used, but also on the properties of a domain. The results show that the following factors have a significant influence on the performance of goal-directed exploration algorithms: the goal distance of the start state, the density of the state space, its connectivity, and a new feature that we call oblongness (the ratio of its diameter to the number of states). For example, goal-directed exploration algorithms exhibit similar performance for exploration problems whose oblongness and goal distances of the start state are similar. Our current work focusses on studying the influence of such domain properties on the efficiency of goal-directed exploration in more detail.

4.5 Multiple Agents

In this section we discuss the goal-directed exploration problem in multi-agent domains. The multi-agent exploration is a rich area by itself. It involves such fundamental issues as cooperation, information exchange and non-determinism among others. In this section we show that VECA can be applied to multi-agent scenarios as well.

In general, hybrid approaches provide a fruitful arena for the multi-agent behavior. Several agents may follow different strategies simultaneously as long as they do not compete for resources. If the strategies were selected to cover different efficiency dimensions, such a setting may cover efficiently both "hard" and "easy" cases. In particular, regarding the goal-directed exploration problem, one of the agents ("cautious") may follow one of the strategies from CS theory that would provide strong performance guarantees. Second agent ("optimist") may try one of more risky AI strategies, third agent ("pragmatist") may choose VECA with one's favorite value of k, etc.

Is this multi-agent setting too prudent? What if one considers the best possible scenario of having powerful "optimists" on the team, who would cooperate in their joint search for a goal in an unknown environment? AC-A* is one of those advanced "optimists" that exploits an efficient strategy even for a single-agent domain. Can it be the case that the team of an arbitrary many cooperating "optimists" would follow the AC-A* strategy and lose to a single "cautious" agent? Unfortunately, the answer is negative. Whatever number of "optimists" decide to cooperate completely and follow the AC-A* strategy, even if they do not compete for resources, i.e. they are allowed to be simultaneously at the same time at the same location and can share edge traversals, there exists a domain in which a single "cautious" agent would outperform the whole "optimistic" team.

One can come up with a modification of Figure 3.3 for the counter-example of the multi-agent exploration scenario. Number of branches of each size is multiplied by N – the number of agents. An example of such graph is presented in Figure 4.10. The team of "optimists" would start at v_0, travel along the stem x times, each time attempting to traverse N identical branches that share a common stem vertex v_i, until one of the agents discovers the goal at the end of one of the N longest branches. On the other hand, if the "cautious" agent that follows Chronological Backtracking or DFS, starts at the stem vertex attached to the goal stem, it will not traverse more than the weight of the graph before reaching the goal. Simple calculations show that each "optimist" would traverse x^x less than a single agent in graph shown in Figure 3.3:

$$length_{``optimistic''} = \frac{x^{x+3} + x^{x+2} - 4x^{x+1} + x + 1}{(x-1)^2}$$

The "cautious" agent would traverse the length of at most the weight of the graph:

$$length_{``cautious''} = \frac{(3N+1)x^{x+2} - (4N+2)x^{x+1} - (N-1)x^x + 2}{(x-1)^2}$$

Thus, as soon as the graphs parameter x satisfies $x \geq 3N + 1$, the "cautious" agent would experience a sweet victory over the team of "optimists." This facts emphasizes the importance of considering worst-case scenarios, since the price that an "optimistic" team may pay, can be multiplied by the number of the agents, besides simply losing to a single agent in time.

Fortunately, VECA provides a good solution to the multi-agent variant of the goal-directed exploration problem. VECA with low parameter k establishes strong performance guarantees for an arbitrary reversible domain. One of the agents may follow such VECA for a "backup." We also found that when VECA's parameter k is equal to 2 or 4, it outperforms all other known goal-directed exploration algorithms in sparse tree-like mazes. Such a "super-cautious" agent might be very helpful in "hard" search problems.

This simple example shows the importance of the team diversification, as one may perceive life as a sequence of goal-directed exploration problems in a partially unknown world.

4.6 Summary

We introduced an application-independent framework for goal-directed exploration, called VECA, that addresses a variety of search problems in the same framework and provides good performance guarantees. VECA can accommodate a wide variety of exploitation strategies that use heuristic knowledge to guide the search towards a goal state. For example, in Chapter 5

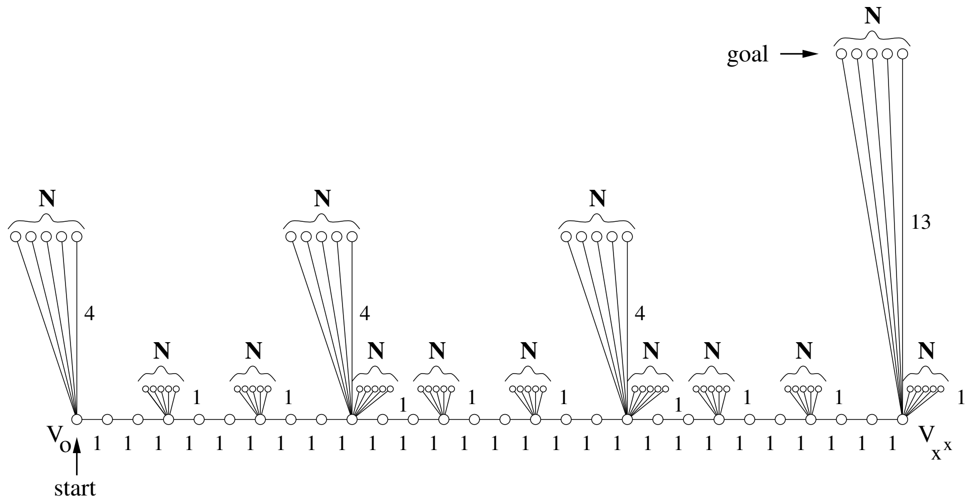

Figure 4.10: A Bad Graph for the "Optimistic" Team

we discuss the application of VECA to the sensor-based planning problem. VECA monitors whether the heuristic-driven exploitation algorithm appears to perform poorly on some part of the state space. If so, it forces the exploitation algorithm to explore the state space more. This combines the advantages of pure exploration approaches and heuristic-driven exploitation approaches: VECA is able to utilize heuristic knowledge and provides also a better performance guarantee than previously studied heuristic-driven exploitation algorithms (such as the AC-A* algorithm). VECA's worst-case performance is always linear in the weight of the state space.

Thus, while misleading heuristic values do not help one to find a goal state faster, they cannot completely deteriorate its performance neither. A parameter of VECA determines when it starts to restrict the choices of the heuristic-driven exploitation algorithm. This allows one to trade-off stronger performance guarantees (in case the heuristic knowledge is misleading) and more freedom of the exploitation algorithm (in case the quality of the heuristic knowledge is good). In its most stringent form, VECA's worst-case performance is guaranteed to be as good as that of BETA, the best uninformed goal-directed exploration algorithm. Our experiments suggest that VECA's strong worst-case complexity does not greatly deteriorate the empirical performance of the combinations of previously studied heuristic-driven exploitation algorithms if they are used in conjunction with VECA. Furthermore, in many cases VECA even improved their performance.

Chapter 5

Agent-Centered Approach for Sensor-Based Planning

In this chapter, we analyze a navigation problem in which a robot has to navigate from a given start location to a given goal location in an unknown terrain. We model this navigation problem as finding a path from a start vertex to a goal vertex in an initially partially or completely unknown graph that represents the terrain. This path planning problem is complicated by the fact that the sensors on-board a robot can typically sense the environment only near its current position, and thus the robot has to interleave planning with moving to be able sense its environment. As the robot moves, it acquires more knowledge about the terrain and consequently reduces its uncertainty about the environment, which also reduces the number of contingent obstacle configurations that the planner has to consider. In the simplest setting we assume that the domain is static and robot's actions are deterministic. Even such a generate scenario prompts for further investigations concerning efficient strategies that would balance action execution and exploring the domain. Thus, sensing during plan execution and using the acquired knowledge for re-planning (often called "sensor-based planning") makes the planning problem tractable.

The above sensor-based planning problem and the goal-directed exploration problem introduced in Chapter 3, are somewhat similar. Robot's sensor limit lookahead of the robot, one's knowledge about the environment is incomplete, one has to explore it sufficiently to reach the goal. However, in the sensor-based planning problem, one has an additional knowledge in a form of already provided map that imperfectly represents the problem domain. The presence of this knowledge promoted the hopes that the sensor-based planning problem is essentially easier than the goal-directed exploration problem.

In this chapter we show that two problems are indeed close and accept variations of VECA as efficient solutions. Furthermore, we modify an example from Chapter 3 to demonstrate that a popular technique of planning with the freespace assumption is not optimal for the sensor-based planning problem, although it provides strong empirical performance. Thus, the presence of the map seems not to facilitate agent's task.

5.1 Sensor-Based Planning with the Freespace Assumption

A popular technique for sensor-based planning is planning with the freespace assumption [17] [56] [73] [82]: The robot assumes that the terrain is clear unless it knows otherwise. It always plans a shortest path from its current location to the goal location. When it detects an obstacle that blocks its path, it replans a shortest path from its current location to the goal location using its knowledge about all obstacles encountered so far. It repeats this procedure until it reaches the goal location or realizes that reaching the goal is impossible. Planning with the freespace assumption has been used both on grids and visibility graphs. This approach allows to omit an expensive procedure of modeling all obstacles of the problem domain and introduce them in a simplified form upon sensing.

In the literature, it has been conjectured that this sensor-based planning approach might be optimal[82], given the lack of initial knowledge about the environment. We show that this is not the case. In particular, we demonstrate that planning with the freespace assumption can make good performance guarantees on some restricted graph topologies (such as grids), but is not optimal in general. For situations in which its performance guarantee is not sufficient, we also describe an algorithm, called Basic-VECA, that exhibits good average-case performance and provides performance guarantees that are optimal up to a constant factor.

5.2 Problem Description

In this section, we formalize the navigation problem that we study and show how it has been used on actual robots. We state the sensor-based planning problem as follows:

> **Sensor-Based Planning Problem**: An agent is given an undirected, finite graph with positive edge lengths and vertices that are either blocked or unblocked (their status does not change over time). One unblocked vertex is labeled the starting vertex; one vertex is labeled the goal vertex. The agent can move from its current vertex to any unblocked neighboring vertex. Initially, it does not know the status of all vertices. However, it always senses the status of its neighboring vertices. The agent is started at the starting vertex and has to either move to the goal vertex, or recognize that this is impossible.

This sensor-based planning problem can be pictured as follows: A robot is given a map and has to move from its current location to a given goal location. Intersections can be

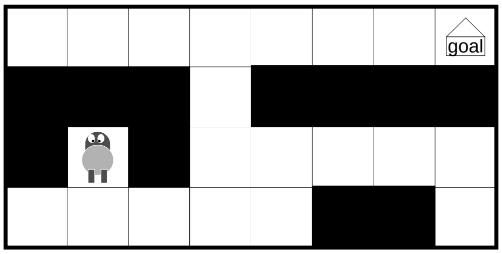

Figure 5.1: Outdoor robot navigation with NAVLAB II

blocked by construction sites, but the robot does not have complete prior knowledge about which intersections are blocked. It can, however, observe the status of all its neighboring intersections.

In the literature, the sensor-based planning problem has been studied in the context of actual robot navigation problems. One of the applications of sensor-based planning problems is an outdoor navigation problem that has been used to formalize the mission of NAVLAB II, Carnegie Mellon's robot HMMWV (high mobility multi-wheeled vehicle) [74].

> **Outdoor Navigation Problem:** An unmanned ground vehicle has to reach specified coordinates in an unmapped static terrain. To do so, it discretizes the unknown area into a coarse-resolution map of square cells. Each cell is either traversable or untraversable. The vehicle always occupies exactly one cell and can move in all eight compass directions to traversable adjacent cells. Its sensor always detects which of its eight adjacent cells are traversable, and integrates all new information into the map.[1]

This outdoor navigation problem is a special case of the sensor-based planning problem on regular 8-connected grids. For example, a vehicle that operates in the terrain of Figure 5.1 initially has the knowledge shown in Figure 5.2. Figure 5.3 shows the corresponding traversability graph.

Another application of sensor-based planning problems is an indoor navigation problem that has been implemented by [56].

[1] NABLAB II does not only distinguish between traversable and untraversable cells, but differentiates more fine-grained by assigning traversal costs to them. It takes a robot, for example, longer to traverse a stretch of uneven or muddy terrain than it takes it to traverse a stretch of paved road of the same length. Re-planning occurs whenever a new traversal cost is assigned to a cell. Our description is a special case of this approach that distinguishes only two traversal costs, one of which is infinite. This does not affect our conclusions, because a sensor-based planning algorithm that is inefficient for the special case is also inefficient in general.

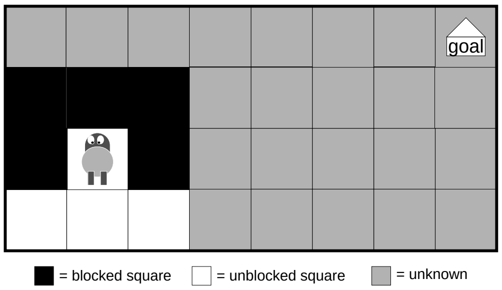

Figure 5.2: Initial knowledge of the robot

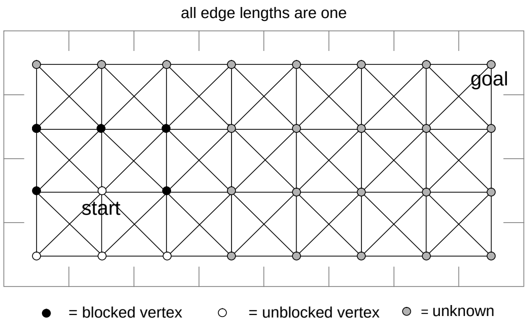

Figure 5.3: Initial graph

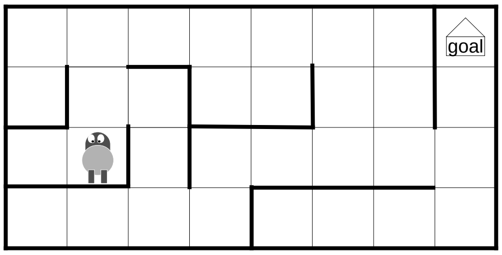

Figure 5.4: Indoor robot navigation with a Nomad

Indoor Navigation Problem: A Nomad-class mobile robot has to reach a goal position in an unknown static maze with walls. The robot can move in the four main compass directions to adjacent cells if no wall blocks its path. Its sonar sensors always detect the presence of walls adjacent to the robot, and integrate all new information into the map.

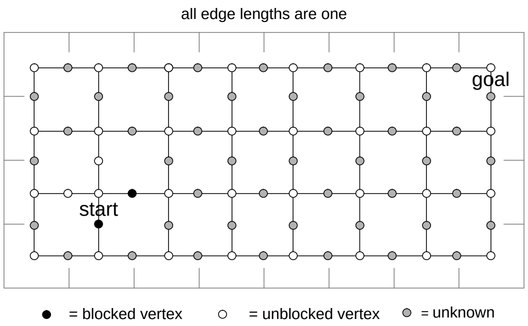

Figure 5.5: Initial graph (1)

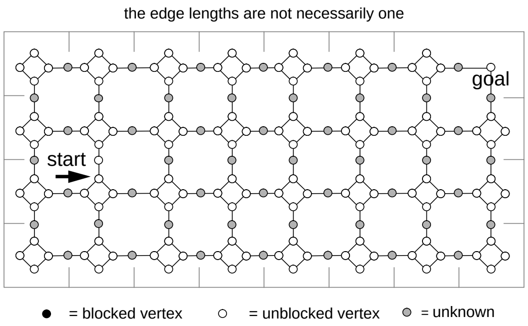

Figure 5.6: Initial graph (2)

This indoor navigation problem is a special case of the sensor-based planning problem on regular 4-connected grids with extra vertices. For example, Figure 5.5 shows the traversability graph that corresponds to the initial knowledge of a Nomad robot that operates in the maze of Figure 5.4. The extra vertices are necessary to convert the sensor-based planning problem from one where the edges are blocked or unblocked to one where the vertices are blocked or unblocked.

As an example for how to model holonomic, but not omni-directional mobile robots with limited sensing direction, we consider a robot that can only sense the wall directly in front of it and solves the indoor navigation problem by repeatedly moving forward, turning left 90 degrees, and turning right 90 degrees. Figure 5.6 shows the corresponding traversability graph. In this case, the edge lengths are not necessarily uniformly one (not shown in the figure), since turning and moving forward can take different amounts of time.

5.3 D* Algorithm

In this section, we describe planning with the freespace assumption as a greedy way of solving sensor-based planning problems. Planning with the freespace assumption can be stated as follows:

> **Planning with the Freespace Assumption:** The robot always makes the optimistic assumption that vertices are unblocked if it does not know their status. It uses this assumption to plan a shortest traversable path (a path that does not contain vertices that are known to be blocked) from its current vertex to the goal vertex and traverses it until it learns about a blocked vertex on the path. At this point, it repeats the procedure, taking into account its knowledge about which vertices are blocked. If it reaches the goal vertex, it stops and reports success. If, at any point in time, it fails to find a traversable path from its current vertex to the goal vertex, it stops and reports that the goal vertex cannot be reached.

The D* algorithm assumes that re-planning takes finite amount of time at any point in time. Since the re-planning step is based on finding a shorting path, the finite assumption is thus based on the finite size of the graph that models the problem domain, which is a common assumption. The most time-consuming step of planning with the freespace assumption is re-calculating a shortest path when new knowledge about obstacles has been acquired. The Dynamic A* (D*) algorithm [72] does this without unnecessary re-calculations.

Theorem 5.1 *Planning with the freespace assumption terminates in finite time and is correct.*

Proof:

- Termination: Every time when the robot cannot follow a planned path, it has learned about at least one additional blocked vertex, and there are only a finite number of them, implying that planning with the freespace assumption terminates in finite time, provided that a re-planning method is used that is capable of re-planning in finite time.

- Correctness: Planning with the freespace assumption reports success only if it is at the goal vertex and thus has solved the sensor-based planning problem. If reports failure only if there is at least one blocked vertex on every path from its current vertex to the goal vertex and thus no traversable path from its current vertex to the goal vertex exists. Since there is a traversable path from its current vertex to the starting vertex (the robot was able to reach its current vertex from the starting vertex and this path can be traversed in reverse since the graph is undirected), there is no traversable path from the starting vertex to the goal vertex either. Consequently, reaching the goal vertex is impossible. ■

Planning with the freespace assumption has been used on actual robots. For example, it has been applied to the outdoor [74] and the indoor [56] navigation problems described above. Planning with the freespace assumption has several advantages: It is easy to implement. Its computations can be done efficiently. It takes advantage of newly acquired knowledge immediately and always uses all of its knowledge. Without any changes to the algorithm, it is able to use prior knowledge about blocked vertices. It learns an optimal trajectory over multiple trials with the same starting and goal vertices, since the freespace assumption encourages the exploration of vertices with unknown status. It exhibits a reasonable goal-oriented behavior in practice. Finally, when it is used in conjunction with grids (as in the outdoor navigation problem), it does not need to make assumptions about the shapes of obstacles.

5.4 Complexity Analysis

We measure the performance of sensor-based planning algorithms by the distance that the robot has to travel before it reaches the goal vertex or discovers that this is impossible. Since the environment is not completely known initially, we cannot expect the robot to follow the omniscient best path. However, it is a common assumption in the literature that planning with the freespace assumption is optimal given the lack of initial knowledge (in other words, that no other uninformed sensor-based planning can do better) [82] [73] [56], although no analytical results to this effect have been reported.

In the following section, we assume completely uninformed sensor-based planning algorithms. We provide lower bounds on the performance guarantee of planning with the freespace assumption (measured as the length of the robot's path until one reaches the goal or finds that the goal is unreachable) and show that, contrary to the existing belief, planning with the freespace assumption is not optimal, since there exists another uninformed sensor-based planning algorithm that provides a better performance guarantee. Section 5.4.2 contains upper bounds for the length of robot's path. (Further research involves closing the gap between the upper and the lower bounds.) In both sections, we need the following notation: $weight(G)$ denotes the weight of graph $G = (V, E)$ (the sum of all its edge lengths), and $dist_G(v_1, v_2)$ denotes the length of a shortest path between $v_1 \in V$ and $v_2 \in V$ in G.

5.4.1 Lower Bounds

In this section, we use two examples to establish lower bounds on the performance guarantee of planning with the freespace assumption when it is completely uninformed. These lower bounds demonstrate that planning with the freespace assumption is not optimal, in the following way: Consider the following uninformed sensor-based planning algorithm.

Chronological Backtracking: The robot always selects an edge that leaves its current vertex and traverses it, according to the following restrictions: If possible, the robot traverses an edge that leads to an unblocked vertex and that it has not yet traversed. If there is no such edge, it instead traverses the edge in the opposite direction with which it entered its current vertex for the first time ("backtracking"). If the robot reaches the goal vertex, it stops and reports success. If, at any point in time, the robot is at the starting vertex and has already traversed all of the edges that leave the starting vertex and lead to unblocked vertices, it stops and reports that the goal vertex cannot be reached.

Chronological backtracking solves any sensor-based planning problem with at most two traversals of every edge. Consequently, it provides a tight performance guarantee of $\Theta(weight(G))$, and no uninformed sensor-based planning algorithm can do better in the worst case.[2] The performance guarantee of planning with the freespace assumption has to be judged against this benchmark. We show that the performance guarantee of planning with the freespace assumption is superlinear in the weight of the graph (even for planar graphs) and thus worse than that of chronological backtracking. Hence it is not optimal.

Our example is a planar graph, because maps are usually planar. It is a simple modification of a graph from Figure 3.3[3].

The example is shown in Figure 5.7: The graph $G_1 = (V_1, E_1)$ consists of a stem with several branches that connect the goal vertex with the stem. All edge lengths are one. The stem has length n^n for some integer $n \geq 1$ and consists of the vertices $v_0, v_1, \ldots, v_{n^n}$, where v_0 is the starting vertex. For each integer i with $1 \leq i \leq n$ there are n^{n-i} branches of length $1 + \sum_{j=0}^{i-1} n^j$ each. These branches attach to the stem at the vertices v_{jn^i} for integers j; if i is even, then $0 \leq j \leq n^{n-i} - 1$, otherwise $1 \leq j \leq n^{n-i}$. All vertices that are not directly connected to the goal vertex are unblocked, and so are the goal vertex and the vertex on the longest branch that is directly connected to the goal vertex. All other vertices are blocked.

Planning with the freespace assumption can behave as follows if it has no initial knowledge: It starts at v_0 and traverses along the whole stem, trying to use the branches of length 2 to get to the goal vertex only to discover that they are blocked. It then switches directions and travels along the whole stem in the opposite direction, this time trying to use the branches of length $n + 2$ to get to the goal vertex (to discover again that they are blocked), and so forth, switching directions repeatedly. It succeeds when it finally attempts to use the longest branch.

[2]Chronological backtracking is often stated as follows: It always traverses untraversed edges that lead to unblocked *and previously unvisited* vertices, and backtracks if such edges do not exist. This improves the average-case performance, but does not change the performance guarantee.

[3]We have assumed that the robot is only able to sense the status of its neighboring vertices. The examples of this section can easily be adapted to sensors with larger lookaheads, say of x vertices, by replacing each edge with x consecutive edges that are connected via $x - 1$ unblocked intermediate vertices.

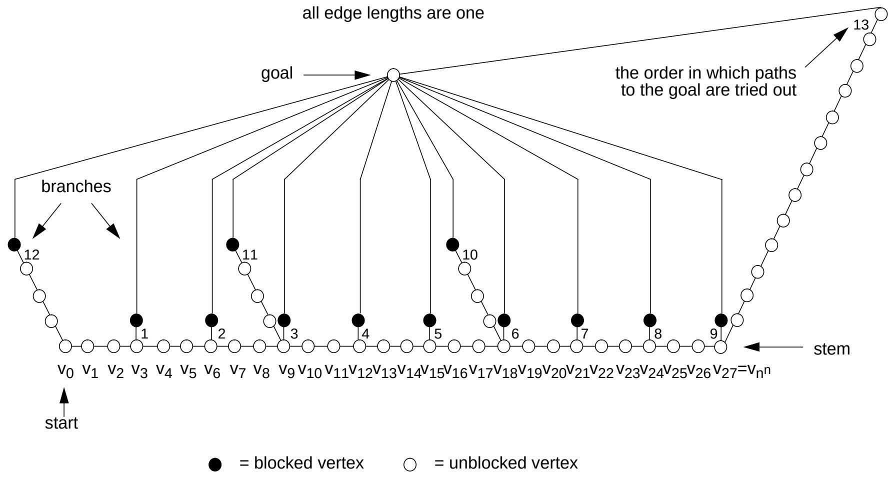

Figure 5.7: Graph G_1 for $n = 3$

To summarize, the edges connected to the goal vertex are tried out in the order indicated in Figure 5.7.

To calculate the performance of planning with the freespace assumption for this behavior, we need the following relationships: First, the total travel distance is at least $\Omega(n^{n+1})$, since the stem of length n^n is traversed n times. Second, the weight of the graph is tight at $weight(G_1) = |E_1| = \Theta(n^n)$. Finally, n is at least $\Omega(\frac{\log|V_1|}{\log\log|V_1|})$. Put together, it follows that the total travel distance is at least $\Omega(n^{n+1}) = \Omega(n \times weight(G_1)) = \Omega(\frac{\log|V_1|}{\log\log|V_1|} \times weight(G_1))$. Since this is superlinear in $weight(G_1)$, the performance guarantee of planning with the freespace assumption is worse than that of chronological backtracking and thus not optimal.

Perhaps, the graph constructed in Figure 5.7 looks very artificial. It was, indeed, a challenging task to come up with an example of the domain where the planning with the freespace assumption algorithm establishes a superlinear behavior, because, in general, this algorithm is very efficient. However, we found that any graph of the type presented in Figure 5.8 is somewhat misleading for the planning with the freespace assumption algorithm. Dashed lines represent non-intersecting paths with at least one vertex blocked along the paths. As long as the sum of the length of the traversable path between vertices v and w and the goal distance from w is greater than the sum of the distance from the current vertex on the upper stem to a dashed path, the length of a dashed path and the distance from a connecting vertex of the dashed path on the lower stem to the goal vertex, the planning with the freespace assumption algorithm will try all dashed paths before traversing the (v, w)-path.

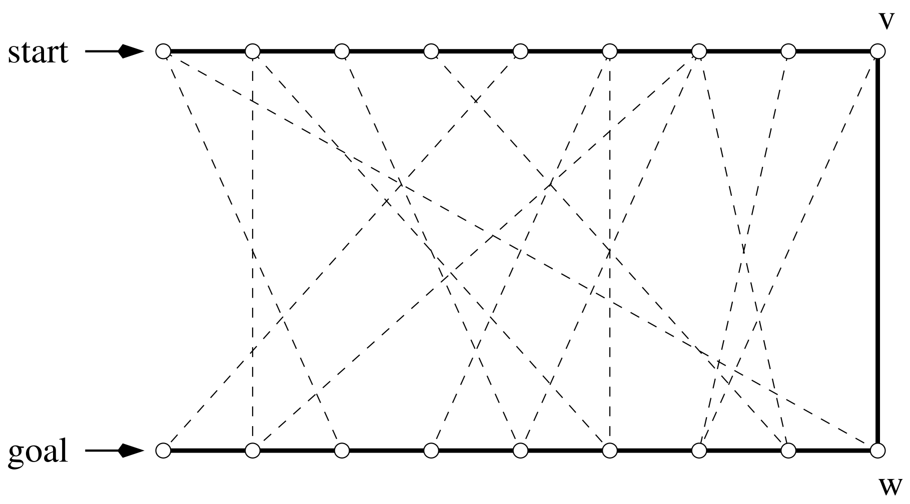

Figure 5.8: A Tough Graph

5.4.2 Upper Bounds

In this section, we prove upper bounds on the performance guarantee of planning with the freespace assumption. We state a bound that holds for all graphs and show how this bound can be improved for some restricted graph topologies, such as grids, that have often been used in conjunction with the freespace assumption. We proceed in two steps: First, we prove properties of the multiple shortest path algorithm, and then we apply them to planning with the freespace assumption.

The multiple shortest path algorithm is defined as follows: The algorithm is given an undirected, finite graph $G = (V, E)$ with positive edge lengths and a sequence $[w_i]_{i=1}^n$ of different vertices $w_i \in V$. When it is started at w_0, it moves on a shortest path to w_1, then moves on a shortest path to w_2, and so forth until it reaches w_n and stops. Any path that can result from this behavior is called a $([w_i]_{i=1}^n, G)$ path.

We make use of the following properties of the multiple shortest path algorithm:

Theorem 5.2 *Any $([w_i]_{i=1}^n, G)$ path on any tree $G = (V, E)$ contains any edge $e \in E$ at most* $\min(2|V_1|, 2|V_2|)$ *times, where $G_1 = (V_1, E_2)$ and $G_2 = (V_2, E_2)$ are the two disconnected graph components that are obtained from G by removing e.*

Proof: Without loss of generality, assume that $|V_1| \leq |V_2|$. The multiple shortest path algorithm traverses e towards G_1 only when it takes a shortest path from a vertex in G_2 to a vertex $w_i \in V_1$ in G_1. Since the graph is a tree, the traversals of e alternate directions. If $w_0 \in V_1$, then the edge is traversed at most $|V_1| - 1$ times towards G_1 and $|V_1|$ times towards

G_2. If $w_0 \in V_2$, then the edge is traversed at most $|V_1|$ times towards G_1 and $|V_1|$ times towards G_2. ■

Theorem 5.3 *Let $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ be any two graphs with $dist_{G_1}(v, v') \leq dist_{G_2}(v, v')$ for all vertices $v, v' \in V$. Then, the length of any $([w_i]_{i=1}^n, G_1)$ path is at most as long as the length of the correspondent $([w_i]_{i=1}^n, G_2)$ path.*

Proof: Since $dist_{G_1}(v, v') \leq dist_{G_2}(v, v')$ for all vertices $v, v' \in V$, the length of any shortest path in G_1 from any vertex to a vertex w_i is at most as long as the length of any shortest path in G_2 between the same vertices. ■

We now show how these properties of the multiple shortest path algorithm can be applied to planning with the freespace assumption:

Consider the sequence $[w_i]_{i=1}^n$ of vertices where w_0 is the starting vertex of planning with the freespace assumption, w_i for $i = 2 \ldots n-1$ are the vertices at which it successfully re-plans paths, and w_n is either the goal state or the state at which it realizes that reaching the goal state is impossible. The vertices w_i are pairwise different, since planning with the freespace assumption only re-plans paths at vertices at which it has never been before. This is the case because it only re-plans when it realizes that its current path is blocked and it would have known that the path was blocked had it been at that vertex before. Furthermore, it moves on shortest paths to vertices w_i, since it always plans shortest paths to the goal. (If the path to vertex w_i were not optimal, neither would be the path to the goal.) Put together, it follows that planning with the freespace assumption on a graph $G = (V, E)$ is a multiple shortest path algorithm on the graph that is obtained from G by removing all edges that border on at least one blocked vertex. For example, the performance guarantee of planning with the freespace assumption is no worse than $|V|\, weight(G)$ on any graph. This follows by summing the values of Theorem 5.2 over all edges, since $\min(2|V_1|, 2|V_2|) \leq |V|$.

Theorem 5.2 can also be used to prove better performance guarantees for restricted graph topologies. The graph shown in Figure 5.9 gives a general idea how one can apply Theorem 5.2 to particular graphs. For example, the shown graph is a spanning tree of a square grid with edge lengths one that does not contain blocked vertices. Summing the values of Theorem 5.2 over all edges results in a bound of $O(|V|^{3/2}) = O(|weight(G)|^{3/2})$ for this graph. A square grid with edge lengths one is supergraph of this graph and thus Theorem 5.3 applies: For each sequence $[w_i]_{i=1}^n$ of vertices, the travel distance of planning with the freespace assumption on the grid is at most as large as that on the spanning tree. Thus, the maximum travel distance on the grid over all sequences $[w_i]_{i=1}^n$ is at most as large as that on the spanning tree. It follows that the performance guarantee of planning with the freespace assumption on square grids with edge lengths one (or any supergraph of such grids) is $O(|weight(G)|^{3/2})$, and planning with the freespace assumption is optimal up to a factor of at most $|weight(G)|^{1/2}$ on these graphs.

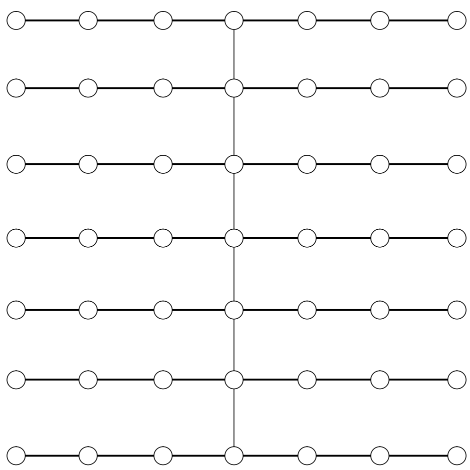

Figure 5.9: Subgraph of a square grid

In general, to determine an upper bound on the performance of planning with the freespace assumption for a given graph topology, one first removes all edges from the graphs that border on at least one blocked vertex. Then, one selects a spanning tree of the resulting graph and sums the values of Theorem 5.2 over all edges of the tree. The resulting bound is also an upper bound for the given graph topology. We found that the diameter of the spanning tree plays an important role in deriving an upper bound through applications of Theorem 5.2. Among all spanning trees containing non-blocked vertices, ones with smaller diameters are usually more preferable.

5.4.3 Applying VECA to Improve Performance Guarantees

In this section, we re-apply Basic-VECA to a sensor-based planning problem and show that it exhibits good empirical performance and provides performance guarantees that are optimal up to a constant factor. We have shown that chronological backtracking can provide a better performance guarantee than planning with the freespace assumption. On the other hand, planning with the freespace assumption exhibits a much better average-case performance than chronological backtracking in typical domains, since chronological backtracking does not actively search for the goal vertex.[4] Basic-VECA is an algorithmic framework that is capable

[4]Experimental results for planning with the freespace assumption are reported in [72]. Its average-case performance is very good, although recent experimental evidence [75] suggests that, at least in some domains, one can improve the total travel distance slightly by assuming that the status of a vertex is similar to the status of its

of accommodating both sensor-based planning algorithms. Its performance guarantee is better than that of planning with the freespace assumption, and its average-case performance is better than that of chronological backtracking (although it can be worse than that of planning with the freespace assumption).

Basic-VECA with $k = \infty$ behaves identically to the goal-directed planning algorithms. On the other hand, Basic-VECA with $k = 0$ behaves identically to chronological backtracking. Values of k between these two extremes produce behaviors that mediate between the goal-directed planning algorithms and chronological backtracking. In all sensor-based planning problem domains, Basic-VECA guarantees that edges are traversed at most $k + 2$ times each and thus a total travel distance of at most $(k+2)weight(G)$. Hence, the performance guarantee is $\Theta(weight(G))$, independent of the value of k. But there is a trade-off: The smaller the value of k, the better is the performance guarantee. A small value of k, however, also restricts the goal-directed planning algorithms earlier. This can force them to explore parts of the graph unnecessarily and increase the average-case performance.

Figure 5.10 shows a version of Basic-VECA that uses planning with the freespace assumption as the goal-directed planning algorithm. In the actual implementation, we use priority lists instead of exponentially decreasing edge costs. Basic-VECA can be used even if no graph is available initially. If Basic-VECA is supplied with a graph, it can use this information to improve its performance. For a more detailed description of VECA and empirical results, see Chapter 4.

Basic-VECA presented in Figure 5.10 guarantees the worst-case complexity of $\Theta(weight(G))$. For dense graphs with $|E| = \Omega(|V|)$ such a guarantee may not be satisfactory. There exists a way of improving the worst-case complexity of Basic-VECA for dense graphs, so that it can guarantee $O(|V|)$: one should keep track and consider costs only for edges of some spanning tree T of the explored portion of the problem domain. Step 3 of Basic-VECA would consider in this case only those paths leading from the current vertex to the goal vertex that can be split into two sub-paths, with the first path containing already visited vertices and edges of the spanning tree, and the second sub-path that goes exclusively through unvisited vertices. Figure 5.11 illustrates the spanning tree improvement of Basic-VECA. The agent is currently at vertex v_c. The shortest path from v_c to the goal goes through vertices x, w, y and z. However, according to the spanning tree modification, since vertex w has been already visited, Basic-VECA ignores two-edge sub-path v_c, x, w and considers a sub-path to w that is routed through already constructed tree, i.e. that goes to w through v_{start}. After reaching w Basic-VECA traverses (w, y), includes (w, y) in T, and then senses the status of vertex z in the attempt to traverse edge (y, z). If z is not blocked, Basic-VECA adds this edge to spanning tree T too, since it will be the first visit of vertex z. If Basic-VECA decides to go to the goal along the path v_c, v_{start}, w, y, z, for the sake of efficiency, the agent may actually move to w through

neighbors instead of assuming that it is unblocked.

Given a graph $G = (V, E)$, Basic-VECA uses the following variables for all $v, w \in V$ with $(v, w) \in E$: $count(v, w)$ keeps track of how often the edge has been traversed from v to w, $reserve(v, w)$ is the reserved VECA cost for its traversal from v to w, and $cost(v, w)$ is the actual VECA cost for its traversal from v to w.

Basic-VECA always makes the optimistic assumption that vertices are unblocked if it does not know their status. Consequently, we call a path traversable if it does not contain vertices that are known to be blocked.

1. Set $count(v, w) := cost(w, v) := 0$ for all $(v, w) \in E$. Set $i := 0$ and s to the starting vertex.
2. If s is the goal vertex, then stop and report success.
3. If no traversable path from s to the goal vertex exists, stop and report that the goal vertex cannot be reached. Otherwise, consider all traversable paths that start at s, end in an untraversed edge, and contain only traversed edges in between. (At least one such path exists. All these paths are guaranteed to contain only unblocked vertices.) Select a path with minimal actual VECA cost from these paths. Break ties by selecting the path with the smallest value, where the value of a path is the length of the shortest traversable path from s to the goal vertex that contains the path as prefix (at least one such path exists).
4. Repeat the following steps until the selected path has been traversed:
   (a) Let $(s, w) \in E$ be the next edge on the path.
   (b) Traverse the edge by moving to vertex w.
   (c) Set $count(s, w) := count(s, w) + 1$.
   (d) If $count(s, w) + count(w, s) = 1$, then set $i := i + 1$ and afterwards $reserve(s, w) := reserve(w, s) := 2^{-i}$.
   (e) If $count(s, w) + count(w, s) \geq k$ and $cost(s, w) = 0$, then set $cost(s, w) := cost(w, s) := reserve(s, w)$.
   (f) If $count(s, w) + count(w, s) > k$, then set $cost(s, w) := \infty$.
   (g) Set $s := w$.
5. Go to step 2.

Figure 5.10: Basic-VECA

vertex x.

Theorem 4.1 guarantees that Basic-VECA with the spanning tree modification has the worst-case complexity of $O(weight(T)) = O(|V|)$. Theorem 5.3 allows us to perform "short-cuts," i.e. to move robot actually along the shortest path to the last vertex of spanning tree T and then to the goal after the path has been chosen. In our example in Figure 5.11, it would correspond exactly to going to vertex x first, then - to w, and then - to y, instead of the estimated detour through v_{start}. The introduction of the spanning tree changes the process of reserving and assigning positive edge costs and thus effects the whole search process.

5.5 Summary

In this Chapter, we investigated goal-directed navigation in unknown environments. A common approach to solving this navigation problem is planning with the freespace assumption, where the robot always plans a shortest path to the goal, assuming that the terrain is clear unless it knows otherwise. It had been conjectured in the literature that this planning approach was

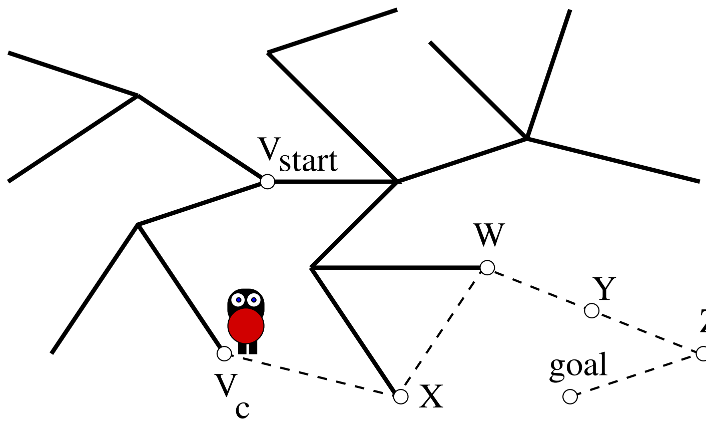

Figure 5.11: Improving Worst-Case Complexity through a Spanning Tree

optimal, given the lack of initial knowledge about the environment. Our results show that this is not the case: Its performance guarantee is not optimal. For situations in which its performance guarantee is not sufficient, we showed how Basic-VECA, a hybrid method based on techniques from AI and CS theory, can be applied to exhibit good empirical performance and provide performance guarantees that are optimal up to a constant factor.

Chapter 6

GenSAT as Agent-Centered Search

GenSAT is a family of local hill-climbing procedures for solving propositional satisfiability problems. In order to "re-utilize" knowledge accumulated about agent-centered search (see Chapters 3-4), we restate it as a navigational search process performed on an N-dimensional cube by a fictitious agent with limited lookahead. Several members of the GenSAT family have been introduced whose efficiency varies from the best in average for randomly generated problems to a complete failure on some realistic, structured problems, hence raising the interesting question of understanding the essence of their different performances. In this paper, we show how we use our navigational interpretation to investigate this issue. We introduce new algorithms that sharply focus on specific combinations of properties of efficient GenSAT variants, and which help to identify the relevance of the algorithm features to the efficiency of local search. In particular, we argue for the reasons of higher effectiveness of HSAT compared to the original GSAT. We also derive fast approximating procedures based on variable weights that can provide good switching points for a mixed search policy. Our conclusions are validated by empirical evidence obtained from the application of several GenSAT variants to random 3SAT problem instances and to simple navigational problems.

6.1 GenSAT

Recently an alphabetical mix of variants of GSAT [34, 64] has attracted a lot of attention from Artificial Intelligence (AI) researchers: TSAT, CSAT, DSAT, HSAT [27, 29], WSAT [66], WGSAT, UGSAT [18] just to name few. All these local hill-climbing procedures are members of the GenSAT family. Propositional satisfiability (SAT) is the fundamental problem of the class of NP-hard problems, which is believed not to admit solutions that are always polynomial on the size of the problems. Many practical AI problems have been directly encoded or reduced to SAT. GenSAT (see Table 6.1) is a family of hill-climbing procedures that are capable of finding satisfiable assignments for some large-scale problems that cannot be attacked by conventional resolution-based methods.

---

procedure GenSAT (Σ)
  for i:=1 **to** Max_Tries
    T:= $initial(\Sigma)$
    for j:=1 **to** Max_Flips
      if T satisfies Σ **then return** T
      else *poss-flips* := $hill\text{-}climb(\Sigma, T)$
          ; compute best local neighbors of T
        V := $pick($*poss-flips*$)$; pick a variable
        T := T with V's truth assignment inverted
    end
  end
return "no satisfying assignment found"

Table 6.1: The GenSAT Procedure.

GSAT [34, 64] is an instance of GenSAT in which *initial* (see Table 6.1) generates a random truth assignment, *hill-climb* returns all those variables whose flips[1] give the greatest increase in the number of satisfied clauses and *pick* chooses one of these variables at random [27]. Previous work on the behavior of GSAT and similar hill-climbing procedures [27] identified two distinct search phases and suggested possible improvements for GenSAT variants. HSAT is a specific variant of GenSAT, which uses a queue to control the selection of variables to flip[2]. Several research efforts has attempted to analyze the dominance of HSAT compared with the original GSAT for randomly generated problem instances. We have developed a navigational search framework that mimics the behavior of GenSAT. This navigational approach allows us to re-analyze the reasons of higher effectiveness of HSAT and other hill-climbing procedures by relating it to the number of equally good choices. This navigational approach also suggests strong approximating SAT procedures that can be applied efficiently to practical problems. An approximation approach can be applied to both "easy" and "hard" practical problems, in the former case it will likely to produce a satisfiable assignment, whereas in the latter case it will quickly find an approximate solution. For a standard testbed of randomly generated 3SAT problems, the transition phase between "easy" and "hard" problem instances corresponds to the ratio value of 4.3 between the number of clauses L to the number of variables N [49, 11].

[1]Flip is a change of the current value of a variable to the opposite value.

[2]See Section 6.3 for the definition of HSAT.

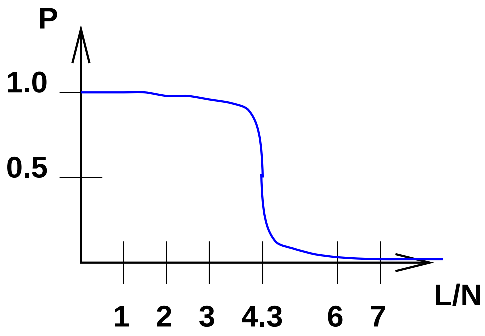

Figure 6.1: The transition phase for random 3SAT problems.

Figure 6.1 demonstrates the probability of generating a satisfying assignment for random 3SAT problems depending on the L/N-ratio.

An approximate solution can be utilized in problems with time-critical or dynamically changing domains. Interestingly, we found that it also provides a good starting point for a different search policy, i.e. serves as a switching point between distinct search policies within the same procedure. Such an approach can be utilized beneficially in multi-processor/multi-agent problem settings.

Our experiments with randomly generated 3SAT problem instances and realistic navigational problems confirmed the results of our analysis.

6.2 GenSAT as Agent-Centered Search

State spaces for boolean satisfiability problems can be represented as N-dimensional cubes, where N is the number of variables. We view GSAT and similar hill-climbing procedures as performing search on these high-dimensional cubes by moving a fictitious agent with limited lookahead. For efficiency reasons, the majority of GSAT-like procedures limit the lookahead of the agent to the neighbors of its current state, i.e., to those vertices of the cube that are one step far from the current vertex. An edge of the cube that links two neighboring vertices within the same face of the cube, corresponds to the flip of a variable. Thus, we reduced the behavior of GSAT to agent-centered search on a high-dimensional cube. Recall, in agent-centered search the search space is explored incrementally by an agent with limited lookahead. Throughout the paper we refer to this navigational version of GenSAT as to NavGSAT.

The worst-case complexity of both informed and uninformed agent-centered search is of the order of the number of vertices, i.e. $O(2^N)$. Moreover, unlike classical AI search where A* is an optimal informed algorithm for an arbitrary admissible heuristic, there are no optimal algorithms for agent-centered search problems[71]. Furthermore, as we have shown in Chapter 3, even a consistent, admissible heuristic can become misleading, and an efficient informed agent-centered search algorithm can demonstrate worse performance than the uninformed (zero heuristic) version of the same algorithm [39].

From the algorithmic point of view, the behavior of LRTA* [42], one of the most efficient agent-centered search methods, is close to NavGSAT's behavior. Both methods look for the most promising vertex among neighbors of the current vertex. In addition to selecting a neighbor with the best heuristic value, LRTA* also updates the heuristic value of the current vertex (see Table 3.3). The efficiency of LRTA* is known to depend on how closely the heuristic function represents the real distance [71]. The vast majority of GSAT-like procedures use the number of unsatisfied (or satisfied) clauses as the guiding heuristic. In general, this heuristic is neither monotone, nor admissible. However, for the most intricate random instances of SAT problems with $L = O(N)$, this heuristic is an $O(N)$ approximation of the real distance. ϵ-search has been introduced in [35] as a modification of LRTA* [42] that can utilize the guidance of non-admissible, non-monotone prior knowledge and still guarantee convergence to a solution that would be a certain approximate of the optimal solution. The convergence requirement is that given heuristic values are at most $1 + \epsilon$ times the goal distance for some $\epsilon \geq 0$, i.e.

$$h(v) \leq (1 + \epsilon)h^*(v) \quad for \;\; all \;\; v \in V$$

Table 6.2 presents the description of the ϵ-search modification of LRTA*. Thus, ϵ-search [35] applies to SAT problems.

---

procedure ϵ-search(V, E)
Initially, $F(v) := h(v)$ for all $v \in V$.
ϵ-search starts at vertex v_{start}:

1. $v :=$ the current vertex.
2. If $v \in Goal$, then STOP successfully.
3. $e := argmin_e F(neighbor(v, e))$.
4. $F(v) := max(F(v), 1 + F(neighbor(v, e)))$.
5. Traverse edge e, update $v := neighbor(v, e)$.
6. Go to 2.

Table 6.2: ϵ-Search Modification of LRTA*.

The description of ϵ-search may represent LRTA* as well, because the only difference between those two methods is step 4. Since LRTA* deals with monotone heuristics, for which

$h(v) \leq 1 + h(neighbor(v, e))$ holds for all vertices $v \in V$ and adjacent edges $e \in E$, the current heuristic value $F(v)$ is at most the estimate for any of its neighbors – $1 + F(neighbor(v, e))$).

Lemma 4 *After repeated problem-solving trials of a soluble propositional satisfiability problem with N variables and $O(N)$ clauses, the length of the solution of ϵ-search converges to $O(N^2)$.*

Proof: After repeated problem-solving trials the length of a solution of ϵ-search converges to the length of the optimal path multiplied by $(1 + \epsilon)$ [35]. On one hand, the length of the optimal path for a soluble propositional satisfiability problem is $O(N)$. On the other hand, for problems with $L = O(N)$ approximating factor ϵ is also $O(N)$. These two facts imply $O(N^2)$ complexity of the final solution after an unknown number of repeated trials. ■

Even though the length of a solution of ϵ-search converges to $O(N^2)$ for satisfiable problem instances, several initial trials can have exponential length. Thus, this approach can be applied only in special circumstances: One is provided possibly exponential memory and possibly exponential time for pre-processing to re-balance the heuristic values, after that the complexity of solving of the pre-processed problem is $O(N^2)$. Unfortunately, the effort spent on preprocessing and knowledge acquired about a particular problem instance seems not to be automatically transferred to other problem instances. Since such an "exponential" pre-processing scenario is not always what AI researchers keep in mind when applying GenSAT, we do not consider ϵ-search as a general navigational equivalent of GenSAT. However, in Section 6.3 we show that one (first) run of ϵ-search coincides completely with the run of HSAT for the majority of soluble SAT problem instances.

The approach of possibly "exponential" pre-processing is not completely hopeless, one can try to reduce the problem by considering smaller number of variables (projection on the correspondent face of the cube), acquire knowledge about the projected problem, and then to expand this knowledge (lifting from the face on the whole cube) to the original problem instance. This can be a promising direction for a series of problems that share common sub-structures.

Thus, the question of the efficiency of GSAT and similar procedures is reduced to the domain-heuristics relations that guide agent-centered search on an N-dimensional cube. Recent works on changing the usual static heuristic – the number of unsatisfied (satisfied) clauses – to the dynamic weighted sums [18] produced another promising sub-family of GenSAT procedures. Our experiments showed that the "quality" of the usual heuristic varies greatly in different regions of the N-dimensional cube, and as the ratio of L to N grows, this heuristic becomes misleading in some regions of the problem's domain. These experiments identified the need to introduce novel heuristics and better analysis of the existing ones.

6.3 New Corners or Branching Factor?

We conducted a series of experiments with the ϵ-search version of LRTA* and the number of unsatisfied clauses as the heuristic values for each vertex (corner) of the N-dimensional cube. We found that the combination of a highly connected N-dimensional cube and such prior knowledge forces an agent to avoid vertices with updated (increased in step 4 of Table 6.2) heuristic values. Exactly the same effect has been achieved by HSAT, a variant of GenSAT, for randomly generated 3SAT problems with a low ratio of the number of clauses to the number of variables. In HSAT flipped variables form a queue, and this queue is used in $pick$ (see Table 6.1) to break ties in favor of variables flipped earlier until the satisfying assignment is found or the amount of flips has reached the pre-set limit of Max_Flips. Thus, we consider ϵ-search as a navigational analogue of HSAT for soluble problem instances.

Previous research identified two phases of GenSAT procedures: steady hill-climbing and plateau phases [27]. During the plateau phase these procedures perform series of sideway flips keeping the number of satisfied clauses on the same level. The reduction of the number of such flips, i.e. cutting down the length of the plateau, has been identified as the main concern of GenSAT procedures. Due to high connectivity of the problem domain and the abundance of equally good choices during the plateau phase, neither HSAT nor ϵ-search re-visit already explored vertices (corners) of the cube for large-scale problems. This property of HSAT has been stated as the reason of its performance advantage for randomly generated problems in comparison with GSAT [29].

To re-evaluate the importance of visiting new corners of the N-dimensional cube, we introduced another hill-climbing procedure that differs from GSAT only in keeping track of all visited vertices and Never Re-visiting them again, NRGSAT. On all randomly generated 3SAT problems, NRGSAT's performance in terms of flips was identical to GSAT's one. Practically, NRGSAT ran much slower, because it needs to maintain a list of visited vertices and check it before every flip. Based on this experiment, we were able to conclude that exploring new corners of the cube is not that important. This increased our interest in studying further reasons for the performance advantage of HSAT over GSAT.

We focused our attention on *poss-flips* – the number of equally good flips between which GSAT randomly picks [28], or, alternatively, the branching factor of GSAT search during the plateau phase. We noticed that on earlier stages of the plateau phase both GSAT and NRGSAT tend to increase *poss-flips*, whereas HSAT randomly oscillates *poss-flips* around a certain (lower) level. To confirm the importance of *poss-flips*, we introduced *variable weights*[3] as a second heuristic to break ties during the plateau phase of NavGSAT. NavGSAT monitors the number of flips performed for each variable and among all equally good flips in terms of the number of unsatisfied clauses, NavGSAT picks a variable that has been flipped the least number of times. In case of second-order ties, they can be broken either randomly, fair – NavRGSAT, or deterministically, unfair, according to a fixed order – NavFGSAT.

Both NavRGSAT and NavFGSAT allow to flip back the just flipped variable. Moreover,

[3]Weight of each variable is the number of times this variable has been flipped from the beginning of the search procedure. Each flip of a variable increases its weight by one.

Problem	Algorithm	Mean	Median	St.Dev.
100	GSAT	12,869	5326	9515
vars,	HSAT	2631	1273	1175
430	NavFGSAT	3558	2021	1183
clauses	NavRGSAT	3077	1743	1219
1000	GSAT	4569	2847	1863
vars,	HSAT	1602	1387	334
3000	NavFGSAT	1475	1219	619
clauses	NavRGSAT	1649	1362	675
1000	GSAT	7562	4026	3515
vars,	HSAT	3750	2573	1042
3650	NavFGSAT	3928	2908	1183
clauses	NavRGSAT	4103	3061	1376

Table 6.3: Comparison of number of flips for GSAT, HSAT, NavRGSAT and NavFGSAT.

the latter procedure often forces to do so due to the fixed order of variables. However, the performance of both NavRGSAT and NavFGSAT is very close to HSAT's performance. Table 6.3 presents median, mean and standard deviation of GSAT, NRGSAT, HSAT, NavRGSAT and NavFGSAT for randomly generated 3SAT problems with 100 and 1000 variables and different ratios L to N. We investigated problems of this big size, because they represents the threshold between satisfiability problems that accept solutions by conventional resolution methods, for example Davis-Putnam procedure, and ones that can be solved by GenSAT hill-climbing procedures.

In the beginning of the plateau phase both NavGSAT methods behave similarly to HSAT: Variables flipped earlier are considered last when NavGSAT is looking for the next variable to flip. As more variables gain weight, NavGSAT methods' behavior deviates from HSAT. Both methods can be perceived as an approximation of HSAT.

We identified that a larger number of *poss-flips* is the main reason why GSAT loses to HSAT and NavGSAT on earlier stages of the plateau phase. As the number of unsatisfied clauses degrades, there are less choices for equally good flips for GSAT, and the increase of *poss-flips* is less visible. However, during earlier sideway flips GSAT picks equally good variables randomly, this type of selection leads to the vertices of the cube with bigger *poss-flips*, where GSAT tends to be "cornered" for a while. Figure 6.2 presents average amounts of *poss-flips* with the 95%-confidence intervals. The *poss-flips* were summed up for each out of four hill-climbing procedures for every step in the beginning of the plateau phases (from $0.25N$ to N) for a range of problem sizes. Since the number of variables and the interval of measuring grow linearly on N, we present sums of *poss-flips* scaled down by N^2. As it follows from Figure 6.2, the

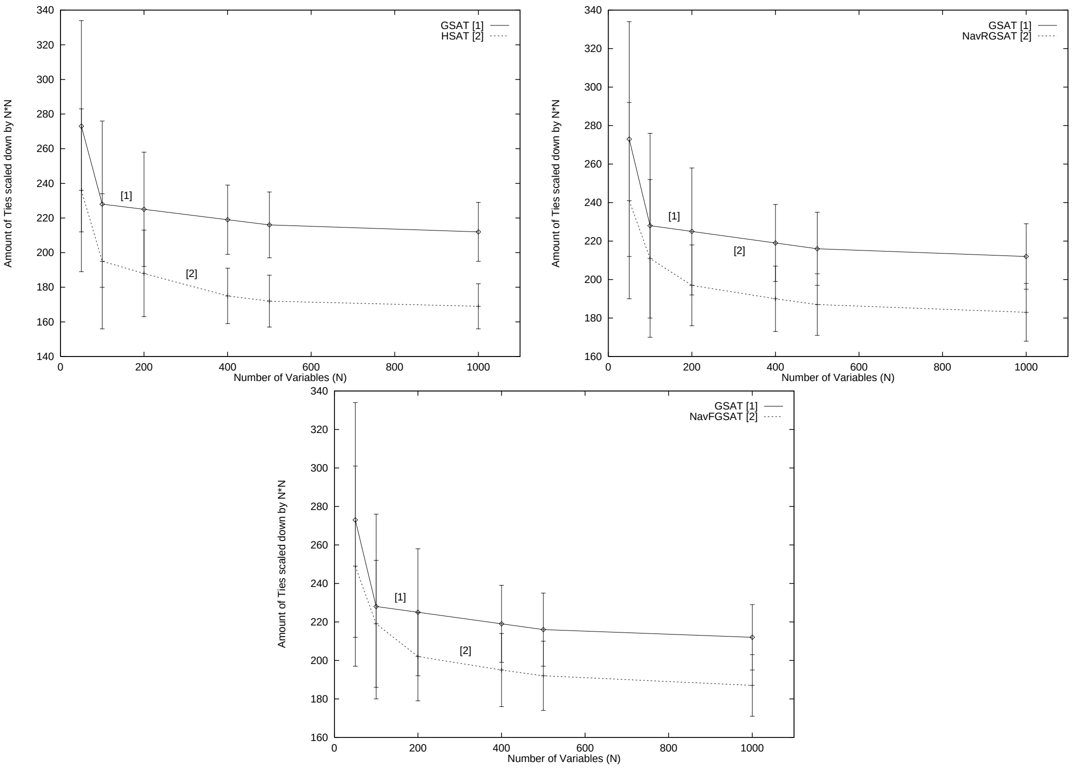

Figure 6.2: Comparison of *Poss-Flips* for GSAT, HSAT, NavRGSAT and NavFGSAT.

original GSAT consistently outnumbers all other three procedures during that phase, although its confidence intervals overlap with NavRGSAT and NavFGSAT's confidence intervals.

Figure 6.3 presents the dynamics of *poss-flips* during a typical run of GSAT. It is easy to see that on early plateaux *poss-flips* tend to grow with some random noise, for example, in Figure 6.3 second, third and fifth plateaux produced obvious growth of *poss-flips* until drops corresponding to the improvement of the heuristic values and, thus, the end of the plateau. During the first and fourth plateaux, the growth is not steady though still visible. Even though flips back are prohibited for NRGSAT, it maintains the same property, because of the high connectivity of the problem domain and the abundance of equally good choices.

Figure 6.4 represents the average percentage of ties for a 3SAT problem with 100 variables and 430 clauses over 100 runs for GSAT and HSAT, and for GSAT and NavRGSAT. The average number of *poss-flips* for GSAT dominates the analogous characteristic for HSAT by a noticeable amount. This type of dominance is similar in the comparison of GSAT with NavRGSAT in the beginning of the plateau phase. In the second part of the plateau phase the number of *poss-flips* for HSAT or NavRGSAT approaches the number of *poss-flips* for GSAT. Lower graph represents second-order ties for NavRGSAT that form a subset of *poss-flips*.

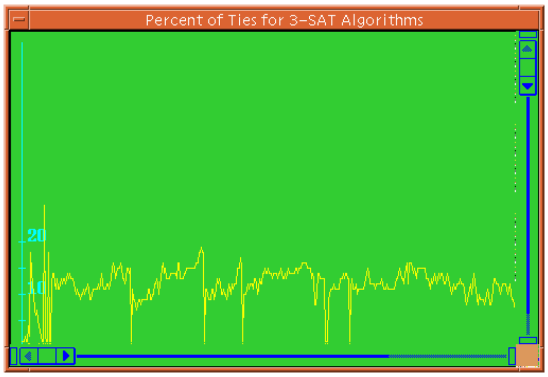

Figure 6.3: Dynamics of *Poss-Flips* for GSAT.

Our experiments confirmed the result obtained in [27] that the whole picture scales up almost linearly in the number of variables and the number of *poss-flips*, although we noticed a tendency of earlier beginning of the plateau phase as the number of variables grew from several hundred to several thousand. In our experiments, the plateau phase began after about $0.2N - 0.25N$ steps. By that moment at most a quarter of the variable set has been flipped, and thus NavFGSAT mimics HSAT up to a certain degree. After $2N$ or $3N$ flips, both versions of NavGSAT diverge significantly from HSAT. After these many steps both NavRGSAT and NavFGSAT still maintain random oscillation of *poss-flips*, whereas GSAT tends to promote higher levels of *poss-flips*. Unfortunately, for problems with larger ratio of the number of clauses to the number of variables NavFGSAT is often trapped in an infinite loop. NavRGSAT also may behave inefficiently for such problems: From time to time the policy of NavRGSAT forces it to flip the same variable with a low weight several times in a row to gain the same weight as other variables from the set of *poss-flips*.

Thus, NavGSAT showed that the number of *poss-flips* plays an important role in improving the efficiency of GenSAT procedures. HSAT capitalizes on this property and therefore constitutes one of the most efficient hill-climbing procedures for random problem instances. However, many real-world satisfiability problems are highly structured and, if applied, HSAT may easily fail due to its queuing policy. NavGSAT suggests another sub-family of GenSAT hill-climbing procedures that does not tend to increase the number of *poss-flips*. Weights of variables and their combinations can be used as a second tie-breaking heuristic to maintain lower level of *poss-flips* and find exact or deliver approximate solutions for those problems for which HSAT fails to solve.

For randomly generated 3SAT problems HSAT proved to be one of the most efficient hill-climbing procedures. There have been reports on HSAT's failures in solving non-random propositional satisfiability problems [29]. We view the non-flexibility of HSAT's queue heuristic as a possible obstacle in solving over-constrained problems. This does not happen in solving

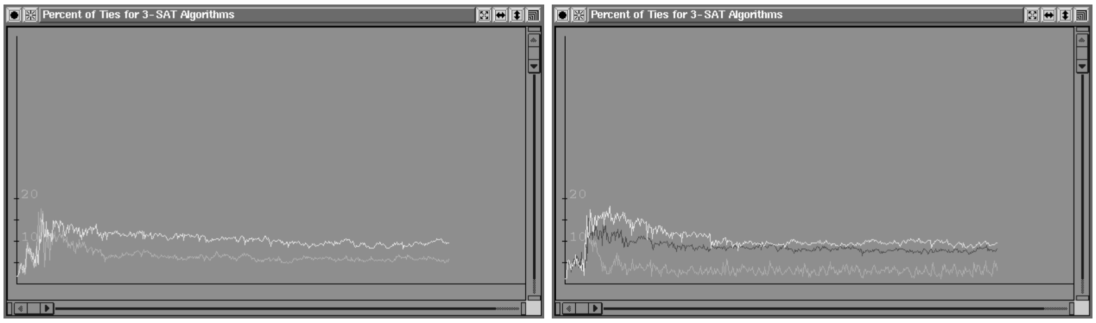

Figure 6.4: Percentage of *Poss-Flips* for GSAT with HSAT and GSAT with NavRGSAT.

random 3SAT problems with low L/N-ratio.

6.4 Approximate Satisfaction

While running experiments with GSAT, HSAT and other hill-climbing procedures, we noticed that GSAT experiences biggest loss in the performance in the beginning of the plateau phase where the amount of *poss-flips* can be as high as 20-25%. On the other hand, HSAT, NavFGSAT and NavRGSAT behave equally good during the hill-climbing phase and the beginning of the plateau phase. We thus concluded that any of the latter three procedures can be applied to provide fast approximate solutions. For some problems, versions of NavGSAT are not as efficient as HSAT. Nonetheless, we introduced NavFGSAT and NavRGSAT to show that HSAT's queuing policy is not the unique way of improving the efficiency of solving propositional satisfiability problems.

Approximate solutions can be utilized in time-critical problems where the quality of the solution discounts the time spent for solving the problem. NavGSAT can be also applied to problems with dynamically changing domains, when the domain changes can influence the decision making process. Finally, approximate solution provide an excellent starting point for a different search policy. For example, WGSAT and UGSAT [18] utilized a promising idea of the instant heuristic update based on the weight of unsatisfied clauses. An approximate solution provided by HSAT or NavGSAT constitutes an excellent starting point for WGSAT, UGSAT or another effective search procedure of a satisfiable solution, for example, ϵ-search (with heuristic updates). Among others we outline the following benefits of employing HSAT or NavGSAT to deliver a good starting point for another search method:

- Perfect initial assignment with a low number of unsatisfied clauses.
- Absence of hill-climbing phase that, for example, eliminates noise in tracking clause weights.

- Efficient search in both steps of policy-switching approach.
- Convenient point in time to fork search in multi-agent/multi-processor problem scenarios.

Although HSAT, itself, is an efficient hill-climbing procedure for randomly generated problems with a low clause-variable ratio, we expect that HSAT might experience difficulties in more constrained problems. NavGSAT provides another heuristic that guides efficiently in the initial phases. On the other hand, the hill-climbing phase may either produce noise in the clause weight bookkeeping or a redundant list of vertices with updated heuristics that slows down the performance of ϵ-search. Search with policy switching can benefit significantly from employing efficient procedures in all of its phases.

6.5 Navigational Planning Problems

To confirm the results of our navigational approach to GSAT, we applied all the discussed above hill-climbing procedures to the following simple navigational problem:

> **Navigational Problem (NavP):** An agent is given a task to find the shortest path that reaches a goal vertex from a starting vertex in an "obstacle-free" rectangular grid.

NavP is a simplistic planning problem. It can be represented as a propositional satisfiability problem with $N = |S| * D$ variables, where S is the set of vertices in the rectangular grid and D is the shortest distance between starting vertex X and goal vertex G. In a correct solution, a variable x_s^d is assigned *True* ($x_s^d = 1$), if s is dth vertex on the shortest path from X to G, and *False* otherwise. There can be only one variable with the *True* value among variables representing grid vertices that are d-far from starting vertex X. This requirement implies $L_1 = \Theta(|S|^2 * D)$ pigeonhole-like constraints:

$$\bigwedge_{d=1}^{D-1} \bigwedge_{s_1 \neq s_2} (\neg V_{s_1}^d \vee \neg V_{s_2}^d)$$

Already these constraints make the domain look "over-constrained," since the ratio of L_1 to N is not asymptotically bounded. Another group of constraints has to force *True*-valued variables to form a continuous path. There can be different ways of presenting such constraints, we chose the easiest and the most natural presentation that does not produce extra variables:

$$\bigwedge_{d=2}^{D-1} \bigvee_{s \in S} (V_s^d \wedge (V_{s_1}^{d-1} \vee V_{s_2}^{d-1} \vee V_{s_3}^{d-1} \vee V_{s_4}^{d-1}))$$

Vertices $s_1, s_2, s_3, s_4 \in S$ are the neighbors of vertex $s \in S$ in the rectangular grid. To reduce the number of variables and clauses, the initial and the goal states are represented by stand-alone single clauses:

$$(V_{s_5}^1 \vee V_{s_6}^1 \vee V_{s_7}^1 \vee V_{s_8}^1)$$

$$(V_{s_9}^{D-1} \vee V_{s_{10}}^{D-1} \vee V_{s_{11}}^{D-1} \vee V_{s_{12}}^{D-1})$$

Vertices $s_5, s_6, s_7, s_8 \in S$ are the neighbors of the starting vertex, $s_9, s_{10}, s_{11}, s_{12} \in S$ are the neighbors of the goal vertex.

Second group of constraints is not presented in the classical CNF form. It is possible to reduce it to 3SAT, but such a reduction will introduce a lot of new variables and clauses and will significantly slow down the performance without facilitating search for a satisfiable assignment. From the point of view of hill-climbing procedures that track clause weights, this would mean only a different initial weight assignment and a linear change in counting clause weights. Therefore, we decided to stay with the original non-3SAT model and considered each complex conjunction $\bigvee_{s \in S}(V_s^d \wedge (V_{s_1}^{d-1} \vee V_{s_2}^{d-1} \vee V_{s_3}^{d-1} \vee V_{s_4}^{d-1}))$ as a single clause. Together with the starting and goal vertex constraints, the second group contains $L_2 = \Theta(D)$ constraints that force *True*-valued variables to form a continuous path.

It is fairly easy to come up with an initial solution, so that all but one constraint are satisfied. Figure 6.5 shows one of such solutions that alternates between the goal vertex and one of its neighbors, and the final path that satisfies all the constraints. The original GSAT has the complexity that is exponential on D. It performs poorly for such domains, because at every step it has more equally good chances than any other algorithm. HSAT was able to solve "toy" problems with less than 200 variables until its search was under the influence of initial states. For larger problems, after an initial search HSAT used to switch to a systematic search that avoided changing recently changed vertices. Since HSAT re-started search from both starting and goal vertices on a regular basis, all the variable corresponding to their neighboring vertices has frequently changed their values. Therefore, paths from the opposite direction attempted to avoid changing these variables again. This was one of the domain where the queuing policy of HSAT played against it.

A slightly modified versions of NavFGSAT and NavRGSAT were capable of solving larger problems using Top-Down Depth-First-Search (TDDFS). TDDFS traverses repeatedly the search tree (the set of vertices reachable in D steps) from the root down, each time attempting to visit the least visited vertex from the current vertex or, if possible, unvisited vertex. The only modification of this behavior was that NavGSAT methods alternated roots between the starting vertex and the goal vertex while performing such search.

Figure 6.5: Initial and final solutions for NavP.

6.6 GenSAT Conclusions

We showed that GenSAT hill-climbing procedures for solving propositional satisfiability problems can be interpreted as navigational, agent-centered search on a high-dimensional cube, NavGSAT. This type of search heavily depends on how well heuristic values represent the actual distance towards the set of goal states. HSAT, one of the most efficient GSAT-like procedures, maintains low level of *poss-flips*. We identified this property as the main benefit of HSAT in comparison with the original GSAT. However, the non-flexibility of HSAT's queuing policy can be an obstacle in solving more constrained problems. We introduced two versions of NavGSAT that also maintain low level of *poss-flips* and can be applied as approximating procedures for time-critical or dynamically changing problems, or serve as a starting phase in search procedures with switching search policies.

Chapter 7

Further Insights into On-Line Search Complexity

In this chapter we discuss a few issues of on-line search that follow directly from the thesis work. A full investigation of these issues constitutes an interesting direction for future work. In particular, in this chapter we argue how some of the domain features may influence the complexity of on-line search. In a certain sense this discussion is also relevant to a much wider spectrum of problems. For example, as we noticed in Chapter 2, precise knowledge of the tight upper bound does not always provide an enlightening hint on constructing an optimal solution for combinatorial optimization problems. Similarly, if one is informed about the existence of a plan of a certain length, for some planning problems it is easy to come up with a feasible plan of this length, whereas for other problems this type of information would not help much.

Thus, we would like to estimate the complexity of building feasible plans in deterministic planning problems too, as some incremental forward or backward-chaining planners that repair plans by backtracking and branching, can be perceived as search controlling mechanisms that move a fictitious agent through a set of feasible plans.

What makes some domains harder to search than others? What are the relevant domain features that make search so hard? Are there any relations between identifying the complexity of search and other methods from different scientific areas? Of course, we are not going to provide full answers to all of these questions. In this chapter we attempt to answer only some of the above question, especially those relevant to on-line search and constructing optimal solutions for combinatorial optimization problems.

There exist an additional interest to this topic, as there have been already several successful efforts in solving classical situated off-line search problems by on-line agent-centered search methods. Recall that in agent-centered search agent's lookahead is limited by a neighborhood of its current location. Among others, such problems include 5x5-puzzle solved by Korf's LRTA* [43] and 3SAT satisfiability problems, with the success in solving which attributed recently to the family of hill-climbing procedures (see Section 6). These “stories of success” have much in common:

- Agent-centered search achieves a reasonable balance between exploration (considering valid plans) and exploitation (selecting and executing sub-plans, acquiring knowledge, and resolving uncertainties).

- Problem domains are well-connected, the cost of recovery from making a wrong decision is low.

Given an admissible, monotone heuristic and unlimited resources, A* search algorithm cannot be outperform by any other algorithm [57]. And, indeed, agent-centered search methods appear to be less efficient than search-in-memory for many modestly sized problems. The benefit of A* algorithm is that it “teleports” its fictitious agent from one location to another for free.

As we emphasized before, unlike the classical approach of situated search-in-memory, in agent-centered search the sequence of executed actions induce a continuous path on the problem domain. Such a difference impose a necessary change of the goal function. Usually the length of this path is viewed as the prime efficiency characteristic, since often the deliberation time is negligible in comparison with time needed to move the agent along the path.

All successful agent-centered search replacements of situated search-in-memory seem to share two common features: Problem domains have relatively small diameters, but are too big for attacking by A*. Therefore, in our research we included the diameter of the problem domain in the list of relevant features. It appears that the diameter, indeed, has an impact on the complexity of on-line search in reversible domains.

7.1 The Oblongness Factor

In off-line search “teleporting” is allowed for a fictitious agent. Problems of this kind accept a greedy approach in exploring promising locations. Exploiting this strategy A* algorithm guarantees at most the same number of explored states in the worst case as any other search algorithm for the same domain and the same admissible heuristic function [57]. In this sense, search-in-memory is a homogeneous search environment with a domain-independent search tool. Furthermore, given a set of admissible heuristic functions, one achieves better efficiency by selecting majorizing heuristic values.

In Chapter 3 we showed that heuristic values provide a slightly different type of guidance for agent-centered search. For example, for some problems an agent would achieve better efficiency with zero heuristic values than with a monotone, admissible heuristic that may be more misleading. Thus, the principle of selecting majorizing heuristics does not extend to agent-centered search.

We would like to find a simple way of estimating the complexity of on-line search problems that would tell us about the chances for agent-centered search to succeed in solving these problems. The empirical evidence obtained from the experiments for the goal-directed exploration problem suggested that the size of the problem domain, its diameter and the distance from the starting state to the goal can affect the efficiency of search. Moreover, we found that for the goal-directed problems with the goal distance from the starting state being of the order of the diameter of the problem domain, the following feature that we call *oblongness*, seems to capture the complexity of agent-centered search. The choice of this feature was not accidental. There have been plenty of on-line search techniques of completely different natures, whose worst-case complexity on undirected graphs is proportional to the product of the size of the graph (number of vertices) and the diameter of the graph. The list of such algorithms include Smell-Oriented Search [80], discounted and undiscounted zero-initialized Q-learning [37] and edge counting [38], in next section we consider random walks, the expected complexity of visiting all vertices for which approximates the same product. Since both the diameter and the size of the graph grow linearly on the number of vertices, to make the complexity parameter scalable, we introduce the following feature:

Definition 1 **Oblongness** *is the ratio of the diameter of an undirected graph to the number of vertices.*

Figure 7.1 shows the dependency of the performance of efficient agent-centered search methods, such as AC-A*, on the value of the *oblongness*. In a certain sense this picture is the inverse to one shown in Figure 3.2 from Chapter 3, where we were gradually changing the density of the domain by adding undirected edges to a maze that initially was a tree. The diameter of the domain was shrinking along with the growing density – from being a constant fraction of the whole domain to becoming a square root of the number of vertices for the complete rectangular maze. In Figure 7.1 we go even further and continue adding edges until the domain becomes a complete graph.

We confirm the relevance of the *oblongness* factor to the complexity of on-line search by considering a series of AI domains that have been studied in the literature. Our analysis of reversible AI domains identified three major groups of problems with drastically different *oblongness*. To make them comparable, we consider the following problems either with $\Theta(N!)$ the size of the problem domain or adjust them to this size for scalable domains.

1. BlocksWorld, Pancake and Burnt Pancake problems, Signed and Unsigned problems of Genetic Mutations by Inversion. In the Pancake problem the task is to sort an arbitrarily shuffled pile of pancakes of different sizes with a help of a spatula that can reverse the

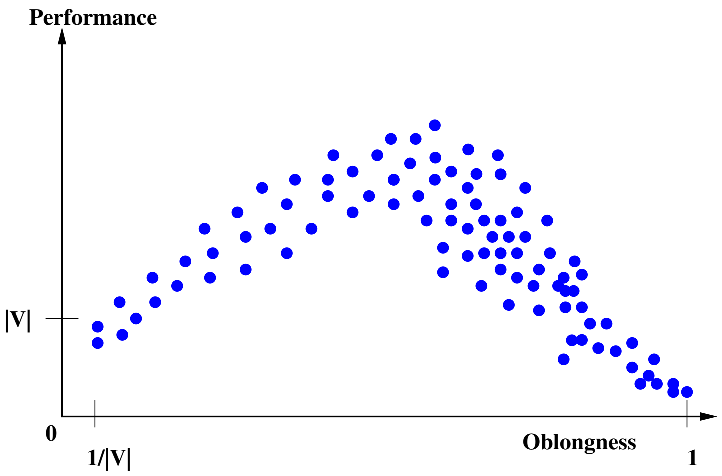

Figure 7.1: The Efficiency of Agent-Centered Search and Oblongness

sequence of pancakes from the top to the place of insertion. In the Burnt Pancake problem one should also preserve the proper orientation of all pancakes in the final sorted sequence (for example, burnt side down). In the Genetic Mutation problems one can create new mutations by extracting an arbitrary subsequence and reversing its order to transform the mouse's gene into the elephant's gene. In the signed version of the problem, both genes have all the proteins specifically oriented in the same sense as in the Burnt Pancake problem.

2. $\sqrt{N} \times \sqrt{N}$-puzzle, SAT. The size of the SAT problem domain is 2^N, to make it $\Theta(N!)$, one should scale up the diameter of the problem by $\Theta(\log N)$.

3. Planar mazes, Navigational problems.

First group of problems has domains with the diameters of size $\Theta(N)$. The diameter of the $\sqrt{N} \times \sqrt{N}$-puzzle is both $\Omega(N)$ and $O(poly(N))$, the same is true for SAT. Third group has the biggest variety of *oblongness* values ranging from $1/|V|$ to 1. Thus, first and second group of problems are located on the same side of the *oblongness* spectrum. Figure 7.2 shows the location of the problems relatively the range of the *oblongness* values of their domains.

7.2 Complexity of On-Line Search

Some search domains are so easy that arbitrary prior knowledge cannot mislead the search process completely. In such domains even Random Walk would easily find a goal. And this is not a coincidence, as we relate the "hardness" of search domains to the complexity of Random

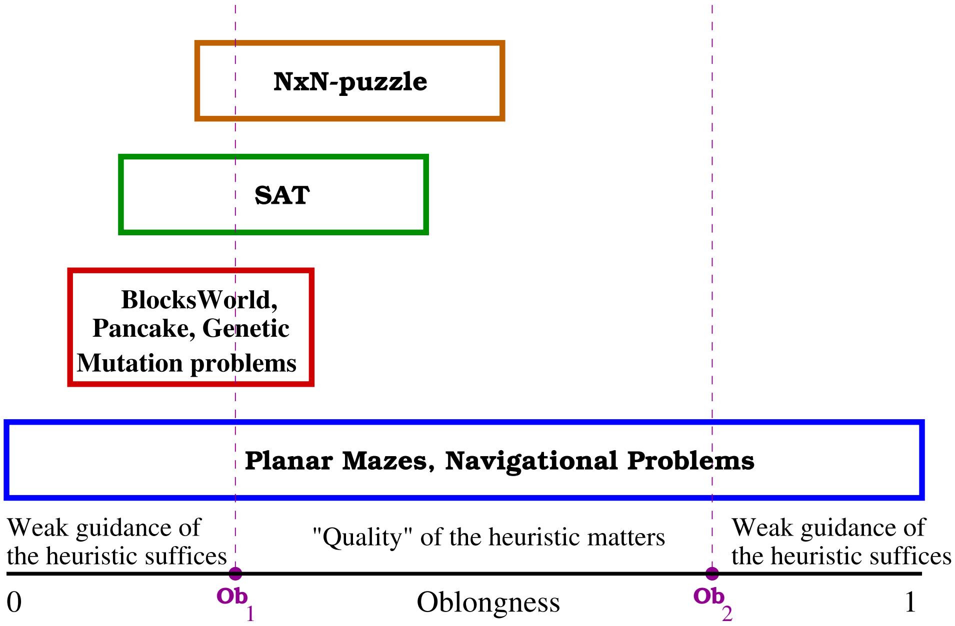

Figure 7.2: Problems and their *Oblongness* Ranges.

Walk on these domains. Throughout this thesis we consider reversible domains, where each action has an opposite (anti)-action. For the sake of simplicity, in this chapter we limit ourselves to undirected graphs that represent problem domains.

Figure 7.3 demonstrate two extreme undirected graphs that are easy to search even without prior knowledge - almost a complete graph and a caterpillar with a long stem and short branches. Easiness of search in such domains comes from low cost of recovering from errors, in the former case an agent is always close to the goal, in the latter case short branches allow the agent to recover and come back to the stem in a relatively small number of steps. This is not accidental that these two extreme cases are coupled together. For a Random Walk that chooses at random any edge emanating from the current location, it takes $O(|V|^2)$ in average to reach the goal from any starting vertex on these domains [52].

However, the expected time of reaching vertex w from vertex v in the *lollipop graph* presented in Figure 7.4 is $\Theta(|V|^3)$ [52]. It shows that *lollipop graphs* and graphs with build-in *lollipops* might be hard domain instances for on-line search. The value of the *oblongness* factor for for the caterpillar is close to one, for the clique - $1/|V|$, for the *lollipop* graph - $1/2$. These domains also confirm the correctness of our *oblongness* hypothesis.

If one skips the heuristic update step in LRTA* algorithm (step 4 in Table 3.3) and heuristic values are the same for all vertices (edges), such “circumsized” LRTA* would coincide completely with Random Walk. Moreover, since in the beginning of on-line search the prob-

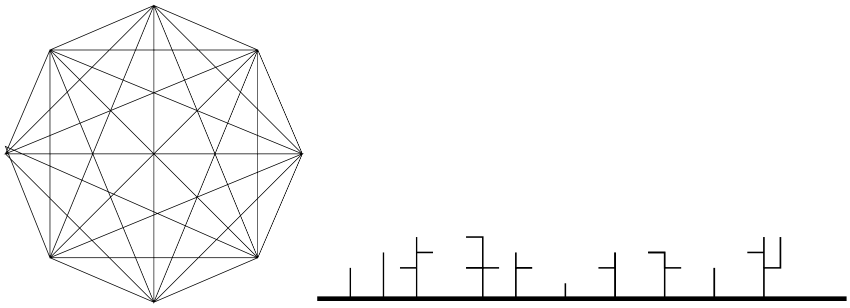

Figure 7.3: Undirected Graphs that Are Easy for Search.

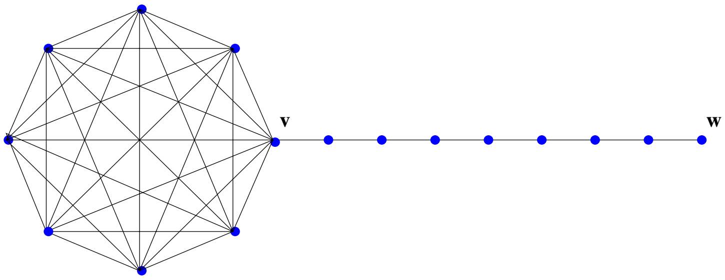

Figure 7.4: A Lollipop Graph

lem domain is not known, first agent's actions have to imitate Random Walk due to the limit of agent's knowledge about the domain, even according to more sophisticated on-line search algorithms. However, efficient agent-centered search methods use memory to keep already acquired knowledge about the domain, either completely or partially. The presence of such knowledge forces the flow of on-line search to deviate from Random Walk.

However, since in its initial phase, on-line search is similar to Random Walk, it can be useful to overview a related work on Random Walk in undirected graphs. In their work on probabilistic algorithms [52], authors drew an analogy between a Random Walk on the graph and the resistance of an electric scheme, where each unit edge is replaced by a unit resistor. This is a nice example of cross-fertilization between the Probability and Electrical Network Theories. We introduced the *oblongness* parameter in Section 7.1. In the nest section, we overview the theory of Random Walk on undirected graphs to use it as an approximation of the agent-centered search procedures, to justify the correctness of the chosen feature and of the identified relation.

Since the variety of algorithmic strategies amenable to on-line search does not allow to put all domain features under a common denominator, i.e. to relate exactly the complexity of search with the shape of the domain for every possible on-line algorithm, we limit ourselves to a more modest problem of the identification of relevant features and stating recommendation on choosing efficient search algorithms.

7.2.1 Resistive Networks

If every undirected edge of the graph is replaced by a unit resistor (see two examples in Figure 7.5), then such an electrical scheme obeys Kirchhoff's and Ohm's Laws. Kirchhoff's Law interprets the electric current as an electric flow by stating that the sum of the currents entering a node in the network equals the sum of emanating currents. Ohm's Law regulates voltages, resistances and currents by stating that the voltage across a resistance equals to the product of the resistance and the current through it.

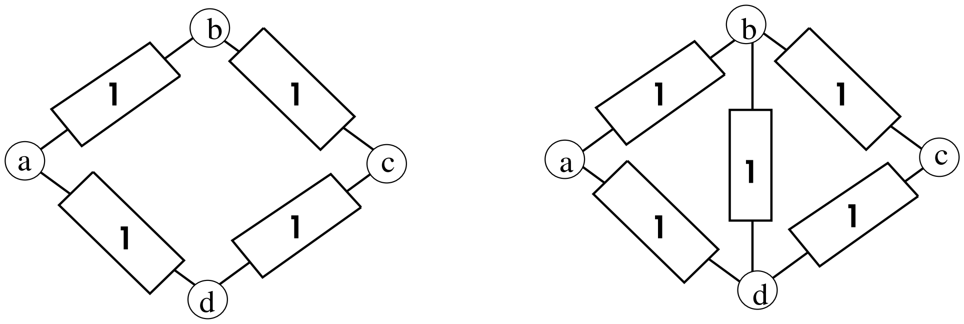

Figure 7.5: Examples of Resistive Networks

If a current of one amper is injected into node a in the left network example in Figure 7.5 and removed from node c in this network, according to Kirchhoff's and Ohm's Laws: Half of the ampere of the current flows through the path abc, and the other half ampere through adc [52]. Interestingly, if we add an edge between nodes b and d, as shown in the right network of Figure 7.5, there is no current along branch bd due to the symmetry of the network and equal voltage levels at nodes b and d.

> **Effective Resistance:** The *effective resistance* R_{ab} between any two nodes a and b in a resistive network is the voltage difference between them when one amper is injected into a and removed from b.
>
> **Network Resistance:** The *network resistance* $R(G)$ for an arbitrary undirected graph is the maximum *effective resistance* for all pairs of nodes: $R(G) = max_{a,b \in V} R_{ab}$

As the networks in Figure 7.5 show, an additional edge does not always reduce the resistance of the network, although it never increase it. The opposite is also true: Removing an edge never increases the resistance, although it may remain the same after the removal.

The introduced "electrical" parameters can be utilized in establishing bounds for Random Walk. If we define the *commute time* C_{ab} between two nodes a and b as the expected time for Random Walk starting at a to come back to a after visiting b at least once, the following theorem ties the value of the *commute time* with the *effective resistance* [52]:

Theorem 7.1 *For any two vertices a and b, the* commute time $C_{ab} = 2|E|R_{ab}$.

In on-line search an agent is supposed to reach the goal state. Since Random Walk is not goal-directed (it only approximates on-line search on earlier search stages), we might be interested in estimating the *cover time* $C(G)$ – the expected time for Random Walk to visit all nodes of graph G including the goal node(s) [52]:

Theorem 7.2 $|E|R(G) \leq C(G) \leq 2e^3|E|R(G)\ln|V| + |V|$.

The diameter of the graph d determines the upper bound for the resistance, because the *effective resistance* R_{ab} between nodes a and b is at most the distance between them $dist_G(a, b)$. Hence, for any pair of nodes a and b, the *effective resistance* R_{ab} is at most d. Consequently, the network resistance of the whole graph $R(G)$ is at most d. The *cover time* $C(G)$ for such a graph is thus $O(|E|d\ln|V|)$.

7.2.2 Simplifying the Estimates

In Section 7.2.1 we showed that the expected time for Random Walk to visit all vertices of the graph is $O(|E|d\ln|V|)$, where d is the diameter of the graph. The resistance of the network was used to derive this upper bound. Since the actual resistance is often much better than the diameter of the graph and this bound is asymptotically tight for dense graphs, for which the network resistance is much lower than the diameter of the graph, for our purposes we will omit the $\ln|V|$-factor.

We would like to simplify the situation even further. The *cover time* for Random Walk has been shown to belong to the interval $[|E|R(G), 2e^3|E|R(G)\ln|V| + |V|]$. For a fixed number of vertices $|V|$, bigger number of edges $|E|$ implies lower expected *resistance*. This is may not be true for particular graph instances, but in average the *resistance* of graphs with bigger number of edges is lower than that of graphs with the same number of vertices and smaller number of edges. Since the actual *resistance* $R(G)$ varies for different graphs and is a computationally-consuming parameter even for moderately sized graphs, for practical purposes we introduced another feature – *oblongness* – that seem to capture the complexity of on-line search.

The motivation of this feature is the following: Both the number of edges $|E|$ and the resistance of the network $R(G)$ grow linearly on the number of vertices $|V|$ for the same type of graphs. Therefore, to make an estimate scalable over domains of different size, we should divide it by $|V|^2$. Furthermore, for the simplicity reasons, we substitute the product $|E|R(G)$ by $|V|d$, where d is the diameter of the graph. The argument in favor of this substitution is that more edges reduce both the *resistance* of the network and the diameter of the graph. The presence of memory and the memorization process during on-line search improves the efficiency of search algorithms in comparison with Random Walk. Thus, we have to re-caliber the scale based on values of $|E|R(G)$ anyway. In our estimate we reflect this by changing $|E|R(G)$ for $|V|d$, which is a fair substitute for regularly structured on-line search problem domains. These simplifications imply exactly the definition of the *oblongness* parameter.

7.3 Estimating Complexity of Planning Problems

We were able to relate the complexity of specific search problem to the *oblongness* factor in Section 7.2, because for many considered problems the domains had a regular structure, the branching factor was within a certain interval, i.e. the number of available actions at every state was approximately the same. Many planning problems demonstrate the same property, the *oblongness* factor can be applied to them as well. However, for some planning problems, the value of the branching factor depends on the stage of the planning process. For example, while solving the N-Queen problem, one can place the first Queen in any square of the chessboard. The more Queens are committed to squares, the less variants are left for the remaining Queens. Nonetheless, we would like to extend the relevance of the *oblongness* factor to such planning problems too.

7.3.1 Using Oblongness to Compare Search Complexities

Thus, according to Section 7.2, we can compare the complexities of on-line search problems by estimating the values of their *oblongness* parameters. We extend this comparison to planning domains and illustrate it through comparing the N-Queen and the “Mutilated” Checkerboard problems. The domain of the N-Queen problem contains N^2 squares, where one is supposed to place N Queens. Hence, the size of the domain is roughly

$$S_Q = \sum_{i=0}^{N} \left(\begin{array}{c} i \\ N^2 \end{array} \right).$$

In the above estimate we ignore symmetries and allow non-feasible partial solutions, because some algorithms may use them too. The estimated length of the N-Queen problem solution L_Q is N.

The size of the "Mutilated" Checkerboard problem is approximately

$$S_M = \sum_{i=0}^{\frac{N-1}{2}} \left(\begin{array}{c} i \\ N \end{array} \right).$$

To make the comparison fair, we count intersecting configurations for S_M too. The estimated length of the "Mutilated" Checkerboard problem solution L_M is also N. When these two problems are considered for approximately the same size of the chessboard, for example $N \times N$, then S_M is significantly bigger than S_Q, therefore $Oblongness_1 = L_Q/S_Q > L_M/S_M = Oblongness_2$. After the comparison of the *oblongness* values has been completed, we need to place problems in a proper *oblongness* spectrum region. It is easier to perform such a placing action for the "Mutilated" Checkerboard problem: S_M is approximately a half of $\sum_{i=0}^{N} C_N^i = 2^N$.

Hence, if compared with the domains from Figure 7.2, the "Mutilated" Checkerboard problem should be placed between the BlocksWorld and SAT. The N-Queen problem should be placed slightly further in the *oblongness* spectrum picture, between SAT and $N \times N$-puzzle. Indeed, finding a solution for the N-Queen problem is known to be a hard task, it was one of the problems that stimulated the development of backtracking methods in AI.

In Chapter 7.2 we introduced the *oblongness* parameter and two threshold level that determine the expected complexity of on-line search. First threshold level Ob_1 can be defined as containing problems with the diameters that are linear on some parameter N and domains of size $N!$. Second level Ob_2 determines the problems that are easy in navigating, although their diameters are relatively large. For problems of this type, the diameter spans a big portion of the domain. Thus, Ob_2 can be defined as a constant close to one. For the problems with the *oblongness* value below Ob_1 or above Ob_2, even non-monotone, non-admissible heuristics could be utilized efficiently, if they provide at least some "guidance" towards the goal. For domains with the *oblongness* values within the interval between Ob_1 and Ob_2, the "quality" of heuristics is crucial, as it determines the efficiency of search algorithms.

We do not determine what the term "quality" means regarding heuristic values, because for different approaches it can possess completely different meanings. For example, for off-line search, the "quality" of the heuristic would mean how close the heuristic values are to the goal distances. For on-line search, it is more important to have heuristic values balanced with respect to guiding a physical or fictitious agent towards the goal. Furthermore, distinct on-line search algorithms need different type of guidance. In our experiments with planar mazes we found that the Manhattan distance is more misleading for LRTA* than the MAX(X,Y)-distance, whereas for AC-A* algorithm it is exactly the opposite.

With respect to deterministic planning problems, we relate the complexity of building a feasible plan with the expected length of the plan and the size of the planning domain. In this estimate we assume that the length of the plan is of the same order as the diameter of the graph representing the planning domain. Thus, the *search entropy* is similar to the *oblongness* parameter of search problem domains.

There exists another reason of splitting search problems into ones with the *oblongness* value within the interval $[Ob_1, Ob_2]$ and out of it. Polemics on the difference between human reasoning and computer-oriented proofs, see for example [68, 63, 30], tend to place the strength of human reasoning in either "easily" navigated, highly connected problem domains, where a human does not need extensive modeling of all the contingencies and is able to recover from making mistakes "en-route," or heavily constrained domains with few branching points. The domains of the intermediate type with a lot of branching and "costly" recovery from making wrong decisions are amenable to computer-oriented search.

Another example of domains with highly variable *oblongness* comes from regulating departmental schedules: The "secretarial" task of assigning conference rooms is easy in the case when a department has an abundance of rooms. If there exists a unique room, this task can be also resolved on the "first to come – first to serve" basis. When the number of conference rooms does not easily satisfy the number of incoming requests, scheduling meetings becomes a challenging task [7].

7.3.2 Using Oblongness in Agent-Centered Search

The classification of problem domains based on the value of their *oblongness* value has an impact on the efficiency of agent-centered search. Recall, in agent-centered search a physical or fictitious agent has limited lookahead, and agent's actions form a continuous path through the problem domain.

Several off-line planning problems have been solved by introducing a fictitious agent and artificially limiting its lookahead by the immediate neighbors of the current state. Those problems have common features: Their *oblongness* parameter lies near or slightly above the Ob_1 boundary. The further the the value of the *oblongness* exceeds Ob_1, the more necessity is for careful selection of prior knowledge. Further increase of the *oblongness* leads to a strict dependency of the efficiency of agent-centered search on the "quality" of the prior knowledge guidance, unless the domain approaches the caterpillar-like shape with the value of the *oblongness* being a constant close to one.

Thus, to estimate the efficiency of agent-centered search for a particular problem, one should just figure out the *oblongness* factor of the problem domain. Based on such an estimate, the need for better heuristics becomes urgent for moderately "oblong" domains. For off-line problems of this type, standard non-agent-centered techniques of "teleporting," rejecting partial solutions and "island search" can be more efficient than applications of agent-centered guided by skewed heuristic values.

7.4 Promising Directions for Cross-Fertilization

In this section we consider both method-driven and problem-driven hybrid approaches and suggest hints on building new successful examples of hybrid algorithms or untraditional application across distinct disciplines.

7.4.1 Problem-Driven Cross-Fertilization

Recall that problem-driven type of the hybrid approach starts with the problem that one is supposed to solve. This is, probably, the most realistic practical application of the hybrid approach. According to the methodology presented in Chapter 1, in such a case one should begin with identifying methods from different scientific areas that are relevant to the problem under investigation. This phase is based solely on the experience of the researcher, we cannot suggest valuable hints on selecting relevant methods besides trying to be creative and keeping all methods that can somehow relate to the given task.

Given a particular problem, one is usually interested in the efficiency of its solution. Very often different methods are focused on different types of efficiency, sometimes their foci are contradicting. For example, reasonably risky algorithms can establish strong empirical performance, but lose to more cautious ones in the worst-case. Even more risky procedures can be sharply focused on specific problem domains and lose a lot when the domain is changed. There also exists a difference between the complexity of deriving the solution and the complexity of the solution itself. Some real-time algorithms can guarantee the convergence, but the solutions that they construct may be much worse than ones obtained by more computationally expensive procedures.

7.4.2 Hints on Building Hybrid Solutions

One of the possibilities for the hybrid approach is to combine methods with different efficiency foci in a single hybrid framework. Usually "cautious" procedures that provide worst-case guarantees, search the problem domain methodically without leaving a room for an uncontested opportunity. They do it either by chopping off guaranteed portions of the domain or by repeating certain steps.

A good opportunity for the **Constructing Hybrid Algorithms** phase is to relax one of such "cautious" procedures. It can be done, for example, through mixing its steps with more risky, more efficient in average (or for this type of problems) procedure by executing steps of the "cautious" procedure every so often. In this case, it will still chop off guaranteed portions of the domain, the described combination with another procedure would only offset its guarantees by a constant. In other cases, a "cautious" procedure may contain an obvious or a hidden parameter that can be relaxed to allow this procedure to be combined with another algorithm.

This was exactly the case with designing VECA, where instead of immediate backtracking we introduced parameter k and allowed the algorithm to traverse edges repeatedly without penalizing its actions until k traversals.

In Chapter 7 we discussed the relation between some features of the problem domain and the efficiency of search. We introduced *oblongness* as a simple feature of the domain that seems to capture the search complexity. By estimating the value of this feature for a particular problem domain, one gets a clear picture about the need in prior knowledge and the type of methods that can be applied efficiently to solve this problem. Some off-line search or deterministic planning problems can be solved effectively by agent-centered methods, i.e. by introducing a fictitious agent with limited lookahead (for the efficiency purposes) and forming a continuous path to the goal state through the problem domain. As we concluded in Chapter 7, such problem domains should have either very low or very high values of their *oblongness*, or the heuristic values should provide an excellent guidance towards the goal.

Another direction for the problem-driven hybrid approach is to combine some of the methods amenable to the discussed with known strong methods from other Sciences that are not traditionally related to the problem. In the next section we overview some of such non-traditional applications across different disciplines.

7.4.3 Method-Driven Hybrid Approach

Unlike the problem-driven hybrid approach, the method-driven approach starts from having methods already selected. Although those methods may come from distinct Scientific areas, can use very different vocabularies and even solve different problems, the intersection of their task directions should hint multi-disciplinary researchers on applying them in a beneficial way.

Our experience shows that the more difference between those methods are, the more exciting can the result of **Classification** or **Constructing Hybrid Algorithms** phases be. However, bigger difference between the selected methods imply possibly much more work in **Creating the Environment** and **Analysis** phases, success of which, in its turn, can result in a big payback from the concluding **Constructing Hybrid Algorithms** phase.

In this section we present hybrid applications that involve two drastically different principles from Mechanical Physics and Mathematics, namely the *perpetual motion* and Binary Division. These principles belong to the basis of both disciplines. Nonetheless, hybrid applications that include these principles, provide nice visual interpretations to otherwise cumbersome solutions.

Perpetual Motion

Even such a "non-computational" concept as the *perpetual motion* can be utilized in the hybrid approach. We introduce a geometrical problem that seems not to relate directly neither to

Physics, nor to Mechanics. However, a simple and nice reduction to the *perpetual motion* elegantly establishes the desired property.

Consider a convex 3D polyhedron. Every internal point may be projected either onto the 2D face of the polyhedron or on the extension of the face. Figure 7.6 shows a 2D slice containing the point in the case when the projection is located on the extension of the polyhedron's face.

Definition 2 *Internal point of a convex polyhedron is* stable*, if there exists a face of the polyhedron, such that this point is projected onto this face.*

Theorem 7.3 *All internal points of a convex 3D polyhedron are* stable*.*

Proof: Suppose that there exists a point, which is not *stable*. Make a firm model of the polyhedron with the mass center at the point under investigation. Put this model on an even surface. If none of the projections of this point get onto a face of the polyhedron, such a model would perform the *perpetual motion* over the surface, which is not possible. ■

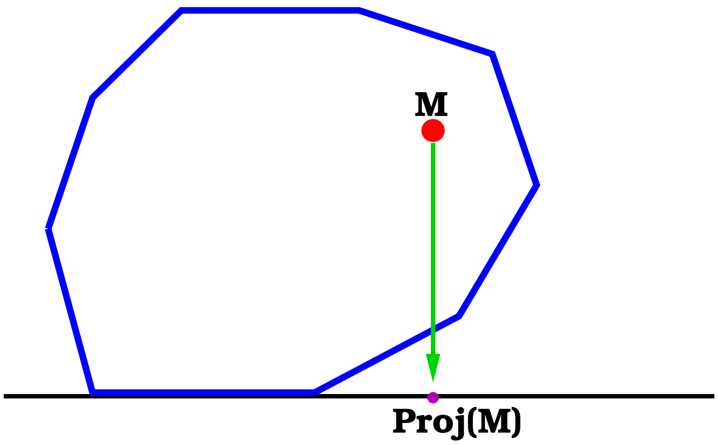

Figure 7.6: Projection of the Mass Center on the Face of a Polyhedron

Binary Division in Evaluating Sums of Two Arrays

Binary Division is another simple, but powerful principle that can be easily overlooked in designing optimal algorithms. In this section we describe applications of Binary Division in two optimal algorithms solving the Two Array Sums problem and Random Access Memory (RAM) troubleshooting. Recall, Binary Division uses the minimum number of binary queries to select the desired element out of a set with complete order.

Two Array Sums Problem: For how many pairs of indices (i, j) the relation $A[i] + A[j] \leq B[i] + B[j]$ holds, where A and B are two equi-sized arrays of real (not necessarily positive) numbers.

If we denote the size of arrays as N, the number of pairs of indices is $N(N-1)/2$. The brute-force solution that accounts all such pairs would have a quadratic complexity. However, an application of Binary Division cuts down the complexity of the Two Array Sums problem to $O(N \log N)$, as the following theorem shows:

Theorem 7.4 *The worst-case complexity of the Two Array Sums problem is $O(N \log N)$.*

Proof: Consider third array C of the same size N, which is the difference of A and B: $C[i] = A[i] - B[i] \;\; i = 1, \ldots, N$. With the help of array C we reduce the two-array problem to a single array problem, since $A[i] + A[j] \leq B[i] + B[j]$ implies $A[i] - B[i] \leq B[j] - A[j]$, which in its turn implies $C[i] + C[j] \leq 0$.

With array C we perform the following counting procedure:

1. Sort all elements of array C.
2. Apply Binary Division for each $i = 1, \ldots, N$ to determine k_i – the number of indices j, so that $C[j] \leq -C[i]$.
3. Sum up $k_i \;\; i = 1, \ldots, N$.

Steps 1-3 require only $O(N \log N)$[1] – the complexity of step 1 (sorting) and step 2 (N Binary Divisions).

Within the same counting procedure, we accomplished even more than promised: For every fixed index $i \in [1, N]$, we counted the number of indices j, so that $C[i] + C[j] \leq 0$, which is equivalent to $A[i] + A[j] \leq B[i] + B[j]$.

To prove that $O(N \log N)$ is tight, we reduce the Two Array Sums problem to sorting: If we set $C[i] \;\; i = 1, \ldots, N$ so that negative elements are alternating with the negations of positive ones, and positive elements are alternating with the negations of negative ones (for example, $\{-2k, -2k-2, \ldots, -2, 1, 3, \ldots, 2k-3, 2k-1\}$), then counting the number of indices j satisfying $C[i] + C[j] \leq 0$ for every $i = 1, \ldots, N$ is equivalent to sorting the array C, because for every fixed index i this number is different and corresponds to the order of this element in the array. ■

[1] Moreover, all steps can be parallelized, the same procedure can be performed in $O(\log N)$ time by N processors.

Binary Division in RAM Troubleshooting

Random Access Memory (RAM) became an essential part of the hardware design. Higher degree of integration allows to place larger memory configuration on the same physical space. However, closer placement of memory transistors make them vulnerable to affecting each other, thus destroying information that memory cells are supposed to keep. Typical memory faults on the cell level are:

- Inability of a particular cell to store a low or high level (0/1 bits).
- Pairwise influence of memory cells.

Hence, for RAM troubleshooting one is supposed to check that both low (0) and high (1) levels can be written to and read from every cell, that all four combinations of low and high levels can be written to and read from all possible pair of cells, since a particular layout of the chip can place distant memory cells (in terms of the address space) as physical neighbors. Second requirement seems like demanding a lot of writing and reading tests. However, we show that an application of Binary Division can significantly cut down the amount of writes and reads. We show that for the simplest case of register memory that can be easily generalized for arbitrary RAM configurations.

Theorem 7.5 *Troubleshooting N registers requires $\Theta(\log N)$ writes and reads.*

Proof: We assume that all N registers can be accessed simultaneously, i.e. in a single write we can attempt to change values of any of them, during a single read we get stored information from all registers. Figure 7.7 illustrates an application of Binary Division to troubleshooting N registers. Exactly $2(\log N + 1)$ writes and reads are needed to check whether all pairs of registers can store all possible pairs of low and high levels, during the last two write/read tests, all but two cells can be assigned arbitrary values (highlighted in golden color). Thus, the complexity of troubleshooting N registers is $\Theta(\log N)$. ■

7.5 Summary

In this chapter we considered the efficiency of solving search and planning problems and attempted to relate some features of the problem domains with the complexity of on-line search. We argued that the complexity of on-line search in a certain sense is similar to random walk, we linked our new introduced parameter with relevant on-line search techniques of different nature. These facts allowed us to bring already developed theory about the expected complexity of random walk and confirm the correctness of the novel feature.

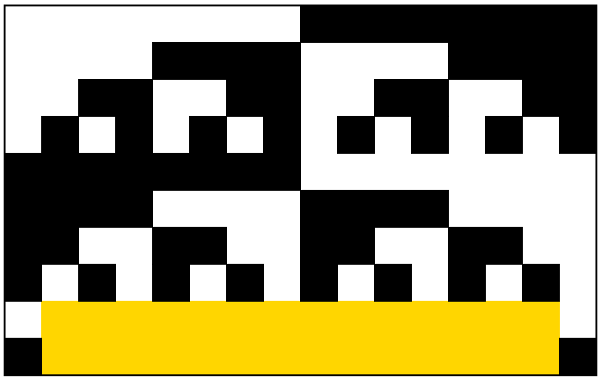

Figure 7.7: Troubleshooting N registers

We went through a sequence of simplifications, because we wanted to consider problems of different sizes, and, hence, to come up with a parameter that could be easily calculated in order to estimate the expected complexity of on-line search problems. Since search methods are very different in their nature, and the dependency of their efficiency on heuristic values varies from one method to another, we were interested only in approximate estimates. Such an estimation is expected to result in recommendations on method selection, whether to spend an extra effort on coming up with "high-quality" heuristics or acquiring more prior knowledge about the domain, on setting up internal parameters of search procedures, etc.

We found that the *oblongness* parameter captures well the complexity of search. It is a simple parameter of the problem domain that splits the domains into three categories. According to two threshold levels Ob_1 and Ob_2, problems with the values of the *oblongness* between these levels are complicated search problems that require prior knowledge with strong guidance towards the goal(s). On the other hand, problems with the values of the *oblongness* outside the interval $[Ob_1, Ob_2]$ can rely efficiently on heuristic values even with a "weak" guidance. For problems of this kind, even a non-monotone, non-admissible heuristic that reflect "common sense" for distantly related types of problems is likely to guide the search process efficiently.

In Section 7.4 of this chapter we considered both the problem-driven and method-driven hybrid approaches and stated a series of hints on building successful hybrid applications. Besides providing various examples of untraditional interdisciplinary applications that involve basic principles from distinct areas of Science, this chapter also states suggestions on how one can actually build hybrid algorithms, and how to utilize the results from Chapter 7 and attack on-line/off-line search/planning problems by agent-centered methods.

Chapter 8

Conclusions

In this thesis work we introduced the methodology of the inter-disciplinary hybrid approaches to solving various groups of problems from different areas of Sciences. We focused the development specifically on hybrid algorithms between Artificial Intelligence, CS theory and Operations Research. Such an approach enabled us to achieve the mutual enrichment and better understanding of many efficient methods that can be applied across distinct disciplines. Enlightened by new vision that comes from multi-facet consideration, we were able to solved several open problems by improving the worst-case and/or average-case (empirical) complexities, in some cases we performed competitive analysis and concluded with the directions of beneficial using known techniques.

Two methods of deriving upper bounds for the values of combinatorial optimization problem solutions – the Pigeonhole Principle and Linear Programming Relaxation – appear to have the same bounding power. Whatever established by either of them, can be derived by the other one. Moreover, these two methods are dual to each other in the sense of Linear Programming. Nonetheless, traditional applications of the Pigeonhole Principle carry more intuitive sense and indicate whether the upper bound is tight, whereas Integer Programming Relaxation can be applied automatically to any instance of Integer Programming problem. The latter approach provides an alternative way of solving combinatorial optimization problems.

For the goal-directed exploration problem we proposed a new systematic, application-independent framework, called VECA. VECA can accommodate a wide variety of exploitation strategies that use heuristic knowledge to guide the search towards a goal state. VECA monitors whether the heuristic-driven exploitation algorithm appears to perform poorly on some part of the state space. If so, VECA forces the exploitation algorithm to explore the state space more. This way VECA combines the advantages of both pure exploration approaches and heuristic-driven exploitation approaches: It is able to utilize heuristic knowledge, but, as opposed to existing heuristic-driven exploitation algorithms, it provides a good performance guarantee: Its worst-case performance over all state spaces of the same size – no matter how misleading the heuristic knowledge is – cannot be worse than that of the best uninformed goal-directed exploration algorithm. Thus, VECA provides better performance guarantees than previously studied goal-directed exploration algorithms, such as the AC-A* algorithm. Our experiments showed that this guarantee does not come at the cost of a deterioration in average-case performance for many previously studied exploitation algorithms: In many cases when used in VECA, their performance even improved.

I considered future work throughout the thesis, the concentration of hints and ideas to be discussed in future can be easily found in Chapter 7 on further insights into on-line complexity.

Bibliography

[1] M. Ajtai. The complexity of the pigeonhole principle. In *29th Annual Symposium on Foundations of Computer Science*, 346–355, 1988.

[2] R. Armstrong, D. Freitag, T. Joachims, and T. Mitchell. A learning apprentice for the world wide web. In *AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments*, 1995.

[3] B. Awerbuch, M. Betke, R. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. In *Proceedings of the Conference on Computational Learning Theory*, 1995.

[4] B.T. Bennett and R.B. Potts. Arrays and brooks. *Journal for the Australian Mathematical Society*, pages 23–31, 1967.

[5] G.D. Benson and A. Prieditis. Learning continuous-space navigation heuristics in real time. In *Proceedings of the Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1992.

[6] M. Betke, R. Rivest, and M. Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18(2/3), 2/3 1995.

[7] A.L. Blum. Empirical support for Winnow and Weighted-Majority based algorithms: results on a calendar scheduling domain. *Machine Learning*, 26:5–23, 1997.

[8] A. Blum, P. Raghavan, and B. Schieber. Navigation in unfamiliar terrain. In *Proceedings of the Symposium on Theory of Computing*, pages 494–504, 1991.

[9] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. In *Proceedings of AJCAI*, pages 1636–1642, 1995.

[10] C. Cheng and S. Smith. Applying constraint satisfaction techniques to job-shop scheduling. Technical Report CMU-RI-TR-95-03, Robotics Institute, Carnegie Mellon University, January 1995.

[11] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of 11th National Conference on Artificial Intelligence (AAAI)*, 21–27, 1993.

[12] X. Deng, T. Kameda, and C.H. Papadimitriou. How to learn an unknown environment. In *Proceedings of the Symposium on Foundations of Computer Science*, 1991.

[13] X. Deng and C.H. Papadimitriou. Exploring an unknown graph. In *Proceedings of the Symposium on Foundations of Computer Science*, pp. 355–361, 1990.

[14] P.G.L. Dirichlet. Vorlesungen über Zahlentheorie. Vieweg, Braunschweig, 1879.

[15] L. Euler. Solution problematis ad geometriam situs pertinentis. *Comment. Academiae Sci.I.Petropoloitanae*, 8:128–140, 1736.

[16] R.E. Filkes and N.J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4), 1971.

[17] G. Foux, M. Heymann, and A. Bruckstein. Two-dimensional robot navigation among unknown stationary polygonal obstacles. *IEEE Transactions on Robotics and Automation*, 9(1):96–102, 1993.

[18] J. Frank. Weighting for Godot: Learning Heuristics for GSAT. In *Proceedings of 13th National Conference on Artificial Intelligence (AAAI)*, 338–343, 1996.

[19] S. Hanks, M. Pollac, and P. Cohen. Benchmarks, testbeds, controlled experimentation and the design of agent architectures. In *AI Magazine*, Winter 1993.

[20] J.N. Hooker. Generalized Resolution and Cutting Planes. *Annals of Operations Research* 12(1988), 214-239.

[21] J.N. Hooker and M.A. Osorio. Mixed Logical/Linear Programming. 1996.

[22] W. Harvey and M. Ginsberg. Limited discrepancy search. In *Proceedings of AJCAI*, 1995.

[23] C. Hierholzer. Uber die moglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Math. Ann.*, 6, 1873.

[24] E.J. Hoffman, J.C. Loessi, and R.C. Moore. Constructions for the solution of the m queens problem. *National Mathematics Magazine*, March-April:66–72, 1969.

[25] M.R. Garey, and D.S. Jonsson Computers and intractability : a guide to the theory of NP-completeness. San Francisco: Freeman, 1979.

[26] C.F. Gauss. Disquisitiones Arithmeticae. Fleischer, Leipzig, 1801.

[27] I. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of 11th National Conference on Artificial Intelligence (AAAI)*, 28–33, 1993.

[28] I. Gent and T. Walsh. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research*, 1993, 1, 47–59, 1993.

[29] I. Gent and T. Walsh. Unsatisfied variables in local search. In J. Hallam, editor,*Hybrid Problems, Hybrid Solutions.* IOS press, 1995.

[30] M. Ginsberg. Do Computers Need Common Sense? In *Proceedings of Fifth International Conference on Knowledge Representation and Reasoning (KR-96)*, 620–626, 1996.

[31] M. Ginsberg. Partially Extended Resolution and the Pigeonhole Problem. CIRL, Tech.report.

[32] R.L. Graham, M. Grotschel, and L. Lovasz. Handbook on Combinatorics. Amsterdam ; New York : Elsevier : Cambridge, Mass.: MIT Press, 1995.

[33] F. Granot, and P.L. Hammer. On the Use of the Boolean Functions in 0-1 Programming. *Methods of Opereations Research* 12(1971), 154-184.

[34] J. Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin* 3(1):8–12, 1992.

[35] T. Ishida and M. Shimbo. Improving the learning efficiencies of realtime search.. In *Proceedings of 13th National Conference on Artificial Intelligence (AAAI)*, 338–343, 1996.

[36] L.G. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokaldy*, 20(1), 1979.

[37] S. Koenig and R.G. Simmons. Complexity Analysis of Real-Time Reinforcement Learning Applied to Finding Shortest Paths in Deterministic Domains. Technical Report CMU-CS-93-106; School of Computer Science, Carnegie Mellon University; 99 pages; 1992.

[38] S. Koenig. Agent-Centered Search: Situated Search with Small Look-Ahead. Ph.D. Thesis Proposal; School of Computer Science, Carnegie Mellon University; 30 pages; 1995.

[39] S. Koenig and Y. Smirnov. Learning graphs with a nearest neighbor approach. In *Proceedings of the Conference on Learning Theory (COLT)*, 19–28, 1996.

[40] S. Koenig and Y. Smirnov. Sensor-Based Planning with the Freespace Assumption. In *International Conference on Robotics and Automation (ICRA)*, 1997.

[41] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[42] R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 3 1990.

[43] J.C. Pemberton, and R.E. Korf. Incremental Path Planning on Graphs with Cycles. In *Proceedings of the AI Planning Systems Conference*, 179–188, 1992.

[44] R.E. Korf. From approximate to optimal solutions: A case study of number partitioning. In *Proceedings of AJCAI*, pages 266–272, 1995.

[45] O. Lassila and S. Smith. Constraint-based tools for complex scheduling applications. In *Proceedings of IEEE 4th Annual Dual-Use Technologies and Applications Conference*, page Utica (NY), 1994.

[46] V.J. Lumelsky, S. Mukhopadhyay, and K. Sun. Dynamic path planning in sensor-based terrain acquisition. *IEEE Transactions on Robotics and Automation*, 6(4):462–472, 8 1990.

[47] J. McCarthy. A Tough Nut for Proof Procedures. Stanford Artificial Intelligence Project, Memo No. 16, July 17, 1964.

[48] K.I.M. McKinnin, and H.P. Williams. Constructing Integer Programming Models by the Pedicate Calculus. *Annals of Opereations Research* 21(1989), 227-246.

[49] D. Mitchell, H. Levesque, and B. Selman. Hard and easy distributions of SAT problems. In *Proceedings of 10th National Conference on Artificial Intelligence (AAAI)*, 459–465, 1992.

[50] A.W. Moore and C.G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

[51] A.W. Moore and C.G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. In *Advances in Neural Information Processing Systems (Proceedings of the Conference on Neural Information Processing Systems)*, 1994.

[52] R. Motwani and P. Raghavan Randomized algorithms. Cambridge ; New York : Cambridge University Press, 1995.

[53] Nauck. Schach. *Illustrierte Zeitung*, 361:352, 1850.

[54] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1985.

[55] A. Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. Tualbee, editors, *Electronic Information Handling*. Spartan Books, Washington, DC, 1965.

[56] Illah Nourbakhsh and M. Genesereth. Assumptive planning and execution: a simple, working robot architecture. *Autonomous Robots*, 3(1):49–67, 1996.

[57] J. Pearl. Heuristics: Intelligent Search Strategies for Computer Problem-Solving. Addison-Wesley, Reading, MA, 1985.

[58] J.C. Pemberton and R.E. Korf. Incremental path planning on graphs with cycles. In *Proceedings of the AI Planning Systems Conference*, pages 179–188, 1992.

[59] The PRODIGY Research Group under the Supervision of Jaime G. Carbonell. Prodigy 4.0: The manual and tutorial. Technical Report CMU-CS-92-150, CMU, 1992.

[60] N.S.V. Rao, S.S. Iyengar, C.C. Jorgensen, and C.R. Weisbin. Robot navigation in an unexplored terrain. *Journal of Robotic Systems*, 3(4):389–407, 1986.

[61] N.S.V. Rao, N. Stoltzfus, and S.S. Iyengar. A “retraction” method for learned navigation in unknown terrains for a circular robot. *IEEE Transactions on Robotics and Automation*, 7(5):699–707, 10 1991.

[62] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal of Computing*, 6(3):563–581, 1977.

[63] J. Schaeffer, N. Treloar, P. Lu, and R. Lake. Man versus Machine for the World Checkers Champonship. *AI Magazine*, 14(20:28–35, 1993.

[64] B. Selman, H.J. Levesque, and D.G. Mitchell. An new method for solving hard satisfiability problems. In *Proceedings of AAAI*, pages 434–439, 1992.

[65] B. Selman and H.A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of AAAI*, pages 46–51, 1993.

[66] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of 12th National Conference on Artificial Intelligence (AAAI)*, 337–343, 1994.

[67] B. Selman and H.A. Kautz. Pushing the Envelope. In *Proceedings of 13 National Conference on Artificial Intelligence (AAAI)*, 1996.

[68] H.A. Simon. Artificial Intelligence: An Empirical Science. Artificial Intelligence, 77:95–127, 1995.

[69] Y. Smirnov, S. Koenig, and M.M. Veloso. Efficient exploration of unknown environments. In *Proceedings of the AMS Meeting in Kent*, page 799, 1995.

[70] Y. Smirnov, S. Koenig, and M.M. Veloso. Heuristic-driven "treasure hunt" with linear performance guarantees. In *Program of XXVII Southeastern International Conference on Combinatorics, Graph Theory and Computing*, page 60, 1995.

[71] Y. Smirnov, S. Koenig, M.M. Veloso, and R.G. Simmons. Efficient goal-directed exploration. In *Proceedings of AAAI*, 292–297, 1996.

[72] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1652–1659, 1995.

[73] A. Stentz. Optimal and efficient path planning for unknown and dynamic environments. *International Journal of Robotics and Automation*, 10(3):89–100, 1995.

[74] A. Stentz and M. Hebert. A complete navigation system for goal acquisition in unknown environments. *Autonomous Robots*, 2(2):127–145, 1995.

[75] A. Stentz. Map-based strategies for robot navigation in unknown environments. In *AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems*, pages 110–116, 1996.

[76] R.S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[77] S.B. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University, 1992.

[78] A. Thucker. *Applied Combinatorics*. John Wiley & Sons, 1980.

[79] M.M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, 1994.

[80] I.A. Wagner, M. Lindenbaum, and A.M. Bruckstein. On-Line Graph Searching by a Smell-Oriented Vertex Process. In *Proceedings of the On-Line Search Workshop at AAAI-97*, pages 122—125, 1997.

[81] H.P. Williams. Logic Applied to Integer Programming and Integer Programming Applied to Logic. *European Journal of Operations Research*, 81(1995), 605–616.

[82] A. Zelinsky. A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation*, pages 707–717, 1992.