UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANDRÉ GRAHL PEREIRA

Solving Moving-Blocks Problems

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Luciana Salete Buriol
Coadvisor: Prof. Dr. rer. nat. Marcus Ritt

Porto Alegre
October 2016

CIP — CATALOGING-IN-PUBLICATION

Pereira, André Grahl

Solving Moving-Blocks Problems / André Grahl Pereira. – Porto Alegre: PPGC da UFRGS, 2016.

120 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2016. Advisor: Luciana Salete Buriol; Coadvisor: Marcus Ritt.

1. Heuristic Search. 2. Single-Agent Search. 3. Moving-Blocks Problem. 4. Sokoban. 5. Pattern Database. 6. Abstraction Heuristic. 7. Computational Complexity. 8. Nondeterministic Constraint Logic. I. Buriol, Luciana Salete. II. Ritt, Marcus. III. Solving Moving-Blocks Problems.

“Me dê a sua força, Pégaso...
Meteoro de Pégaso!”
— SEIYA DE PÉGASO

ACKNOWLEDGEMENTS

I would like to thank you all people that helped to make this research possible. I would like to express my gratitude to my advisors Luciana Buriol and Marcus Ritt for their intellectual and personal support through this entire process. I also would like to thank my supervisors abroad Robert Holte and Jonathan Shaeffer. I extended my gratitude to the institutions UFRGS, UFSM and INPE and the people that are part of them.

I would like to thank my family. In especial my father Ledio Placido Pereira, my mother Ieda Grahl Pereira and my brother Ramon Grahl Pereira. I also would like to thank my uncles and aunts Salezio, Carla, Cátia e Orival for their immeasurable support.

I would like to thank my former advisors Claudio Tiellet, Cesar Pozzer and Adriano Petry. I also would like to thank all my professors during my entire life.

Finally, I would like to thank all my friends. In special my friends from my home city Araranguá, from Santa Maria, from Porto Alegre, from Edmonton and the ones that were my roommates during my graduate studies.

This research is supported by CNPq (462813/2014-2, 478847/2013-0) and by the National Center of Supercomputing at UFRGS.

ABSTRACT

In this thesis, we study the class of *moving-blocks problems*. A moving-blocks problem consists of k movable blocks placed on a grid-square maze where there is an additional movable block called the *man*, which is the only block that can be moved directly. In particular, each moving-blocks problem is defined by the set of moves available, by the goal description and by what happens when the man attempts to move a block. *Sokoban* is the best known and researched moving-blocks problem. We study moving-blocks problems in theory and practice.

We investigate the computational complexity of problems of moving-blocks. Prior to this thesis, most of the scientific literature addressed moving-blocks problems with PUSH moves only, in most of the cases proving that these problems are PSPACE-complete. We consider two sets of problems: PULL moves only, and PUSH and PULL moves combined. Our reductions are from Nondeterministic Constraint Logic. We prove that many problems with PULL moves only are PSPACE-complete. In addition, we prove that the entire set of PUSH and PULL moves is PSPACE-complete. Our contribution in this research line is to enhance the knowledge on the complexity landscape of moving-blocks problems.

Our main objective in this thesis is to optimally solve moving-blocks problems with a focus on Sokoban. Methods based on heuristic search and abstraction heuristics such as pattern databases are the most effective approaches to optimally solve these problems. We make many contributions in this research line. We introduce novel heuristic functions using pattern databases with the idea of intermediate goal states. We propose a technique based on pattern databases to detect deadlocks. We propose tie-breaking rules that exploit the structure of the problem. Using these heuristic functions and tie-breaking rules we increase the number of optimally solved instances of Sokoban and other problems compared to previous methods.

Keywords: Heuristic Search. Single-Agent Search. Moving-Blocks Problem. Sokoban. Pattern Database. Abstraction Heuristic. Computational Complexity. Nondeterministic Constraint Logic.

Resolvendo de Problemas de Blocos-Móveis

RESUMO

Nesta tese, nós estudamos a classe de *problemas de blocos-móveis*. Um problema de blocos-móveis consiste em k blocos móveis dispostos em um labirinto em grade quadrangular onde há um bloco móvel adicional chamado de o *homem*, que é o único bloco que pode ser movido diretamente. Em particular, cada problema de blocos-móveis é definido pelo conjunto de movimentos disponíveis, pela descrição do objetivo e pelo o que acontece quando o homem tenta mover um bloco. *Sokoban* é o problema de blocos-móveis mais conhecido e pesquisado.

Nós investigamos a complexidade computacional de problemas de blocos-móveis. Antes desta tese, a maior parte da literatura cientifica abordou problemas de blocos-móveis apenas com movimentos de EMPURRAR, na maioria dos casos provando que esses problemas são PSPACE-complete. Nós consideramos dois conjuntos de problemas: apenas movimentos de PUXAR, e movimentos de EMPURRAR e PUXAR combinados. Nossas reduções usam a Lógica de Restrições Não Determinística. Nós provamos que muitos problemas apenas com movimentos de PUXAR são PSPACE-complete. Além disso, nós provamos que o conjunto de problemas com movimentos de EMPURRAR e PUXAR é PSPACE-complete. A nossa contribuição nessa linha de pesquisa é aprimorar o conhecimento sobre o panorama da complexidade de problemas de blocos-móveis.

Nosso principal objetivo com essa tese é resolver otimamente problemas de blocos-móveis com foco em Sokoban. Métodos baseados em busca heurística e heurísticas de abstrações como banco de dados de padrões são as abordagens mais efetivas para resolver otimamente esses problemas. Nós fazemos muitas contribuições nessa linha de pesquisa. Nós introduzimos novas funções heurísticas usando bancos de dados padrão com a ideia de estados objetivos intermediários. Propomos uma técnica baseada em bancos de dados padrão para detectar impasses. Propomos regras de desempate que exploram a estrutura do problema. Usando estas funções heurísticas e regras de desempate nós aumentamos o número de instâncias resolvidas de forma ótima de Sokoban e outros problemas em comparação com os métodos anteriores.

Palavras-chave: Busca Heurística, Busca de Único Agente, Problema de Blocos-Móveis, Sokoban, Banco de Dados de Padrão, Heurística de Abstração, Complexidade Computacional, Lógica com Restrições Não Determinística.

LIST OF ABBREVIATIONS AND ACRONYMS

PDB	Pattern Database
IPDB	Intermediate Pattern Database
MPDB	Multiple Goal Pattern Database
CEGAR	Counterexample-Guided Cartesian Abstraction Refinement
IDA*	Iterative Deepening A*
IPC	International Planning Competition
NCL	Nondeterministic Constraint Logic
EMM	Enhanced Minimal Matching

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

A *planning* problem is defined by an initial description of the environment, a set of rules that defines how to transform the environment and a desired description of the environment. A solution to a planning problem is a sequence of actions that transforms the initial description into the desired description. This thesis studies a particular class of planning problems that is challenging and fun – *moving-blocks problems*.

Moving-Blocks Planning Problems

In a moving-blocks problem, movable blocks are placed on a grid-square maze defined by immovable blocks. There is a distinguished block, called the *man*, that is the only block that can be moved directly and is able to move other blocks. The initial state is given by the positions of the movable blocks and the man.

Different moving-blocks problems are based on what happens when the man attempts to move a block, the set of actions, and the definition of the goal state. *Sokoban* is the best known and studied moving-blocks problem. In Sokoban the man can push blocks to adjacent squares and the goal state is to achieve a defined placement of the movable blocks.

Moving-blocks problems are both challenging theoretically and practically. In general, problems in the class of moving-blocks are PSPACE-complete, and only artificial constraints make them tractable. Thereby, solving moving-blocks problems is at least as hard as solving planning in general when the set of states is finite.

Moving-blocks problems are fun. They have a simple and concise description, and are easy to understand, but still intellectually challenging. There is a considerable number of people interested in solving them. In addition, there is a community interested is developing automated approaches to solve instances of moving-blocks problems.

Objective

Our goal with this research has two aspects. The first is to improve the understanding of the theoretical hardness of deciding if a problem of moving-blocks is solvable. The second, and, in fact, our main focus, is to better solve optimally moving-blocks problems with a focus on Sokoban.

Theoretical Approach

In this thesis, we investigate the computational complexity of deciding if a moving-blocks problem is solvable. A large number of articles investigated the complexity of moving-blocks problems. However, the majority of the literature addressed problems with PUSH moves, in most of the cases proving that these problems are PSPACE-complete. Other versions of the problem with different types of moves have been proved to be NP-hard. The Nondeterministic Constraint Logic (NCL) is a framework developed by Hearn and Demaine (2005) to decrease the effort of proving PSPACE-hardness results. Usually, it is used to prove that puzzles and games are PSPACE-hard. We use this framework to investigate the hardness of the problems in the class of moving-blocks problems.

Practical Approach

Moving-blocks problems can be formalized as *state space* problems. Exploration (*systematic search*) is the most common approach to solve state space problems – the set of states is systematically explored until a goal state is reached. *Heuristics* improve the efficiency of searches guiding the exploration to more promising regions of the state space. Abstraction heuristic functions are responsible for an expressive progress in the areas of heuristic search and domain-independent planning. Our aim is to increase the number of optimally solved instances of some moving-blocks problems, in particular Sokoban. The most effective approaches to solve these problems so far are based on heuristic search techniques. We investigate new heuristic search techniques and new abstraction-based heuristic functions to optimally solve these problems.

1.1 Contributions to the Literature

The scientific publications to be considered for the partial fulfillment of the requirements for the Ph.D. degree are:

- PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Finding Optimal Solutions to Sokoban using Instance Dependent Pattern Databases. **Symposium on Combinatorial Search**, p. 141—148. 2013.
- PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Solving Motion Planning Problems. **International Joint Conference on Artificial Intelligence School - Doctoral Consortium**. 2014.
- PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Solving Sokoban Optimally using Pattern Databases for Deadlock Detection. **Encontro Nacional de Inteligência Artificial**. 2014.
- PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Optimal Sokoban Solving using Pattern Databases with Specific Domain Knowledge. **Artificial Intelligence**, p. 52–70. 2015.
- PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Pull and PushPull are PSPACE-Complete. **Theoretical Computer Science**, p. 50–61. 2016.
- PEREIRA, A. G. HOLTE, R.; SCHAEFFER J.; BURIOL, L. S.; RITT, M. Improved Heuristic and Tie-Breaking for Optimally Solving Sokoban. **International Joint Conference on Artificial Intelligence**, p. 662–668. 2016.

1.2 Contributions of this Research

In this thesis, we advance the computational complexity results and the heuristic search techniques to optimally solve the class of moving-blocks problems. Our contributions can be summarized in five items: (i) we prove that problems with PULL, and PUSH and PULL moves are PSPACE-complete; (ii) we introduce a pattern database approach to detect deadlocks; (iii) we propose a novel method to apply pattern databases to intermediate goals; (iv) we develop domain-dependent techniques that exploit the structure of the problem; and (v) we further improve the proposed pattern database heuristics and the domain-dependent techniques.

1.2.1 Moving-Blocks Problems with PULL and PUSHPULL Moves are PSPACE-complete

Dor and Zwick (1999) initiated a research line studying the computational complexity of the class of moving-blocks problems. Culberson (1999) proved that Sokoban is PSPACE-complete. Demaine, Demaine and O'Rourke (2000), Demaine, Hearn and Hoffmann (2002), Demaine et al. (2003), Demaine, Hoffmann and Holzer (2004) proved complexity results for several moving-blocks problems with PUSH moves. Most of the literature is focused on problems with PUSH moves. There is a single result for problems that combine PUSH and PULL moves. Problems with PULL moves have been proved to be NP-hard while equivalent problems with PUSH moves are known to be PSPACE-complete.

Our contributions are that we improve the known NP-hardness results of problems with PULL moves to PSPACE-completeness results. Therefore most of the versions with PULL moves are as hard as the versions with PUSH moves. We also were able to show that the whole class of problems that combines PUSH and PULL moves is PSPACE-complete.

1.2.2 Pattern Databases for Deadlock Detection

Sokoban is harder to solve when compared to other traditional state space problems, and the presence of deadlocks is one of the reasons for that. Deadlocks are states that are reachable from the initial state but cannot reach any goal state. In recent years one of the most effective approaches to create heuristic functions for state space problems are

pattern databases (PDBs) (CULBERSON; SCHAEFFER, 1996). An effective heuristic function for Sokoban must detect deadlocks, otherwise the search method could spend great effort exploring parts of the state space that will not lead to a goal state.

Our main contribution is to propose a PDB heuristic function to Sokoban for deadlock detection. We also apply standard PDBs as an admissible heuristic function for Sokoban. For this, we proposed an abstraction transformation, a method to build the PDB efficiently and to compute the heuristic value. However, due to domain-specific characteristics of Sokoban standard PDBs are ineffective as a heuristic function. We show that a standard PDB nevertheless can effectively detect deadlocks and may be used together with other heuristic functions to increase the number of optimally solved instances. Using the proposed approach, we are able to detect five times more deadlocks than the standard heuristic function of Sokoban, solving optimally two more instances, while exploring an order of magnitude fewer nodes.

1.2.3 Pattern Databases to Intermediate Goals

The effectiveness of the search method is directly related with the heuristic function used. A straightforward application of PDBs in Sokoban results in an ineffective heuristic function. We propose an alternative approach, by introducing the idea of an instance decomposition to obtain an explicit intermediate goal state which allows an effective application of PDBs. Similar to the previous contribution, we proposed an efficient method to construct the PDB and to compute the heuristic function. We also prove theoretical properties of the proposed heuristic. When applied on the standard set of instances of Sokoban this approach improves heuristic values on initial states, detects considerably more deadlocks in random states, and increases the number of optimally solved instances compared to previous methods.

1.2.4 Using Domain-Specific Structure Information

The heuristic function, in general, is based on the computation of distances in the state space. We propose a domain-specific tie breaking rule that gathers information from the structure of the problem. It uses the information from the state space related to the placement of goals to improve the effectiveness of the search method. Based on this idea

we propose a specific tie-breaking rule to Sokoban and other problems.

1.2.5 Improved Pattern Databases Heuristic and Domain-Dependent Technique

We extend the idea of intermediate goal state to allow the use of multiple intermediate goal states and show that the previous proposed heuristic is no longer effective. We solve this problem and show that the new heuristic is effective when using multiple intermediate goal states. Our new method increases the number of optimally solved instances and the number of proved optimal solutions of Sokoban.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces the basic concepts about heuristic search as well as recent research progress in designing more informative heuristic functions with a focus on PDBs. It also introduces the domains studied in this thesis. Chapter 3 presents detailed complexity results about the class of moving-blocks problems. Chapter 4 presents theoretical and experimental results about Sokoban, and introduces a domain-dependent solver and heuristic search techniques. Chapter 5 presents the improved pattern database heuristic and tie-breaking rule for optimally solving Sokoban with theoretical and experimental results. Chapter 6 presents concluding remarks and future research directions.

2 BACKGROUND

In this chapter we review the basic concepts of heuristic search, the definitions that are used in this thesis and recent developments in designing admissible heuristic functions. We also present Sokoban in detail and, in general, the class of moving-blocks problems.

2.1 State Space Problem

Definition 2.1.1 (Weighted State Space Problem)**.** *A weighted state space problem is a tuple* $P = (S, A, s, T, w)$*, where:*

- S *is a set of states,*
- A *is a finite set of actions,*
- $s \in S$ *is the initial state,*
- $T \subseteq S$ *is a set of goal states, and*
- $w : A \to \mathbb{R}_{\geq 0}$ *is a cost function.*

An action $a \in A$ is a function $a : S \to S$ that transforms a state u into a successor state v incurring cost $w(a)$. The set of goal states may be given explicitly, or implicitly by a goal condition. A solution $\pi = (a_1, ..., a_k)$ is an ordered sequence of actions that transforms the initial state s into some goal state. The solution cost of π is defined as $\sum_{i=1}^{k} w(a_i)$. A solution is *optimal* if it has the minimum cost among all solutions. F is a finite set of state variables. Each variable $f \in F$ has an associated domain of possible values $D(f)$. A complete assignment of values $d_f \in D(f)$ to variables f defines a state.

To find an optimal solution to a state space problem corresponds to solve a single-source shortest path problem in a graph. A state space problem P corresponds to a graph $G = (V, E, s, T)$ with vertices $V = S$, initial vertex s, goal vertices T and edges $E \subseteq V \times V$. There is an edge $(u, v) \in E$ iff $v = a(u)$ for some action $a \in A$. The objective is to find a path in G from s to some goal vertex in T. The main difference between a graph problem and a state space problem is that a graph problem assumes that the graph is given explicitly and in a state space problem the graph is given implicitly.

2.2 Heuristic Functions

In general, a *heuristic* is a rule that is effective in finding a solution to a problem, but the solution is not guaranteed to be optimal. In the heuristic search and domain-independent planning literature, heuristic refers to a specific class of heuristic evaluation functions. For a state space problem, the *heuristic function* estimates the solution cost from some given state to some goal state. A heuristic search algorithm uses this information to orient the search into the direction of some goal state. The heuristic function has a large influence on the performance of the heuristic search algorithm.

Definition 2.2.1 (Heuristic)**.** *A heuristic h is a function that maps a state $u \in S$ to $\mathbb{R}_{\geq 0}$.*

The *perfect heuristic* h^* gives the exact solution cost of each state to the closest goal state (∞ if no solution exists). A heuristic is particularly useful if it is a lower bound, i.e., never exceeds the optimal solution cost.

Definition 2.2.2 (Admissibility)**.** *A heuristic h is admissible if $h(u) \leq h^*(u)$ for all $u \in S$.*

Another important property of a heuristic function is *consistence*.

Definition 2.2.3 (Consistence)**.** *A heuristic h is consistent if, for all states $u, v \in S$, $h(u) \leq h(v) + w(u, v)$.*

If $h(u) \geq h'(u)$ for all states $u \in S$, h dominates h'. In general, a heuristic search algorithm with h will have better performance than using h', exploring fewer nodes. There are however cases, where this does not hold (HOLTE, 2010; HOLTE et al., 2016).

2.3 Heuristic Search Algorithms

A search algorithm solves the task of single-source path in a given graph through a trial-and-error exploration. This task can be solved by algorithms like breadth-first search, depth-first search or Dijkstra's algorithm. A* (HART; NILSSON; RAPHAEL, 1968) and Iterative Deepening A* (IDA*) (KORF, 1985) perform heuristic search to reduce the exploration effort. They use the function $f(u) = g(u) + h(u)$ where $g(u)$ is the distance from the initial state s to state $u \in S$, and $h(u)$ is the heuristic function. The f-value is the estimated solution cost from s through u to some goal state.

A* and IDA* with an admissible heuristic are guaranteed to return an optimal solution if one exists. A state is a configuration of a problem and a *node* is a data structure that represents a state together with f, g, h-values and a parent link. Several nodes can represent the same state in a heuristic search algorithm. To *generate* a node corresponds to create the data structure for a given state. To *expand* or *explore* a node corresponds to generate all its successors.

2.3.1 The A* Algorithm

The A*algorithm was proposed by (HART; NILSSON; RAPHAEL, 1968). According to Edelkamp and Schrödl (2012) the A* algorithm is "the most prominent heuristic search algorithm". Every node generated by A* is stored in memory. Thus, with a consistent heuristic every node is explored only once. However, this benefit leads to the main drawback of A*: it uses linear space in the number of states, which is usually exponential. To manage the nodes in memory is also costly.

Figure 2.1 shows the A* algorithm. All generated (yet to be expanded) nodes are stored in the *Open* set and all expanded nodes in the *Closed* set. At every step the algorithm removes the node u with the smallest f-value from *Open* and stores it in *Closed*. If u is a goal, then the algorithm terminates and returns the optimal solution. Otherwise, it expands u generating all its successors managing the generated nodes according to function *Improve*. Improve receives two nodes, u and its successor v. If v is in *Open* and a shortest path was found, then the *parent* of v and its f-value are updated. In case of an *inconsistent* heuristic, if v is in *Closed* and a shorter path to v was found v is *reopened*. The algorithm removes v from *Closed* and inserts into *Open* updating its parent and f-value. In the last case, a new node is generated with a parent and f-value.

2.3.2 The IDA* Algorithm

The Iterative Deepening A* algorithm proposed by (KORF, 1985) solves the main drawback of A*. IDA* needs linear space in the solution length. The algorithm uses the f-value to perform a bounded exploration iteratively in a depth-first manner. IDA* does not detect duplicates. Thus, it is possible to re-explore nodes. Asymptotically, IDA* explores the same number of nodes as A*, given that the number of explored nodes grows

Figure 2.1: A* algorithm.

Implicit Search
Input: State space problem P and heuristic function h.
Output: Optimal solution from s to $t \in T$, or $\emptyset$ if no solution exists.

$Closed \leftarrow \emptyset$
$Open \leftarrow \{s\}$
while $Open \neq \emptyset$ **do**
    Select some u from $Open$
    Insert u into $Closed$
    if $Goal(u)$ **then return** $Path(u)$
    $Succ(u) \leftarrow Expand(u)$
    for each v **in** $Succ(u)$ **do**
        $Improve(u, v)$
return $\emptyset$

Improve
Input: Nodes u and v, v successor of u.
Output: Updates parent of v, $f(v)$, $Open$, and $Closed$.]

if v **in** $Open$ **then**
    if $g(u) + w(u, v) < g(v)$ **then**
        $parent(v) \leftarrow u$
        $f(v) \leftarrow g(u) + w(u, v) + h(v)$
else if v **in** $Closed$ **then**
    if $g(u) + w(u, v) < g(v)$ **then**
        $parent(v) \leftarrow u$
        $f(v) \leftarrow g(u) + w(u, v) + h(v)$
        Remove v from $Closed$
        Insert v into $Open$
else
    $parent(v) \leftarrow u$
    $f(v) \leftarrow g(u) + w(u, v) + h(v)$
    Insert v into $Open$

exponential with depth search – the last iteration dominates the previous ones. The optimality in the number of nodes explored by A* implies the asymptotical optimality of IDA* in number of explored and stored nodes. However, in practice IDA* can explore many more nodes than A* especially in domains where small cycles are common.

Figure 2.2 shows the IDA* algorithm. It performs a series of depth-first explorations bounded by U'. In the first iteration, the value of U' is equal to the heuristic value in s the initial state. If $h(s) = h^*(s)$ then the algorithm has only one iteration. During one iteration only nodes with f-values no greater than U are explored. The value of U is updated with the smallest f-value of a generated but not yet expanded node in the last iteration. This update guarantees that at least one new node will be explored in the next iteration and once a solution is found it must be optimal.

Figure 2.2: IDA* algorithm.

IDA*
Input: State space problem P and heuristic function h.
Output: Optimal solution from s to $t \in T$, or $\emptyset$ if no solution exists.

$U' \leftarrow h(s)$
$Path \leftarrow \emptyset$
while $Path = \emptyset$ **and** $U' \neq \infty$ **do**
    $U \leftarrow U'$
    $U' \leftarrow \infty$
    $Path \leftarrow Iteration(s, 0, U)$
return $Path$

Iteration
Input: Nodes u, path length g and upper bound U.
Output: Optimal solution from s to $t \in T$, or $\emptyset$ if no solution exists. Update threshold U'.

if $Goal(u)$ **then return** $Path(u)$
$Succ(u) \leftarrow Expand(u)$
for each v **in** $Succ(u)$ **do**
    if $g + w(u, v) + h(v) > U$ **then**
        if $g + w(u, v) + h(v) < U'$ **then**
            $U' \leftarrow g + w(u, v) + h(v)$
    else
        $Path \leftarrow Iteration(v, g + w(u, v), U)$
        if $Path \neq \emptyset$ **then return** $(u, Path)$
return $\emptyset$

2.4 Abstraction-Based Heuristic Functions

Abstraction-based heuristic functions led to a major advancement in the heuristic search and domain-independent planning in recent years. Informally, an abstraction ignores some information or constraints of the state space obtaining a more "coarse-grained" version of the state space. An abstraction heuristic function corresponds to the exact distances in the abstract state space.

Definition 2.4.1 (Abstraction transformation)**.** *An abstraction transformation $\phi : S \rightarrow S'$ maps states u in the concrete state space S to abstract states u' in the abstract state space S'.*

If the abstraction transformation defines an abstract state space where the distance between all abstract states u' and v' is smaller or equal to the distance between states u and v in the concrete state space – $w(u', v') \leq w(u, v)$. The distance of the abstract state space can be used as an admissible heuristic function in the original state space. The abstract heuristic h is defined as the distance to the closest abstract goal state.

2.4.1 Pattern Databases

The *Pattern Database* (PDB) heuristic was introduced by Culberson and Schaeffer (1996). A PDB is defined by an abstraction transformation, also called *pattern*, that projects away all information of the state outside of the abstraction. It selects a subset of variables F' of the set variables F of states and considers only the information of variables in F'. Then, two states u and v that only differ in variables outside of the abstraction are mapped to the same abstract state. This abstraction defines abstract states u' and the abstract state space S'.

A PDB stores the distances of abstract states to abstract goal states in a lookup table. It precomputes the optimal solution cost from the set of abstract goal states for all abstract states by reverse search. The effectiveness of the PDB depends on the chosen abstraction. Different abstractions lead to different heuristic functions and to abstract state spaces with different sizes – usually the abstract state space must be small enough to be explored exhaustively.

One challenge for the use of PDBs is to choose the most informative abstraction. Given that multiple abstractions are possible one way to combine them is take their maximum – since the maximum of admissible heuristic results in an admissible heuristic. Culberson and Schaeffer (1996) applied PDBs to the Fifteen Puzzle and explored this idea. They use two different abstractions, generating two PDBs. They take the maximum value provided by the two PDBs and the Manhattan distance. Using the resulting heuristic they were able to reduce the number of explored nodes by three orders of magnitude compared to using only the Manhattan distance. Korf (1997), for the first time, solved random initial states of Rubik's Cube taking the maximum of three different PDBs. These two achievements were made possible due to abstraction-based heuristics.

A more informative approach to combine heuristic is to add their values instead of taking their maximum. In general, however, this does not lead to an admissible heuristic. Given a set of PDBs defining a set of heuristic functions $h_1, \cdots, h_k$ it would be desired instead of taking their maximum to add the heuristic values resulting in an admissible heuristic. This can be accomplished by ensuring that the cost of each action a in A is accounted in only one PDB. Then, the sum of the set of heuristic values results in an admissible estimate.

Korf and Felner (2002), Felner, Korf and Hanan (2004) apply this idea to specific domains. Yang et al. (2008) give the theoretical foundation for additive abstractions. Katz

and Domshlak (2008a) introduce the notion of *cost partition* and show how to obtain optimal admissible partition of PDBs in polynomial time. The cost partition approach splits the cost of each action to the set of abstractions.

An important question is that if is better to use one large PDB or several smaller ones. Holte et al. (2006) consider how to best use a fixed amount (m units) of memory for storing PDBs. They examine whether using n PDBs of size m/n instead of one PDB of size m improves search performance. They compare several PDBs of size m/n for various values of n taking the maximum between them and for a fixed total size of m. The experiments showed that large and small values of n are suboptimal. There is an intermediate value of n that reduces the number of nodes generated by up to two orders of magnitude over $n = 1$ (one PDB of size m).

Another approach to address the space limitation of PDBs is to use compression. Felner et al. (2007) explore the idea of merging adjacent entries of the PDB into a single one. This allows the use of larger abstractions for building the PDBs. Using an appropriate storage method, adjacent entries are highly correlated, and most of the information is preserved. PDBs in all previous studies have had one entry for each state in the abstract state space. They compress cliques in the PDB, i.e., states in the abstract state space that are reachable from each other by one edge. Thus, the PDB for these entries will differ from one another by no more than one unit (when all actions have cost one). Using this idea they obtained several improvements in many domains.

2.4.2 Hierarchical Search

Holte et al. (1996), Holte, Grajkowski and Tanner (2005) proposed the *hierarchical heuristic search* method. The main idea is to compute, on demand, only the abstract states that will be employed to solve a given initial state. In general, PDBs are built exploring the whole abstract state space in a preprocessing phase. Building a PDB can be time-consuming, but the time can be amortized if there are many initial states of the same problem to be solved, i.e. if the PDB can be reused. This is not the case for moving-blocks problems – since each instance is a different problem. Hierarchical search solves this problem using a hierarchy of abstractions.

A hierarchical search algorithm computes the heuristic for a state s by applying an abstraction, obtaining an abstract state s' and an abstract state space S'. Then, the algorithm explores S' and the optimal solution cost of s' is used as the heuristic value for s.

To efficiently search on S', the next abstraction in the hierarchy is applied obtaining the abstract state s'' and an abstract state space S''. The number of abstractions used is unlimited. This process continues until the abstract state space generated is small enough to be enumerated exhaustively. The optimal solution cost of abstract states found in previous searches are stored in lookup tables for efficiency. Hierarchical search outperforms PDBs by orders of magnitude in domains where each problem has a unique initial state to be solved.

2.4.3 Merge-and-Shrink Abstractions

Helmert, Haslum and Hoffmann (2007), Nissim, Hoffmann and Helmert (2011) generalized the idea of abstraction-based heuristics to *merge-and-shrink* abstractions. The abstract state space is built incrementally. The process starts with a set of atomic abstract state spaces – each abstract state space is built on a single state variable. Then, two abstract state spaces are merged and replaced by their synchronized product, which is greater and probably more informative. After the merging, the resulting abstraction is shrunk – aggregating pairs of abstract states into one – to control the exponential growth of the abstract state spaces. The shrinking process continues until a desired size of the abstract state space is achieved. The incremental process finishes when all abstractions were merged. The framework of merge-and-shrink abstractions gives great freedom in the abstraction design. Theoretically, it dominates most other known admissible heuristics (HELMERT; DOMSHLAK, 2009). However, the theoretical power of merge-and-shrink abstraction arises from the perfect decisions which abstractions to merge and abstract states to aggregate. Better merge and shrink methods are an open research problem.

2.4.4 Domain and Cartesian Abstractions

Domain abstractions (HERNáDVöLGYI; HOLTE, 2000) partition each variables domain in sets of values that are considered equal. Domain abstractions are more general than PDBs because a domain abstraction that considers all values of a variable equal corresponds to a PDB. This allows a more fine-grained definition of an abstraction and perhaps a more informative heuristic. In domain abstractions, the partition is made for all abstract states where the more general *Cartesian abstraction* (SEIPP; HELMERT, 2013;

SEIPP; HELMERT, 2014) allows to partition each state individually. Thus, one abstract state may correspond to a single concrete state while another abstract state corresponds to half of the states from the state space.

Cartesian abstractions are built in an iterative process based on the counterexample-guided abstraction refinement methodology. It starts from an abstraction that generates an abstract state space with a unique abstract state where all states are mapped to a single abstract state. The algorithm iteratively computes solutions to the abstract state space and checks if the computed solution is also a solution for the concrete state space. If it fails, it refines the abstract state space such that the same failure cannot occur in future iterations. Space and time limit this iterative process. The resulting abstractions were shown experimentally to be efficient admissible heuristics.

2.4.5 Implicit Abstractions

Previous abstraction heuristics such as PDBs, merge-and-shrink and domain abstractions require a moderate size abstract state space that can be explored exhaustively. Implicit abstractions, proposed by Katz and Domshlak (2008b), Katz and Domshlak (2009), impose a different restriction on the abstract state space. Instead of size it requires tractability. An implicit abstraction transforms a state space into an abstract state space such that optimal solutions for the abstract state space can be computed in polynomial time. This type of abstraction does not limit the size of the abstract state space and proved to be effective in practice.

2.5 Landmark Heuristic Functions

A *landmark* is something that every solution must accomplish at some point to achieve the goal. A *fact landmark* is a variable value that must be true at some point in every solution (HOFFMANN; PORTEOUS; SEBASTIA, 2004). The values of variables on initial and goal states are fact landmarks. An *action* landmark is an action that must occur in every solution. A disjunctive set of landmarks is a set in which at least one of the landmarks must occur in every solution. Landmarks are implicit sub-goals. They were initially used to decompose the problem into subproblems, enabling to try to solve each subproblem individually. Orderings between landmarks are possible, for example,

a landmark A must be achieved before another landmark B; or a landmark A must be achieved directly before a landmark B.

Initially, landmarks were proposed as a method to decompose problems. Recently, many heuristic functions were proposed based on landmarks. Richter, Helmert and Westphal (2008) proposed a non-admissible path dependent heuristic function that estimates the distance to the goal state of a state by the number of landmarks yet to be achieved. Karpas and Domshlak (2009) introduced an admissible landmark multi-path dependent heuristic function using a cost-partition approach to guarantee admissibility. The landmark-cut proposed by Helmert and Domshlak (2009) is the result of theoretical research on the relation between different heuristic functions. Landmark-cut is an admissible heuristic that uses an advanced strategy to compute sets of disjunct action landmarks. Bonet and Helmert (2010) and Pommerening and Helmert (2013) improved the landmark-cut introducing new approaches to combine the information of landmarks admissibly and to compute them. Landmark-cut is one the most successful heuristics in domain-independent planning.

To decide if a fact is a fact landmark is PSPACE-complete. Thus, in general, to decide if a fact is a fact landmark methods use a sufficient criterion in a relaxed version of the problem. An action landmark can be efficiently detected. If a relaxed problem without an action a in unsolvable then, a is an action landmark. There is a fundamental difference between abstractions and landmarks: an abstraction heuristic explores distances in the state space, whereas a landmark heuristic explores the structure of the state space.

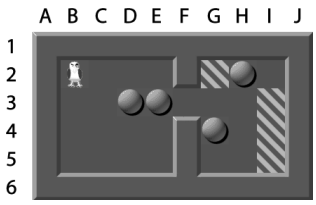
2.6 Moving-Blocks Problems

In this section, we present the class of moving-blocks problems. We start reviewing in detail Sokoban and related literature. Then, we present generalizations of Sokoban. Finally, we present other examples of similar problems.

2.6.1 Sokoban

Sokoban can be solved as a state space problem. It is set on a maze grid, which is defined by squares occupied by immovable blocks (*walls*) and free squares. There are k movable blocks called *stones* and k goal squares. The *man* (Sokoban) is a movable block

Figure 2.3: Heuristic function of Sokoban computed by a minimum cost perfect matching in a bipartite graph with domain-dependent enhancements.



(a) Example of an instance.

	Goals			
Stones	G2	I3	I4	I5
H2	1	∞	∞	∞
D3	4	5	6	7
E3	3	4	5	6
G4	2	3	4	5

(b) Naive distances.

	Goals			
Stones	G2	I3	I4	I5
H2	1	∞	∞	∞
D3	**6**	5	6	7
E3	**5**	4	5	6
G4	2	**9**	**10**	**11**

(c) Backout conflicts.

that can traverse free squares and push stones to adjacent free squares. A solution of Sokoban is a sequence of such actions that move the stones from their initial positions to the goal squares. The most common objective is to find a solution which minimizes the number of pushes, without accounting for the moves of the man. In this thesis, we are interested in an admissible solver for this objective. An admissible solver always finds an optimal solution to an instance if one exists, while a non-admissible solver can return any feasible solution.

In Sokoban, a state is defined by the positions of the *k* stones and by the *reachable component* of the man, i.e., the set of free squares reachable by the man without pushing stones. A reachable component can be represented by a normalized position of the man, e.g. the leftmost upper free square of the component. A *goal state* is defined implicitly as a state in which each stone is on a different goal square. Sokoban has $k!$ goal states since the stones are unlabeled and can be placed on any goal square. A *deadlock* is a state $u \in S$ which is reachable from s but cannot reach any goal state. In general deadlocks are hard to detect. A particular situation of deadlocks in Sokoban is caused by *dead squares*. A free square is *dead* if a stone on it cannot be pushed to any goal square. By this definition, a goal square is never a dead square, since if a stone is on it, the stone is already located at a goal square and then it does not need to be pushed.

Sokoban is a simplified model of general robot motion planning that computes a collision-free path between origin and destination points, which is a fundamental problem in robotics and has a large range of applications (DOR; ZWICK, 1999). There is a standard set of 90 problem instances used in the literature, ordered roughly from easiest to hardest in difficulty for a human to solve. Table 2.1 shows some search space properties of these instances. Sokoban is a challenging problem, and one of the remaining puzzles which humans solve better than computers. All instances of the standard set have been solved by humans, but even the best non-admissible solvers are not able to solve all of

them. Sokoban is PSPACE-complete (CULBERSON, 1999), and is harder to solve than other well-known single-agent search problems like Rubik's cube or the 24-puzzle, due to its large branching factor, greater solution length, larger search space size, and a more complex computation of the heuristic value (JUNGHANNS; SCHAEFFER, 2001). Real world problem characteristics like the presence of deadlocks, states that are more complex to represent and generate, and a lack of symmetry also contributes to the difficulty of solving Sokoban. For this reason, only a few instances have been solved with admissible techniques, and most of the Sokoban solvers are concerned only with finding a solution using techniques without optimality guarantees.

Table 2.1: Search space properties of the standard set of instances of Sokoban.

	Branching Factor	Solution Length	Search Space Size	Stones	Free Squares	Non-dead Squares
Min.	0	97	10^8	6	49	41
Avg.	12	260	10^{18}	16	113	77
Max.	136	674	10^{31}	34	181	133

2.6.1.1 Rolling Stone

Rolling Stone is one of the best-known solvers for Sokoban. It comes in an admissible and a non-admissible version. Both use an IDA* search and multiple domain-independent and domain-dependent enhancements. The admissible version, which we call RS*, uses techniques like an enhanced heuristic, move ordering, tunnel macros, transposition and deadlock tables. When limited to explore 20 million nodes and with a deadlock table of approximately 22 million entries it solves six instances. Prior to the work in this thesis, RS* was the best admissible Sokoban solver (JUNGHANNS; SCHAEFFER, 2001). In an attempt to solve more instances, the non-admissible version, referred to as RS, applies techniques that do not guarantee optimality such as goal cuts, pattern searches, relevance cuts, overestimation, and rapid random restart. RS is able to solve 57 instances within the same limits.

2.6.1.2 Other Solvers

Botea, Müller and Schaeffer (2002) proposed planning on an abstraction of the search space. Applied to Sokoban they obtain a non-admissible solver which is able to solve ten instances from the standard set in less than three minutes and exploring

fewer than one million nodes. Demaret, Lishout and Gribomont (2008) describes a non-admissible solver that uses a hierarchical planning strategy along with deadlocks learning to solve Sokoban. Using this approach, they were able to solve 54 instances from the standard set. In this case, the stopping criterion was eight hours of running time per instance.

The state-of-the-art non-admissible solvers have been developed by the Sokoban community. Among the ones with the best results, JSoko is able to solve 71, YASC 79, and Takaken 86 instances (MEGER, 2014; DAMGAARD, 2014; TAKAHASHI, 2014; Sokoban Wiki, 2014). Since these results have not been published formally, it is not always clear which techniques have been used to achieve them.

2.6.1.3 Heuristics for Sokoban

The best heuristic for Sokoban was proposed by Junghanns and Schaeffer (1998b). To obtain a lower bound, it relaxes the restriction that a stone can block the path of another stone. The distance of a stone to every goal square can be computed by solving an instance with only one stone. For the example in Figure 2.3a these "naive distances" can be seen in Figure 2.3b. Since each stone has to be assigned to a different goal square, the smallest number of pushes needed to bring each stone to a different goal square is a valid lower bound. This number is equal to the minimum cost of a perfect matching in the complete bipartite graph between stones and goal squares where the weight of a stone-goal edge is the naive distance of the stone from the goal. A minimum cost perfect matching can be found in $O(k^3)$, where k is the number of stones (KUHN, 1955). We call this heuristic function MM. MM is the optimal solution value of an instance where the capacity constraints of the free squares have been relaxed, allowing several stones and the man to have the same position.

Two enhancements of MM were proposed by Junghanns and Schaeffer (1998b). The first takes *backout conflicts* into account. Backout conflicts consider the position of the man when his movement is restricted in articulation squares by a stone. An articulation square is a square that if removed disconnects the connected component of the instance. Figure 2.3c shows the distances obtained when considering backout conflicts. Improved distances are shown in bold. For example, the distance of the stone at $G4$ to goal $I3$ increases from three to nine, since when considering the position of the man, the shortest path taking backout conflicts into account is $G4$-$G3$-$F3$-$E3$-$D3$-$E3$-$F3$-$G3$-$H3$-$I3$.

The second enhancement is *linear conflicts* (LC) which increases the heuristic

value by two when a pair of adjacent stones is in the optimal path of each other: since for adjacent stones either the horizontal or the vertical movements are blocked, some move in the orthogonal direction must reduce the distance to some goal. Otherwise, the lower bound can be increased by two. In Figure 2.3a a linear conflict occurs between stones at $D3$ and $E3$.

Backout and linear conflicts preserve admissibility. When combined with MM we obtain the *enhanced MM* (EMM). EMM is the heuristic function used by RS* (JUNGHANNS; SCHAEFFER, 2001).

2.6.1.4 Pattern Databases for Sokoban

Deadlock tables and pattern searches, proposed by Junghanns and Schaeffer (2001), and other techniques that store the exact solution of subproblems (BOTEA; MÜLLER; SCHAEFFER, 2002), can be seen as precursors of PDBs in Sokoban. Deadlock tables exhaustively enumerate all possible configurations of a small rectangular area, and analyze, for each configuration, if the stones can be removed from the area. Configurations for which this is not the case are considered deadlocks. RS uses areas of five by four squares. The resulting 22 million entries are stored in a deadlock table. A deadlock table is independent of the instance and has to be computed only once. A state is a deadlock if a part of its configuration is found in the deadlock table. Pattern searches are computed by sub-searches for each instance during the main search trying to identify speculatively deadlocks and penalties to add to the heuristic value. These penalties are added in a non-admissible way (JUNGHANNS, 1999, p. 88) and thus can lead to non-optimal solutions.

However, these approaches miss important characteristics of PDBs like an abstract goal state and the computation of the distance of a set of abstract states to the abstract goal state by backwards search. They are also not designed to be used as an admissible heuristic function. For example, RS with deadlock tables and pattern searches, using as lower bound only the value found by pattern searches, is unable to find a solution for any of the instances of the standard set.

The seminal article of Edelkamp (2001) introduced PDBs to domain-independent planning. One of the test beds used was Sokoban on a set of 52 automatically generated, small instances. The author transformed Sokoban into a problem with an explicit goal state by mapping stones one-to-one to goals. In this way, some instances may have longer solutions or even become unsolvable. Compared to other optimal domain-independent planners using PDBs produced significantly better results. Haslum et al. (2007) improved

the application of PDBs in domain-independent planning by presenting an approach to select good abstractions automatically. Again Sokoban was used as a test bed, but without mapping stones to goal squares. They were able to solve 28 of 40 instances selected from "microban", which is considered an easy Sokoban test set. Recently Sievers, Ortlieb and Helmert (2012) introduced an efficient implementation of the approach proposed by Haslum et al. (2007) in the state-of-the-art domain-independent planner Fast Downward (HELMERT, 2006a).

2.6.2 Generalizations

The class of moving-blocks problems can be obtained by changing three components of the Sokoban problem: the goal, the type and restrictions of moves. The standard move is to push one stone to an adjacent free square. We can change the type of the move from push-only to pull-only, or allow to push and pull stones. We can relax the restriction of pushing at most one stone at a time and allow the man to push up to a fixed number k or an unlimited number (denoted by $*$) of stones at once. We can also restrict the moves to slide versions: a pushed stone slides until it hits an obstacle, and a pulled stone slides together with the man until the man hits an obstacle.

Two decision problems are associated with a problem. In the *storage* decision problem, each stone must be at a distinct goal square in the goal configuration. In the *path* decision problem, there is only one goal square, which the man must reach in the goal configuration. Stones do not have goal squares in this version. Formally we have:

Definition 2.6.1 (Moving-blocks problem)**.** *Given a maze with a set of k stones, the man with a type of move (push, pull or, push and pull) and restriction (1, k, $*$ and unit or slide), and a goal configuration, is there a sequence of moves that reaches the goal configuration?*

Definition 2.6.2 (Storage version of a moving-blocks problem)**.** *Given a moving-blocks problem with k stones and a set of k goal squares, is there a sequence of moves that places each stone at a distinct goal square?*

Definition 2.6.3 (Path version of a moving-blocks problem)**.** *Given a moving-blocks problem and a goal square, is there a sequence of moves that enables the man to reach the goal square?*

The literature uses the nomenclature MOVE-N, where MOVE stands for the type of the move and N stands for the number of stones that the man can move at once (DEMAINE; DEMAINE; O'ROURKE, 2000). The type of the move can be PUSH, PULL or PUSHPULL and the number of stones moved at once 1, k or $*$. MOVEMOVE problems have slide restriction. For example, PUSHPUSHPULLPULL is a problem with push and pull moves restricted to slide versions. We add the suffix -S for problems with a storage goal and the suffix -P for problems with a path goal. We omit suffix if a statement is valid for both problem variants. PUSH-1-S corresponds to Sokoban.

Taking in account all possibilities for these three components we end up with 36 different problems. One common generalization in the literature, not studied in this thesis, is to remove walls, leaving only movable blocks.

2.6.3 Examples

In this section, we present another problem that is part of the class moving-blocks problems. We also present two problems that are very similar to problems in the class of moving-blocks problems.

2.6.3.1 Pukoban

Pukoban is a moving-blocks problem similar to Sokoban. The main difference is the set of available actions. The man can also pull stones to adjacent free squares. In Sokoban, the state space is directed; thus deadlocks can be formed. In Pukoban, the state space is undirected; every action is reversible. Thus, deadlocks cannot be formed.

These additional actions make the two problems different – many techniques that are effective in Sokoban cannot be used in Pukoban. The four additional actions increase the branching factor, making the problem harder. However, the absence of deadlocks enable us to make other assumptions that could lead to new techniques. Pukoban corresponds to the problem PUSHPULL-1-S. In Chapter 3 we prove that Pukoban is PSPACE-complete.

There is no hard standard set of instances for Pukoban. However, since every instance of Sokoban is also solvable under the rules of Pukoban, the xSokoban set is used in the literature. Zubaran and Ritt (2011) proposed a solver based on A* for Pukoban. It uses several domain-dependent techniques. The heuristic function used is the MM.

The basic version of the solver using only the heuristic function and a tie breaking rule can solve 15 instances while a similar configuration in Sokoban can solve 10. They also propose a technique that is similar to linear conflicts. It computes penalties between two stones in a preprocessing phase. These penalties are added to the heuristic value during the search. The final version of their solver can increase the number of solved instances to 20.

2.6.3.2 Atomix

Atomix is a sliding-block problem defined on a maze grid. The maze is defined by walls, and there are k movable blocks (*atoms*). Different from Sokoban, the atoms are labeled, but there are atoms with the same label. The goal is to form a molecule i.e. a predefined pattern from the atoms. The actions available in Atomix could be understood as a PUSHPUSHPULLPULL without a man – a PUSHPULL problem with slide moves. Each atom can be moved up, down, right and left; it keeps on moving until it hits another atom or wall. The state space is directed. Thus, deadlocks are possible. Holzer and Schwoon (2004) show that Atomix is PSPACE-complete.

There is a standard set of 97 instances. Hüffner et al. (2001) developed a domain-dependent solver for Atomix based on A*. They investigate several heuristic search techniques and were able to solve optimally 35 instances. The main difficulty in developing a heuristic search approach to Atomix is to create an effective admissible heuristic function. The distances of atoms to the goals depends on the interaction with other atoms. Hüffner et al. (2001) propose a heuristic based on two ideas. They relaxed the capacity constraint of squares, allowing atoms to pass through other atoms. Additionally, they removed the slide constraint: atoms can stop at any desirable square. The distances of the relaxed problem are used as a heuristic function for the original problem. The proposed heuristic is the weakest point of the Hüffner et al. (2001) solver, and a better heuristic will likely improve their results. There are many similarities between moving-blocks problem and Atomix. We believe that some of our techniques could be used to improve the results in this problem – in particular the heuristic function.

2.6.3.3 Airport Ground Traffic Planning

Airport (TRUG; HOFFMANN; NEBEL, 2004) is a real-world problem introduced in the 4th International Planning Competition. It models the ground traffic planning of

airplanes. Each airplane must be moved from its original location to its destination. The problem is defined on a segment graph. There are k airplanes, m park segments and n runway segments. Each airplane is moved by a set of complex actions. The purpose of these actions is to model realistic safety conditions (HOFFMANN et al., 2006). In a more abstract view, these action can be understood as push and pull actions. There is a set of 50 instances that range from simple instances with one airplane to instances that model realistically the Munich Airport with 15 airplanes. A state-of-the-art heuristic can solve 38 of 50 instances optimally (HELMERT; DOMSHLAK, 2009). This problem is PSPACE-complete (HELMERT, 2006b).

3 MOVING-BLOCKS PROBLEMS ARE PSPACE-COMPLETE

In this chapter, we study the computational complexity of a PUSH and PUSHPULL problems. The chapter is organized as follows. In Section 3.1 we review the literature presenting the hardness results for moving-blocks problems. In Section 3.2 we prove that PUSH and PUSHPULL problems are PSPACE-complete using the Nondeterministic Constraint Logic presented in Subsection 3.1.1. In Section 3.3 we discuss the results and we conclude in Section 3.4.

3.1 Previous Works

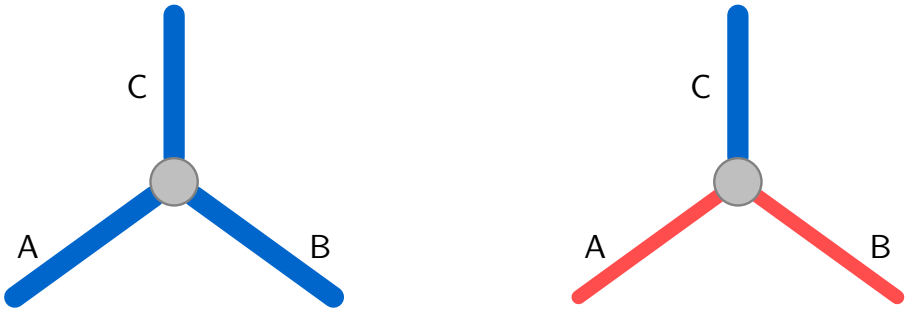
In this section we first introduce Nondeterministic Constraint Logic and, in Subsection 3.1.2, we review the previous hardness results for related moving-blocks problems.

3.1.1 Nondeterministic Constraint Logic

Nondeterministic Constraint Logic (NCL) is a framework developed by Hearn and Demaine (2005) to decrease the effort of proving PSPACE-hardness results. NCL is usually used to prove that puzzles and games (e.g. sliding-blocks puzzles and Rush Hour) are PSPACE-hard. It is based on a *constraint graph*. A constraint graph is an oriented graph with edge weights in $\{1, 2\}$. An edge with weight 1 is called *red* and an edge with weight 2 is called *blue*. Each vertex has a *minimum inflow* constraint of 2, i.e., the sum of the weights of inward-directed edges must be at least 2. A move on the constraint graph is to reverse an edge orientation resulting in a valid configuration.

The framework has two base components: *OR* and *AND* vertices. Figure 3.1(a) shows an *OR* vertex. It has three blue edges, A, B and C, and behaves similar to a logical *OR*. A blue edge can be directed outward if and only if at least one of the other two blue edges is directed inward. Figure 3.1(b) shows an *AND* vertex. It has two red edges, A and B, and one blue edge C and behaves similar to a logical *AND*. The blue edge can be directed outward if and only if both red edges are directed inward. Different from logical *OR* and *AND* gates, in NCL there is no notion of inputs and outputs. For example, in an *AND* vertex when the blue edge is directed inward, both red edges can be directed

Figure 3.1: Basis vertices of NCL.



(a) *OR* vertex. (b) *AND* vertex.

outward.

There are two basic decision problems in the NCL framework. In the *Configuration–Configuration* version we have to decide if there is a sequence of moves that transforms a given constraint graph in configuration A into a configuration B. In the *Configuration–Edge* version we have to decide if there is a sequence of moves on a given constraint graph that terminates reversing a given edge e. Hearn and Demaine (2005) prove the PSPACE-hardness of both problems by reduction from the PSPACE-complete problem Quantified Boolean Formulas (GAREY; JOHNSON, 1979).

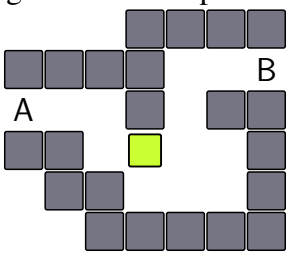
In the NCL framework every constraint graph is equivalent to a planar constraint graph. Thus, to show that a particular problem is PSPACE-hard one needs to show how to construct *OR* and *AND* gadgets in the problem of interest and how to connect them into an arbitrary planar constraint graph.

3.1.2 Previous Hardness Results for Moving-Blocks Problems

Dor and Zwick (1999) introduced the family of moving-blocks problems based on Sokoban and proved that PUSH-1-S is NP-hard by giving a polynomial time reduction from the Satisfiability Problem (SAT). They also proved that PUSHPULL-k-S for k greater or equal to five is NP-hard by a reduction from SAT. They leave open the question if Sokoban is PSPACE-complete or NP-complete.

Culberson (1999) solved the open problem posed by Dor and Zwick (1999). He was able to prove that the problem PUSH-1-S or Sokoban is PSPACE-complete showing how to construct a Sokoban problem corresponding to a linear space-bounded Turing machine. The central idea of the proof is the use of the unrecoverable configurations present in Sokoban to build the gadgets. The unrecoverable configurations are used in the con-

Figure 3.2: One way passage gadget used in the proof of PSPACE-hardness of Sokoban.



struction of the gadgets to restrict movements of the man. Some moves can be declared prohibited even if the move is technically allowed, since no solution of the problem contains such moves. Figure 3.2 shows one gadget used in the proof with that property. In this gadget, the man can only pass from A to B, but not from B to A: the only possible move of the stone to the left results in an unrecoverable configuration.

In a series of articles Demaine, Demaine and O'Rourke (2000), Demaine, Hearn and Hoffmann (2002), Demaine et al. (2003), Demaine, Hoffmann and Holzer (2004) proved complexity results for several moving-blocks problems considering the path decision problem. Demaine, Demaine and O'Rourke (2000) proved the NP-hardness of PUSH-1-P and PUSHPUSH-1-P, and both reductions are from SAT. Demaine, Hearn and Hoffmann (2002) proved that PUSH-k-P with k greater or equal than two and PUSH-$*$-P are PSPACE-complete using a reduction from NCL. Demaine et al. (2003) proved that PUSHPUSH-k-P and PUSHPUSH-$*$-P are NP-hard with reductions from planar 3-coloring and SAT, respectively. Later, they proved that PUSHPUSH-k-P is PSPACE-complete (DEMAINE; HOFFMANN; HOLZER, 2004).

Ritt (2010) proved NP-hardness for several generalizations with PULL moves considering the path decision problem. Using a reduction from planar 3-coloring he proved that PULL-1-P, PULL-k-P, PULL-$*$-P, PULLPULL-1-P, PULLPULL-k-P, and PULLPULL-$*$-P are NP-hard.

Table 3.1 shows the complexity of some of these problems. The results for problems PUSH-$\{k, *\}$-P, PULL-$\{k, *\}$-P, PULLPULL-$\{1, k, *\}$-P, and PUSHPULL-k-$\{P, S\}$ require walls. The remaining results are stronger, since they do not depend on walls, but imply the hardness of the corresponding variant with walls (RITT, 2010).

3.2 Moving-Blocks Problems are PSPACE-complete

In this section, we prove that several moving-blocks problems with PULL and PUSHPULL moves are PSPACE-complete by reduction from NCL.

Table 3.1: Hardness results for moving-blocks problems. Problems hard for PSPACE are also complete for PSPACE since all problems are in PSPACE.

Problem	Storage		Path	
	Hard for	Reference	Hard for	Reference
PUSH-1	PSPACE	Culberson (1999)	NP	Demaine and Hoffmann (2001)
PUSH-k with $k \geq 2$	Open		PSPACE	Demaine, Hearn and Hoffmann (2002)
PUSH-$*$	Open		PSPACE	Demaine, Hearn and Hoffmann (2002)
PUSHPUSH-1	NP	O'Rourke *et al.* (1999)	PSPACE	Demaine, Hoffmann and Holzer (2004)
PUSHPUSH-k	Open		PSPACE	Demaine, Hoffmann and Holzer (2004)
PUSHPUSH-$*$	Open		NP	Demaine, Hoffmann and Holzer (2004)
PULL-1	Open		NP	Ritt (2010)
PULL-k	Open		NP	Ritt (2010)
PULL-$*$	Open		NP	Ritt (2010)
PULLPULL-1	Open		NP	Ritt (2010)
PULLPULL-k	Open		NP	Ritt (2010)
PULLPULL-$*$	Open		NP	Ritt (2010)
PUSHPULL-1	Open		Open	
PUSHPULL-k with $k \geq 5$	NP	Dor and Zwick (1999)	NP	Dor and Zwick (1999)
PUSHPULL-$*$	Open		Open	
PUSHPUSHPULLPULL-1	Open		Open	
PUSHPUSHPULLPULL-k	Open		Open	
PUSHPUSHPULLPULL-$*$	Open		Open	

Figure 3.3: Blocks used to build the gadgets, in order from left to right: red stone, blue stone, yellow stone, highlighted square and wall.

The gadgets are built using five basic blocks (Figure 3.3). Red, blue and yellow stones are "triggers" whose motion serves to satisfy the vertex constraint. Blue and red stones represent blue and red edges. Yellow stones are used in the interior of the gadgets and do not correspond to elements of the NCL. Highlighted squares are used to specify some set of squares related to the transition of the state of an edge. Dark gray blocks are walls. Finally, the man is not shown since the gadgets below are designed such that the man can freely move to any empty (white) square in all valid configurations.

Definition 3.2.1 (Input)**.** *An input configuration is given by a $n \times m$ matrix where each cell is marked free, wall, stone, goal or man.*

An input configuration can be stored using three bits for each cell. Different configurations resulting from moves of the man or stones can still be stored using the same space. All problems considered in this thesis are in PSPACE. A configuration and the total number of configurations can be represented in polynomial space. So the problem is in PSPACE, since we can non-deterministically execute a sequence of valid moves up to an upper bound on the number of configurations by using a counter. Containment in

PSPACE then follows from Savitch (1970)'s theorem.

3.2.1 PULL Problems are PSPACE-complete

In this subsection, we prove that PULL-$\{1, k, *\}$-S and PULLPULL-$\{1, k, *\}$-$\{P, S\}$ are PSPACE-complete. We show how to build *OR* and *AND* gadgets, and how to connect them into an arbitrary planar constraint graph. We demonstrate that the gadgets maintain the constraints required to emulate NCL. First, we give the constructions for the PULL-1-S problem – the dual problem of Sokoban. Later, we show that the same set of gadgets can be employed to demonstrate PSPACE-completeness of a broader set of problems with pull-only moves.

PULL problems have a common kind of unrecoverable configuration when the man gets trapped in the interior of the gadget. There are some moves that are reversible in isolation, but due to interactions with other stones the man can get trapped, surrounded by stones and walls, and be unable to reverse the move. This type of unrecoverable configuration will be central to our proof in order to maintain the inflow constraint at each gadget. When considering problems with the storage goal we construct the gadgets such that each stone can be pulled to only one goal square. This leads to another unrecoverable configuration, where a stone may be unable to reach its goal position. These additional restrictions facilitate the construction of the gadgets, compared to the path versions. The storage decision problem directly corresponds to the *Configuration–Configuration* decision problem of NCL. Each edge has a defined direction in the goal configuration as each stone has a defined goal square in our construction.

We show an NCL *OR* vertex gadget of a PULL-1-S problem in Figure 3.4. In the vertex gadget, stones A and B represent outward-directed edges, and the stone C represents an inward-directed edge. A switches state being pulled two units left, B two units right, and C two units down.

Lemma 3.2.1. *The construction of Figure 3.4 satisfies the constraints of an NCL* OR *vertex.*

Proof. We need to show that C may switch state if and only if A or B switch state first. If C switches state without first A or B switching state, then the gadget will be in an unrecoverable configuration, since the man must pull C two units down and will be trapped in the interior of the gadget. Other moves to change the state of C are possible, like pulling

Figure 3.4: PULL *OR* gadget.

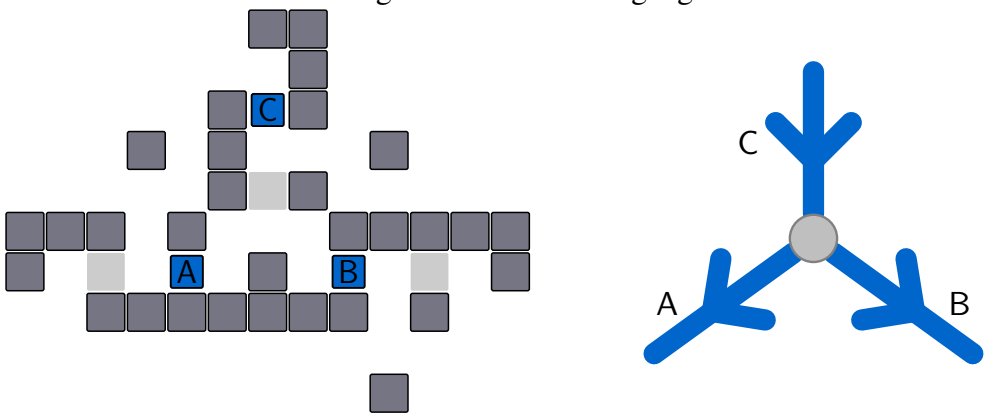
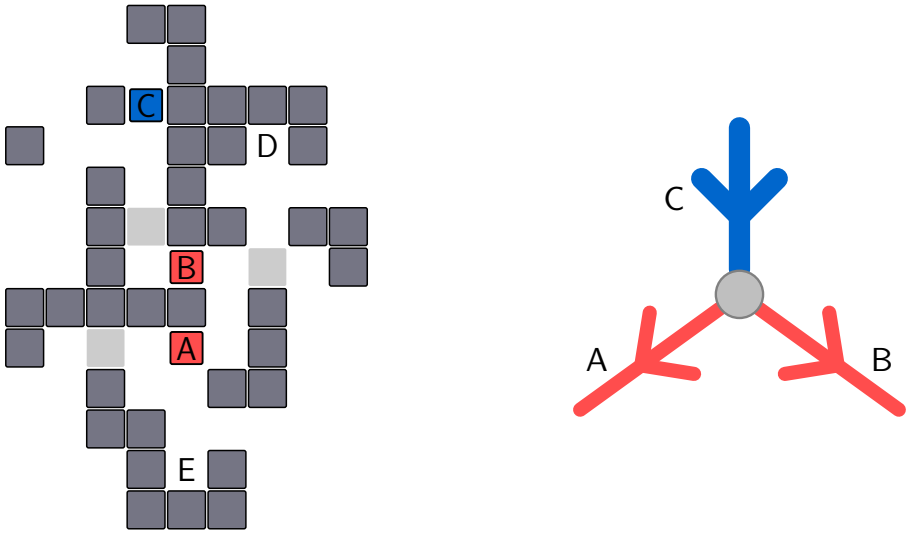


Figure 3.5: PULL *AND* gadget.



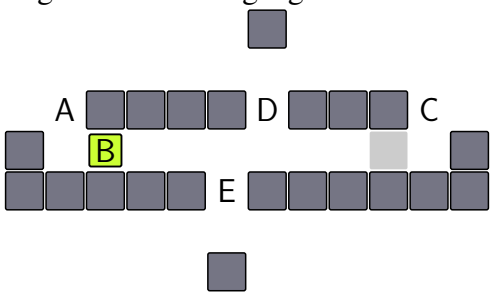
A one unit left and one unit up, but these moves lead to an unrecoverable configuration. A symmetric reasoning is also valid for A and B. □

Next, we construct an NCL *AND* vertex gadget of the PULL-1-S problem in Figure 3.5. In the gadget, stones A and B represent outward-directed red edges, and stone C represents an inward-directed blue edge. Stone B switches state by moving two units right, and stone A by moving two units left. Stone C switches state by moving three units down.

Lemma 3.2.2. *The construction of Figure 3.5 satisfies the constraints of an NCL* AND *vertex.*

Proof. We need to show that C may switch state if and only if A and B switch state first. If C switches state without A and B changing state, then the gadget will be in an unrecoverable configuration, since the man will be trapped, surrounded by stones B and C, and walls. If only A changes state then we still cannot switch C. If only B switches

Figure 3.6: PULL gadget connection.

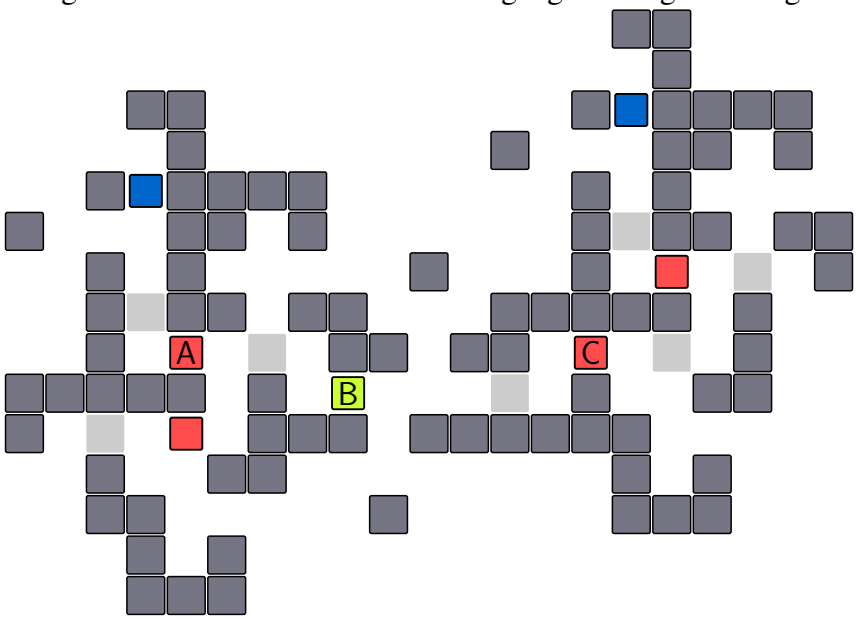


state, and then C switches state, again the gadget will be in an unrecoverable configuration and now stone A will block the path of the man. If C is outward-directed, and A or B changes state, the gadget will be in an unrecoverable configuration. When C is inward-directed, A and B may freely change state. There are other moves that would enable C to change state like moving A one unit down, but these movements are non-reversible. □

We showed how to construct *OR* and *AND* vertices for PULL-1-S problems. We now show how to connect the vertices into an arbitrary planar constraint graph. The basic idea is to connect gadgets using a tunnel. We use the tunnel of Figure 3.6 to propagate the "signal" between two vertices that share an edge. Figure 3.6 is constructed from partial terminal positions of stones in *OR* and *AND* gadgets. Stone B switches state by being pulled to the highlighted square. For B to switch state the path through C must be open. After B changes state, the path through A is open. Pulling B one unit right is an intermediate position and does not have an effect in the gadgets. The orthogonal path (D and E) serves only to guarantee global access to the man. Without orthogonal paths the man could be locked in a cycle. Squares D and E could be used to store stone B and open paths from A and C at the same time, but since this move is non-reversible it is prohibited.

Figure 3.7 shows an example of a connection between *AND* gadgets through a red edge. Stones A, B and C represent a single red edge from the NCL. To change the state of A, first B must be pulled, but for this first C must be pulled to the right. This propagates the "signal" between vertices. After C is moved to the right, A and C will block their respective gadgets. At this moment, the edge is outward-directed in both gadgets and other edges would be responsible to maintain the inflow constraint. This intermediate state does not give more freedom to the constraint graph. At each step the edge will be inward-directed to only one gadget.

Figure 3.7: Connection of two *AND* gadgets using a red edge.



Theorem 3.2.1. PULL-1-S *is PSPACE-complete.*

Proof. By reduction from the *Configuration–Configuration* version of NCL. Given a planar *OR* and *AND* constraint graph, we build a PULL-1-S problem as described above, corresponding to the initial configuration of the constraint graph. We place a goal square for each corresponding edge direction of the goal configuration of the constraint graph. By the construction of the gadgets, the constructed problem has a solution if and only if the original NCL problem has a solution. □

Corollary 3.2.1. PULL-k-S *and* PULL-$*$-S *are PSPACE-complete.*

Proof. The man has access to only one stone at a time, and there is no sequence of moves that leaves two stones in contact. Thus, giving more strength to the man does not change the functionality of the gadgets. □

Corollary 3.2.2. PULLPULL-1-S, PULLPULL-k-S *and* PULLPULL-$*$-S *are PSPACE-complete.*

Proof. The sliding versions do not lead to new configurations, and all moves needed by the gadget functionality can be performed by slide moves. □

We now turn to the path decision problem. In the path decision problem, we do not have goal squares for stones. Thus, we are not able to prohibit some non-reversible moves that could break the gadgets. However, when only slide moves are allowed the same set of gadgets can be used to prove PSPACE-completeness for path versions.

In the *OR* gadget only moves that switch the state of edges are possible, and no moves that could lead to an invalid configuration can be performed. The same is valid for the connection gadget. In the *AND* gadget, pulling stone B up or stone A down would lead to an invalid configuration, leaving the man trapped in position D or E. Using slide moves the man is not able to destroy the gadgets.

In the path version, the problem has one single goal square, which the man must reach. Thus it makes sense to use the *Configuration–Edge* decision problem of NCL in the reduction. We place the goal square at the initial square of the stone that corresponds to the edge that must be reversed. With this construction, the man can reach the goal square if and only if the stone switches state.

Theorem 3.2.2. PULLPULL-1-P *is PSPACE-complete.*

Proof. By reduction from *Configuration–Edge* version of NCL. Given a planar *OR* and *AND* constraint graph, we build a PULLPULL-1-P problem as described above, corresponding to the initial configuration of the constraint graph. We place a goal square at the initial position of the stone that corresponds to the edge that must be reversed. The man can reach the goal square if and only if the edge is reversed. □

Corollary 3.2.3. PULLPULL-k-P *and* PULLPULL-$*$-P *are PSPACE-complete.*

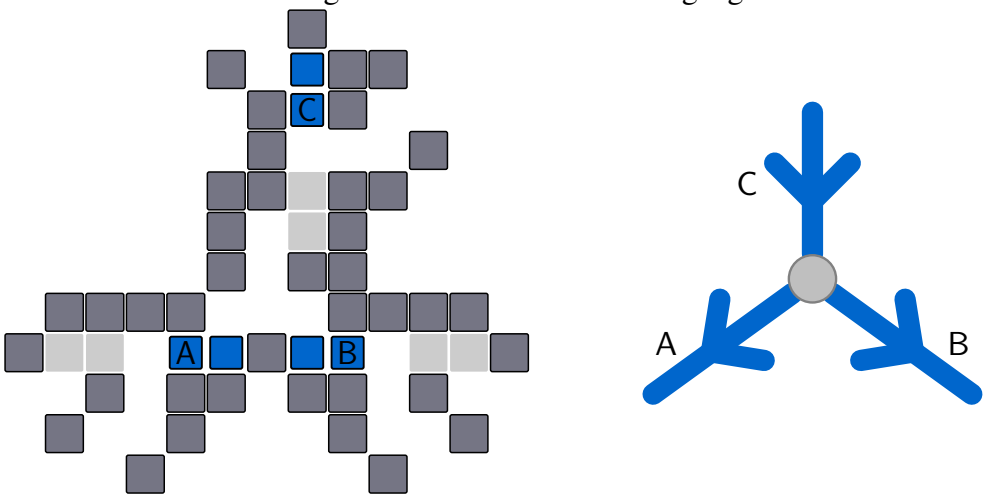
Proof. The man has access to only one stone at a time, and there is no sequence of moves that leaves two stones in contact. □

3.2.2 PUSHPULL Problems are PSPACE-complete

In this subsection, we prove that all PUSHPULL problems are PSPACE-complete. We show how to construct *OR* and *AND* gadgets, and how to connect them into an arbitrary planar constraint graph. In PULL problems we have a single set of *OR*, *AND* and connection gadgets used in all proofs. In PUSHPULL problems we have two sets of *AND* and connection gadgets, a specific set for PUSHPULL-1 and its slide versions, and another set for PUSHPULL-$\{k, *\}$ and its slide versions. All proofs use the same *OR* gadget. We start with a general discussion, then present the proofs for PUSHPULL-1 and PUSHPULL-$\{k, *\}$.

PUSHPULL problems do not have unrecoverable configurations. Every move is reversible: a push move can be undone through a pull move and *vice versa*. In PULL prob-

Figure 3.8: PUSHPULL-1 *OR* gadget.



lems, we use non-reversible moves to prohibit some moves, while in PUSHPULL problems this is not possible. The absence of unrecoverable configurations makes it harder to construct the gadgets. Because of this, using the storage decision problem also does not help to construct the gadgets. Every gadget that we use to prove that a PUSHPULL path version is PSPACE-complete can also be used to prove that the equivalent storage version is PSPACE-complete. This is done by changing the decision problem.

Another difference of the gadgets when compared to the ones of PULL problems is that we use two or more stones for every edge. With push and pull moves the man is able to store stones in squares that could not be used before. Therefore, we use more stones to fill the space and guarantee that always at least one stone is blocking the main path in the gadget. We will refer to every set of adjacent stones by the single label in one of them.

The unrecoverable configurations, when the man gets trapped in the interior of the gadgets, were central to our proof for PULL problems. The same idea is central to our proof for PUSHPULL problems. In the gadgets when the man makes a move and gets trapped in the interior of the gadget he cannot change the state of any other edge. Moreover, he cannot complete the sequence of actions that changes the state of the edge, without first undoing the last move. Therefore, in every configuration in which the man is able to switch the state of an edge, the global inflow constraint is satisfied.

We show an NCL *OR* vertex gadget for the PUSHPULL-1 problem in Figure 3.8. The *OR* gadget for PUSHPULL is similar to the *OR* gadget for PULL. A, B and C switch state being moved to the extreme opposite position. For A to switch state for example, first, the labeled stone must be pulled two units left and then pushed one unit left. Next, the second stone must be pulled one unit left and then pushed two units left.

Lemma 3.2.3. *The construction of Figure 3.8 satisfies the constraints of an NCL* OR *vertex.*

Proof. We need to show that C may switch state if and only if A or B switch state first. When the man tries to switch the state of C, without first A or B switching state, the only move allowed is to pull the first stone one unit down. This does not help since the man would be locked in the interior of the gadget and would have to undo the move. For C to switch state, he must first switch state of A, then C may switch state. Similarly, for B. A symmetric reasoning is also valid for A and B. Other moves are possible, e.g. pulling B one unit right and then one unit down. However, this does not help since the other stone will continue blocking the path. □

Next, we show an NCL *AND* vertex gadget for the PUSHPULL-1 problem in Figure 3.9. Stones A, B and C switch state moving to the maximum possible opposite position. For example, for the edge A to switch state, the first stone could be pulled two units left and then pushed one more unit left. Next, the second stone could be moved similarly. When C is inward-directed, the red edges may freely switch state as in an NCL *AND* vertex.

We use two sets of yellow stones D and E to enforce the restriction of the gadget and to act as a buffer that makes the red edges independent. To C switch state, first red edges A and B must be inward-directed by moving A to the left and B to the right. The next step is to move D and E. To D change state the three stones must be placed on the highlighted squares, but this is only possible if the stones E change state first. In this way, an order is enforced to move the yellow edges.

Lemma 3.2.4. *The construction of Figure 3.9 satisfies the constraints of an NCL* AND *vertex.*

Proof. We need to show that C may switch state if and only if A and B switch state. If C switches state without first A and B switch state, then the man will be trapped in the interior of the gadget. To C change state first D and E must change state. The only way to open the path to switch the state of C is to place D and E in their respective highlighted squares. To D and E change state, A and B must switch state. □

We showed how to construct *AND* and *OR* vertices for PUSHPULL-1. It remains to show how to connect the vertices into arbitrary planar graphs. The basic idea shown in Figure 3.10 is to connect the gadgets using tunnels with three stones that propagate

Figure 3.9: PUSHPULL-1 *AND* gadget.

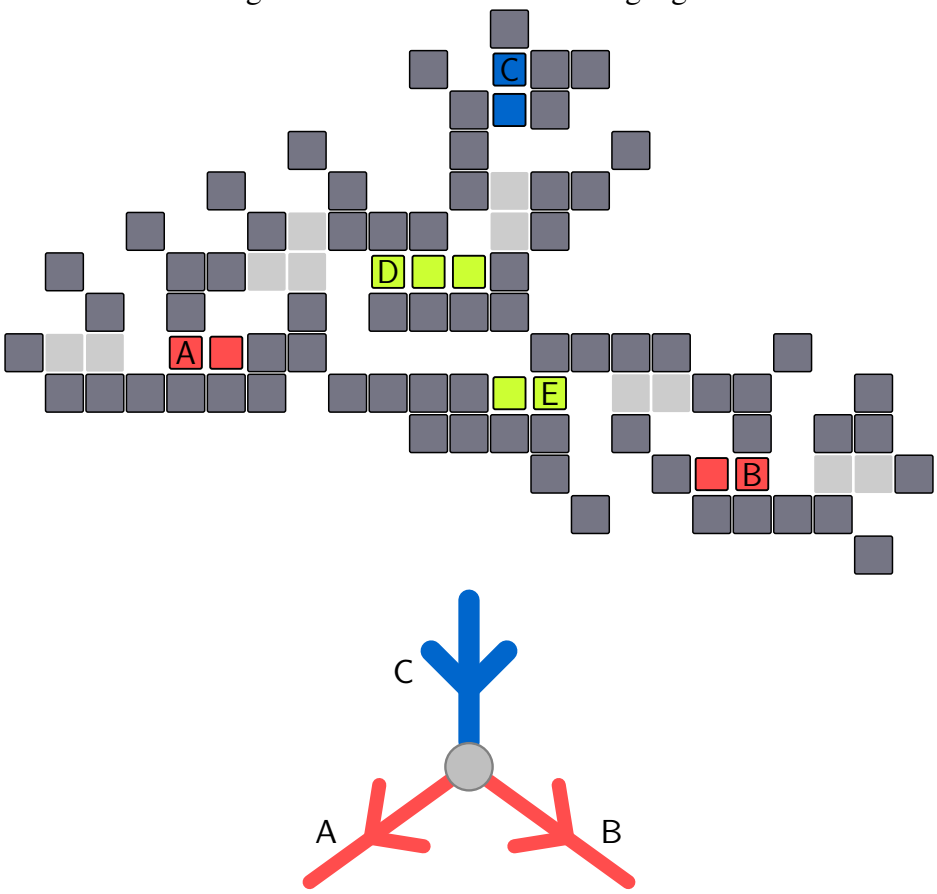
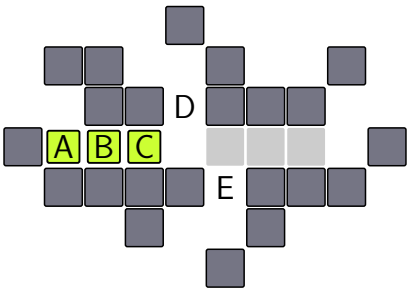


Figure 3.10: PUSHPULL-1 connection.



the "signal" between gadgets. The three stones switch state moving to the highlighted squares on the right. Three stones are necessary because two stones could be stored in the orthogonal path, at squares D and E, but using three stones, at least one will block the path in the tunnel.

Theorem 3.2.3. PUSHPULL-1 *is PSPACE-complete.*

Proof. By reduction from the *Configuration–Edge* version of NCL. Given a planar *AND* or *OR* constraint graph, we build a PUSHPULL-1-P problem as described above, corresponding to the initial constraint graph configuration. We place the goal square at the initial square of the labeled stone that corresponds to the edge that must reverse. The man

can reach the goal square if and only if the edge is reversed. Similar for a PUSHPULL-1-S problem, by reduction from the *Configuration–Configuration* version of NCL, we place a set of goal squares for each corresponding edge direction of the goal configuration of the constraint graph. □

Corollary 3.2.4. PUSHPUSHPULLPULL-1 *is PSPACE-complete.*

Proof. All moves necessary to the gadget functionality can be performed by slide moves, with the exception of the connection gadget in Figure 3.10. In the gadget, the man must use squares D or E to switch the state, and using only slide moves he cannot perform these actions. We remove stone A and the leftmost highlighted square maintaining stones B and C in the squares shown in the gadget. Then, the man can pull C to the right and later push B to the right. Since the man cannot store stones on squares D and E, two stones are sufficient to guarantee that at least one of them must be pulled, ensuring the properties of the gadget. □

The gadgets used to prove that PUSHPULL-1 is PSPACE-complete cannot be used to prove that PUSHPULL-k and PUSHPULL-$*$ are PSPACE-complete. For example, in the connection gadget from Figure 3.10 with $k = 3$, the man can pull the three stones one unit right and then push them further to the right. This would open the path to the gadget connected at the left without enforcing that the gadget connected to the right changes state. A similar situation occurs in the *AND* gadget. The blue edge can switch state without the left red edge be inward-directed. With $k = 3$, we change the state of D without switching state of A. The *OR* gadget can be used for k and $*$ versions since the first pull move already locks the man inside of the gadget. Thus, we have to show how to construct the *AND* and the connection gadget for the k and $*$ versions.

We show an NCL *AND* vertex gadget using the PUSHPULL-$\{k, *\}$ problem in Figure 3.11. This gadget is similar to a PULL *AND* gadget. In the gadget, the stones A and B represent outward-directed red edges, and the two stones C represent an inward-directed blue edge. Stones A, B and C switch state moving to the highlighted squares. Note that for B to switch state independently of A the man must pull two stones at once. Thus, this gadget cannot be used in the proof of PUSHPULL-1.

Lemma 3.2.5. *The construction of Figure 3.11 satisfies the constraints of an NCL* AND *vertex.*

Proof. We need to show that C may switch state if and only if A and B switch state. If C switches state without first A and B switching state, then the man will be locked in the

Figure 3.11: PUSHPULL-$\{k, *\}$ *AND* gadget.

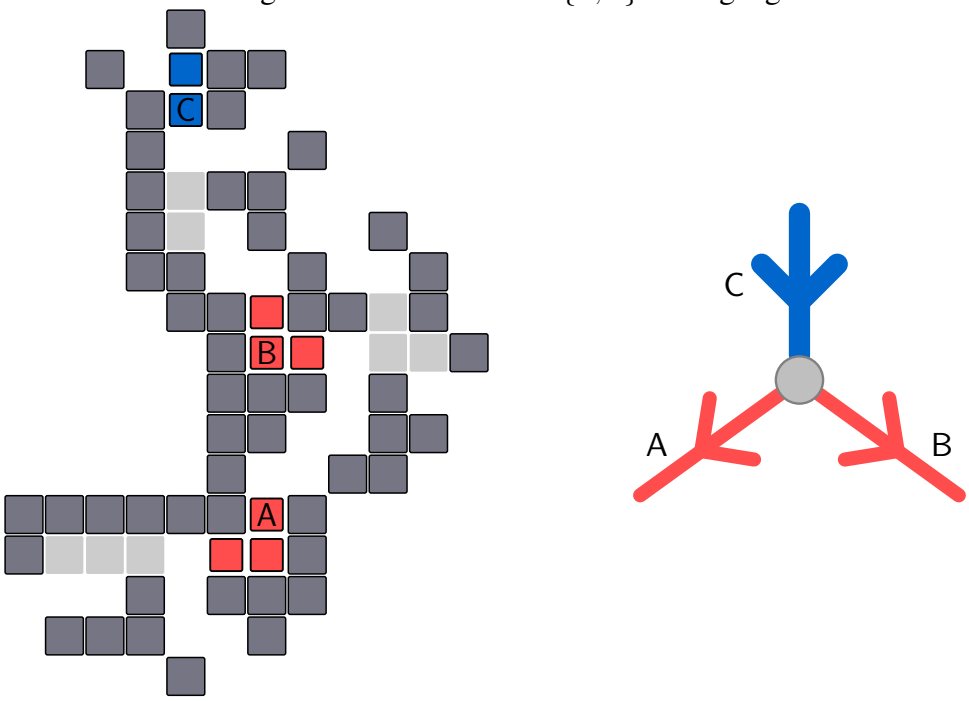
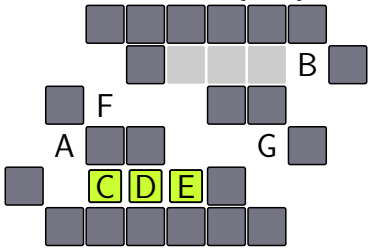


Figure 3.12: PUSHPULL-$\{k, *\}$ connection.



interior of the gadget. If only B switches state and then C switches state, the man will be locked by A. We must first switch the state of A and B. Then, C may switch state using the path opened, and the man may leave the interior of the gadget. □

Figure 3.12 shows the connection gadget. Similar to the one used for PUSHPULL-1 it uses the same three stones, has the same type of orthogonal path, and end connections. The main difference is that the gadget enforces that the man uses paths from A and B to switch state of the three yellow stones. Thus, the man must first switch the state of the gadget connected to B to change the state of the stones. In the same way squares D and E could be used to store stones, but at least one stone will block the path of the tunnel.

We describe in detail how to switch the state of the connection gadget using only slide moves. First, E must be pulled and then pushed up. Next, C and D can be pushed to right and D pulled up and to the right. Then, C can be pushed to the right, pulled and pushed up. Then, D can be pushed back to the left. In this point the three stones form a column from top to bottom E, C and D. Now E can be pulled to the right and C and D

can be pushed up. E can be pushed to the left leaving C and E in contact, then C and E can be pulled together to the right. Finally, D can be pushed up.

Theorem 3.2.4. PUSHPULL-$\{k, *\}$ *is PSPACE-complete.*

Proof. By reduction from NCL *Configuration–Edge*. Given a planar *AND* and *OR* constraint graph, we build a PUSHPULL-$\{k, *\}$-P problem as described above, corresponding to the initial constraint graph configuration. We place a goal square at the initial position of the labeled stone that corresponds to the edge that must reverse. The man can reach the goal square if and only if the edge is reversed. Similar for a PUSHPULL-$\{k, *\}$-S problem, by reduction from the *Configuration–Configuration* version of NCL, we place a set of goal squares for each corresponding edge direction of the goal configuration of the constraint graph. □

Corollary 3.2.5. PUSHPUSHPULLPULL-$\{k, *\}$ *is PSPACE-complete.*

Proof. All moves necessary to the gadget functionality can be performed by slide moves. □

3.3 Discussion

We have shown that several PULL problems are PSPACE-complete. Thereby, most of the versions with PULL moves are as hard as the PUSH versions. We were able to show the PSPACE-completeness for the whole class of PUSHPULL problems. Table 3.2 shows the new complexity results obtained in this work.

We were able to show that PULLPULL-$*$-P is PSPACE-complete. The equivalent PUSH version is known to be NP-hard. We also were able to show that PUSHPULL-1-P is PSPACE-complete while PUSH-1-P and PULL-1-P are NP-hard. The gadgets used to prove the PSPACE-hardness of PUSHPULL-1-P are more complex than those used in more relaxed versions of the problems. This reinforces the evidence that proving PSPACE-completeness of PUSH-1-P and PULL-1-P is harder than proving the same hardness results for k and $*$ versions.

Our results only hold for versions with walls. We chose to analyze these variants, since they still are a large class of relevant problems, and walls allow simpler gadgets and proofs. In problems with only movable blocks, the gadgets would need additional components to restrict moves. Moreover, using only movable blocks would not allow us to work with problems where the man can push or pull an unlimited number of stones.

Table 3.2: Hardness of moving-blocks problems with results of this work shown in boldface. Problems hard for PSPACE are also complete for PSPACE since all problems are in PSPACE.

Problem	Storage		Path	
	Hard for	Reference	Hard for	Reference
PUSH-1	PSPACE	Culberson (1999)	NP	Demaine and Hoffmann (2001)
PUSH-k with $k \geq 2$	Open		PSPACE	Demaine, Hearn and Hoffmann (2002)
PUSH-$*$	Open		PSPACE	Demaine, Hearn and Hoffmann (2002)
PUSHPUSH-1	NP	O'Rourke *et al.* (1999)	PSPACE	Demaine, Hoffmann and Holzer (2004)
PUSHPUSH-k	Open		PSPACE	Demaine, Hoffmann and Holzer (2004)
PUSHPUSH-$*$	Open		NP	Demaine, Hoffmann and Holzer (2004)
PULL-1	**PSPACE**	**This work**	NP	Ritt (2010)
PULL-k	**PSPACE**	**This work**	NP	Ritt (2010)
PULL-$*$	**PSPACE**	**This work**	NP	Ritt (2010)
PULLPULL-1	**PSPACE**	**This work**	**PSPACE**	**This work**
PULLPULL-k	**PSPACE**	**This work**	**PSPACE**	**This work**
PULLPULL-$*$	**PSPACE**	**This work**	**PSPACE**	**This work**
PUSHPULL-1	**PSPACE**	**This work**	**PSPACE**	**This work**
PUSHPULL-k	**PSPACE**	**This work**	**PSPACE**	**This work**
PUSHPULL-$*$	**PSPACE**	**This work**	**PSPACE**	**This work**
PUSHPUSHPULLPULL-1	**PSPACE**	**This work**	**PSPACE**	**This work**
PUSHPUSHPULLPULL-k	**PSPACE**	**This work**	**PSPACE**	**This work**
PUSHPUSHPULLPULL-$*$	**PSPACE**	**This work**	**PSPACE**	**This work**

One could think that a simple duality from PUSH-S problem is enough to prove PSPACE-completeness for PULL-S problems. Since in PULL-S problems every stone has a goal square we could place every stone at a goal square and solve the dual problem with push moves, and thus answering yes if the dual problem has a solution. In fact, PULL-S has goal squares for stones, but not for the man. We could solve the dual problem for every possible position of the man and answering yes if one of them has a solution. However, this ignores that the position of the man matters. For some positions of the man, the instance may have a solution and for others not. Thus, a simple duality is not enough to prove PSPACE-completeness.

3.4 Conclusion and Future Work

We have shown the PSPACE-completeness of a broad class of moving-blocks problems. In particular, we improve the known NP-hardness results of PULL problems to PSPACE-completeness results. We also were able to show the PSPACE-completeness of the whole class of PUSHPULL problems.

In further research the complexity of PULL-k-P, PULL-$*$-P, PUSHPUSH-$*$-S,

and PUSHPUSH-$*$-P remains to be studied. After that the knowledge gap in Table 3.2 would be filled. Another possibility is to extend the results presented here to versions of the problems without walls.

Two important research questions are still open. First, the exact complexity of PUSH-1-P and PULL-1-P. This would enhance known complexity results of several problems. Second, is there an interesting but still tractable moving-blocks problem?

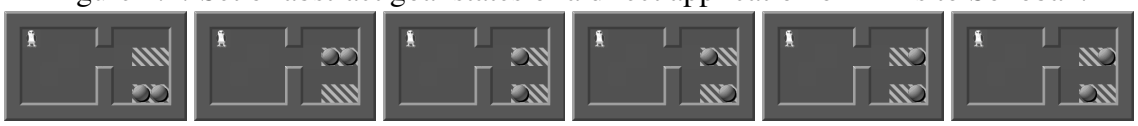
4 SOLVING OPTIMALLY MOVING-BLOCKS PROBLEMS WITH HEURISTIC SEARCH

In this chapter, we propose a heuristic search approach to optimally solve Sokoban and Pukoban. The chapter is organized as follows. In Section 4.1 we present the main contribution of this chapter the heuristic function based on the idea of applying PDBs to intermediate goal states. In the Subsection 4.1.5, we present a standard application of PDBs to Sokoban. In Section 4.2, we describe a domain-dependent tie breaking rule. Next, in the Section 4.3, we present and discuss the computational results considering Sokoban. Section 4.4 discusses how the techniques previously presented can be used in Pukoban and present computational results. In Section 4.5, we discuss how heuristic distances to intermediate goal states could be used in other problems. Finally, in Section 4.6 we discuss results and future work.

4.1 Heuristic Functions Based on Pattern Databases

Consider a direct application of PDBs to Sokoban. A natural abstraction is to keep only k' of all k stones. This is admissible, since the cost of solving a subset of stones never exceeds the cost of solving the whole instance. We can construct a PDB which stores the shortest distance to the nearest abstract goal state. Since in Sokoban the goal state is defined implicitly, every placement of the k' stones on the k goal squares is an abstract goal state. Figure 4.1 shows the abstract goal states for a toy instance with $k = 4$ and $k' = 2$. We call this PDB the *multiple goal state PDB* (MPDB).

Figure 4.1: Set of abstract goal states of a direct application of PDBs to Sokoban.



(a) (b) (c) (d) (e) (f)

It turns out that MPDBs are ineffective as heuristic function. In this chapter we propose a different approach which introduces the idea of an intermediate goal state. We use PDBs to find a good heuristic value for the number of pushes necessary to reach the intermediate goal state, and EMM to estimate the number of pushes from there to a goal state. We compare the resulting heuristic with EMM, with MPDBs, and with the solvers RS* and Fast Downward using PDBs.

We show that applying PDBs to intermediate goal states can solve optimally more problem instances in less time. On initial states the proposed heuristic finds 62 better results and over a set of random initial states detects four times more deadlocks when compared to EMM. We increase the number of instances solved optimally from 10 to 20. All of them were solved using considerably fewer explored nodes and computation time than previous methods.

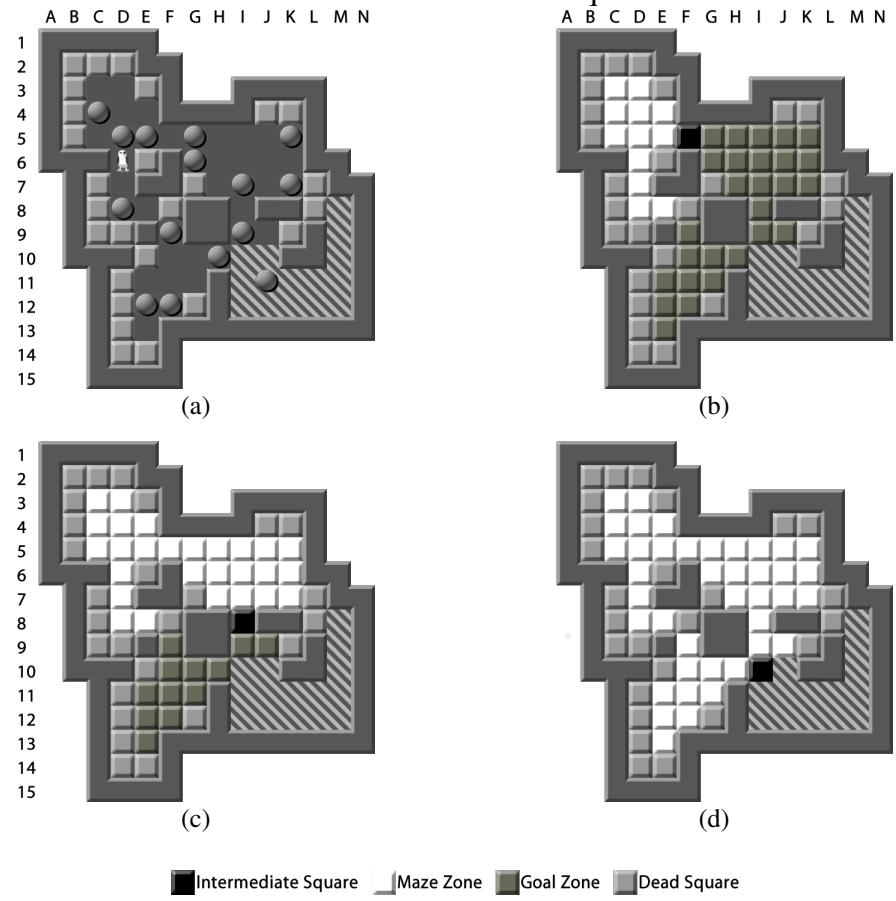
In this section, we describe how instances can be decomposed into a maze zone and a goal zone and define an intermediate goal state, explain the construction and storage of the PDB, introduce a new heuristic, and show how to compute it efficiently. We also demonstrate its consistency.

4.1.1 Instance Decomposition

Intuitively, if every stone has to pass over some fixed intermediate square to reach any of the goal squares, we can decompose an instance into two subproblems. The first is to bring all stones to the intermediate square, and the second is to move them from there to the goal squares. This is a relaxation of the original problem and thus cannot be used to solve it directly, but we can solve the two subproblems separately to obtain a lower bound on the length of an optimal solution. The main advantage of this approach for Sokoban is that it enables us to effectively apply PDBs to the first subproblem, since this subproblem has a single, well-defined goal state. For the second subproblem any heuristic can be used.

A sequence P of squares is a *stone-path* from square a to square b with man at square m, if there exists a solution of an instance with a single stone at a, a single goal square at b and the man initially at m such that the stone visits exactly the squares in P. Consider a fixed square c. We call a square a a *maze square*, if for all goal squares b and all positions of the man m all stone-paths from a square a to b with man at m contain c. The set of all maze squares forms the *maze zone*. Note that, by definition square c is part of the maze zone. All other non-dead and non-goal squares are called *goal zone squares* and form the *goal zone*. Each non-dead square c of the instance defines a different decomposition into maze and goal zones. We call c the *intermediate square* of that decomposition. A given decomposition corresponds to a new abstract state space for the maze subproblem, which is obtained by relaxing the capacity constraints of the intermediate square: we allow to place any number of stones and the man on it at the same time. In the corresponding intermediate abstract goal state all stones are placed at

Figure 4.2: A Sokoban instance (a) and three different instance decompositions (b,c,d) defined by three different intermediate squares. The instance decomposition in (d) has the largest possible maze zone and thus it defines the cut square.



the intermediate square, and the man is in the single reachable component. By definition of an intermediate square a stone can be pushed from the maze zone to the goal zone only by passing over the intermediate square.

We define a *cut square* as the intermediate square that maximizes the size of the maze zone. The set of cut squares can be obtained by analyzing all non-dead squares c. For each square c a reverse search from all goal squares is run. During the reverse search, we prohibit to place a stone on c, to limit the search to the goal zone. Then, all squares reachable from goal squares belong to the goal zone, and all other non-dead squares to the maze zone. Among all such squares, we choose one that maximizes the size of the maze zone, since the PDB provides better heuristic values for the maze zone.

Figure 4.2 shows an example of an instance and three different decompositions. The instance shown in Figure 4.2a has 40 non-dead and non-goal squares. Figure 4.2b shows a possible decomposition into 13 maze squares (including the cut square) and 27 goal zone squares. The decomposition of Figure 4.2c has 28 maze squares and 12 goal zone squares, and the one of Figure 4.2d has 41 maze squares and an empty goal zone.

The decomposition of Figure 4.2d leads to the largest maze zone, and the black square at I10 is the unique cut square of this instance. Note that the decompositions concern only pushed stones, not the movement of the man. Even when placing a stone on the intermediate squares in Figure 4.2 the reachable component of the man still includes all free squares.

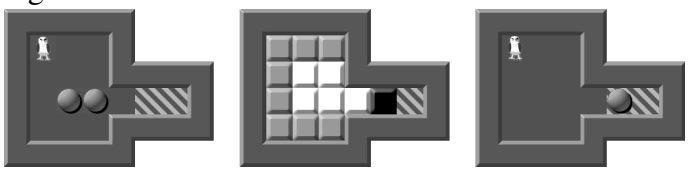
4.1.2 Construction and Storage of the PDB

The application of PDBs to Sokoban is different from other puzzles in two aspects. First, in Sokoban each instance has a different state space and goal state. Thus, the PDB must be constructed for each instance, and the construction cost cannot be amortized over the solution of multiple initial states. This is also true for applications of PDB in domain-independent planning and there is no simple solution for this problem. For example, when considering the domain of Sokoban Haslum et al. (2007) report that 85% of the total time is spent building the PDB. Therefore, PDBs must be constructed efficiently and should be effective when used to guide the search. Second, in Sokoban the stones are unlabeled, and thus a single PDB for a fixed number of k' stones, can be used for every subset of k' stones.

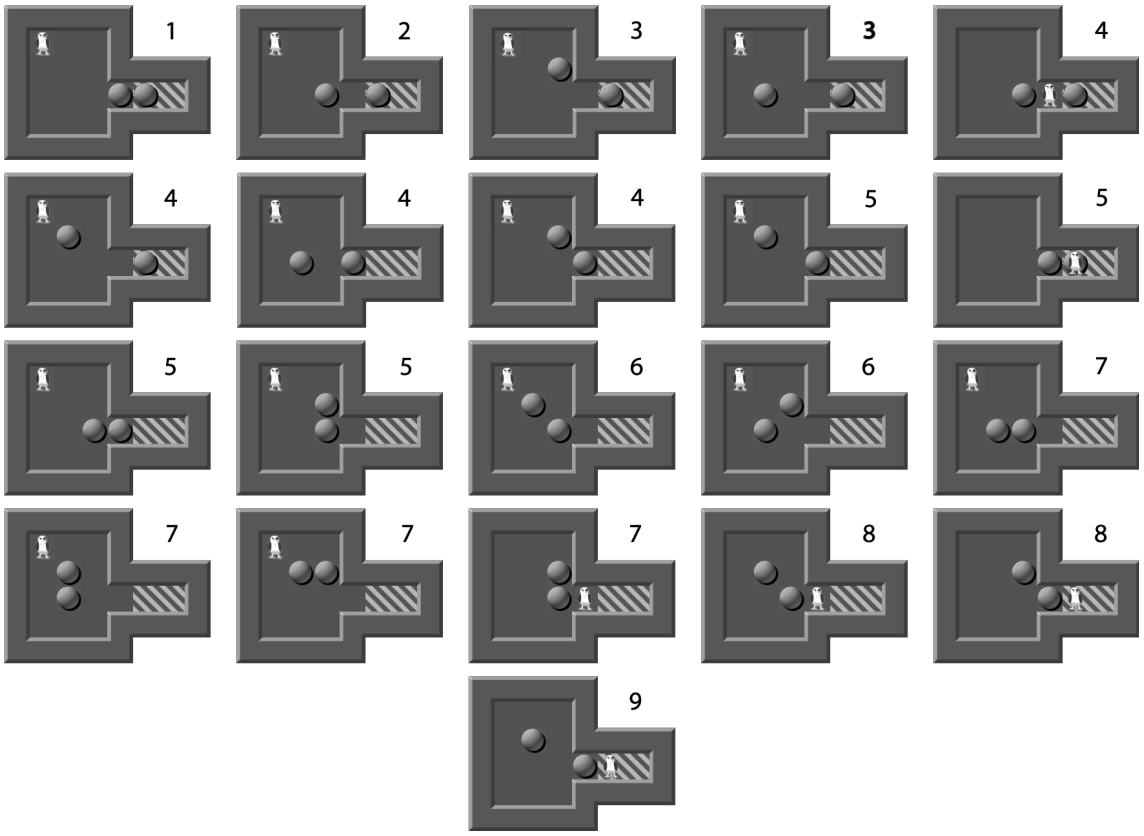
We call the PDB constructed using the intermediate abstract goal state an *intermediate pattern database* (IPDB). The number of stones in the abstraction determines the size of the IPDB. An IPDB-k' uses an abstraction of k' stones. It is constructed by a reverse search starting from the intermediate abstract goal state. Since we are only interested in distances to the maze zone, and stones on the cut square do not restrict the movement of the man, moving them to the goal zone can neither increase the man's freedom nor reduce the distance to the intermediate abstract goal state. Thus, we only allow movements from and to maze zone squares, without losing admissibility.

Figure 4.3a shows an instance, its decomposition and the intermediate abstract goal state for the construction of the IPDB. Figure 4.3b shows all the abstract states stored in the IPDB-2 that were generated by the reverse search starting from the abstract goal state. The distance to the abstract goal state is shown in the upper right corner of each abstract state. The position of the man is normalized to the leftmost upper free square of its reachable component. In this instance six squares are reachable by pull moves from the cut square. Thus, there are $\binom{6}{2} = 15$ possible placements of the stones, and for each placement thirteen squares remain free. So in the worst case, the IPDB stores

Figure 4.3: Generation of the PDB for the maze zone.



(a) Example of an instance (left), its decomposition (center), and the intermediate abstract goal state for the IPDB (right).



(b) All states generated by the construction of the IPDB.

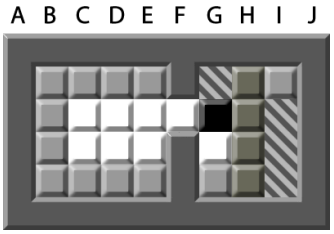
$15 \times 13 = 195$ abstract states. In practice the number of abstract states is much smaller, since the number of reachable components of the man is less than the number of free squares, usually 1 or 2, and not every placement of stones leads to a valid abstract state. In the example the PDB contains only 21 abstract states.

When the IPDB is built from an abstract goal state with two stones, it can be stored in a three-dimensional array with two indices for the position of each stone and one index for the non-normalized position of the man. This is possible because an IPDB built from two stones has a small number of entries. This storage enables fast queries since the index for the positions of the stones and the man can be computed in constant time. For larger IPDBs built from more than two stones ($k' > 2$), this approach cannot be used, and we store each abstract state in a hash table. In this way, the PDB fits into memory, but the

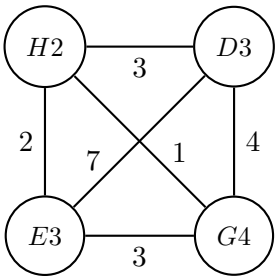
cost of a query is higher, queries have to determine the reachable component of the man.

4.1.3 Computation of the Heuristic Function

Figure 4.4: Computation of the proposed heuristic in the maze zone. (a) Decomposition of the instance of Fig 2.3a. (b) Matching graph for computing the maze zone lower bound for an IPDB-2.



(a)



(b)

The heuristic value of a state is the sum of the heuristic value for the maze zone, obtained by the IPDB, a heuristic value for the goal zone, and global linear conflicts between stones adjacent to the cut square and a stone at the cut square. For the goal zone we use a heuristic function similar to the standard heuristic function EMM of Sokoban.

To obtain a lower bound for the maze zone using an IPDB-k', we have to partition all k stones into parts of size k'. For any partition of the set of k stones into disjoint subsets of k' stones, the sum of the costs for solving each subset is an admissible heuristic value, since the cost of solving the whole subproblem is at least the total cost of solving each disjoint subproblem with k' stones independently. If k' does not divide k, we place $k \bmod k'$ additional artificial stones on the cut square. Defining a fixed partition of the stones for the whole search is not a promising strategy since local interactions of the stones depend on their placement. For this reason the PDB is partitioned dynamically. This is particularly easy in Sokoban, since we need only a single PDB. Thus, among all possible partitions, we want to find one that maximizes the lower bound. This problem can be solved in polynomial time by a maximum weight matching when $k' = 2$, but is NP-complete, for $k' > 2$.

When the abstraction is composed by two stones we use a maximum weight matching algorithm in order to obtain the highest heuristic value in polynomial time. To obtain the maze heuristic value a complete graph with one vertex for each stone is build. Each pair of stones is connected by an edge whose weight is the cost of pushing

both stones to the cut square, obtained by querying the IPDB. If the cost of some pair is not stored in the IPDB a deadlock is identified. If only one stone is in the maze zone the weight of the edge corresponds to the cost of pushing this stone to the cut square. If both stones are in the goal zone the edge weight is zero. When the number of stones is odd we add one artificial vertex that is connected to each other vertex by an edge whose weight corresponds to the cost of pushing the stone to the cut square. Figure 4.4 shows the decomposition of the instance in Figure 2.3a and the corresponding graph. The value of the maze heuristic function is the value of a maximum weight matching in this graph.

When the abstraction is composed by more than two stones, the problem of computing the highest heuristic value becomes equivalent to the maximum weight exact cover, an NP-complete problem (GAREY; JOHNSON, 1979). Since the heuristic value has to be computed for each state generated during the search, we use a simple and fast greedy randomized constructive procedure to approximate it. We start by querying the distance of every subset of k' of the k stones in the IPDB. Then we sort these distances in order of non-increasing increments, where the increment is the difference of the value stored in the IPDB and the sum of the distances for each stone to reach the cut square. Each of the k stones in the state can be part of only one selected subset of k' stones. Thus, selecting a subset will exclude other subsets which include the same stone. We heuristically generate $k+1$ partitions of the k stones as follows. The first partition is obtained by repeatedly choosing a subset of highest increment, until all stones are covered. The remaining k partitions are obtained by a greedy randomized strategy. This strategy chooses some random subset from the first m subsets, and then completes the partition greedily. We repeat this strategy k times with $m = i\binom{k}{k'}/k$ in iteration $i = 1, \ldots, k$. Thus, in the last iteration we choose randomly among all subsets. The highest heuristic value obtained in all $k+1$ partitions is used as the heuristic value. Observe that the heuristic obtained in this way is admissible, since every partition of the stones yields a lower bound on the shortest distance to bring the stones to the cut square. The constructive heuristic queries all $\binom{k}{k'}$ subsets of k' from k stones in the IPDB. Depending on k and k' there can be a large number of queries, and for each one we have to find the reachable component of the man. Thus, this approach has a large computational cost. However, it detects all deadlocks formed by k' stones and provides good heuristic values.

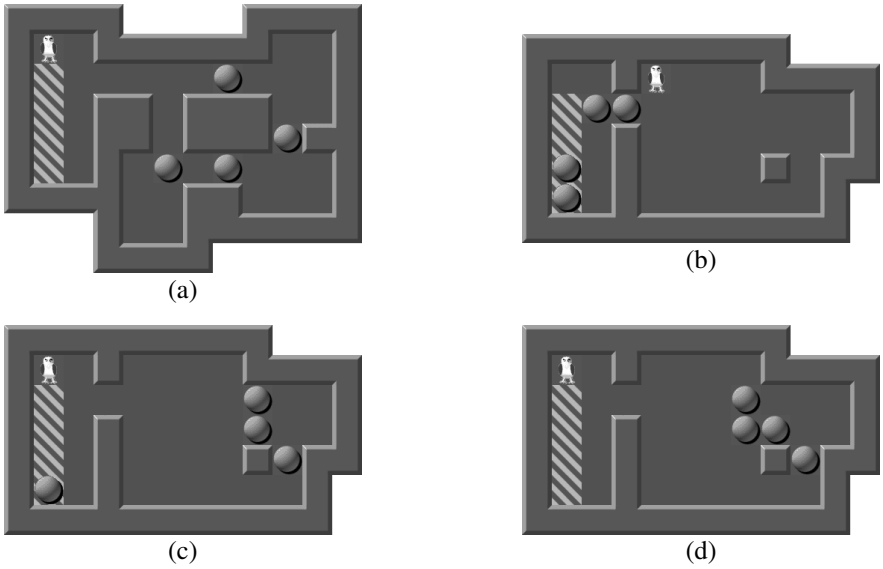
For the goal zone we use the EMM, i.e. the value of a minimum matching of the stones and the goal squares and penalties for linear conflicts. The minimum matching is computed on the same bipartite graph used for EMM, shown in Figure 2.3a, but the

distances are computed in a slightly different way, as follows. All the stones in the maze zone are treated as if they were positioned at the cut square. For each such stone, the position of the man is defined as the original position of the stone in the maze zone. The stones in the goal zone or at goal squares remain at their current position and use the current position of the man. As for EMM, these distances already include backout conflicts. Finally, the detection of linear conflicts is limited to pairs of stones in the goal zone or at goal squares. For example, when computing the distances for the instance of Figure 2.3a, only the position of the stone at $H2$ is used and the other stones in the maze zone are placed on the cut square $G3$. Note how the placement of the man at the original position of a stone influences its distance to the goals. For example, the stones originally at squares $D3$ and $E3$ have shortest distance 3 to goal square $G2$, but the shortest distance to $G2$ of the stone originally at square $G4$ is only 2. Finally, linear conflicts that include stones at the cut square are neither penalized by the maze lower bound nor by the goal lower bound. Such *global linear conflict* conflicts are penalized separately.

When passing from a state $u \in S$ to a successor $v \in \mathrm{Succ}(u)$ the heuristic value has to be updated. Most of the time only the goal or the maze heuristic value changes, so we can avoid unnecessary computations. If a push is completely inside the maze zone, the goal heuristic value does not change since the distances in the bipartite graph do not change. Similarly, if a movement is done completely in the goal zone the maze heuristic value does not need to be computed again.

EMM identifies implicit deadlocks caused by two or more stones that can be pushed only to a single goal square. However, it does not identify a deadlock in which a position of one stone blocks all feasible paths of another. An IPDB-k' detects all possible interactions of at most k' stones and the position of the man in the maze zone. Since it is constructed exhaustively, if some configuration of the man and stones is not stored in the IPDB it must be a deadlock and thus can be discarded. An IPDB detects, for example, linear and backout conflicts, and the implicit deadlocks of the EMM. Figure 4.5a shows an example of an instance where the lower bounds of EMM, IPDB-2, IPDB-3, and IPDB-4 are strictly increasing, and Figures 4.5b–4.5d show a sequence of instances with deadlocks detected only IPDB-2, IPDB-3, and IPDB-4, but no weaker lower bound.

Figure 4.5: Examples of instances with different lower bounds and deadlocks. (a) The lower bound obtained by EMM, IPDB-2, IPDB-3, and IPDB-4 is 42, 46, 48, and 50, respectively. The lower bound of IPDB-4 matches the optimal solution length. (b) A deadlock detected by IPDB-2, but not EMM. (c) A deadlock detected by IPDB-3, but not IPDB-2 or EMM. (d) A deadlock detected only by IPDB-4.



4.1.4 Consistency

A heuristic h is *consistent* if, for all states $u, u' \in S$, $h(u) \leq h(u') + w(u, u')$. The A* algorithm using a consistent heuristic will find an optimal solution visiting each state at most once. To show consistency of h it is sufficient to show that this relation holds for $u' \in \mathrm{Succ}(u)$ (PEARL, 1984). In Sokoban, $w(u, u') = 1$ for all states $u \in S$, $u' \in \mathrm{Succ}(u)$. Therefore, to establish consistency it is sufficient to show, that for any valid push of a stone the heuristic h decreases by at most 1 (see e.g. Edelkamp and Schrödl (2012, p. 21)).

In this section we show that the proposed heuristic is consistent. We start by introducing some notation for Sokoban instances and define states formally, and then proceed to show consistency of the individual and the combined heuristics. The correspondence of state space problems and graphs allows us to use graph terminology. For example, a path of length k is a subgraph $P = (V, E)$ of G with vertex set $V = \{u_0, \ldots, u_k\}$ and edge (or action) set $E = \{(u_0, u_1), \ldots, (u_{k-1}, u_k)\}$ and we refer to a path by either its vertex or its edge set. Its cost is $w(P) = w(E) = \sum_{e \in E} w(e)$. For a pair of states $u, u' \in S$, let $w(u, u') = w(P)$ for some path P from u to u' of minimum cost.

Let B be a set of stones, Q be the set of free squares of an instance, and $G \subseteq Q$ the set of goal squares. A state $u \in S$ is a pair $u = (m, p)$, where $m \in Q$ is the leftmost

upper free square in the reachable component of the man, and $p : B \to Q$ is an injective map from the stones to the free squares. State $u = (m, p) \in S$ is a goal state if the set of squares occupied by the stones $p(B) = \cup_{b \in B} p(b) = G$. For squares $q, r \in Q$ let $\delta(q, r, m)$ be the shortest distance of a stone-path from q to r when the man is at m. Note that this definition already includes backout conflicts.

To a state $u = (m, p) \in S$ corresponds a complete bipartite graph $MM = (B \dot\cup G, E, d)$ between the stones and the goals, with edge set $E = \{\{b, g\} \mid b \in B, g \in G\}$, and weights $d(e) = \delta(p(b), g, m)$ for $e = \{b, g\} \in E$. The cost of a matching $M \subseteq E$ in MM is

$$c(M) = \sum_{e \in M} d(e). \tag{4.1}$$

We write M^* for the minimum cost perfect matching in MM. For a state $u \in S$ and a corresponding bipartite graph MM with minimum cost perfect matching M^*, the heuristic h^{MM} is defined as

$$h^{MM}(u) = c(M^*). \tag{4.2}$$

It is well-known that the minimum matching heuristic for Sokoban is consistent (see e.g. Edelkamp and Schrödl (2012, ch. 1.7.3)). We begin to show, by a similar argument, that this also holds for h^{MM} which includes backout conflicts.

Theorem 4.1.1. *The heuristic h^{MM} is consistent.*

Proof. Consider states $u \in S$, $u' \in \mathrm{Succ}(u)$, with corresponding bipartite graphs $MM = (B \dot\cup G, E, d)$ and $MM' = (B \dot\cup G, E, d')$, matching costs c and c' and minimum cost perfect matchings M^* and M'^*. Since u' is a successor of u only one stone has been moved. Its distance to any goal squares decreases by at most 1, and all other distances do not change. Thus we have $c'(M) \geq c(M) - 1$ for all matchings $M \subseteq E$. Therefore,

$$h^{MM}(u') = c'(M'^*) \geq c(M'^*) - 1 \geq c(M^*) - 1 = h^{MM}(u) - 1$$

where the last inequality follows from the minimality of M^* in state u . □

Definition 4.1.1 (Linear conflict)**.** *A triple $(q, r, m) \in Q^3$ where squares q and r are adjacent is a* linear conflict *if for an instance in state $u = (m, p) \in S$ with only two stones placed at q and r, and for all states $u' \in \mathrm{Succ}(u)$, $h^{MM}(u') \geq h^{MM}(u) + 1$. Two*

stones $b, b' \in B$ of an arbitrary instance in state $u = (m, p) \in S$ are in a linear conflict, if $(p(b), p(b'), m)$ is a linear conflict.

Let $L(u)$ be the maximum number of linear conflicts in state $u \in S$ such that each stone is part of at most one linear conflict. (The value $L(u)$ is the size of a maximum matching in the conflict graph over the stones, where stones in linear conflict are connected by an edge.) The heuristic h^{EMM} is defined as

$$h^{EMM}(u) = h^{MM}(u) + 2L(u). \tag{4.3}$$

Theorem 4.1.2. *The heuristic h^{EMM} is consistent.*

Proof. Consider states $u \in S$, $u' \in \mathrm{Succ}(u)$. We have to show that $h^{EMM}(u') \geq h^{EMM}(u) - 1$. Since at most one stone moves, $L(u') \geq L(u) - 1$ and since h^{MM} is consistent, it is sufficient to show that if L decreases by 1 then h^{MM} must increase. If $L(u') = L(u) - 1$ one linear conflict has been resolved. Then, $h^{MM}(u') \geq h^{MM}(u) + 1$ by definition of a linear conflict, and

$$h^{EMM}(u') = h^{MM}(u') + 2L(u') = h^{MM}(u') + 2L(u) - 2 \geq h^{MM}(u) + 2L(u) - 1 = h^{EMM}(u) - 1.$$

□

4.1.4.1 The maze zone subproblem

Consider a decomposition of an instance into a maze zone and a goal zone, defined by some cut square. The maze zone subproblem is to bring all stones in the maze zone to the cut square, where the capacity constraints of the cut square have been relaxed. Suppose we know, for all subsets of squares $P \subseteq Q$ of size k' and positions of the man m the minimum cost $\delta(P, m)$ of solving this subproblem with the k' stones placed at squares P. (As explained above, these values are computed by an exhaustive backwards search and stored in a PDB.) Then,

$$h^M(u) = \sum_{P \in \mathcal{B}} \delta(p(P), m)$$

is an admissible heuristic for the maze zone subproblem, where $\mathcal{B}$ is a partition of the set of stones in the maze zone B in state $u = (p, m) \in S$ into subsets of size k'. The

admissibility of h^M follows from the decomposition into independent subproblems of size k' and the optimality of δ. If $|B|$ is not a multiple of k' we add $|B| \bmod k'$ artificial stones and place them at the cut square.

Lemma 4.1.1. *If, in each state $u \in S$ a partition $\mathcal{B}$ that maximizes $h^M(u)$ is chosen, then heuristic h^M is consistent for the maze zone subproblem.*

Proof. The proof is similar to that of Theorem (4.1.1). Consider states $u = (m, p) \in S$, $u' = (m', p') \in \mathrm{Succ}(u)$, with corresponding optimal partitions $\mathcal{B}^*$ and $\mathcal{B}^{*\prime}$ of the stones. Since u' is a successor of u only one stone has been moved. Thus, for any partition of the stones, the distance of the part containing this stone decreases by at most 1, while all other distances do not change. Thus we have $\sum_{P\in\mathcal{B}} \delta(p'(P), m') \geq \sum_{P\in\mathcal{B}} \delta(p(P), m) - 1$ for all partitions $\mathcal{B}$. Therefore,

$$h^M(u') = \sum_{P\in\mathcal{B}^{*\prime}} \delta(p'(P), m') \geq \sum_{P\in\mathcal{B}^*} \delta(p'(P), m') \geq \sum_{P\in\mathcal{B}^*} \delta(p(P), m) - 1 = h^M(u) - 1,$$

where the last inequality follows from the optimality of $\mathcal{B}^*$. □

Observe that for $k' = 2$ the optimal partition in Lemma (4.1.1) corresponds to a maximum weight matching in the complete graph over the stones B, where each edge $e = \{b, b'\}$, $b, b' \in B$ has weight $\delta(e, m)$. For $k' > 2$ the problem of finding the optimal partition is NP-complete, and we usually cannot afford to compute it. In this case h^M is not guaranteed to be consistent, which is the case for the greedy heuristic explained in Section 4.1.3.

4.1.4.2 The goal zone subproblem

The goal zone subproblem is to bring all stones to the goal squares, after placing all stones of the maze zone at the cut square. To obtain the goal zone heuristic h^G, the problem is relaxed into an independent subproblem for each stone, where the man is placed at the original position of the stone in the maze zone, or remains at his current position in the goal zone or at goal squares. These distances are lower bounds on the shortest distance of each stone to the goal squares, as explained in Section 4.1.3. Therefore, for any placement of the stones, such that all stones are either at the cut square or in the goal zone, EMM, where linear conflicts are restricted to pairs of stones in the goal zone, is an admissible heuristic. By an argument very similar to Theorem (4.1.2) it can be shown to be consistent.

4.1.4.3 Consistency of the complete heuristic

Finally, we show the consistency of the complete heuristic h^D, which is the sum of the maze zone heuristic h^M and the goal zone heuristic h^G with an additional penalty for global linear conflicts.

Definition 4.1.2 (Global and local linear conflicts)**.** *For a given instance decomposition into a maze and a goal zone, a linear conflict of two stones in state $u \in S$ is called* global*, if one of the stones is placed at the cut square.*

It is admissible to consider global linear conflicts, because neither the heuristic of the goal zone subproblem nor that of the maze zone subproblem accounts for them. For the goal zone heuristic, this holds by definition, for the maze zone heuristic by relaxation of the capacity constraints of the cut square. However, the same stone cannot be part of a global linear conflict and a linear conflict in the goal zone. Thus, we redefine $L(u)$ to be the maximum number of global linear conflicts or linear conflicts in the goal zone in state $u \in S$, such that each stone is part of at most one linear conflict. (As above, the value $L(u)$ is the size of a maximum matching in a corresponding conflict graph.) Let $h^G(u) = h^{EMM}(u) + 2L(u)$ be the heuristic value of the goal zone subproblem, as described above. Again, by an argument similar to Theorem (4.1.2), h^G with the new definition of L can be shown to be consistent. The complete heuristic is defined by

$$h^D(u) = h^M(u) + h^G(u). \tag{4.4}$$

It is admissible since it has been obtained by relaxing the capacity constraints of the cut square, which allows us to decompose the problem into two independent subproblems, and the heuristics of the subproblems are admissible. Finally, from the independence of the subproblems we also have

Theorem 4.1.3. *If heuristic h^M is consistent, then heuristic h^D is consistent.*

Proof. Consistency of h^D follows since h^M and h^G are independent: if h^M decreases then h^G cannot decrease and *vice versa*. Indeed, if h^M decreases, either a stone in the maze zone has been moved, and h^G does not change, since the goal zone subproblem is the same. Furthermore, the number of linear conflicts cannot decrease in this case, by definition of linear conflicts. Conversely, if h^G decreases a stone in the goal zone, on a goal square or on the cut square has been moved. If its new position is in the goal zone or on a goal square h^M does not change, since the maze zone subproblem is the same.

Otherwise, if its new position is in the maze zone, h^M cannot decrease, since the maze zone subproblem is either the same, or has one stone more. □

4.1.5 Multiple Goal State PDB

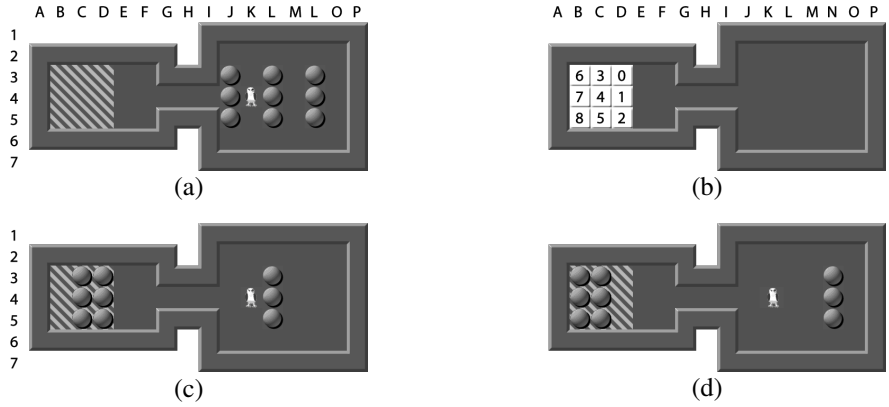
As mentioned before, a natural approach for a PDB would be to use an abstraction that keeps only k' of the k stones, and stores the distances to the nearest abstract goal state. In this case, every of the $\binom{k}{k'}$ possible placements of k' stones on goal squares together with a reachable component for the man is an abstract goal state. Figure 4.1 shows an example of the set of abstract goal states for a small instance. This is one of the main differences between IPDBs and MPDBs: an MPDB has a usually large number of goal states, whereas the IPDB has only one.

A reverse search from the set of abstract goal states builds the MPDB. The search continues until the whole abstract state space is explored. Every explored abstract state is stored in the MPDB with its distance to the closest abstract goal state. The storage is the same as for IPDBs.

In the MPDB, we have the distance from each abstract state to its closest abstract goal state. We can partition k stones into disjoint parts of size k', and each part represents an abstract state if we add the reachable component of the man. In this case, we can sum up the distance of each abstract state and get an admissible heuristic value. Note that several abstract states could be mapped to the same abstract goal state. The methods used to compute the partition are the same for IPDBs and MPDBs.

The lookup in MPDBs is simpler since it stores the heuristic value for the whole state. We query the distance of each abstract state, and the sum of the distances is the heuristic value. However, as in IPDBs, there is a particular situation when k' does not divide k. Consider the case when $k = 8$ and $k' = 3$. After the partition, we get two parts of size three, and one part of size two. The MPDB has distances for abstract states of size three, but not for two. This problem is not as trivial as it is for IPDBs. In this situation we do not know where to place an artificial stone: a stone on a different goal square may result in different distance for the abstract state. In such cases, we build a second MPDB with abstraction of size $k \bmod k'$. The construction of the second, smaller MPDB is usually not significant compared to the construction cost of the main MPDB.

Figure 4.6: Example of the fill order tie breaking rule. (a) shows the initial state of the instance and (b) the fill order priorities. Both states shown in (c) and (d) have the same f-value with $g = 59$ and $h = 34$, however (c) has fill order of 63 and (d) has fill order of 504.



4.2 A Domain-Dependent Tie Breaking Rule

The A* algorithm often must choose one from multiple nodes with the same f-value. Without explicit tie breaking rules the implementation of the priority queue decides which node is explored next. A stable priority queue, for example, will explore tied nodes in the order of their generation. We can use domain-dependent knowledge to explicitly prioritize nodes with the same f-value. A traditional approach is to break ties in favor of nodes with the smallest h-value. Junghanns and Schaeffer (2001) propose a tie breaking scheme which uses inertia as a first-level tie breaker, and the heuristic value as a second-level tie breaker. Inertia gives priority to a node which has been generated by a longer sequence of moves of the same stone. Here, we propose a new tie breaking rule called *fill order*.

Compared to other problems, Sokoban tends to generate a larger number of ties, since there are multiple combinations of stones on goal squares and the rest of the instance which have the same f-value. For example, in Figure 4.6 the initial node 4.6a has descendants 4.6c and 4.6d with $f = 93$. Using the proposed heuristic function and breaking ties by generation order explores more than 5 million nodes to solve the instance, since several possible configurations of the stones on goal squares are explored before reaching the goal node. Some of these configurations are deadlocks like configuration 4.6c.

But in fact the goal squares cannot be filled in an arbitrary order. Goal squares $B3$, $B4$ and $B5$, for example, have to be filled before any of the next columns of goal squares. Thus, we define a fill order for the goal squares which respects the restrictions that result from the placement of the goal squares in the maze.

The order is defined algorithmically as follows. Starting from an placement of all stones on the goal squares, we repeatedly remove a set of stones, until all stones have been removed. In each iteration, we process all remaining stones in an arbitrary order. For each stone we test if a reverse move can be applied to it. If a test succeeds we assign a priority of 2^i to the corresponding goal square, where i is current total number of successful tests. Then we remove the set of all stones for which the test succeeded in the current iteration. On termination a priority value has been assigned to each goal square. The priority values represent a guess of the order in which the goal squares are filled in an optimal solution. The fill order of a node is defined as the sum of the priority values of the goal squares which are occupied by a stone. Thus, the fill order gives preference to nodes that place the stones at the goal squares of highest priority value first. The idea of the fill order is similar to goal macros (JUNGHANNS; SCHAEFFER, 2001) since both techniques try to use information from the state space related to the placement of goal squares to improve the effectiveness of the solver. The fill order, however, is admissible, since it is only used to break ties.

Figure 4.6b shows the priorities of the goal squares of the example above. A state with a stone at $B5$ has priority 256 and a state with eight stones in the other goal squares has priority 255. In this way we get an absolute order for filling the goal squares. Using this rule of tie breaking we solve the example instance exploring fewer than $5,000$ nodes.

4.3 Computational Results: Sokoban

In this section we provide results of computational experiments with the proposed methods. We compare the heuristic function EMM to MPDBs and to IPDBs. To be comparable, all three heuristics were implemented using the same basic algorithms and data structures.

We first evaluate the effectiveness of the instance decomposition and discuss the construction of the PDBs. Next, we compare the heuristics EMM, MPDB, and IPDB on the initial states and randomly generated states of the standard set of instances. We also investigate the impact of the standard tie breaking rules based on inertia, lower bound and fill order. We compare the performance of several Sokoban solvers using the new heuristics and the tie breaker to Rolling Stone with all admissible techniques. Furthermore, we test the performance of the Fast Downward domain-independent planner with PDBs in solving instances from the standard set. Finally, we present experiments of the use of

MPDB as a deadlock detection technique.

All experiments were performed on a PC with an AMD Opteron CPU running at 2.39 GHz with 32 GB of RAM on the standard set of 90 instances. We use Blossom V to compute the matchings (KOLMOGOROV, 2009). All experiments, if not otherwise stated, have been run with a limit of 20 million explored nodes, one hour of computation time and 32 GB of memory. In tables that report nodes, a prefix ">" means that an optimal solution has not been found because they reached the time limit. Runs which reached the node limit of 20 M only have an entry ">". In those tables, the memory limit of 32 GB was never reached in any run. The number of nodes does not include the nodes explored when constructing the PDB, but reported times include the construction time.

4.3.1 Instance Decomposition and Construction of the PDB

Figure 4.7 shows the results of the decomposition of the 90 instances from the standard set. For each instance the y-axis shows the percentage of squares in the maze zone of all non-dead squares. The x-axis plots the instances in order of decreasing percentages. The figure shows that all instances can be decomposed into goal and non-trivial maze zones (of size at least 3). Since the IPDB contains the exact distances from stones in the maze zone to the cut square we can expect our method to perform better for larger maze zones. This is the case for the standard instances: one-third has over 90% of the squares belonging to the maze zone and on average 67% of the squares belong to the maze zone. However, the effectiveness of the IPDB also depends on the characteristics of the instance. Some instances may have large goal zones, but the IPDB still can be effective if the "hard" part of the problem is in the maze zone, which is often the case in human-designed instances.

Table 4.1 shows the average and maximum time for constructing the PDB and the average and maximum number of entries. We have tested MPDB and IPDB with abstractions of size two and four, and with an abstraction of variable size k'. The value of k' for each instance corresponds to the largest PDB that can be constructed within the time and memory limits, that is one hour of computation time and 32 GB of memory.

The IPDB is built only in the maze zone, thus it can be built faster and has fewer entries than the MPDB. The IPDB-2 and MPDB-2 are always built in less than a second, and have fewer than $10,000$ entries. The IPDB-4 uses never more than 3.1 M entries, and can be built in less than 200 seconds, while the IPDB-k' uses at most 85 M entries. An

Figure 4.7: Percentage of squares in the maze zone. The number of the instance in the standard set is plotted above each data point.

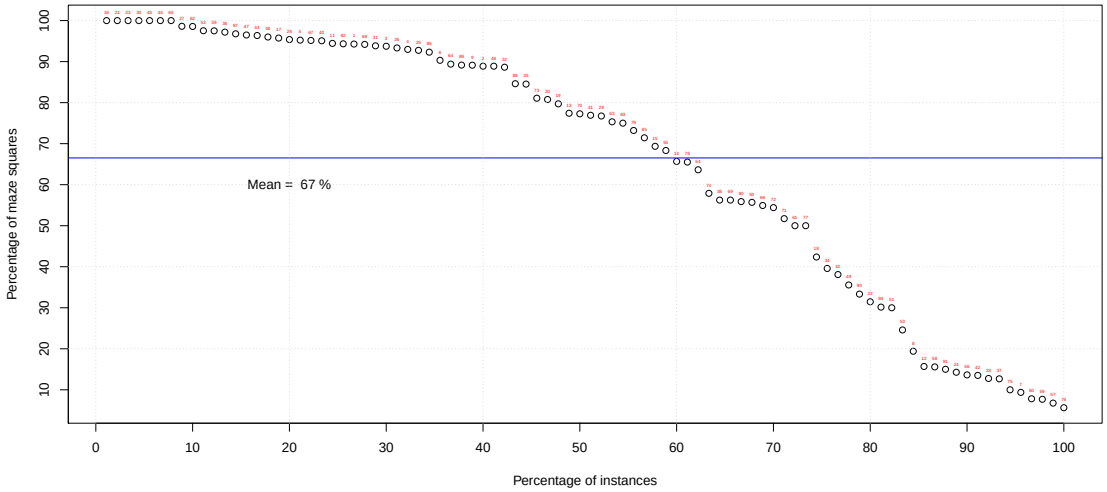


Table 4.1: Building time and memory requirements for the pattern databases.

	MPDB			IPDB		
	2	4	k'	2	4	k'
Avg. time (s)	0	176	1,334	0	23	1,069
Max. time (s)	1	1,286	3,466	0	199	3,555
Avg. entries	3,455	2,286,486	20,978,849	1,135	323,572	19,459,034
Max. entries	9,557	15,172,945	63,192,937	4,633	3,073,969	83,425,673

MPDB-4 uses up to 15.2 M entries and never takes more than $1,300$ seconds to build, and the MPDB-k' uses up to 64 M entries. The limit in constructing larger PDBs in the variable size case was always the time for constructing the PDB. For a variable size of the abstraction, MPDB results in an average of $k' = 5.0$ and IPDB of $k' = 7.9$. IPDB is able to build databases for larger abstractions because the maze zone is smaller.

4.3.2 Heuristics Function on Initial States

Table 4.2 compares the value of the heuristics, EMM, MPDB, and IPDB on the initial states of the instances from the standard set. Column "#" gives the number of the instance and column "UB" the best known upper bound according to the global Sokoban score file[1]. For MPDB and IPDB we show the results for fixed abstractions of two and four stones (columns "2" and "4") and of an abstraction of variable size (column "k'"). The PDBs have been built as explained above. The best values found for each instance are shown in bold.

The table shows that EMM provides good results, MPDB provides weak heuristic values, and IPDB is able to significantly improve the values. MPDB-2 is always worse than EMM, MPDB-4 finds only one better result, and MPDB-k' improved EMM in nine instances. This shows that MPDB is not effective for Sokoban. Comparing with EMM, IPDB-2 improves the results of 31 instances, IPDB-4 of 58 instances, and IPDB-k' of 61 instances. Moreover, IPDB-k' provides the best results in 85 instances, and matches the upper bound value in 15 instances, compared to 8 instances for EMM.

The weakness of the lower bound of the MPDB has the same reason as the weakness of using a heuristic based on shortest distances of individual stones to goal squares compared to a minimum matching: the MPDB estimates the shortest distance to the nearest goals for each subset of stones, but does not require that the union of the selected goals over all subsets exactly covers the goal squares. Thus, usually multiple stones are assigned to the nearest goal squares, which leads to a weaker lower bound.

To estimate the behavior of the heuristics during the search we ran an experiment with $10,000$ randomly generated states for each of the 90 instances. Each initial state was generated by placing the stones at random non-dead squares, and the man at a random free square. The number of stones was the same as that of the original instance. Table 4.3 shows the mean heuristic value h of EMM, MPDB, and IPDB, and the mean number

[1] <http://www.cs.cornell.edu/andru/xsokoban/scores.txt>

Table 4.2: Heuristic values on the initial states of the standard set of 90 instances.

#	EMM	MPDB			IPDB			UB	#	EMM	MPDB			IPDB			UB
		2	4	k'	2	4	k'				2	4	k'	2	4	k'	
1	95	91	94	**97**	95	**97**	**97**	97	46	223	191	198	203	223	227	**229**	247
2	129	117	119	123	**131**	**131**	**131**	131	47	199	163	173	176	199	201	**203**	209
3	132	116	119	123	**134**	**134**	**134**	134	48	**200**	150	154	154	**200**	**200**	**200**	200
4	**355**	311	316	321	**355**	**355**	**355**	355	49	104	72	96	**120**	104	106	106	124
5	139	121	124	129	**141**	**141**	**141**	143	50	100	83	93	92	**102**	**102**	**102**	370
6	106	94	96	104	106	106	**110**	110	51	**118**	84	93	98	**118**	**118**	**118**	118
7	80	68	77	**83**	80	80	80	88	52	367	319	331	331	369	377	**381**	421
8	**220**	192	196	200	**220**	**220**	**220**	230	53	**186**	164	167	170	**186**	**186**	**186**	186
9	229	206	212	215	231	231	**233**	237	54	177	160	164	166	181	181	**183**	187
10	**510**	349	372	372	**510**	**510**	**510**	512	55	**120**	102	110	114	**120**	**120**	**120**	120
11	207	174	189	194	207	213	**217**	241	56	**193**	170	179	182	**193**	**193**	**193**	203
12	**206**	174	182	186	**206**	**206**	**206**	212	57	**217**	183	195	195	**217**	**217**	**217**	225
13	220	165	179	184	220	**224**	**224**	238	58	**197**	178	182	182	**197**	**197**	**197**	199
14	231	206	212	217	231	231	**233**	239	59	**218**	194	202	203	**218**	**218**	**218**	230
15	96	86	97	**106**	96	100	100	122	60	**148**	127	134	135	**148**	**148**	**148**	152
16	162	118	126	134	166	**170**	**170**	186	61	245	197	211	221	249	253	**257**	263
17	201	197	200	**213**	201	203	**213**	213	62	237	196	206	208	239	241	**243**	245
18	**106**	87	93	95	**106**	**106**	**106**	124	63	427	383	391	391	**429**	**429**	**429**	431
19	**286**	254	257	263	**286**	**286**	**286**	302	64	367	341	353	354	373	379	**381**	385
20	446	360	392	392	446	**450**	**450**	462	65	203	170	181	187	203	207	**209**	211
21	129	89	103	107	129	**137**	**137**	147	66	187	158	173	173	187	193	**199**	325
22	**308**	250	264	264	**308**	**308**	**308**	324	67	377	323	338	341	385	393	**395**	401
23	426	383	393	393	430	**432**	**432**	448	68	321	283	298	302	325	**333**	**333**	341
24	**518**	414	434	434	**518**	**518**	**518**	544	69	219	183	199	205	219	**223**	**223**	433
25	368	261	278	278	370	378	**380**	386	70	**329**	281	284	288	**329**	**329**	**329**	333
26	165	138	155	172	167	175	**185**	195	71	294	261	279	287	294	**298**	**298**	308
27	353	295	309	309	355	359	**359**	363	72	288	196	214	225	288	292	**296**	296
28	286	208	223	230	**290**	**290**	**290**	308	73	437	408	414	414	**441**	**441**	**441**	441
29	122	107	118	122	128	130	**132**	164	74	176	164	175	**186**	178	182	182	212
30	359	320	357	376	385	407	**419**	465	75	**263**	216	228	238	**263**	**263**	**263**	295
31	232	176	189	193	232	**236**	**236**	250	76	**194**	156	167	167	**194**	**194**	**194**	204
32	113	94	108	111	**115**	**115**	**115**	139	77	**360**	255	308	314	**360**	**360**	**360**	368
33	**152**	119	138	143	**152**	**152**	**152**	174	78	**136**	124	128	128	**136**	**136**	**136**	136
34	154	128	145	150	154	**164**	**164**	168	79	166	144	152	155	168	170	**172**	174
35	364	316	332	332	364	**368**	**368**	378	80	**231**	213	216	218	**231**	**231**	**231**	231
36	507	447	461	459	507	511	**517**	521	81	**167**	145	155	156	**167**	**167**	**167**	173
37	**242**	187	206	206	**242**	**242**	**242**	284	82	135	117	122	124	135	135	**137**	143
38	73	60	72	**81**	73	79	79	81	83	**194**	182	184	187	**194**	**194**	**194**	194
39	652	535	565	565	652	**658**	**658**	672	84	149	135	141	142	151	**153**	**153**	155
40	310	275	285	286	312	314	**316**	324	85	305	238	252	259	305	**307**	**307**	329
41	221	166	182	188	223	**227**	**227**	237	86	122	97	105	115	122	124	**130**	134
42	**208**	139	152	152	**208**	**208**	**208**	218	87	221	201	209	212	221	223	**225**	233
43	132	116	126	132	134	138	**140**	146	88	336	246	269	269	336	342	**344**	390
44	167	158	162	164	169	169	**173**	179	89	353	274	302	302	353	361	**363**	379
45	284	224	236	243	286	290	**294**	300	90	442	244	260	260	442	446	**450**	460
Averages										241	200	211	215	242	244	246	262

Table 4.3: Average heuristic value and number of deadlocks over $10,000$ randomly generated initial states for the standard set of instances.

	EMM	MPDB			IPDB		
		2	4	k'	2	4	k'
h	163.66	136.02	143.29	145.62	164.41	165.23	165.47
DL	1,525	8,062	8,870	8,922	6,346	6,942	6,984

of detected deadlocks ("DL") over all 90 instances. The mean values are only over the states where none of the heuristic functions detected a deadlock to exclude artificially high heuristic values.

As in the standard initial states, EMM presents good heuristic values, while those of the MPDB are much lower, and IPDB is able to improve slightly over the EMM lower bound. We can also see by the average value of IPDB-k' that abstractions of more than four stones yield only marginal improvements. The difference between the methods in the capacity of detecting deadlocks is much more significant than the difference of the heuristic values. EMM detects in average over all instances a deadlock in 15% of the randomly generated states, MPDB in more than 80%. IPDBs detect a deadlock in about 65% of the states.

These differences come directly from the differences in the methods. The EMM detects deadlocks caused by sets of stones for which there is an insufficient number of goal squares. These deadlocks are also detected by the IPDB, since it applies EMM in the goal zone, and all stones must pass over the cut square. Thus, the IPDB never detects fewer deadlocks than EMM, and often more (see the examples in Figure 4.5). In contrast, the MPDB is applied to the whole instance, and thus usually detects more deadlocks than the IPDB, since it also considers interactions of the stones inside the goal zone (an example would be any of the patterns in Figure 4.5b–4.5d occurring in the goal zone). However, since the MPDB does not enforce a one-to-one mapping of stones to goals, it cannot detect all deadlocks detected by EMM and IPDB.

The MPDB-k' detects on average 89% of the randomly generated states as deadlocks. This percentage of deadlocks seems surprisingly high, but can be explained by the fact that the instances of Sokoban usually are built such that most of the stone placements generate deadlocks. Therefore, during the search, a promising sequence of movements can bring a group of stones near to the goals, but produce a deadlock in another group of stones. This reinforces the importance of deadlock detection, because if a heuristic function fails to detect deadlocks early, a large search effort can be spent in states that never

Table 4.4: Results for an IPDB-2 with different tie breaking rules. A dot in the table means the instance indicated in the column was solved by the configuration indicated in the row.

Rule	Nodes	Instance#																	Solved
		1	2	3	4	6	7	17	38	43	49	51	53	73	78	80	82	83	
GO	281,607,359	•						•	•										3
IN	281,217,035	•						•	•										3
LB	127,733,711	•	•	•		•		•	•	•	•				•	•	•	•	12
FO	75,446,188	•	•	•	•	•	•	•	•	•		•		•	•	•	•	•	15
INLB	131,358,093	•	•	•		•		•	•	•	•				•	•		•	11
INFO	62,616,370	•	•	•	•	•	•	•	•	•		•	•	•	•	•		•	15
LBIN	125,548,205	•	•	•		•		•	•	•	•				•	•	•	•	12
LBFO	125,374,431	•	•	•		•		•	•	•	•				•	•	•	•	12
FOIN	95,281,106	•	•	•	•	•	•	•	•	•		•			•	•	•	•	14
FOLB	134,412,436	•	•	•		•	•	•	•	•					•	•	•	•	12
INLBFO	94,720,546	•	•	•		•		•	•	•	•	•		•	•	•		•	13
INFOLB	105,731,790	•	•	•		•	•	•	•	•		•		•	•	•		•	13
LBINFO	107,504,210	•	•	•		•		•	•	•	•	•			•	•	•	•	13
LBFOIN	107,504,214	•	•	•		•		•	•	•	•	•			•	•	•	•	13
FOINLB	116,543,217	•	•	•		•	•	•	•	•		•			•	•	•	•	13
FOLBIN	116,542,868	•	•	•		•	•	•	•	•		•			•	•	•	•	13
Totals		16	14	14	3	14	7	16	16	14	7	9	1	4	14	14	10	14	

lead to solutions. In instance #48, for example, 99% of the states are classified as deadlocks by IPDB-4 and MPDB-4, while EMM detects no deadlocks. Similar cases occur in several instances. These results show that PDBs have a significant capacity of detecting deadlocks.

4.3.3 Tie Breaking Rules

This section presents results for three tie breaking rules: inertia (IN), lower bound (LB), and fill order (FO). We also test the six cases where one rule serves as a secondary tie breaker, and another six cases where the remaining rule serves as a third-level tie breaker, since these configurations can result in a different number of explored nodes and solved instances. All tests use the IPDB-2 as the heuristic function.

The results are reported in Table 4.4. The first column gives the tie breaker used. The name of the first-level rule is followed by the name of the second- and third-level rule, if any. Any remaining ties are broken by giving preference to the earliest generated node (generation order, GO). The second column gives the total number of explored nodes. In this sum unsolved instances enter with 20 million nodes.

In total, 17 out of 90 instances were solved by at least one of the 16 tie breakers.

Instances 1, 17 and 38 were solved by all of them. Rules IN and GO solve three instances, but IN explores slightly fewer nodes. Rule LB solves 12 instances and explores significantly fewer nodes. Finally, rule FO solves 15 instances exploring slightly more than half of the number of nodes explored by LB. This is the best result for any simple tie breaker.

Among the rules with a second-level tie breaker the two combinations of FO and IN obtain the best results. Rules that use LB in general present the worst results. The best rule is INFO. This configuration improves the results obtained using only the rule FO. Using a third-level tie breaker does not increase the number of solved instances or decrease the number of explored nodes.

4.3.4 Solving Sokoban with Pattern Databases

In this section, we compare the state-of-the-art Rolling Stone solver with all admissible techniques (RS*) to solvers using the heuristic functions IPDB and MPDB. We first compare all heuristic functions using the INLB tie breaking rule of Rolling Stone. Next, we compare the heuristic functions using the tie breaking rule INFO. We also investigate to what extent PDBs with an abstraction of variable size can solve more instances. Finally, we describe experiments with the Fast Downward planner using PDBs in Sokoban. We do not report the solution length of optimally solved instances, since they always were equal to the best upper bounds shown in Table 5.1.

Our implementation uses an A* search, since the memory consumption of the PDBs is low, and we can afford to store all generated nodes. The A* search avoids exploring duplicated nodes, which makes the use of transposition tables unnecessary. To provide a fair comparison with RS*, we also report the results of an A* search using the same heuristic function EMM as RS*. This solver is called EMM in the tables below.

Table 4.5 shows the results for all instances which could be solved by one of the solvers breaking ties by INLB. It reports for each solver the number of explored nodes and the time to find the optimal solution. The number of explored nodes for RS* are from Junghanns and Schaeffer (2001). To the best of our knowledge, no times have been published for RS*. Both MPDB and IPDB are built from an abstraction with two stones.

RS* is able to solve 6 instances. The A* search with the same heuristic function and tie breaking rule solves ten instances, and explores about a factor of 5 fewer nodes. This shows that the A* search can overcome domain-dependent techniques such as transposition and deadlock tables, and the tunnel macros used by RS*. This is expected, since

Table 4.5: Number of explored nodes and computation time (in seconds) for different Sokoban solvers breaking ties by rule INLB.

#	RS*	EMM		MPDB-2		IPDB-2	
	Nodes	Nodes	Time	Nodes	Time	Nodes	Time
1	223	160	0	319,196	14	117	0
2	620,030	161,834	25	>	2,590	184	0
3	>	1,187,486	126	>	2,540	1,525,943	115
6	10,107,620	1,354,432	146	>	2,364	403,974	31
17	10,672,805	1,471,533	71	1,744,788	83	12,914	1
38	415,485	93,423	8	8,260,380	575	35,268	3
43	>	>	2,114	>	1,898	7,763,870	652
49	>	1,596,896	191	>	2,260	1,596,896	141
78	871	8,544	1	>	2,087	7,971	1
80	>	27,708	3	>	1,559	10,594	1
83	>	559	0	>	1,328	362	0
	6	10	2,685	3	17,298	**11**	**945**

RS* was designed for reduced memory usage. The results for the MPDB show that the direct application of PDBs to Sokoban is ineffective. It solves only three instances and in general uses more computation time and explores more nodes than the other techniques. The IPDB solves eleven instances, explores more than a factor of two fewer nodes than EMM, and is about three times faster. Furthermore, the more complex computation of the lower bound does not lead to a higher cost per node, and IPDB processes more nodes per second than EMM. Taken together, the IPDB is more than able to amortize the cost of constructing the PDB. Although MPDB detects many more deadlocks, its weaker lower bound leads to a poor performance. The IPDB seems to be a good compromise between quality of lower bound and deadlock detection.

We now turn to the analysis of the solvers when breaking ties by rule INFO. Table 4.6 shows the number of explored nodes and the solving time for all instances that could be solved by at least one of the solvers (RS*, EMM, MPDB, IPDB). MPDB and IPDB were tested with abstractions of two and four stones. The test was limited to abstractions of at most four stones, since for some instances a PDB with five stones could not be built in an hour.

The cost for computing tie breakers INLB and INFO is similar, and therefore all solvers process about the same number of nodes per second. The performance of the MPDB does not improve: MPDB-2 explores about the same number of nodes as for tie breaking rule INLB, and MPDB-4 explores fewer nodes, but is more than two times slower than MPDB-2. Both solve only three instances. EMM and IPDB perform better when breaking ties by INFO. EMM solves three more instances, and explores significantly

Table 4.6: Number of explored nodes and computation time (in seconds) for different Sokoban solvers breaking ties by rule INFO.

#	RS*	EMM		MPDB				IPDB			
				2		4		2		4	
	Nodes	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
1	223	169	0	318,212	13	187,472	37	125	0	122	1
2	620,030	147,281	22	>	2,417	>858,240	3,600	215	0	199	3
3	>	20,839	2	>	2,306	>906,030	3,600	2,203	0	232	2
4	>	1,590	1	>	3,356	>16,920	3,600	1,053	1	879	250
5	>	>	3,582	>	2,397	>301,916	3,600	>	2,677	349	8
6	10,107,620	1,334,586	132	>	2,209	>1,001,761	3,600	399,940	36	1,358	4
7	>	12,477,687	1,129	>	2,094	>842,650	3,600	12,477,687	1,271	12,477,687	1,271
9	>	>	3,467	>	2,067	>263,527	3,600	>	2,331	15,686	191
17	10,672,805	1,349,943	62	1,732,569	80	1,106,769	277	12,138	1	298	3
38	415,485	361,852	31	8,228,382	539	1,921,579	1,290	144,125	11	144,094	39
43	>	>	2,090	>	1,862	>1,353,230	3,600	7,292,104	758	2,669	13
51	>	2,283,280	493	>	3,350	>149,696	3,600	2,283,280	591	2,283,280	2,176
53	>	1,071	1	>9,050,495	3,600	>30,254	3,600	1,071	1	1,071	7
65	>	>15,085,591	3,600	>	3,055	>93,265	3,600	>17,276,451	3,600	494	52
73	>	>	3,538	>	2,961	>82,393	3,600	835	1	743	46
78	871	340	0	>	2,080	>1,812,340	3,600	287	0	286	3
79	>	>12,988,110	3,600	>	3,324	>146,317	3,600	>11,926,273	3,600	4,664	31
80	>	939	1	>	1,583	>286,205	3,600	773	1	646	15
82	>	>	2,606	>	2,125	>216,431	3,600	>	2,718	269	21
83	>	869	0	>	1,261	>1,360,629	3,600	524	0	454	7
	6	13	24,360	3	42,681	3	62,804	15	17,599	**20**	**4,143**

fewer nodes. IPDB-2 solves 15 instances, in considerably less computation time than EMM. IPDB-4 solves 20 instances. It reduces the number of explored nodes, and it is about a factor of four faster than the other solvers. Computing the heuristics for PDBs with more than two stones takes much more time. In all of the unsolved instances, both PDBs could not explore 20 million nodes in one hour, which can be observed in the results for MPDB-4. For the IPDB-4, however, this cost can be compensated by a much lower number of processed nodes.

To investigate the potential of using PDBs with larger abstractions, we determined for all instances the smallest k' such that the optimal solution can be found within our standard resource limits (one hour, 20M nodes). If the construction of the PDB needs more than one hour or more than 32 GB, the instance is considered unsolvable. For the MPDB the results from Table 4.6 do not change. No other instance could be solved with larger abstractions. Table 4.6 shows that the IPDB-4 solves five instances more than IPDB-2 (namely 5, 9, 65, 79, and 82). We found that two of them, instance 79 and 82, can be solved already with an abstraction of size 3, and one more instance (62) can be solved with an abstraction of size 5. So, larger abstraction sizes are not able to compensate the weak lower bound of the MPDB. The improvement in the IPDB is also rather limited, and it seems that better lower bounds and more detected deadlocks that come from interactions of five or more stones are rare and cannot compensate for the larger cost for building the

PDB.

Now we turn our attention to experiments that explore the performance of a state-of-the-art domain-independent planner using PDBs. We chose the Fast Downward planner, because it contains one of the most efficient implementations of pattern databases for domain-independent planning. In this experiment we use a time limit of four hours, and no limit on the number of nodes, since Fast Downward processes many more nodes per second. The planner was run with an A* algorithm and the PDBs from Haslum et al. (2007) as implemented by Sievers, Ortlieb and Helmert (2012). All other parameters have been left at their default values, which are commonly accepted as a good choice (POMMERENING; RöGER; HELMERT, 2013). For the experiment we used the domain description from the International Planning Competition that comes with Fast Downward.

The planner could not solve any instance within the resource limits used. For all instances, either the construction of the PDB did not terminate, or the memory did exhaust before finding the optimal solution. This behaviour has probably two reasons. First, the movement of the man is specified as an action of cost 0, which leads to a faster processing of the nodes, but also considerable more nodes must be explored. For example, in instance #1 Fast Downward explores more than 200 million nodes and reaches an f-value of 94, three below the optimal solution length. In contrast, the most expensive operation in our solver is the normalization of the position of the man, but this greatly reduces the number of unique states. Second, based on heuristic value on initial states, the lower bound is probably weak, for the same reason that the MPDB is weak in Sokoban. For example, the heuristic values of Fast Downward for the initial states of instances #1, #2, and #3 are 88, 100, and 105, respectively, compared to 97, 131, and 134 obtained by the IPDB.

Our last experiment compares the IPDB and the Fast Downward planner on instances of medium difficulty, with the same configuration as in the previous tests, and a time limit of $30\,\mathrm{m}$. The Microban test set, which Haslum et al. (2007) used in their experiments turned out to be too simple: all solvers were able to solve at least 150 of the 155 instances. To find instances of medium difficulty, we have screened several other test sets and chosen Boxxle1. We have further removed 18 instances from the set which Fast Downward could solve by blind search in 5 min to avoid floor effects, leaving us with 90 test instances. In these instances in average 65% of the non-dead non-goal squares belong to the maze zone, similar to the xSokoban test set. The IPDB could solve 35 instances, exploring in average $1.2\,\mathrm{M}$ nodes in $65\,\mathrm{s}$, and Fast Downward 14, exploring in average $8.8\,\mathrm{M}$ nodes in $22.8\,\mathrm{s}$. IPDB-2 solves the latter 14 instances exploring in average

6911 nodes in $0.2\,\mathrm{s}$. This shows that IPDB-2 solves the puzzles much faster than Fast Downward, and explores fewer nodes. On the instances which could not be solved the IPDB always failed due to time constraints, while Fast Downward failed due to memory constraints.

4.3.5 Using Pattern Databases for Deadlock Detection

MPDBs are highly effective in the task of deadlock detection – much more than IPDBs and EMM. We believe that we could build a better heuristic function if we combine an effective heuristic function with an effective deadlock detection technique like MPDB. However, there is an overhead associated with the construction of the MPDB and with the process of verifying if a state is a deadlock by looking up at the MPDB. In this subsection we aim to investigate this trade-off.

The construction of the MDPB for deadlock detection is done as usual with the exception that we do not store the distances of abstract states to abstract goal states. We only store which abstract states are reachable from abstract goal state. Through the search, we check if every abstract state is contained in the MPDB if not a deadlock is identified. This method is computationally costly for $k' > 2$. Because the process of lookup abstract state in the MPDB is the most costly part of our method to compute the heuristic function.

We use a naive approach to combining MPDBs with another heuristic function. For example, combining IPDB and MPDB we get IPDB + MPDB. In IPDB + MPDB, first we check in the MPDB if the state is a deadlock. If yes the state is discarded. If not, the state receives the value computed by the IPDB.

Table 4.7 shows the results of a heuristic that combines EMM and MPDB using INLB as the tie-breaking rule. EMM uses INLB as tie breaking rule and corresponds to the same solver as before. EMM + MPDB shows the results when using the EMM as the heuristic function and the MPDB only for deadlock detection. Two sizes of abstractions have been tested. The tests have been performed with a limit of one hour of computation time and 5 million explored nodes. EMM + MPDB-2 solves eleven instances and EMM + MPDB-4 twelve instances. EMM + MPDB-4 explores fewer nodes than the other approaches. Ten of the eleven instances that EMM + MPDB-2 solves, it solves faster than the other approaches. These results show that the use MPDB for deadlock detection could be effective even considering the overhead.

One could argue that the improvement of IPDBs in the heuristic value is marginal

Table 4.7: Number of explored nodes and computation time (in seconds) for different Sokoban solvers breaking ties by rule INLB using MPDBs only for deadlock detection.

#	EMM		EMM + MPDB			
			2		4	
	Nodes	Time	Nodes	Time	Nodes	Time
1	160	0	160	0	153	3
2	161,834	25	86,840	11	68,927	222
3	1,187,486	126	950,950	98	22,268	42
6	1,354,432	146	366,955	36	1,641	7
7	>5,000,000	428	223,956	20	127,840	276
17	1,471,533	71	19,633	1	775	7
38	93,423	8	24,196	2	8,919	6
49	1,596,896	191	1,381,596	147	>900,425	3,600
51	>5,000,000	905	>5,000,000	929	223,106	3,068
78	8,544	1	8,387	1	7,646	39
80	27,708	3	10,594	2	480	109
81	>5,000,000	1,055	>5,000,000	1,196	198,935	2,365
83	559	0	362	1	305	39
	10	2,959	11	2,444	12	9,783

and that the increase in the number of solved instances is due to the deadlock detection. To assess this, we compare two heuristic EMM + MPDB-4 and IPDB-4 + MPDB-4 in Table 4.8. Both heuristics break ties by INLB, in the experiment we use our usual limits. The results show that the claim is wrong. IPDB-4 + MPDB-4 solves sixteen instances while EMM + MPDB-4 solves twelve.

Table 4.9 shows our last experiment regarding the use of MPDBs to deadlock detection. We want to assess if the MPDB could improve our current best results that are obtained by IPDB-4 using INFO rule. Thus, we run two configurations, EMM + MPDB-4 and IPDB-4 + MPDB-4, both breaking ties by INFO. In the last line of the table, we provided the total number of explored nodes. EMM + MPDB-4 solves 13 instances, the same as EMM, however exploring two orders of magnitude fewer nodes. IPDB-4 + MPDB-4 also solved the same number of instances as IPDB-4, but it solves exploring considerably fewer nodes and uses less time in total.

We have proposed an effective approach for optimally solving Sokoban, by combining the use of an effective heuristic function such as IPDBs with MPDBs. We solved the same number of instances exploring two orders of magnitude fewer nodes and using less time in total. We could improve these results by investigating better approaches to combining MPDBs and IPDBs since we used a naive approach to combining them.

Table 4.8: Number of explored nodes and computation time (in seconds) for different Sokoban solvers breaking ties by rule INLB using MPDBs only for deadlock detection.

	EMM + MPDB-4		IPDB-4 + MPDB-4	
#	Nodes	Time	Nodes	Time
1	153	3	115	7
2	68,927	222	184	14
3	22,268	42	318	12
4	>61,800	3,600	1,164	600
6	1,641	7	165	7
7	127,840	276	127,840	288
17	775	7	267	18
21	>437,991	3,600	89,098	1,796
38	8,919	6	5,280	9
43	>2,832,999	3,600	331,581	1,180
51	223,106	3,068	>212,440	3,600
78	7,646	39	197	49
79	>395,134	3,600	4,548	112
80	480	109	478	158
81	198,935	2,365	198,935	2,578
82	>500,918	3,600	253	68
83	305	39	305	69
	12	24,183	16	10,565

4.4 Computational Results: Pukoban

In this section, we present results of the proposed techniques, IPDB and Fill Order, in Pukoban. We use the same implementation, algorithms, and data structures that were used in the experiments of Sokoban – we only consider the four additional pull moves. If not otherwise stated, the description of the techniques provided to Sokoban is also valid for Pukoban. In Pukoban, there are no deadlocks and the improvement of the solver cannot come from deadlock detection. Because of that we do not perform experiments with MPDBs.

In Subsection 4.4.1, we discuss the construction of the IPDB and the computation heuristic function. We describe the similarities briefly and the differences in detail. Next, in Subsection 4.4.2, we propose a variation of the Fill Order. In Subsection 4.4.3, we present experiments related to the instance decomposition and heuristic function on initial states. Finally, in Subsection 4.4.4, we compare different Pukoban solvers.

All experiments were performed on a PC with an AMD Opteron CPU running at 2.39 GHz with 32 GB. The limit for each experiment was one hour of computation

Table 4.9: Number of explored nodes and computation time (in seconds) for different Sokoban solvers breaking ties by rule INFO using MPDBs only for deadlock detection.

	EMM		EMM + MPDB-4		IPDB-4		IPDB-4 + MPDB-4	
#	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
1	169	0	125	5	122	1	122	7
2	147,281	22	66,157	190	199	3	199	14
3	20,839	2	4,988	16	232	2	232	11
4	1,590	1	1,033	293	879	250	879	500
5	>	3,582	>564,055	3,600	349	8	349	34
6	1,334,586	132	1,645	9	1,358	4	1,308	13
7	12,477,687	1,129	103,233	201	12,477,687	1,271	103,233	236
9	>	3,467	>207,166	3,600	15,686	191	15,684	325
17	1,349,943	62	776	13	298	3	297	22
38	361,852	31	133,515	61	144,094	39	98,317	70
43	>	2,090	>2,054,841	3,600	2,669	13	2,563	55
51	2,283,280	493	>373,946	3,600	2,283,280	2,176	>393,249	3,600
53	1,071	1	895	153	1,071	7	895	144
65	>15,085,591	3,600	>114,370	3,600	494	52	488	229
73	>	3,538	>169,527	3,600	743	46	743	260
78	340	0	302	58	286	3	286	49
79	>12,988,110	3,600	>374,629	3,600	4,664	31	4,639	111
80	939	1	718	180	646	15	646	159
81	>16,119,819	3,600	350	97	>183,944	3,600	350	76
82	>	2,606	>576,972	3,600	269	21	269	67
83	869	0	491	64	454	7	454	71
	13	27,960	13	30,140	20	7,743	20	6,054
	162,173,966		4,749,755		15,119,445		625,223	

time and 32 GB of memory. We perform the experiments on the standard set of instances xSokoban. As in the previous tables, in columns that report nodes, a prefix ">" means that an optimal solution has not been found because the solver reached the time limit. The number of explored nodes doesn't include the nodes explored when constructing the PDB, but reported times include the construction time.

4.4.1 Pattern Database and Heuristic Function

The computation of the IPDB in Pukoban is exactly the same as in Sokoban. First, we find the instance decomposition that maximizes the size of the maze zone. Then, having the cut square, we compute the IPDB by a reverse search starting from the intermediate abstract goal state. The reverse search continues until the whole abstract state

space is explored. The storage method is identical.

The computation of the heuristic function has two differences. First in Pukoban, there are no linear conflicts and, second the heuristic EMM for the goal zone subproblem is computed differently. When computing the heuristic for the goal zone subproblem, in Sokoban, stones in the maze zone, are treated as if they were positioned at the cut square, the same is done in Pukoban. In Sokoban, for each such stone, the position of the man is defined as the original square of the stone in the maze zone. In Pukoban the same approach does not produce an admissible heuristic, since the man could have pushed or pulled the stone to the cut square. Thus, we position the man at the reachable square that produces the minimum distance for each goal square.

4.4.2 Tie breaking Rule

The Fill Order proposed for Sokoban is used in the same manner in Pukoban. However, in Pukoban we have more flexibility to fill the goal squares, because of the four additional pull moves. Thus, we propose a variation of the Fill Order named Fill Order Improved (FI). FI uses the same procedure to compute the linearization of the goal squares used by the FO. The difference comes in the priorities assigned to each node.

In FI, the node receives a priority according to the number of stones on goal squares that strictly obey the order. Consider an instance with four stones. If a state has stones on goal squares with orders three, two and zero it will receive a priority two. If a node has stones on goals with orders three, one and zero it will receive a priority one. If a node has stones on goals with orders two, one and zero it will receive a priority zero. FI is a strict version of FO.

4.4.3 Instance Decomposition and Heuristic Function on Initial States

Figure 4.8 shows the results of the decomposition of the 90 instances from the standard set. For each instance, the y-axis shows the percentage of squares in the maze zone. The x-axis plots the instances in order of decreasing percentages. In Pukoban, two instances have maze zone with size one, and four with size two. The remaining instances have non-trivial maze zones. The mean size of the maze zone now is 46%. There are no dead squares in Pukoban. Thus, the number of free squares increases. These additional

Figure 4.8: Percentage of squares in the maze zone. The number of the instance in the standard set is plotted above each data point.

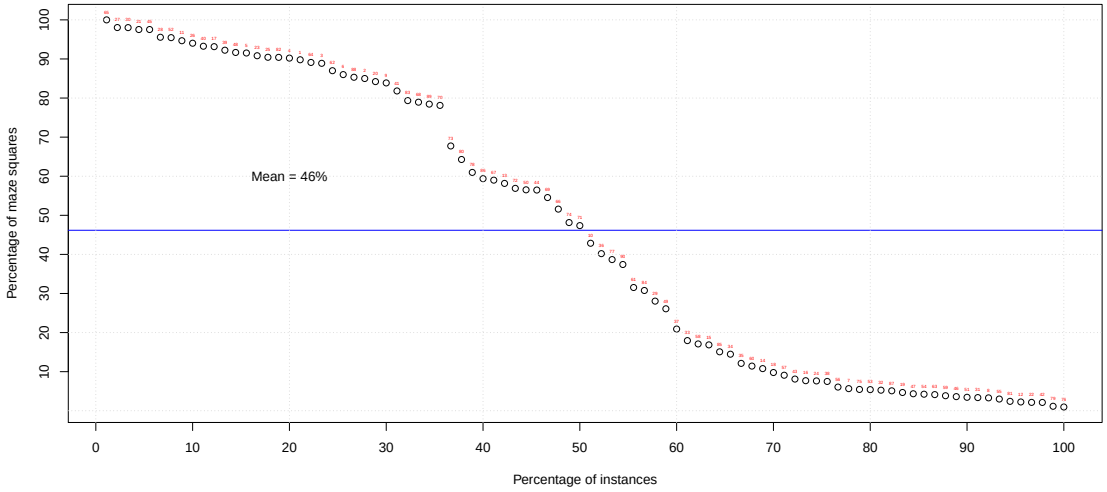


Table 4.10: Mean heuristic values on the initial states of the standard set of 90 instances. Pukoban versus Sokoban.

	EMM	IPDB-2	IPDB-4	HF
Pukoban	210.32	210.61	210.79	211.26
Sokoban	240.54	241.81	244.32	248.03

free squares increase the number of alternatives that stones can be moved to goal squares. This, results in a decrease in the mean size of the maze zone.

Table 4.10 compares the mean heuristic value for Pukoban and Sokoban on the initial states of the instances from the standard set. Different from Sokoban, in Pukoban we do not have upper bounds on the solution length for all instances. Thus, in the last column "HF', we present the highest f value found during the search by an experiment using IPDB-4 and tie breaking by rule INFO.

The table shows that HF for Pukoban is lower than for Sokoban. This is expected since with the additional pull moves shorter solutions could be found. In Pukoban, IPDBs of size 2 and 4 were able to improve marginally the results of EMM. This can be explained by the reduction in the mean size of the maze zone. The difference between EMM and HF in Pukoban is unexpected. In Sokoban, this difference is almost eight while in Pukoban is less than one. The xSokoban set is hard for Sokoban and not necessarily for Pukoban. This could explain the small improvement of the IPDBs.

4.4.4 Solving Pukoban with Pattern Databases

In this subsection, we present experiments regarding our techniques on the task of optimally solving Pukoban. We compare our results to the ones obtained by Zubaran and Ritt (2011). Zubaran and Ritt (2011) present two sets of results for optimally solving Pukoban. In the first, they solve Pukoban using an A^* guided by EMM and tie breaking by LBIN. We call this solver ZR. In the second, which is their complete solver, they add to ZR a more sophisticated scheme of tie breaking, an approach to dynamically update the heuristic function and, an admissible memory-based technique to improve the heuristic function that is similar to linear conflicts. We call this solver ZRI.

We present three sets of experiments. In the first, we compare ZR to our solver guided by an IPDB built from abstractions with two and four stones. We use the same tie breaking rule LBIN as ZR. In this experiment, we want to verify if an IPDB by itself could improve the results when comparing to the standard heuristic function for Pukoban. Then, we compare ZRI to our solver tie breaking by INFO and guided by IPDB in the same setting as before. Finally, we present our results using INFI and IPDB with abstractions with two and four stones.

Table 4.11 shows the results of the two Pukoban solvers using LBIN as tie breaking rule. ZR is able to solve 15 instances, IPDB-2 solves 22 and IPDB-4 solves 21 instances. More instances were solved by the Pukoban solver when compared to a solver with the same techniques in the Sokoban setting. In Sokoban, an IPDB-2 tie breaking by LBIN solves 12 instances. This difference could be explained by the fact that the solution lengths under Pukoban rules are shorter than under Sokoban rules. The most likely explanation is that the xSokoban set is hard for Sokoban, but not for Pukoban. The subset of instances solved is not the same when comparing the two solvers. This is due to implementation details that define the order in which the successors of a node were generated. Our solver can explore more nodes per second than ZR, but this is not the reason for the increase in the number of solved instances. An IPDB-4 limited to explore one million nodes solves 20 instances, five more than ZR.

In Table 4.12 we present the results of our solver breaking ties by INFO and the best version of the ZRI (ZUBARAN; RITT, 2011). Our solver is able to improve the results obtained by ZRI. ZRI solves 20 instances the same number that an IPDB-4 using LBIN solves when limited to explore 1 million nodes. Clearly our best technique to solve Pukoban is INFO. Using INFO, our solver is able almost to double the number

of instances solved. Using larger abstractions does not improve the number of solved instances.

Finally, Table 4.13 we presents the results of our solver breaking ties by INFI and ZRI (ZUBARAN; RITT, 2011). The modified Fill Order produces a significant improvement in the results. Our solver increases the number of solved instances by nine.

4.5 Generalization of IPDBs

In this section we discuss how IPDBs could be generalized to other problems. We give concrete examples of problems where IPDBs may lead to better heuristics. We also highlight some similarities between landmarks (HOFFMANN; PORTEOUS; SEBASTIA, 2004) and cut squares, and discuss how our approach could be an example of an abstraction based heuristic function enhanced by landmarks.

We have shown that an instance decomposition can improve the effectiveness of PDBs in Sokoban. Thus an instance decomposition is very likely to lead to improved heuristic functions in similar moving-block puzzles. These include all variants of Sokoban that have been studied in the literature, which allow to push several or even an unlimited number of stones, add the capacity of pulling stones, or require push and pull moves to slide until hitting the next obstacle (DEMAINE; HEARN; HOFFMANN, 2002; RITT, 2010; DEMAINE; HOFFMANN; HOLZER, 2004). Another example of a related puzzle where our idea may be applied is Atomix, where the player has to assemble a molecule in a maze grid. Atomix has no man and allows only slide moves, but like Sokoban has multiple goal states, since the assembly site of the molecule is not fixed.

More generally to apply IPDBs to a concrete problem, three properties are required. The problem must have a set of objects, a set of locations, and the objective is to move the objects to goal locations, in such a way that the mapping of objects to goal locations is not fixed beforehand. The effectiveness of IPDBs depends on the existence of a cut location that the objects must pass over to reach the goal locations. Such a cut location divides the locations into two sets: those reached before the cut, and those reached after the cut. In the first set, a PDB can be applied without suffering from the multiple abstract goal states. In the second set, any heuristic can be used. The admissibility of the resulting heuristic function is guaranteed by a cost partition (KATZ; DOMSHLAK, 2008a; YANG et al., 2008): each heuristic has actions with zero cost in the opposite set of locations.

Several problems from the International Planning Competition (IPC) which are

transportation problems satisfy these properties. Examples include the STORAGE and TIDYBOT domains. Both have objects with several goal locations and a topology which likely permits to find cut locations. For more realistic variants of some domains this also holds. An example is the AIRPORT domain (TRUG; HOFFMANN; NEBEL, 2004) from the IPC-4, which models an airport ground traffic planning. In this domain, airplanes must be moved from their original locations to a goal locations (runway or parking). In the definition of the domain used in the IPC, each airplane has a specific goal position. However, an airplane could as well go to one among several parking positions, instead of a specific one. The same applies to runways. Thus, we could change the objective to only define as the goal state for airplanes that they would be airborne or parked. In this setting, the planner can select the optimal park or runway locations. And thus, our approach could be used to enhance the effectiveness of PDBs.

It has been recognized that combining critical path methods, abstractions, and landmarks, three of the most successful techniques for obtaining good heuristic functions, can lead to better heuristics (DOMSHLAK; KATZ; LEFLER, 2012). In particular, the combination of landmarks and abstractions has been pursued in lines of research like cartesian (SEIPP; HELMERT, 2013; SEIPP; HELMERT, 2014) or implicit (KATZ; DOMSHLAK, 2010; DOMSHLAK; KATZ; LEFLER, 2012) abstractions. Domshlak, Katz and Lefler (2012) have shown that abstractions are highly sensitive to the goal specification, and that making landmarks explicitly available (by what they call an L-reformulation), the quality of an abstraction-based heuristic can be improved.

The effectiveness of our approach can be understood in the context of landmarks and abstractions. The cut squares used in our decomposition are related to landmarks (HOFFMANN; PORTEOUS; SEBASTIA, 2004), which are implicit sub-goals that must be accomplished in every optimal plan. Since every box b in the maze zone must pass over the cut square c, $\mathrm{at}(b, c)$ is a fact landmark, and the set of pushes which achieve $\mathrm{at}(b, c)$ form a (disjunctive) action landmark in the Sokoban domain. We use the cost partition mentioned above to obtain the subproblem of achieving this specific set of landmarks, and another subproblem of reaching the goal state from a state where all landmarks have been achieved. Landmarks usually make the heuristic path-dependent, since a landmark that has been achieved may not be true in a later state. We avoid this by relaxing the first subproblem in such a way that there exists an optimal solution where all landmarks are true in the corresponding goal state. Finally a PDB abstraction is used to find a better heuristic function for the first subproblem. It seems possible that this idea could be generalized to

be applied in a domain-independent way.

4.6 Conclusion and Future Work

We have shown that PDBs can be applied successfully to find optimal solutions to Sokoban and Pukoban and improve the current state of the art. This result is not obvious, since a PDB must be built for each instance, and these problems have $k!$ goal states, which lead to weak lower bounds. Indeed, MPDBs, a straightforward application of PDBs to Sokoban, lead to ineffective heuristic functions.

To be able to effectively apply PDBs to Sokoban and Pukoban, we partition the search space into a maze and a goal zones, to obtain an explicit intermediate goal state. In this way, we can apply PDBs to compute a lower bound on the distance to the intermediate goal state, and use any existing lower bound for the distance from the intermediate goal state to some final goal state. We also propose a new domain-specific tie breaking rule called fill order.

In Sokoban, we have shown that the lower bound can be effectively computed, dominates the state-of-the-art lower bound in 62 of the 90 instances of the standard set, and identifies four times more deadlocks in tests over random states. Our experiments also show that the fill order is an effective tie breaker. Applied together in an A* search, these techniques explore fewer nodes, and are an order of magnitude faster than other approaches. In one hour our solver finds the optimal solutions of 20 instances, compared to 6 for the state-of-the-art solver RS*, and 10 for EMM. We also improved the results in Pukoban, mainly due to our proposed tie breaking rule.

Although MPDBs result in weak lower bounds, they detect more deadlocks than the other approaches. We have shown that MPDBs can serve the role of an efficient deadlock detector and combined with other heuristic function can increase the number of optimally solved instances of Sokoban. It is further possible to use instance decompositions with several, smaller maze zones. PDBs for smaller maze zones have fewer entries which allow for larger abstractions, so we capture more interactions of stones within maze zones, but lose the interactions between maze zones. It would be also interesting to apply PDBs and better tie breaking rules in non-admissible solvers.

Table 4.11: Number of explored nodes and computation time (in seconds) for different Pukoban solvers breaking ties by rule LBIN.

#	ZR		IPDB			
			2		4	
	Nodes	Time	Nodes	Time	Nodes	Time
1	125	0	88	0	88	6
2	6,900	2	172	0	118	15
3	161,348	148	7,275	2	136	6
4	>747,745	3,600	>4,329,902	3,600	6,651	1,471
5	411,147	821	991,532	271	183	22
6	>753,529	3,600	2,523,327	515	107	8
7	97	0	2,982	1	2,982	1
8	296	1	7,461	4	7,461	4
9	>794,601	3,600	774,066	254	>110,237	3,600
11	>558,184	3,600	28,822	8	28,614	866
17	>1,498,085	3,600	124	38	124	59
21	>589,015	3,600	1,039	1	1,036	150
27	>292,889	3,600	3,029,716	2,022	>11,662	3,600
36	968,816	3,311	>3,429,532	3,600	>14,805	3,600
38	361	0	38	0	38	0
51	>564,492	3,600	5,670	1	5,670	2
56	40,727	45	>5,973,150	3,600	>22,345	3,600
76	>364,254	3,600	1,227,191	398	1,227,191	496
78	131	0	127	1	127	18
79	166	0	167	1	167	1
80	>941,829	3,600	12,570	4	225	73
81	179	0	168	1	168	1
82	33,199	28	639	1	269	68
83	1,379,790	3,578	5,940	1	195	98
84	201	0	>9,020,739	3,600	>114,641	3,600
86	>681,512	3,600	125	0	125	7
	15	47,534	22	17,925	21	21,372

Table 4.12: Number of explored nodes and computation time (in seconds) for different Pukoban solvers breaking ties by rule INFO.

#	ZRI		IPDB			
			2		4	
	Nodes	Time	Nodes	Time	Nodes	Time
1	88	0	149	0	149	6
2	2,089	1	1,369	1	1,333	21
3	12,072	1	269	0	269	8
4	328	0	1,620	5	1,585	908
5	13,728	2	293	1	293	27
6	>557,734	3,600	3,450	1	176	8
7	97	0	20,427	4	20,427	3
8	296	1	486	2	486	2
9	>533,730	3,600	435	1	433	54
11	4,555	1	3,092	2	3,083	171
13	>427,501	3,600	704	2	681	171
17	>1,502,845	3,600	181	23	179	48
19	>613,026	3,600	540	2	540	71
21	>602,391	3,600	14,844	7	14,500	373
23	7,739	3	30,700	17	30,690	3,113
33	>696,681	3,600	2,720	31	2,720	157
35	>270,892	3,600	518	3	518	126
36	897,979	3,105	697	3	696	374
38	359	0	45	0	45	0
45	>335,943	3,600	439	1	407	191
51	>559,787	3,600	230	0	230	0
52	>664,429	3,600	622	2	592	583
54	>430,857	3,600	445	2	445	97
56	40,727	43	>8,965,704	3,600	>48,408	3,600
62	236	0	389	1	389	295
63	>440,801	3,600	749	2	749	208
64	>377,204	3,600	535	1	535	332
65	>491,234	3,600	695	2	341	721
67	>898,984	3,600	641	2	641	265
68	>518,070	3,600	524	2	524	334
70	>585,147	3,600	777	2	755	318
73	>1,153,156	3,600	717	2	703	255
78	131	0	225	1	225	17
79	166	0	385	1	385	1
80	936,075	3,278	2,565	2	1,823	110
81	178	0	319	1	319	1
82	4,585	1	702	1	600	71
83	74,091	16	3,596	2	3,253	120
84	157	0	>7,585,369	3,600	>116,350	3,600
86	>740,512	3,600	1,661,768	326	>647,756	3,600
	20	78,452	38	7,656	37	20,360

Table 4.13: Number of explored nodes and computation time (in seconds) for different Pukoban solvers breaking ties by rule INFI.

#	ZRI		IPDB			
			2		4	
	Nodes	Time	Nodes	Time	Nodes	Time
1	88	0	149	0	149	6
2	2,089	1	1,369	1	1,333	21
3	12,072	1	269	0	269	8
4	328	0	1,620	4	1,585	944
5	13,728	2	293	1	293	30
6	>557,734	3,600	3,450	1	176	10
7	97	0	214	0	214	0
8	296	1	568	2	568	2
9	>533,730	3,600	435	1	433	60
11	4,555	1	17,810	7	17,798	729
13	>427,501	3,600	1,123	3	1,036	263
16	>562,795	3,600	4,648	2	4,648	246
17	>1,502,845	3,600	181	0	179	57
19	>613,026	3,600	540	1	540	85
21	>602,391	3,600	334	1	329	139
23	7,739	3	10,953	6	10,946	1,639
31	>412,396	3,600	3,900,173	1,788	3,900,173	1,949
32	>306,827	3,600	4,091,736	1,872	4,091,736	2,043
33	>696,681	3,600	447	17	447	64
34	>404,778	3,600	58,194	36	58,194	3,420
35	>270,892	3,600	656	2	656	167
36	897,979	3,105	18,768	30	>7,294	3,600
38	359	0	388	0	388	0
45	>335,943	3,600	439	1	407	231
48	>223,040	3,600	275,391	693	>1,294	3,600
51	>559,787	3,600	318	0	318	1
52	>664,429	3,600	622	2	592	650
54	>430,857	3,600	646	2	646	146
55	>676,478	3,600	418	1	418	27
56	40,727	43	7,769	7	7,769	1,318
59	>484,496	3,600	988	2	988	214
61	>344,446	3,600	716	2	716	506
62	236	0	389	1	389	301
63	>440,801	3,600	749	2	749	201
64	>377,204	3,600	535	1	535	335
65	>491,234	3,600	633	2	341	721
67	>898,984	3,600	641	2	641	263
68	>518,070	3,600	524	2	524	337
70	>585,147	3,600	1,526	2	1,259	408
73	>1,153,156	3,600	717	2	703	257
76	>365,590	3,600	607	2	607	2
78	131	0	225	1	225	18
79	166	0	385	1	385	1
80	936,075	3,278	9,198	5	5,107	198
81	178	0	355	1	355	1
82	4,585	1	702	1	600	70
83	74,091	16	3,596	2	3,253	117
84	157	0	>7,673,373	3,600	>109,088	3,600
	20	107,252	47	8,111	45	29,003

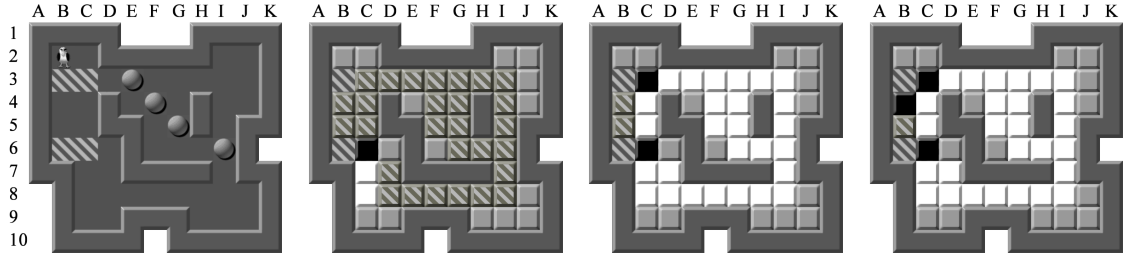
5 IMPROVED HEURISTIC AND TIE-BREAKING FOR OPTIMALLY SOLVING MOVING-BLOCKS PROBLEMS

In this chapter, we present a novel admissible pattern database heuristic (D) and tie-breaking rule (L) for Sokoban, allowing us to increase the number of optimally solved standard Sokoban instances from 20 to 28 and the number of proved optimal solutions from 25 to 32 compared to previous methods. The previously best heuristic for Sokoban (I) – presented in Chapter 4 – used the idea of an intermediate goal state to enable the effective use of pattern database heuristics in transportation domains, where the mapping of movable objects to goal locations is not fixed beforehand. We extend this idea to allow the use of multiple intermediate goal states and show that heuristic I is no longer effective. We solve this problem and show that our heuristic D is effective in this situation. We first reintroduce heuristic I with a new notation. This presentation will be useful to introduce the new heuristic D in Section 5.2. In Section 5.5 we discuss the proposed techniques and present preliminary results that show that our heuristics and tie-breaking rules are effective in other domains.

5.1 Background

Let Q be the set of free squares of an instance, $G \subseteq Q$ the set of goal squares and B the set of stones. A state u is a pair $u = (p(B), m)$, where p is a map from the stones to the free squares and m is the position of the man. For a stone at square $p(b)$ let $\delta(p(b), g)$ be the minimum cost to push the stone at $p(b)$ to square g when the man is at m in an instance with only one stone. This cost ignores all other stones. The enhanced minimum matching (EMM) is the standard heuristic of Sokoban, it is based on a minimum cost perfect matching in the complete bipartite graph between stones and goal squares with edge set $\{(b, g) \mid b \in B, g \in G\}$ and weights $\delta(p(b), g)$. In this bipartite graph let M^* be a minimum cost perfect matching. The matching cost is enhanced with linear conflicts that increase the heuristic value by two when a pair of adjacent stones is in the optimal path of each other, each stone can be part of only one linear conflict. Let L be the number of linear conflicts in a state. Then, the value of EMM for a state u is,

Figure 5.1: Decompositions with one (5.1b), two (5.1c) and three (5.1d) cut squares of the instance shown in (5.1a). Cut squares are shown in black, maze zone squares in white, goal zone squares in brown with diagonal lines and dead squares in gray.



(a) Instance. (b) One cut square. (c) Two cut squares. (d) Three cut squares.

$$\texttt{EMM} = \sum_{(b,g)\in M^*} \delta(p(b), g) + 2L.$$

A series of articles (PEREIRA; RITT; BURIOL, 2013; PEREIRA; RITT; BURIOL, 2014b; PEREIRA; RITT; BURIOL, 2015) introduced admissible PDB heuristics to Sokoban. The heuristic `I` (PEREIRA; RITT; BURIOL, 2015) provides the best previous results. It uses an instance decomposition to obtain an intermediate goal state. If all stones on a set of non-dead squares M have to pass over a fixed *cut square* c to be pushed to all reachable goal squares, the cut square c is used as an intermediate goal state. The squares in M are the *maze zone* squares and all other non-dead and non-goal squares are the *goal zone* squares. Figure 5.1b shows an instance decomposition: the cut square shown at $6C$ is part of the maze zone, the two maze zone squares at $7C$ and $8C$ and for example a goal zone square at $7D$.

The heuristic `I` is the sum of the cost to solve two independent subproblems that are a relaxation of the original problem. The cut square defines the intermediate goal state and it can store many stones and the man simultaneously (this is the relaxation). The maze subproblem corresponds to the cost to push stones that are in the maze zone to the cut square. The goal subproblem corresponds to the cost to push stones at the cut square and in the goal zone to the goal squares. Let h^M and h^G be respectively the heuristic functions for the maze and goal subproblems. The value of heuristic `I` for a state u is defined as the sum of h^M and h^G.

A PDB is used to compute h^M. The abstraction used maintains only k' of all k stones in the instance, a PDB-k' uses an abstraction of k' stones. For this PDB, there is a unique abstract goal state that has k' stones placed at the cut square. To build the PDB the algorithm performs a reverse search from the abstract goal state to enumerate all

reachable abstract states with stones k' in the maze zone. Let $\delta(p(P))$ be the cost stored in the PDB for an abstract state u' with stones $P \subseteq B$. To compute h^M the algorithm creates a partition $\mathcal{B}$ of all stones in the maze zone into parts P of size k'. Among all possible partitions $\mathcal{B}$ the heuristic tries to find one that maximizes the heuristic value as described in (PEREIRA; RITT; BURIOL, 2015). The value of h^M for a state u is,

$$h^M = \sum_{P \in \mathcal{B}} \delta(p(P)).$$

A modified state is used to compute h^G. Stones in the maze zone are placed at the cut square and the position of the man is changed accordingly, all stones in the goal zone remain the same. A complete description of the procedure can be found in (PEREIRA; RITT; BURIOL, 2015). Let d be this new map from the stones to free squares and position of the man. Let L be the number of linear conflicts in a state considering only stones in the goal zone and at the cut square. Finally, the value of h^G for a state u is defined as,

$$h^G = \sum_{(b,g) \in M^*} \delta(d(b), g) + 2L.$$

5.2 Proposed Pattern Database Heuristic based on Multiple Intermediate Goal States

In this section, we describe a simple method to extend the heuristic `I` to use multiple intermediate goal states and we show why it is ineffective. We present a more complex approach our novel admissible PDB heuristic `D` and describe how to compute the it efficiently in Section 5.2.2.

5.2.1 Instance Decomposition and Independent Subproblems

We want to find a set of cut squares $\mathcal{C}$ that maximizes the size of the maze zone M. To find such a set we analyze all possible sets of cut squares of fixed size. Given a set $\mathcal{C}$ we perform k reverse searches one for each goal square in which we place a single stone at the goal square. In all searches the squares in $\mathcal{C}$ are blocked for the stone, it cannot be placed at them. Then, all reachable non-dead and non-goal squares in these searches

are part of the goal zone, and all other non-dead squares are part of the maze zone. By construction, all stones in the set of non-dead squares M can only be pushed to the goal zone passing over the squares in $\mathcal{C}$.

We place a number in front of the letter `I` (and later `D`) to define the number of cut squares used. `1I` uses one cut square and `2I` uses two cut squares. The PDB construction for heuristic `I` using more cut squares is the same as the original approach the only difference is that there are more abstract goal states. Each combinations of k' stones on the cut squares generates an abstract goal state. With two cut squares, x stones on one cut square and y on the other, and $k' = 4$ stones, we have five abstract goal states $(x, y) = (0, 4), (1, 3), (2, 2), (3, 1)$ and $(4, 0)$. Each pair (x, y) is a unique abstract goal state. When computing h^G for each stone in the maze zone, it is unknown which cut square will be used in pushing that stone into the goal zone. Then, when placing stones in the maze zone at the cut squares, the stone will be placed at the closest cut square for each goal square. The remainder of the heuristic computation is the same.

An instance decomposition of Figure 5.1a using one cut square, Figure 5.1b, finds a maze zone with three squares ($6C$, $7C$ and $8C$). Because of that `1I` and plain `EMM` have the same heuristic value of 27 for that instance. In an instance decomposition using two cut squares (Figure 5.1c), the maze zone comprises almost the whole instance, a three cut squares decomposition increases the maze zone by one square (Figure 5.1d). `2I` with a PDB-$k' = 4$ provides the optimal solution cost of 23 for the maze subproblem, where all stones are pushed to the cut square on $3C$. When computing h^G, two stones are placed at each cut square and the cost of this subproblem is two. `2I` provides the heuristic value of 25 which is lower than plain `EMM`; because of that small difference, `2I` expands 100 times more nodes to solve the instance.

The solution to improve the heuristic value is to solve the subproblems recognizing that they are dependent – the number of stones in each cut square in both subproblems has to be the same. We call this heuristic `D`. Using it in the instance shown in Figure 5.1a it provides the heuristic value of 31 – the optimum solution cost.

5.2.2 PDB Construction and Heuristic Computation

In heuristic `D`, we have a unique PDB for each abstract goal state generated by the combinations of k' stones on the cut squares. One PDB for each pair (x, y) of the previous example. Let a be an assignment that maps stones in the maze zone to cut

squares, stones in the goal zone remain the same. Let A be the set of all possible such assignments. Let $\delta_{a(P)}(p(P))$ be the cost in one of the PDBs for an abstract state with subset of stones P, the number of stones in each cut squares defined by $a(P)$ selects the PDB. To guarantee admissibility of heuristic D we have to find an assignment a that minimizes the sum of h^M and h^G. Thus, the value of heuristic D for a state u is defined as,

$$\mathtt{D} = \min_{\forall a \in A} \left[\sum_{P \in \mathcal{B}} \delta_{a(P)}(p(P)) + \sum_{(b,g) \in M^*} \delta(a(b), g) \right] + 2L.$$

In the following, we describe how to compute the partition $\mathcal{B}$ and an optimal assignment efficiently.

5.2.2.1 Partitioning Computation

When computing $\mathcal{B}$ it is unknown at the time which cut square will be assigned to each stone. We assume that each subset P will be assigned to the set of cut squares that minimizes its cost and thus selecting the PDB with minimum cost. If $k' = 2$, the optimal $\mathcal{B}$ can be found in polynomial time by a maximum weighted matching. If $k' > 2$, we use a greedy randomized constructive method based on the one proposed by (PEREIRA; RITT; BURIOL, 2015). The method starts by querying the cost of every subset, all $\binom{k}{k'}$ subsets. Then, it ranks the subset according to the number of conflict pushes: the difference between the cost of the subset and the cost to push each stone in the subset individually to its closest cut square. Then, a greedy randomized method selects a disjoint partition trying to maximize the sum of the conflict pushes. Only subsets with all stones in the maze zone are included in $\mathcal{B}$.

5.2.2.2 Assignment Computation

The assignment computation, in general, is the costly part of the heuristic. If the sum of conflict pushes in $\mathcal{B}$ is zero, we return the value of EMM. We only assign cut squares for stones in parts P with conflict pushes greater than zero, all other parts are removed from $\mathcal{B}$. The intuition is that we just have to select cut squares for stones that are likely to increase the heuristic value. The simplest approach is to compute the cost of all possible assignments. When checking all possible assignments we call the heuristic the exhaustive

heuristic $\texttt{D}^E$. The difficulty is that the number of stones in $\mathcal{B}$ could be k and thus, we have to check the cost of $|\mathcal{C}|^k$ assignments for a single state. To find an optimal assignment more efficiently we propose the use of a branch and bound computation.

We use a best-first branch and bound computation (BB). At each step in the BB a stone is assigned to a cut square. At the beginning, no stones have cut squares assigned and thus the lower bound is equal to EMM. If all stones in a part have cut squares then we compute the lower bound, otherwise, we just use the lower bound of the parent in the BB tree. The lower bound is defined as the cost of the two subproblems, but only parts where all stones have assigned cut squares use the cost of the PDB. The upper bound is defined as the cost of EMM plus the number of conflict pushes. The heuristic D using the BB is called $\texttt{D}^B$.

5.2.3 Admissibility

In this subsection, we show that the heuristic D is admissible. Let h^* be the perfect heuristic. It is witnessed by some optimal sequence of actions S. For each stone in any part of $\mathcal{B}$, consider the corresponding subsequence of S that brings it for the first time to some cut square. Such a subsequence must exist, by definition of the cut squares. The final position of each stone in these subsequences defines an assignment a of the stones in $\mathcal{B}$ to the cut squares. For all the stones in $\mathcal{B}$, there must be a subsequence disjoint from the subsequence above, that brings it from the cut square to some goal square. Similarly, for all remaining stones there must be such a sequence. These subsequences define a matching M of stones to goal squares. Therefore, we have a pair (a, M) and the value of h^* can be defined as $h^*(a, M) = \delta(p(\mathcal{B}), a(\mathcal{B})) + \delta(a(B), M(B))$. Let $h^D(a^*, M^*) = h^M(p(\mathcal{B}), a^*(\mathcal{B})) + h^G(a^*(B), M^*(B))$ be the heuristic D with an optimal assignment a^* that minimizes the total cost given an optimal matching M^*.

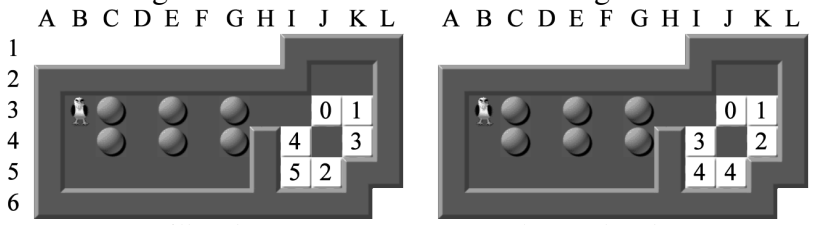
Theorem 5.2.1. *$h^D(a^*, M^*)$ is admissible.*

Proof. For any state, we want to show that $h^D(a^*, M^*) \leq h^*(a, M)$.

By definition, the pair (a^*, M^*) minimizes the value of h^D, any other pair (a, M) cannot provide a lower value. Thus, we have that $h^D(a^*, M^*) \leq h^D(a, M)$.

Now, we want to show that $h^D(a, M) \leq h^*(a, M)$ where (a, M) is the pair extracted from the optimal sequence S. First, consider h^M and $\delta(p(\mathcal{B}), a(\mathcal{B}))$. Both heuristics compute the cost to push the same set of stones from their original squares $p(\mathcal{B})$ to

Figure 5.2: Two different tie-breaking rules.



(a) Fill Order F. (b) Level Order L.

the same set of cut squares $a(\mathcal{B})$. The value of $\delta(p(\mathcal{B}), a(\mathcal{B}))$ accounts for conflicts between all stones in $\mathcal{B}$, but the value of h^M accounts only for conflicts that occur within each part P and thus it must be a lower bound on $\delta(p(\mathcal{B}), a(\mathcal{B}))$. A similar argument also applies for $\delta(a(B), M(B))$ and h^G. All stones B have the same original $a(B)$ and destination $M(B)$ squares. The value of h^G accounts only for linear conflicts while $\delta(a(B), M(B))$ accounts for all conflicts which include linear conflicts. Thus, h^G must be a lower bound on $\delta(a(B), M(B))$. Therefore,

$$h^D(a^*, M^*) \leq h^D(a, M) \leq h^*(a, M).$$

□

5.3 Tie-Breaking Rule

Having the f-value equal to the optimal solution length does not necessarily mean a solution is close at hand; there can be a prohibitively large number of states remaining to be expanded. This makes tie-breaking rules important. When comparing two Sokoban states, the one with more stones on goal squares, in general, is closer to being solved. Moreover, there is an optimal order to place stones on goal squares such that a solution can be found or the solution cost is minimized. The tie-breaking rule F (PEREIRA; RITT; BURIOL, 2015) explores these ideas to speed up the process of finding solutions. Using F the search expands fewer nodes and solves more instances, but it has three main sources of errors: the total order of the goal squares, the rule used to define the order, and the assignment of partial priorities.

A total order may be attempting too much. When there is insufficient information to define an order to fill the goal squares one should not be preferred to the other. Figure 5.2a shows an order defined by F. In this instance according to the rule F, the goal square with number three has to receive a stone before the goal square with number

two, which is not a feasible solution and thus the order is wrong. The rule used to define the order could be strengthened to avoid this type of error. Also, giving partial priorities may prioritize nodes with stones on goal squares with lower priorities such that those with higher priorities cannot be filled anymore. These problems may cause the search to explore a large portion of the search space without finding a solution.

To solve these problems we propose the tie-breaking rule level order (`L`). We compute `L` by placing all stones on goal squares. Then, iteratively we remove a stone with reverse moves. A stone is considered removed if it can reach a square that has a stone on it on the initial state of the instance without moving other stones on goal squares. Goal squares that have their stones removed in the same iteration receive the same priority. If after one iteration no stone can be removed, all the remaining goal squares with stones receive the highest priority. Figure 5.2b shows the order defined by `L`. During the search, a node receives a priority of the goal square with a stone only if all the goal squares with higher priority (bigger numbers) already have stones. For example, (a) a node with a single stone on a goal square with priority three will not receive any priority, and (b) a node with two stones on goal squares with priority four and one stone on goal square with priority zero will receive a priority of two.

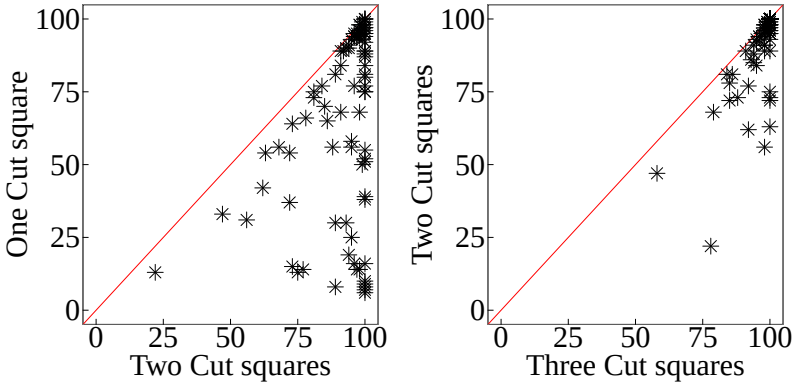
5.4 Experimental Results

In this section, we compare our proposed heuristic `D` and tie-breaking rule `L` with the state-of-the-art heuristic `I` and tie-breaking rule `F`. As well, we include extending heuristic `I` to multiple intermediate goal states. All experiments were run on a PC with an AMD FX-8150 CPU running at 3.6 GHz with 32 GB of RAM. We use the standard limits for Sokoban of 20 million expanded nodes or one hour of CPU time. In our experiments a search ended either because a solution was found or the time limit was reached, the node limit was never reached. The standard test set *xSokoban* of 90 instances is used.

5.4.1 Instance Decomposition and PDB Construction

Our first experiment is regarding instance decomposition. We compare the percentage of maze zone squares obtained by one, two, and three cut squares. These results are shown in Figure 5.3. Over the 90 instances, decompositions with one, two, and three

Figure 5.3: Comparison of the percentage of maze zone squares considering different numbers of cut squares: (5.3a) two cut squares compared to one cut square has a $>$ 10% increase in the size of the maze zone in 48 instances, and (5.3b) three cut squares compared to two cut squares has it in 14 instances.



(a) One vs. two cut squares.

(b) Two vs. three cut squares.

cut squares on average include respectively 66%, 92% and 97% of the squares in the instance are maze zone squares. Using two cut squares has 100% of the maze zone squares in 33 instances, while using one cut has only three instances.

A two cut squares decomposition provides a considerable improvement in the percentage of maze zone squares over one cut square. Because of that we perform experiments to at most two cut squares. PDBs built with $k' = 4$ are the largest that can be built for all instances in one hour, and previously provided the best results. Thus, we fix the size of $k' = 4$ in all our experiments. On average, PDB-4 for `2I` has $323,572$ entries and takes 15 seconds to build, while `2D` has $695,606$ entries and takes 108 seconds.

5.4.2 Heuristics on Initial States and Proved Optimal Solutions

Table 5.1 shows the heuristic values for the initial states of the 90 instances. Column `UB` shows the best-known upper bound. The first information to be noted is that heuristic `I` when extended from one cut square `1I` to two cut squares `2I` has worse results in general, but it is still able to increase the heuristic value on some instances (e.g. #7). `2D` improves the heuristic value on average by 1.71 compared to `1I` over instances where `1I` doesn't provide the best-known solution. For some instances it may be hard to improve the heuristic value. For example, consider #10: the heuristic has not improved, but `1I` has 32% maze zone squares while `2D` has 100%. Thus during the search `2D` will detect more deadlocks. `2D` improves the heuristic value in 25% of the instances compared to `1I`, including an enormous improvement of 18 for instance #33.

Table 5.1: Heuristic values for the initial states of the standard set of 90 instances. Highlighted cells in columns 1I, 2I and 2D have values equal to the best-known solution. Improved values over 1I are shown in bold. Highlighted cells in column column UB show proved optimal solution lengths: when the time limit is reached, the lowest f-value on the open list is equal to the best-known solution. 1I proves the optimal solution length of 25 instances and $2D^B$ of 32 instances.

#	1I	2I	2D	UB	#	1I	2I	2D	UB	#	1I	2I	2D	UB
1	97	93	97	97	31	236	233	236	250	61	253	**255**	**255**	263
2	131	123	131	131	32	115	**122**	**129**	139	62	241	236	241	245
3	134	126	134	134	33	152	140	**170**	174	63	429	424	429	431
4	355	343	355	355	34	164	164	164	168	64	381	373	381	385
5	141	133	141	143	35	368	352	368	378	65	207	203	207	211
6	106	100	106	110	36	511	502	511	521	66	193	193	193	325
7	80	**86**	**86**	88	37	242	225	**246**	284	67	395	392	395	401
8	220	220	220	230	38	79	79	79	81	68	333	329	333	341
9	231	222	231	237	39	658	598	658	672	69	223	217	223	433
10	510	510	510	512	40	314	306	314	324	70	329	327	329	333
11	213	209	213	241	41	227	227	227	237	71	298	**300**	**302**	308
12	206	198	**208**	212	42	208	204	208	218	72	294	262	294	296
13	224	224	224	238	43	138	130	138	146	73	441	437	441	441
14	231	231	231	239	44	169	165	169	179	74	182	176	**190**	212
15	100	**106**	**108**	122	45	290	282	290	300	75	263	**267**	**273**	295
16	170	162	170	186	46	227	217	227	247	76	194	**196**	**198**	204
17	203	199	203	213	47	201	183	201	209	77	360	238	**364**	368
18	106	103	106	124	48	200	186	200	200	78	136	136	136	136
19	286	278	286	302	49	106	88	**114**	124	79	170	149	170	174
20	450	374	450	462	50	102	100	100	370	80	231	201	231	231
21	137	126	137	147	51	118	100	118	118	81	167	141	**173**	173
22	308	307	**310**	324	52	379	359	379	421	82	137	131	137	143
23	432	428	432	448	53	186	182	186	186	83	194	184	194	194
24	518	517	**534**	544	54	181	178	181	187	84	153	149	**155**	155
25	378	361	378	386	55	120	115	120	120	85	307	307	307	329
26	175	162	175	195	56	193	193	**201**	203	86	124	112	124	134
27	359	350	359	363	57	217	**219**	**219**	225	87	223	217	223	233
28	290	287	290	308	58	197	189	197	199	88	342	336	342	390
29	132	130	132	164	59	218	**222**	**222**	230	89	361	351	361	379
30	407	400	407	465	60	148	147	**150**	152	90	446	**447**	**448**	460

Highlighted entries in Table 5.1 are the instances for which the optimal solution cost is now known. If the heuristic value of the initial state or the lowest f-value on the open list when the time limit is reached is equal to the best-known solution, then the optimal solution length is known. Considering only the heuristic value for the initial states 14 instances have the optimal solution proved. 1I, 2I and 2D prove respectively 12, 1 and 14. The ones proved by 1I and 2I are a subset of 2D. Considering the f-value when the time limit is reached, 32 instances have their optimal solution cost proven. $2D^B$ proves 32, and 1I proves 25, a subset of $2D^B$. For some instances, we may not prove the optimality of the solution cost because the upper bound is loose (best human solution). However, we can compare which heuristic is closer to solve the instance: over all algorithms, 1I has the highest or equal highest f-value on the open list at the end of the search for 64 instances while $2D^B$ has it for 87 instances.

5.4.3 Solved Instances

All methods use the same basic code infrastructure and an A* search. We applied an additional improvement to heuristic I to make the comparison fairer: only use the cost in the PDB if the whole subset of stones is in the maze zone. Doing this 1I can solve one more instance.

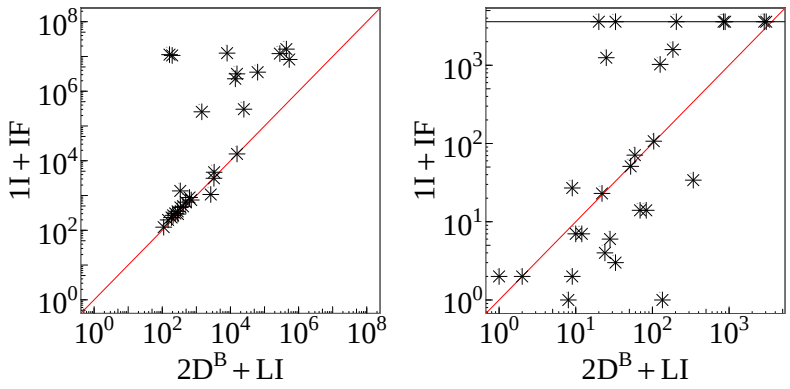
Table 5.2 shows all instances solved by at least one of the methods. We use two combinations of tie-breaking rules: IF corresponds to the previous best tie-breaking rule using *inertia* (JUNGHANNS; SCHAEFFER, 2001) as first order rule, and *fill order* as second order rule, and LI corresponds to our proposed tie-breaking rule L as first level rule, and *inertia* as second order rule. For each heuristic (1I, 2I, $2D^E$, $2D^B$) there are two columns, one for each tie-breaking rule (IF, LI). A dot in a specific column indicates that the instance defining the row was solved by that combination of heuristic and tie-breaking rule. With the exception of 2I all heuristics solve more instances using LI. The improvement is more significant in $2D^B$ solving five more instances, but even 1I benefits from LI. For example in instance #21 it expands 100 times fewer nodes and reduces the time by more than half an hour.

Table 5.2: Solved instances for different heuristics and tie-breaking rules. Only instances solved by at least one of the methods are shown.

#	1I		2I		2D^E		2D^B	
	IF	**LI**	**IF**	**LI**	**IF**	**LI**	**IF**	**LI**
1	•	•	•	•	•	•	•	•
2	•	•			•	•	•	•
3	•	•			•	•	•	•
4	•	•			•	•	•	•
5	•	•			•	•	•	•
6	•	•			•	•	•	•
7	•	•	•	•	•	•	•	•
9	•	•			•	•	•	•
17	•	•	•	•	•	•	•	•
21	•	•					•	•
33								•
38	•	•	•	•	•	•	•	•
43	•	•			•	•	•	•
48		•						•
51	•						•	•
53	•	•			•	•	•	•
55		•				•		•
57								•
60								•
65	•	•			•	•	•	•
73	•	•			•	•	•	•
78	•	•	•	•	•	•	•	•
79	•	•			•	•	•	•
80	•	•			•	•	•	•
81					•	•	•	•
82	•	•			•	•	•	•
83	•	•			•	•	•	•
84							•	•
Tot.	21	22	5	5	20	21	23	**28**

Comparing the heuristics, `2I` can only solve five instances, showing that the extension to multiple intermediate goal states is ineffective with heuristic `I`. Even with a

Figure 5.4: Comparison of the heuristics 1I+IF (previous state-of-the-art) and 2D^B+LI.



(a) Number of expanded nodes. (b) Total time in seconds.

more accurate heuristic, 2D^E cannot solve more instances than 1I due to the cost of expanding nodes. 2D^B can solve more instances than 1I with both tie-breaking rules. In computing Table 5.2, 2D^B on average expands 10 times fewer nodes per second than 1I, but it solves seven more instances. In large search spaces the comparison of 1I and 2D^B indicates that a heuristic that is more informed can be beneficial, even if more computationally expensive. Another point highlighted by the results of 2D^B is the effort to prove the optimality and to find the solution. 2D^B can prove the optimality of 32 instances. However, it can only find the optimal solution for 28.

Figure 5.4 shows the detailed results of 1I+IF, the previous state-of-the-art, compared to 2D^B+LI. 2D^B uses at most 307 seconds more than 1I to solve any instance (#73). Comparing only the instances solved by both methods, 2D^B uses more time in 13 instances while 1I uses more time in 15 instances. Regarding time, there is no clear winner. Regarding expanded nodes 2D^B is the clear winner. Comparing only the solved instances by both methods 2D^B expands 200 times fewer nodes on average.

The main limitation of our approach is that an increase of the number of cut squares will make the heuristic computation more costly. If the number of cut squares is similar to the number of goal locations the heuristic D is unlikely to improve the results. It is reasonable to increase the number of cut squares given that we are not using any specific method to prune the BB and that many selections of cut squares will not produce an optimal solution. In Sokoban for example, it is often the case that if more than one stone chooses the same cut square the solution is already infeasible. We could detect this infeasibility early without EMM. Pruning methods based on these cases could increase the efficiency of our method.

5.5 Experiments in Other Domains

We have shown how to effectively apply PDB heuristics in transportation domains where the mapping of movable objects to goal locations is not fixed beforehand and where multiple intermediate goal states are helpful. We use our heuristic `D` and tie-breaking rule `L` and solve optimally more instances than previous methods. Domains like ATOMIX (HÜFFNER et al., 2001) and AIRPORT-IPC-4 (TRUG; HOFFMANN; NEBEL, 2004) require `D` instead of heuristic `I` because in these domains the heuristic `1I` in general doesn't provide maze zones with effective size. Tie-breaking rules inspired in `L` could also improve the results in these domains. Other domains with similar characteristics like STORAGE and TIDYBOT (both from IPC) are likely to benefit from our techniques.

In this section, we present preliminary experiments related to applying our techniques in domain-independent planning. In Subsection 5.5.1 we compare a domain-independent PDB heuristic to a simple matching heuristic in a realistic version of AIRPORT-IPC-4. In Subsection 5.5.2 we present preliminary results of simple versions of our tie-breaking rules applied as domain-independent techniques.

5.5.1 Solving Airport

We perform two modifications to the AIRPORT-IPC-4 domain making it more similar to the real-world problem. First, we modify the goal state. In the original formulation, all airplanes have a defined goal (parking and take off) positions. In a more realistic formulation, the controller would be able to assign goal positions to airplanes. Thus, we modify the goal state to define as the goal for airplanes only to be parked or airborne, leaving free the choice of the specific position. Second, we add an action to land. In the original formulation, there are many actions to move airplanes through the airport, but there is no action to land. For this reason, there is a strong imbalance between the number of airplanes that have to park compared to the number of airplanes that have to take off. Thus, we add an action to land airplanes. For all instances, we add k airplanes that have as initial state airborne and as goal state to park, where k is the number of airplanes that have to take off in the original formulation. We perform experiments using the modified version of the domain.

We compare three heuristic functions `iPDB`, `Closest` and `MM`. `iPDB` (HASLUM et al., 2007; SIEVERS; ORTLIEB; HELMERT, 2012) is an efficient implementation of

PDBs for domain-independent planning. Closest and MM are simple domain-dependent heuristics. Both heuristics are based on the cost to an airplane a at position u move to position v if a is the only airplane in the airport. These distances are computed in a preprocessing phase and stored in a lookup table. Closest is defined as the sum of the cost for all airplanes reach the closest position that satisfies their goal state. Closest ignores the fact that all airplanes with park goals must end up in different park positions. MM solves this problem using a minimum cost perfect matching between airplanes and park positions. Thus, MM is defined as the sum of the cost for all airplanes reach their closest take off position and the cost of the minimum matching.

The iPDB heuristic may have two sources of ineffectiveness in this domain: the pattern selection and the multiple abstract goal states. Being a domain-independent heuristic iPDB has to select informative patterns without domain-dependent knowledge, a poor selection of patterns may result in a weak heuristic function. Furthermore, as we showed in Chapter 4 standard PDBs are ineffective in transportation domains where the mapping of movable objects to goal locations is not fixed beforehand. We use the heuristic Closest to emulate a PDB heuristic with atomic projections and a perfect pattern selection. Compared to MM, the only source of ineffectiveness of Closest are the multiple abstract goal states.

We implemented Closest and MM in Fast Downward (HELMERT, 2006a), thus, all heuristics share the same basic framework. All experiments use the same hardware as before and we use the time limit of 30 minutes and the memory limit of 4 GB for each instance. We use an A* algorithm and the version of iPDB implemented in Fast Downward with default parameters. Table 5.3 compares the initial heuristic value and the best f-value of the heuristics. The best f-value is the lowest f-value on the open list when time or memory limits are reached, or the solution cost if the instance is solved. Closest and iPDB solve 18 instances, a subset of the instances solved by MM. MM solves 19 instances, #19 is the additional instance solved. Closest and MM can improve the best f-value for many instances when compared to iPDB. In general, we can observe that the multiple abstract goal states reduce both the initial heuristic value and best f-value of Closest and iPDB, and that MM provides better results. These results indicate that our heuristics could improve the results in this domain.

Table 5.3: Values of the initial heuristic (h) and best f-value (f) for Airport instances. Improved values over iPDB are shown in bold. Highlighted cells in column # are solved instances.

#	iPDB		Closest		MM	
	h	f	h	f	h	f
1	8	8	8	8	8	8
2	19	19	16	19	16	19
3	17	17	16	17	16	17
4	20	20	20	20	20	20
5	43	43	40	43	40	43
6	63	63	60	63	60	63
7	63	63	60	63	60	63
8	84	84	80	84	80	84
9	71	71	69	71	69	71
10	18	18	18	18	18	18
11	41	41	38	41	38	41
12	59	59	56	59	56	59
13	57	57	54	57	54	57
14	100	102	94	102	96	102
15	98	100	92	100	94	100
16	139	139	130	134	134	138
17	126	126	119	123	123	126
18	145	145	137	140	141	144
19	108	110	104	108	108	**112**
20	133	133	127	130	131	133
21	146	148	145	148	**147**	148
22	193	197	192	197	**196**	197
23	195	195	194	**197**	**202**	**202**
24	309	309	306	306	**314**	**314**
25	359	359	356	356	**368**	**368**
26	349	349	346	346	**364**	**364**
27	451	451	**490**	**490**	**508**	**508**
28	500	500	**538**	**538**	**562**	**562**
29	481	481	**534**	**534**	**566**	**566**
30	612	612	**654**	**654**	**686**	**686**
31	615	615	**702**	**702**	**744**	**744**
32	566	566	**716**	**716**	**768**	**768**
33	602	602	**787**	**787**	**839**	**839**
34	555	555	**803**	**803**	**865**	**865**
35	604	604	**783**	**783**	**857**	**857**
36	105	105	104	105	104	105
37	198	198	194	195	198	198
38	171	171	168	169	170	170
39	249	249	246	246	**254**	**254**
40	240	240	237	237	**241**	**241**
41	189	189	186	186	**190**	**190**
42	293	293	289	289	**297**	**297**
43	259	259	257	257	**265**	**265**
44	229	229	226	226	**232**	**232**
45	244	244	242	242	**252**	**252**
46	310	310	309	309	**321**	**321**
47	293	293	**368**	**368**	**388**	**388**
48	374	374	**507**	**507**	**539**	**539**
49	373	373	**510**	**510**	**544**	**544**
50	514	514	**819**	**819**	**907**	**907**
Avg.	239.80	240.04	270.92	272.44	285.00	286.18
Solved	18		18		19	

5.5.2 Domain-Independent Tie-Breaking Rules

In this subsection, we perform experiments with all domain from the first to the seventh IPC. In total, there are 1050 instances divided among 33 domains, many of these domains are based on real-world problems. The tie-breaking rule *inertia* is simple to define as a domain-independent technique: it prioritizes states generated by the longest number of actions that change the value of the same variable in sequence. We have implemented this tie-breaking rule in Fast Downward. The simplest possible version of fill/level order prioritizes states that have more goal conditions satisfied. This tie-breaking rule corresponds to the goal count heuristic which is already available in Fast Downward.

Table 5.4 shows the comparison of inertia `I`, goal count `C` to the standard approach used in the community of planning and heuristic search that prioritizes states with lower h-values (ASAI; FUKUNAGA, 2016). We run these experiments using the Fast Downward an A* search and LM-Cut as heuristic function. We set as limits 5 minutes and 2 GB of memory. Each column presents the number of solved instances for different domains with a given tie-breaking rule. `H` prioritizes states with lower h-values. `I` and `C` corresponds to inertia and goal count respectively, `IC` uses inertia as first order and goal count as the second order tie-breaking rule. Finally, `ICH` is the same as `IC`, but adds `H` as a third level tie-breaking rule. The main findings of this experiment are that simple tie-breaking rules can increase the number of solved instances when compared to the most common approach used in the community. Given that goal count is a simple approximation of fill/level order a more refined technique may provide even better results.

5.6 Conclusion

We extend the effectiveness of PDB heuristics in transportation domains and use this to increase the number of optimally solved instances of Sokoban. Further improvements in Sokoban could be produced by better strategies to select the subsets and by increasing the number of stones in the PDB. Another improvement could be related to the tie-breaking rule. We have proven the optimal solution cost for 32 instances, but we were not able to find a solution for four of them mainly because of tie-breaking.

Table 5.4: Number of solved instances by domain using an A* and LM-Cut as heuristic for each tie-breaking rule on IPC benchmarks with 5 minutes as time limit and 2 GB.

Domains	H	I	C	IC	ICH
airport	26	25	21	25	26
barman-opt11-strips	0	0	0	0	0
blocks	27	27	27	27	**28**
cybersec-strips	2	**4**	1	2	**3**
depot	5	**6**	**6**	**6**	**6**
driverlog	13	13	13	13	13
elevators-opt11-strips	15	14	15	15	15
floortile-opt11-strips	6	6	6	6	6
freecell	9	9	9	9	9
grid	1	1	1	1	1
gripper	6	6	6	6	6
logistics00	20	19	20	20	20
miconic	140	66	140	140	140
mystery	15	15	15	15	15
nomystery-opt11-strips	13	12	13	13	13
openstacks-opt11-strips	11	**18**	**18**	**18**	**18**
parcprinter-opt11-strips	13	13	13	13	13
parking-opt11-strips	1	1	1	1	1
pathways	5	5	5	5	5
pegsol-opt11-strips	17	17	17	17	17
pipesworld-notankage	13	13	13	**14**	**14**
pipesworld-tankage	8	8	7	8	8
psr-small	48	48	48	48	48
rovers	7	7	7	7	7
scanalyzer-opt11-strips	10	10	10	10	10
sokoban-opt11-strips	19	19	19	19	19
storage	14	14	14	14	14
tidybot-opt11-strips	11	**12**	11	11	11
tpp	6	6	6	6	6
transport-opt11-strips	6	6	6	6	6
visitall-opt11-strips	10	10	10	10	10
woodworking-opt11-strips	9	**12**	9	**10**	**10**
zenotravel	11	10	11	11	11
1050	517	452	518	526	**529**

6 CONCLUSIONS AND FUTURE WORK

In this thesis, we considered the class of moving-blocks problems introducing heuristic search techniques and proving computational complexity results. First, we improved the performance of heuristic search techniques in the task of obtaining optimal solutions to the moving-blocks problems. We proposed PDB heuristics and tie-breaking rules that were able to increase the number of optimally solved instances of Sokoban. Second, we enhanced the theoretical understanding about moving-blocks problems. We were able to prove that the whole class of PUSHPULL problems is PSPACE-complete and that several PULL problems are PSPACE-complete (PEREIRA; RITT; BURIOL, 2014a).

In Chapter 3 we proved that several PULL and PUSHPULL problems are PSPACE-complete. Our reductions are from the Nondeterministic Constraint Logic. We presented two sets of gadgets for PULL and PUSHPULL problems. We introduced three gadgets that are used to show that many versions of PULL problems with two decision problems are PSPACE-complete. We developed gadgets to show that PUSHPULL problems are PSPACE-complete including specific gadgets for PUSHPULL-1 versions (PEREIRA; RITT; BURIOL, 2016).

In Chapter 4 we propose a heuristic search approach to optimally solve Sokoban and moving-blocks problems. We showed that the natural abstraction used by PDBs when applied to Sokoban is ineffective. We introduced an instance decomposition to obtain an intermediate goal state which enabled the effective application of PDBs in Sokoban. We also showed that tie-breaking rules are important when solving Sokoban. We introduced the novel tie-breaking rule called fill order. Using both the PDB heuristic and the tie-breaking rule we improved the number of optimally solved instances of Sokoban compared to previous methods. The same techniques applied to Pukoban provide similar results. Finally, we conclude the chapter discussing how the idea of intermediate goal states is general and can be applied for the class of moving-blocks problems (PEREIRA; RITT; BURIOL, 2013; PEREIRA; RITT; BURIOL, 2014b; PEREIRA; RITT; BURIOL, 2015).

In Chapter 5 we improved the techniques presented in Chapter 4 increasing the number optimally solve instances of Sokoban. We begin the chapter showing that the heuristic proposed in Chapter 4 is ineffective when applied to multiple intermediate goal states. We show that the solution to this problem is not trivial and propose a strategy to solve it. We introduce a branch and bound method to solve the problem efficiently. We

also show that the tie-breaking rule fill order can be improved. Using both techniques we again increase the number of optimally solved instances of Sokoban and have currently the best results (PEREIRA et al., 2016).

6.1 Future Work

In this section we present ongoing and future work related to this thesis.

6.1.1 Admissible Pattern Search and Learning

PDB heuristics are informative for Sokoban. However, Sokoban can be viewed as an adversarial problem. Instances in the standard set are designed to have conflicts between all stones in the maze. To build the PDB some stones must be projected away and consequently some of these conflicts may be missed. Another interesting point to observe are the strategies that humans use to solve Sokoban. One such strategy is to learn from mistakes, after trying some move and realize the lower bound has increased or a deadlock has been created one might avoid such move and try something new.

Techniques as Pattern Search (JUNGHANNS; SCHAEFFER, 1998a) in Sokoban and CEGAR (SEIPP; HELMERT, 2014) in domain-independent planning try to solve these issues by learning these high order conflicts from counter-examples during the search. Pattern Search is one of the most successful techniques introduced in Rolling Stone (JUNGHANNS, 1999). The technique speculatively perform small searches during the main search trying to prove counter-examples to learn deadlocks and penalties that enhance the heuristic value. Pattern Search in Sokoban doubles the number of solved instances. The problem is that because of the abstraction used the resulting enhanced is not admissible.

We proposed a new abstraction that is similar to Pattern Search but it is admissible. We use this new abstraction to during the search to find high order conflicts through counter-examples learning deadlocks and penalties improving the heuristic value. In preliminary experiments, the new technique used by itself – no PDBs – can improve the heuristic value on initial states and double the number of optimally solved instances.

6.1.2 Domain-Independent Tie-Breaking Rules

Another possible line of research is to apply our tie breaking rules in domain-dependent planning. The tie-breaking rules used in Sokoban seem to be crucial to increase the number of optimally solved instances. Asai and Fukunaga (2016) introduced tie-breaking rules in domain-dependent planning for domains that have zero-cost actions and were able to improve the results. We performed preliminary experiments with simple versions of our tie-breaking rules and we were able to improve the results obtained by Asai and Fukunaga (2016) in domains without zero-cost actions. This is an interesting line of research due to the general belief in the planning and heuristic search community that tie-breaking by lower h-value is a good strategy.

6.1.3 Improving the Sokoban Solver

The first point to investigate is better methods to compute the heuristic value for abstractions with k' greater than two. Our current method is computationally inefficient. A more efficient method would enable us to use abstractions with more stones. Another possible improvement is the domain-dependent tie-breaking rules. The proposed rules are effective for several instances, but they are not for many others.

6.1.4 Applying the Proposed Methods to Other Domains

Another possible research line is to apply our methods to other domains. Our first attempt is to apply our techniques to moving-blocks problems related to Sokoban. Atomix is a good candidate. The best domain-dependent solver of Atomix (HÜFFNER et al., 2001) uses a weak heuristic function. The authors stated that the most likely aspect to improve the solver is an improvement of the heuristic function. Preliminary experiments show that PDBs and tie-breaking rules improve the number of optimally solved instances of Atomix.

The Airport domain (TRUG; HOFFMANN; NEBEL, 2004) is another candidate. In the standard definition of this problem airplanes have defined goal locations. However, an airplane could as well go to one among several goal locations, instead of a specific one. Thus, we could change the objective to only define as the goal state for airplanes

that they would be airborne or parked. In this setting, the planner can select the optimal goal location. Thus, our approach could be used to enhance the effectiveness of PDBs. A simple matching heuristic outperforms iPDB (HASLUM et al., 2007) and the lm-cut (HELMERT; DOMSHLAK, 2009) heuristics. Tie-breaking strategies also reduce the number of expanded nodes to solve instances.

6.1.5 Applying the Proposed Methods to Domain-Independent Planning

Besides specific problems, we could apply our heuristics to domain-independent planning. A cut square is similar to a fact landmark (HOFFMANN; PORTEOUS; SEBASTIA, 2004). Thus, our heuristics can be seen as the use of landmarks to enhance the performance of an abstraction based heuristic function. The high-level idea would be to investigate how to incorporate landmarks information to improve PDBs.

6.1.6 Better Heuristic Function for Sokoban

Finally, we could investigate a novel heuristic function for Sokoban. One possible line of research are domain (HERNáDVöLGYI; HOLTE, 2000) and Cartesian abstractions (SEIPP; HELMERT, 2013). We believe that the direct application of these ideas would not produce competitive results in Sokoban. These abstractions miss the information of the exact position of each stone. Thus, they could be too uninformative when compared to the heuristics present in this thesis. However, the flexibility provided by these abstractions could be one way to solve the problem of multiple abstract goal states since they could be abstracted to a unique high-level abstract state.

Our main challenge when applying abstractions to Sokoban is that each stone can go to multiple goal squares. This generates the multiple abstract goal state problem. Abstraction heuristics explore distances in the state space. Differently, landmarks heuristics explore the structure of the problem, and this could be the key to solve the multiple abstract goal state problem. Landmark-based heuristics are currently one the most effective approaches in the domain-independent planning. Since we already explored PDBs it makes sense to explore now landmarks. The idea is to produce a domain-dependent heuristic function based on landmarks for Sokoban. This was already done for the Pancake problem (HELMERT, 2010). The best PDB approach for the Pancake problem can

solve instances with 20 disks in a day. The domain-dependent landmark heuristic dramatically outperforms this result, being able to solve instances with 60 disks in seconds.

REFERENCES

ASAI, M.; FUKUNAGA, A. Tiebreaking strategies for A* search: How to explore the final frontier. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2016.

BONET, B.; HELMERT, M. Strengthening landmark heuristics via hitting sets. In: **European Conference on Artificial Intelligence**. [S.l.: s.n.], 2010. p. 329–334.

BOTEA, A.; MÜLLER, M.; SCHAEFFER, J. Using abstraction for planning in Sokoban. In: **Computers and Games**. [S.l.: s.n.], 2002. p. 360–375.

CULBERSON, J. C. Sokoban is PSPACE-Complete. In: **Fun With Algorithms**. [S.l.: s.n.], 1999. p. 65–76.

CULBERSON, J. C.; SCHAEFFER, J. Searching with pattern databases. In: **Canadian Conference on Artificial Intelligence**. [S.l.: s.n.], 1996. p. 402–416.

DAMGAARD, B. **Yet another Sokoban clone**. 2014. Last accessed in September 2016. Available from Internet: <sourceforge.net/projects/sokobanyasc>.

DEMAINE, E.; DEMAINE, M.; O'ROURKE, J. PushPush and Push-1 are NP-hard in 2D. In: **Canadian Conference on Computational Geometry**. [S.l.: s.n.], 2000. p. 211–219.

DEMAINE, E. D. et al. Pushing blocks is hard. **Computational Geometry**, v. 26, p. 21–36, 2003.

DEMAINE, E. D.; HEARN, R. A.; HOFFMANN, M. Push-2-F is PSPACE-Complete. In: **Canadian Conference on Computational Geometry**. [S.l.: s.n.], 2002. p. 31–35.

DEMAINE, E. D.; HOFFMANN, M. Pushing blocks is NP-complete for noncrossing solution paths. In: **Canadian Conference on Computational Geometry**. [S.l.: s.n.], 2001. p. 65–68.

DEMAINE, E. D.; HOFFMANN, M.; HOLZER, M. PushPush-k is PSPACE-Complete. In: **International Conference on FUN with Algorithms**. [S.l.: s.n.], 2004. p. 159–170.

DEMARET, J.-N.; LISHOUT, F. V.; GRIBOMONT, P. Hierarchical planning and learning for automatic solving of Sokoban problems. In: **Belgium-Netherlands Conference on Artificial Intelligence**. [S.l.: s.n.], 2008. p. 57–64.

DOMSHLAK, C.; KATZ, M.; LEFLER, S. Landmark-enhanced abstraction heuristics. **Artificial Intelligence**, v. 189, p. 48–68, 2012.

DOR, D.; ZWICK, U. Sokoban and other motion planning problems. **Computational Geometry**, v. 13, n. 4, p. 215–228, 1999.

EDELKAMP, S. Planning with pattern databases. In: **European Conference on Planning**. [S.l.: s.n.], 2001. p. 13–24.

EDELKAMP, S.; SCHRÖDL, S. **Heuristic search – Theory and applications**. [S.l.]: Morgan Kaufmann, 2012.

FELNER, A.; KORF, R. E.; HANAN, S. Additive pattern database heuristics. **J. Artificial Intelligence Res.**, v. 22, p. 279–318, 2004.

FELNER, A. et al. Compressed pattern databases. **J. Artificial Intelligence Res.**, v. 30, p. 213–247, 2007.

GAREY, M. R.; JOHNSON, D. S. **Computers and intractability: A guide to the theory of NP-completeness**. [S.l.]: Freeman, 1979.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Trans. Systems Science and Cybernetics**, v. 4, n. 2, p. 100–107, 1968.

HASLUM, P. et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2007. p. 1007–1012.

HEARN, R.; DEMAINE, E. D. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. **Theoretical Computer Science**, v. 343, n. 1-2, p. 72–96, 2005.

HELMERT, M. The fast downward planning system. **J. Artificial Intelligence Res.**, v. 26, p. 191–246, 2006.

HELMERT, M. New complexity results for classical planning benchmarks. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2006. p. 52–62.

HELMERT, M. Landmark heuristics for the pancake problem. In: **Symposium on Combinatorial Search**. [S.l.: s.n.], 2010.

HELMERT, M.; DOMSHLAK, C. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2009. p. 162–169.

HELMERT, M.; HASLUM, P.; HOFFMANN, J. Flexible Abstraction Heuristics for Optimal Sequential Planning. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2007. p. 176–183.

HERNáDVöLGYI, I.; HOLTE, R. Experiments with automatically created memory-based heuristics. In: **Abstraction, Reformulation, and Approximation**. [S.l.: s.n.], 2000. p. 281–290.

HOFFMANN, J. et al. Engineering benchmarks for planning: The domains used in the deterministic part of ipc-4. **Journal of Artificial Intelligence Research**, AI Access Foundation, USA, v. 26, n. 1, p. 453–541, aug. 2006.

HOFFMANN, J.; PORTEOUS, J.; SEBASTIA, L. Ordered landmarks in planning. **J. Artificial Intelligence Res.**, p. 215–278, 2004.

HOLTE, R. C. Common misconceptions concerning heuristic search. In: **Symposium on Combinatorial Search**. [S.l.: s.n.], 2010.

HOLTE, R. C. et al. Maximizing over multiple pattern databases speeds up heuristic search. **Artificial Intelligence**, v. 170, n. 16-17, p. 1123–1136, 2006.

HOLTE, R. C. et al. Bidirectional search that is guaranteed to meet in the middle. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2016.

HOLTE, R. C.; GRAJKOWSKI, J.; TANNER, B. Hierarchical heuristic search revisited. In: **Abstraction, Reformulation and Approximation**. [S.l.: s.n.], 2005. p. 121–133.

HOLTE, R. C. et al. Hierarchical A*: Searching abstraction hierarchies efficiently. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 1996. p. 530–535.

HOLZER, M.; SCHWOON, S. Assembling molecules in ATOMIX is hard. **Theoretical Computer Science**, v. 313, n. 3, p. 447 – 462, 2004. Special issue on Algorithmic Combinatorial Game Theory.

HÜFFNER, F. et al. Finding optimal solutions to Atomix. In: **KI 2001: Advances in Artificial Intelligence**. [S.l.: s.n.], 2001. p. 229–243.

JUNGHANNS, A. **Pushing the Limits: New Developments in Single-Agent Search**. Thesis (PhD) — University of Alberta, 1999.

JUNGHANNS, A.; SCHAEFFER, J. Single-agent search in the presence of deadlocks. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 1998.

JUNGHANNS, A.; SCHAEFFER, J. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In: **Canadian Conference on Artificial Intelligence**. [S.l.: s.n.], 1998. p. 1–15.

JUNGHANNS, A.; SCHAEFFER, J. Sokoban: Enhancing general single-agent search methods using domain knowledge. **Artificial Intelligence**, v. 129, n. 1–2, p. 219–251, 2001.

KARPAS, E.; DOMSHLAK, C. Cost-optimal planning with landmarks. In: **International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2009. p. 1728–1733.

KATZ, M.; DOMSHLAK, C. Optimal Additive Composition of Abstraction-based Admissible Heuristics. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2008. p. 174–181.

KATZ, M.; DOMSHLAK, C. Structural patterns heuristics via fork decomposition. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2008. p. 182–189.

KATZ, M.; DOMSHLAK, C. Structural-pattern databases. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2009. p. 186–193.

KATZ, M.; DOMSHLAK, C. Implicit abstraction heuristics. **J. Artificial Intelligence Res.**, v. 39, p. 51–126, 2010.

KOLMOGOROV, V. Blossom V: A new implementation of a minimum cost perfect matching algorithm. **Math. Progr. Comput.**, v. 1, n. 1, p. 43–67, 2009.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. **Artificial Intelligence**, v. 27, n. 1, p. 97–109, 1985.

KORF, R. E. Finding optimal solutions to Rubik's cube using pattern databases. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 1997. p. 700–705.

KORF, R. E.; FELNER, A. Disjoint pattern database heuristics. **Artificial Intelligence**, v. 134, n. 1–2, p. 9–22, 2002.

KUHN, H. W. The Hungarian Method for the assignment problem. **Naval Research Logistics Quarterly**, v. 2, p. 83–97, 1955.

MEGER, M. **JSoko**. 2014. Last accessed in September 2016. Available from Internet: <sourceforge.net/projects/jsokoapplet>.

NISSIM, R.; HOFFMANN, J.; HELMERT, M. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In: **International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2011. p. 1983–1990.

PEARL, J. **Heuristics – Intelligent search strategies for computer problem solving**. [S.l.]: Addison-Wesley, 1984. (Addison-Wesley Series in Artificial Intelligence).

PEREIRA, A. G. et al. Improved heuristic and tie-breaking for optimally solving Sokoban. In: **International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2016. p. 662 – 668.

PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Finding optimal solutions to Sokoban using instance dependent pattern databases. In: **Symposium on Combinatorial Search**. [S.l.: s.n.], 2013. p. 141–148.

PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Solving motion planning problems. In: **International Joint Conference on Artificial Intelligence School - Doctoral Consortium**. [S.l.: s.n.], 2014.

PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Solving sokoban optimally using pattern databases for deadlock detection. In: **Encontro Nacional de Inteligência Artificial**. [S.l.: s.n.], 2014.

PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Optimal sokoban solving using pattern databases with specific domain knowledge. **Artificial Intelligence**, v. 227, p. 52 – 70, 2015. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0004370215000867>.

PEREIRA, A. G.; RITT, M.; BURIOL, L. S. Pull and pushpull are pspace-complete. **Theoretical Computer Science**, v. 628, p. 50 – 61, 2016. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S030439751600205X>.

POMMERENING, F.; HELMERT, M. Incremental lm-cut. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2013.

POMMERENING, F.; RöGER, G.; HELMERT, M. Getting the most out of pattern databases for classical planning. In: **International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2013.

RICHTER, S.; HELMERT, M.; WESTPHAL, M. Landmarks revisited. In: **AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2008. p. 975–982.

RITT, M. Motion planning with pull moves. **CoRR**, abs/1008.2952, p. 1–9, 2010.

SAVITCH, J. W. Relationships between nondeterministic and deterministic tape complexities. **Journal of Computer and System Sciences**, v. 4, n. 2, p. 177 – 192, 1970.

SEIPP, J.; HELMERT, M. Counterexample-Guided Cartesian Abstraction Refinement. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2013. p. 347–351.

SEIPP, J.; HELMERT, M. Diverse and Additive Cartesian Abstraction Heuristics. In: **International Conference on Automated Planning and Scheduling**. [S.l.: s.n.], 2014.

SIEVERS, S.; ORTLIEB, M.; HELMERT, M. Efficient Implementation of Pattern Database Heuristics for Classical Planning. In: **Symposium on Combinatorial Search**. [S.l.: s.n.], 2012. p. 105–111.

Sokoban Wiki. **Solver Statistics**. 2014. Last accessed in September 2016. Available from Internet: <http://www.sokobano.de/wiki>.

TAKAHASHI, K. **Takaken Solver**. 2014. Last accessed in September 2016. Available from Internet: <www.ic-net.or.jp/home/takaken/e/soko>.

TRUG, S.; HOFFMANN, J.; NEBEL, B. Applying automatic planning systems to airport ground-traffic control–a feasibility study. In: **KI: Advances in Artificial Intelligence**. [S.l.]: Springer, 2004. p. 183–197.

YANG, F. et al. A general theory of additive state space abstractions. **J. Artificial Intelligence Res.**, v. 32, p. 631–662, 2008.

ZUBARAN, T.; RITT, M. Agent motion planning with pull and push moves. In: **Encontro Nacional de Inteligência Artificial**. [S.l.: s.n.], 2011. p. 358–369.

APPENDIX A RESOLVENDO PROBLEMAS DE BLOCOS-MÓVEIS

Nesta tese, consideramos a classe de *problemas de blocos-móveis* para a qual propomos técnicas de busca heurística e provamos resultados de complexidade computacional. Nós melhoramos o desempenho de técnicas de busca heurística na tarefa de obter soluções ótimas para problemas de blocos-móveis. Propomos funções heurísticas de abstração baseadas em bancos de dados de padrão e regras de desempate que foram capazes de aumentar o número de instâncias de Sokoban resolvidas com garantia de otimalidade. Nós ampliamos o conhecimento teórico sobre problemas de blocos-móveis provando que toda a classe de problemas com movimentos de EMPURRAR e PUXAR é PSPACE-complete e que vários problemas com movimentos de PUXAR são PSPACE-complete.

Um problema de blocos-móveis consiste em k blocos móveis dispostos em um labirinto em grade quadrangular onde há um bloco móvel adicional chamado de o *homem*, que é o único bloco que pode ser movido diretamente. Em particular, cada problema de blocos-móveis é definido pelo conjunto de movimentos disponíveis, pela descrição do objetivo e pelo o que acontece quando o homem tenta mover um bloco. *Sokoban* é o problema de blocos-móveis mais conhecido e pesquisado.

Problema de blocos-móveis são ao mesmo tempo desafiadores de forma teórica e pratica. Em geral, problemas na classe de blocos-móveis são PSPACE-complete, e apenas restrições artificiais os torna tratáveis. Estes problemas têm uma descrição simples e concisa, são fáceis de entender, mas ainda intelectualmente desafiadores. Há um número considerável de pessoas interessadas em resolvê-los. Além disso, há uma comunidade interessada em desenvolver abordagens algorítmicas para resolver instâncias destes problemas.

Antes desta tese, a maior parte da literatura cientifica estudou a complexidade computacional de problemas de blocos-móveis apenas com movimentos de EMPURRAR, na maioria dos casos provando que esses problemas são PSPACE-complete. Outras versões do problema com diferentes movimentos foram provadas ser NP-hard. A Lógica de Restrições Não Determinística (NCL) é um *framework* desenvolvido por Hearn and Demaine (2005) para diminuir o esforço de provar resultados de PSPACE-hardness. Normalmente, ela é usada para provar que *puzzles* e jogos são PSPACE-hard. Nós usamos NCL para investigar a dificuldade de resolver problemas de blocos-móveis.

No Capítulo 3 provamos que vários problemas com movimentos de PUXAR e com movimentos de EMPURRAR e PUXAR são PSPACE-complete. Nós apresentamos dois conjuntos de *gadgets* para problemas com movimentos de PUXAR e com movimentos de EMPURRAR e PUXAR. Nós introduzimos três *gadgets* que são usados para provar que muitas versões de problemas com movimentos de PUXAR com duas descrições de objetivo são PSPACE-complete. Nós também desenvolvemos *gadgets* para mostrar que a classe de problemas com movimentos de EMPURRAR e PUXAR é PSPACE-complete, incluindo *gadgets* específicos para versões em que o homem pode mover apenas um bloco simultaneamente. A nossa contribuição nessa linha de pesquisa é aprimorar o conhecimento sobre o panorama da complexidade de problemas de blocos-móveis.

Nosso principal objetivo com essa tese é investigar abordagens para resolver com garantia de otimalidade problemas de blocos-móveis com foco em Sokoban. Métodos baseados em busca heurística e heurísticas de abstrações como bancos de dados de padrão são as abordagens mais efetivas para resolver otimamente esses problemas. Nós fazemos muitas contribuições nessa linha de pesquisa. Nós introduzimos novas funções heurísticas usando bancos de dados padrão com a ideia de estados objetivos intermediários. Propomos uma técnica baseada em bancos de dados padrão para detectar impasses. Propomos regras de desempate que exploram a estrutura do problema. Usando estas funções heurísticas e regras de desempate nós aumentamos o número de instâncias resolvidas de forma ótima em Sokoban e em outros problemas de blocos-móveis em comparação com os métodos anteriores.

No Capítulo 4 propomos uma abordagem de busca heurística para resolver de forma ótima Sokoban e problemas de blocos-móveis. Nós mostramos que a abstração natural utilizada por heurísticas baseadas em bancos de dados de padrão quando aplicado a Sokoban é ineficaz. Nós introduzimos uma decomposição instâncias para obter um estado objetivo intermediário que permitiu a aplicação efetiva de heurísticas baseadas em bancos de dados de padrão em Sokoban. Nós também mostramos que as regras de desempate são importantes na resolução de Sokoban. Assim, propomos uma regra de desempate que usa a informação da disposição dos quadrados objetivos e dos blocos no labirinto para decidir durante a busca quais estados são mais promissores. Usando tanto a heurística baseada em bancos de dados de padrão quanto a regra de desempate melhoramos o número de instâncias resolvidas de forma ótima de Sokoban em comparação com os métodos anteriores. As mesmas técnicas utilizadas em Pukoban proporcionam resultados semelhantes. Finalmente, concluímos o capítulo discutindo como a ideia de

estados objetivo intermediário é geral e pode ser aplicada para a classe de problemas de blocos-móveis.

No Capítulo 5 melhoramos as técnicas apresentadas no capítulo 4 aumentando o número de instâncias resolvidas de forma ótima em Sokoban. Começamos o capítulo mostrando que a função heurística proposta no Capítulo 4 é ineficaz quando aplicada a vários estados objetivos intermediários. Nós mostramos que a solução para este problema não é trivial e propomos uma estratégia para resolvê-lo introduzindo um método de *branch and bound* para resolver o problema de forma eficiente. Mostramos também que existem variações da regra de desempate que geram melhores resultados. Com ambas as técnicas novamente aumentamos o número de instâncias resolvidas de forma ótima em Sokoban e possuímos atualmente os melhores resultados publicados.