Na prawach rękopisu

Wydział Informatyki i Zarządzania
Politechnika Wrocławska

Classical Planning Supported by Plan Traces for Video Games

Seria: PRE nr …………

Bartłomiej Józef Dzieńkowski

Słowa kluczowe:
planning, data mining, heuristics, formal concept analysis, video games

Krótkie streszczenie:
The thesis introduces a new approach that utilises plan traces, which represent plans executed by human players. A set of input plan traces is used for constructing an abstraction model generalising a planning domain in a video game. The model has the form of hierarchically nesting regions that partition the state space. Regions of the state space are defined implicitly, which allows identifying sets of states without explicitly specifying and storing them in the memory. Such a hierarchical structure can be applied to estimating the distance between any pair of states in a state-space graph. In practice, the model is prepared before planning is performed. It is employed by the heuristic to accelerate the process of solving planning problems that dynamically appear during the game.

Wrocław, 2018

Na prawach rękopisu

Wydział Informatyki i Zarządzania
Politechnika Wrocławska

Planowanie klasyczne w grach komputerowych na bazie zapisów planów

Seria: PRE nr …………

Bartłomiej Józef Dzieńkowski

Słowa kluczowe:
planowanie, ekstrakcja danych, heurystyki, formalna analiza pojęć, gry komputerowe

Krótkie streszczenie:
W pracy zaproponowano nowe podejście, które wykorzystuje zbiór historycznych przebiegów gry zawierających ślady planów wykonanych przez graczy ludzkich. Są to dane wejściowe dla metody budowania modelu przestrzeni stanów. Model ten ma strukturę hierarchicznie zagnieżdżających się regionów, które partycjonują przestrzeń stanów dla zadanej domeny problemów planowania w grze. Regiony są zdefiniowane w sposób niejawny co pozwala na identyfikowanie zbiorów stanów bez ich specyfikowania i przechowywania w pamięci. Zbudowany model jest wykorzystywany podczas gry przez nową heurystykę do szacowania odległości pomiędzy dowolnymi stanami w przestrzeni stanów, co pozwala przyspieszyć proces planowania.

Wrocław, 2018

Wrocław University
of Science and Technology

Doctoral Thesis

Classical Planning Supported by Plan Traces for Video Games

Author:

Bartłomiej Józef Dzieńkowski

Supervisor:

Prof. WrUT, dr hab. Urszula Markowska-Kaczmar

Department of Computational Intelligence

Faculty of Computer Science and Management

July 2017

Abstract

Classical Planning Supported by Plan Traces for Video Games

by Bartłomiej Józef DZIEŃKOWSKI

Planning in modern video games is challenging because of complex virtual worlds and difficult problems that must be solved during runtime. Problems addressed in the dissertation are in the class of classical planning and involve combinatorial optimisation. For solving such problems, state-space search methods are employed. Search-based planners produce solutions of high quality, which is required for building a believable computer-controlled player replacing a human one. However, such an approach is computationally expensive. Heuristic estimation of the cost of reaching a planning goal plays a crucial role in improving the performance of search algorithms because it can guide the search towards the goal and significantly reduce the search space.

The thesis introduces a new approach that utilises plan traces, which represent plans executed by human players. A set of input plan traces is used for constructing an abstraction model generalising a planning domain in a video game. The model has the form of hierarchically nesting regions that partition the state space. Regions of the state space are defined implicitly, which allows identifying sets of states without explicitly specifying and storing them in the memory. Such a hierarchical structure can be applied to estimating the distance between any pair of states in a state-space graph. In practice, the model is prepared before planning is performed. It is employed by the heuristic to accelerate the process of solving planning problems that dynamically appear during the game.

The original contribution of this work is a novel technique of partitioning the state space, an original planning heuristic, and two new state search algorithms that rely on the properties of the heuristic. The experimental study includes, among others, tests conducted in an author's testbed environment designed as a video game. The optimality of the heuristic was shown formally. In the final phase of the study, an automatic method of tuning the structure of regions for an arbitrary planning domain was demonstrated. The approach is universal, and it can be applied to metric and non-metric spaces as well.

Contents

List of Figures

List of Tables

Symbols

$\mathbb{S}$	Def. 2.3	a state-space graph
s	Def. 2.3	a state
S	Def. 2.3	a set of states
t	Def. 2.4	a planning task
p	Def. 2.5	a plan
c	Def. 2.6	a cost
K	Def. 2.8	a formal context
G	Def. 2.8	a set of objects
M	Def. 2.8	a set of attributes
I	Def. 2.8	incidence relation
$\langle G_i, M_j \rangle$	Def. 2.10	a formal concept
$L(K)$	Def. 2.12	a concept lattice
$d(\cdot)$	Def. 5.1	a state descriptor
D	Def. 5.1	a set of state descriptors
$L(K^{\mathbb{S}})$	Def. 6.5	a state-space lattice
X_i	Eq. 2.9	a chromosome
x	Eq. 2.9	a gene

Chapter 1

Introduction

The discussion in this dissertation begins with the motivation behind the undertaken problem. It provides the problem background and the justification for elaborating a new approach to planning. The idea of the proposed approach is sketched. Subsequently, the goal of the study and assumptions of the introduced method are clarified. Promising application fields of the described concept are indicated. The final part of the chapter outlines the structure of the document.

1.1 Background

Virtual reality entertains, teaches, and helps to solve practical problems. That is the essential element of video games, which encompass commercial games, computerised board games, and serious games. The widespread use of video games contributes to generally understood informatisation in the modern society.

The growing desire to realistically recreate the real world has helped to drive development of hardware, Computer Graphics, and Artificial Intelligence. In recent decades, a significant effort has been invested in research on virtual beings. They are an integral part of a realistic virtual world. Building a believable representation of a virtual human is a central problem, in which the primary challenge can be located in simulating intelligent behaviour.

Humans solve difficult problems using their knowledge and intelligence [1]. Information stored in memory becomes knowledge when it is interpreted and understood. Intelligence is an ability to process the knowledge and generate an output representing a problem solution. Many activities in a human brain that take part in the process of problem solving are unconscious.

If a problem is too complex to be solved immediately, then it requires conscious and deliberate action [2]. Usually, a larger problem is divided into smaller instances. Tasks and objectives are outlined during this process. To build a solution, certain activities leading to the desired goal are chosen based on their expected effects. World states describing future progress of the chosen activities are anticipated. The accuracy of anticipated states highly depends on knowledge about the world, which includes information regarding the objects and the relations between them.

The described procedure is referred to as planning. Without a doubt, it is crucial for achieving demanding goals. Because planning is complicated and time-consuming, it is applied when the benefits are higher than the costs. Therefore, it is especially suitable for problems in which actions are distributed in time, their effects have a great impact on the outcome, the cost of failure is high, and the risk is considerable. On the other hand, planning is less important if tasks are well-known and trained or can be performed instinctively, even though they appear to be complex.

Despite the fact that a human brain mostly provides good problem solutions, rather than optimal, its performance is a constant inspiration for AI researchers. Automated planning in computer science shares many principles with planning as the thought process. Similarly, such a procedure relies on a structured knowledge about the world. Analogously, a resource-consuming planning can be replaced by trivial and faster methods if the compromise is acceptable.

The biggest contrast between manual and automated planning is that a computer can process a vast number of possible action outcomes, but without a true understanding of their meaning. Thus, a human is immensely less capable of analysing a large number of the world states but his or her solution search is unquestionably more directed towards the goal. In effect, computers can barely compete with the brain solving difficult planning problems in real-time. Although it is possible to accumulate enough computational power to demonstrate superiority over human problem-solving capabilities, an organic brain is still characterised by an unsurpassed efficiency considering the energy consumption. In many practical applications, available computational resources are limited. Therefore, most of the research in this field is focused on optimising the performance of planning methods to fit practical limitations.

It can be concluded that focusing on collecting and utilising knowledge about a problem and its solution space may be the key to advancement in the area of automated planning.

There is a broad range of studies related to formalising, modelling, acquiring, and storing knowledge [3]. The research in this field is aimed at imitating an ability of a living creature to learn about the rules of a surrounding environment. It is a promising direction and an opposite approach to using rigidly predefined knowledge.

A self-learning system that is capable of acquiring knowledge, which quality equals that of the domain experts, has been a fantasy and goal of researchers since the early beginnings of AI. Automatic knowledge acquisition has spotted many obstacles. The fundamental issue arises from the fact that useful knowledge is difficult to be located in a rich stream of information. There is always a significant amount of noise and data that is irrelevant or redundant. Assessing which information is sound for solving a problem requires reliable background knowledge. In nature, this type of knowledge is sourced from evolved instincts and lifetime experience.

There is evidence that imitating human knowledge and intelligence is beyond present capabilities of science [1]. One of the arguments is that the processes in the brain are not fully investigated. Formal models describing the world and representing knowledge simplify the nature of objects. It is imposed by the problem of complexity and computational power limitations. In the future, a breakthrough in this domain may be triggered by a technological leap.

For these reasons, high-grade planners intended for demanding environments such as video games rely mainly on rigid knowledge [2]. It is considered safer and more reliable than a general approach, in which the additional room for flexible knowledge can lead to unexpected results and potentially cause performance issues. Systems that employ cognitive modelling have limited applicability, and they are still researched.

1.2 Rationale

In the view of the previously mentioned issues and obstacles, a promising direction may be located in the analysis of solution space in planning problems existing in video games, rather than trying to simulate the human brain. The solution space is described by the state space, which is an abstract model that underlies the methodology of solving state search problems in classical planning [2]. It formalises the behaviour of a discrete system as a state-space graph in which nodes are system states, and edges are state transitions invoked by actions. Each planning problem can be modelled in formal terms of a state transition system. A sequence of state transitions that begins in the current system state and ends in the desired one represents the solution of a planning problem. Such a plan should have a minimal cost, which is associated with actions. Proper knowledge about the structure of a state-space graph can be used to reduce the complexity of the solution search. An example of solving a planning problem by searching for the goal state is illustrated in Fig. 1.1. In the picture, circles are states, and edges are transitions between neighbouring states.

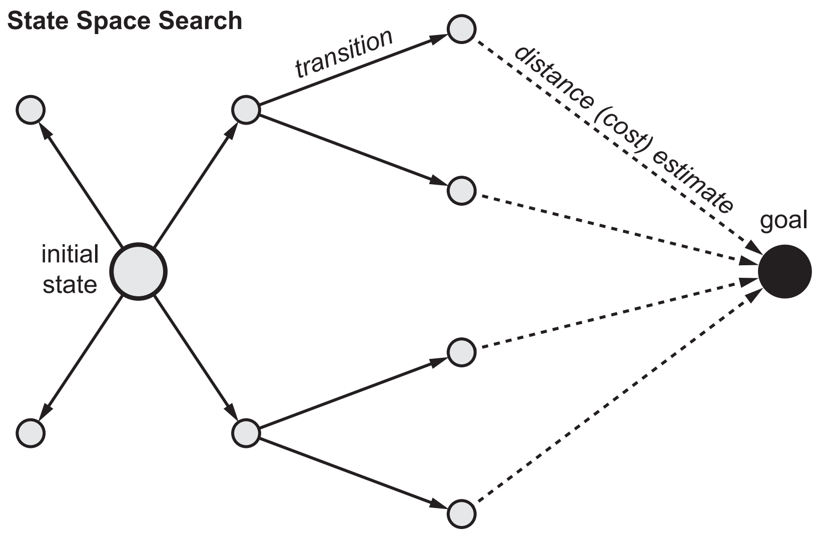

FIGURE 1.1: The concept of solving planning problems by state-space search

Depending on the specificity of a problem, certain optimisations can be employed in a state search method. A heuristic estimation of the distance to a goal state and partitioning of the state space (hierarchical clustering) are the leading techniques [4, 5]. They enable a search algorithm to reduce the number of processed states, which has a direct impact on the consumption of computational resources. The estimated distance between two state nodes in a state-space graph is usually expressed by the minimal cost of traversing a path that connects them. Information about an approximate distance to the goal is vital for guiding the search towards the solution. Without such information, the algorithm is forced to expand states blindly. Uninformed search is costly as the complexity is exponential.

Traditionally, a state can be defined by a tuple of state variables. The difficulty of determining the distance between states is related to the dimensionality of the tuple and the type of state variables. The distance can be estimated easily if state variables are independent, they denote coordinates in Euclidean space, or the tuple belongs to a metric space in general. However, if a state representation incorporates additional variables expressing the effect of abstract actions that are interdependent, then determining the distance becomes a nontrivial problem.

There are two main streams of addressing this issue. The first one is covered by domain-specific planning [6]. It exploits information about the specificity of a planning problem, its solution space, and the structure of the state space. A domain expert's knowledge is utilised for designing a specialised tool for a particular planning problem. An alternative approach is employing domain-independent techniques [2]. There is a set of well-known tools that can be applied to any generally defined planning problem. In principle, deploying a ready-to-use planning system is often cheaper than implementing a dedicated one, but it is usually done at the cost of decreased efficiency.

In many practical cases, domain-independent planners have limited application. Transforming system rules into a domain-independent planning problem language may be the first barrier. It is because effective systems often use complex data structures while the domain-independent representation of a planning domain is limited to a collection of symbols (propositions) expressing facts. Manipulating the symbolic representation is associated with overhead computation, which may be a considerable issue in demanding application areas. Finally, a multipurpose heuristics can hardly compete with specialised ones. Domain-specific knowledge often have to be introduced to a domain-independent planner to improve its efficiency [7].

The approach elaborated in this dissertation joins both streams. It offers a multipurpose heuristic with a high degree of automation, and it does not impose the symbolic representation in the same moment. The method is formally defined on the most general level of a planning problem, which is the state space. The framework is universal, and it is suitable for domain-specific solutions employed in demanding applications. In the future, the method may be also adapted to a domain-independent planner as an extension.

The idea of the proposed heuristic originated from an attempt to learn how to solve complex planning problems by observing human actions. The human brain can quickly learn rules of an arbitrary game, gain experience by playing it, and solve dynamically appearing problems. Human planners have a great intuition and experience that enables them to unconsciously produce plans of a good quality in a short time. The challenge is to extract general knowledge from the observed traces of their executed plans – information that could be generalised and effectively reused to improve the performance of an automated planning process.

A plan trace represents a solution of an individual instance of a planning problem. A planning problem instance is determined by input parameters, rules, and objectives in a simulated environment. A realistic virtual world is characterised by a large number of possible planning problem instances. Each observed problem solution pertains to a different situation in the environment. Game replays record the progress of a game and store plan traces.

There are several arguments why game replays should be considered as a valuable source of information that is worth processing. First, they hold the outcome of problem-solving processes that occur inside the brain of an intelligent being. Although the player's knowledge, strategy, and possible communication with other players may remain unknown, his or her reaction to all game states, which occurred during the play, is perfectly known. Second, they are easy to obtain as they store only changes in the game logic data, which are small enough to be accumulated and uploaded to a server. In fact, recording the match is a popular feature in multiplayer games, which potentially generates vast quantities of data worldwide. Finally, it would be inefficient to process other possible sources of information. For instance, communication between the players is usually vague as they are focused on events that occur in the game. Asking the players to formalise and describe their thought process could be done only in a lab environment on a small scale, thus making it impractical to collect large quantities of data.

With a great potential comes a great challenge. Game replays are raw data streams that have been originally used to reproduce historical games. They contain precise information on how games progressed, but any general knowledge that could be utilised to support the decision process is hidden and must be extracted. Player intentions, a map of goals and subgoals, cooperation patterns are just examples of valuable information that can be potentially mined in the data. It was assumed that the researched method would serve best if any useful information is mined automatically involving minimum domain-specific knowledge. Therefore, it should not require any form of manual annotation of the input data done by domain experts.

The proposed approach is aimed at maximising the overall efficiency of a planner solving complex planning problems in a virtual world. The fewer computational resources a planning method consumes, the more challenging problems it can address. The improvement also translates into the quality of returned plans. It is essential for developing a believable agent characterised by intelligent behaviour.

1.3 Goal

The goal of this thesis is declared as: *elaborating an efficient method of solving planning problems in video games by utilising plan traces.* The following assumptions apply:

- Considered **planning problems** are in the class of classical planning and involve combinatorial optimisation.
- Addressed **video games** are demanding domains characterised by complex environments and challenging planning problems that must be solved during runtime.
- A set of **plan traces** is provided as input. A plan trace holds complete information of world states describing a plan executed by a human player.
- The method **outputs** plans represented as sequences of actions executed by AI-controlled players who substitute human players.
- The **efficiency** of the method is expressed as the consumption of computational resources such as processor time and memory during planning.

The expected outcome of achieving the goal is a new planning method that builds a model capturing the abstraction of a planning domain by processing input plan traces. The model is then used during runtime by a new planning heuristic to reduce the search space and accelerate solving planning problems.

1.4 Potential Application

Automated planning has a broad range of application in many fields. However, planning is sometimes replaced by inferior approaches if there are not enough computational resources to use it effectively. By taking advantage of plan traces, the introduced method may achieve a better performance than the current planning methods. The reduction of resource consumption translates into the improvement of solution quality in demanding domains. Thus, the discussed approach may expand the applicability of planning in many areas by overcoming performance barriers.

Although the proposed planning method is universal, its benefits may be particularly valuable for video games in which planning is performed in real-time because they are an excellent source of plan traces. Each run in a competitive game is affected by human input and progresses differently. Therefore, it is practically impossible to precompute a set of plans for every possible situation in a game. Solving a planning problem must fit in restrictive hardware limitations. It is especially challenging in the case of modern virtual environments that aim to simulate the world realistically.

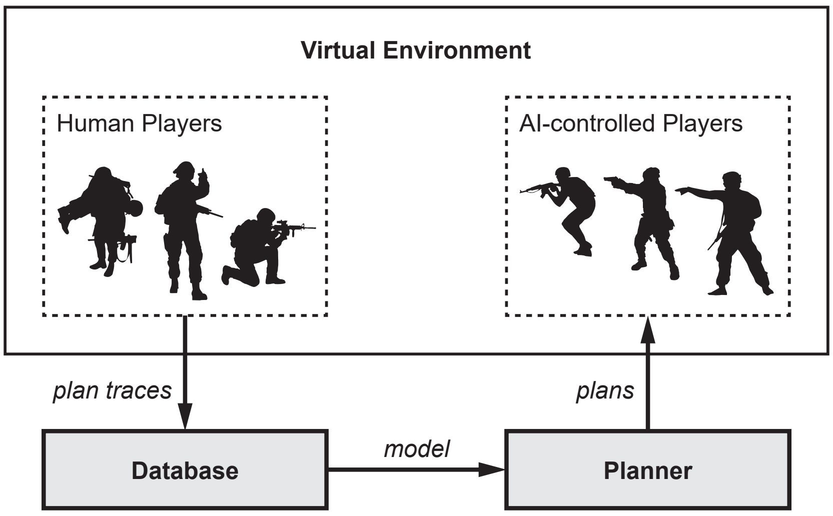

FIGURE 1.2: Supporting automated planning by plan traces observed in a virtual environment

A promising application area of the proposed concept of a planning method is serious games used in the military. It is a popular practice that soldiers exercise their tactics by competing with enemy forces in a virtually simulated battlefield [8]. In a realistic simulation, a single tactical scenario has multiple planning problem instances, a large solution space, and each run is unique. Trained officers solve real planning problems on-line and leave traces of their actions. Their solutions can be used to model the state space of a particular tactical scenario. In the proposed approach, the obtained model could support a planning system and augment its capability to control symmetrical enemy units intelligently – Fig. 1.2. Competing with an intelligent opponent has a great impact on delivering a reasonable level of realism to the simulation and, consequently, improving the training effects.

In a similar manner, the approach can be successfully employed for improving user experience in the area of commercial games [9]. Delivering a believable opponent to a system that meets target machine requirements and fits the budget is a challenging task. Computer-controlled players are a desired feature of multiplayer games. They help to continue a match while there are not enough human players on the server and enable the players to practise in an off-line mode. Planning plays a crucial role in cooperative modes in which human players play together against AI players that also cooperate. In principle, multiplayer servers can accumulate a large number of game replays, which represent plan traces, in a short time. Therefore, they are a good source of input data for the proposed planning method.

The described areas of application are some of the solid examples, but there are many other opportunities for utilising the proposed scheme. A similar solution can be employed in security and disaster mitigation systems. The introduced model of the state space may also contribute to widely understood analysis of the behaviour of users operating in virtual environments.

1.5 Outline

The thesis is divided into six chapters and an appendix. The following **Chapter 2 – Basic Concept** introduces the fundamental concepts on the basis of which the proposed method was built. The description begins with a short discussion referring to the intelligence of agents in Multi-Agent Systems. The next part positions classical planning in the context of Automated Planning. A classical planning problem is defined, and current planning methods are described. Subsequently, a reference to Formal Concept Analysis and constructing a concept (Galois) lattice are provided. The chapter ends with an introduction to Genetic Algorithms.

Chapter 3 – Related Work summarises the application areas of AI methods for video games and emphasises the importance of classical planning. Subsequently, state-of-the-art classical planning algorithms employed in video games are described. In the next part, the subject of game replay analysis is discussed. The chapter is concluded by promising research directions and potential advantages of the proposed approach in the context of the current solutions.

Chapter 4 – Research Problem formulates assumptions referring to classical planning supported by plan traces in video games and formally states the undertaken research problem. The next part of the chapter summarises the results of early studies, which helped to understand the nature of the problem and localise the main challenges. The initial research provided grounds for developing the core algorithm performing state-space partitioning, which is discussed subsequently.

Chapter 5 – State-space Tree Search Heuristic describes the first phase of the evolution of the proposed method. The chapter introduces the notion of state descriptors representing implicit regions of the state space. Next, the model of a region tree built from plan traces is presented. It is used for partitioning the state space. Subsequently, a new heuristic estimator employing the model is defined. The heuristic was tested in an author's testbed environment designed as a video game.

Chapter 6 – State-space Lattice Search Heuristic presents the final form of the proposed method, in which the model of a region tree was replaced with a region lattice. A new heuristic estimator employing the improved model, and two new state search algorithms, which rely on the properties of the heuristic, are introduced. The complexity and the optimality of the estimator are shown formally. The experimental study demonstrates the characteristics of the method. An automatic procedure of tuning the structure of a region lattice is examined.

The final **Chapter 7 – Summary** reviews the researched approach. The chapter summarises the researched approach. It outlines the main features of the proposed method. The results of the experimental study are compiled. The impact on planning in video games is discussed. The original contribution of the work is emphasised. In the final part of the chapter, promising development directions of the method are indicated.

Supplementary materials in the **Appendix** contain the pseudocodes of algorithms that have been implemented and used in the experimental study.

Chapter 2

Basic Concepts

The purpose of this chapter is providing foundations to all basic concepts, upon which the proposed method was built. The chapter begins with a short discussion explaining connections between in-game players and agents in Multi-Agent Systems. As the approach contributes to planning, the field of Automated Planning is introduced. Its particular area related to classical planning is addressed by defining a classical planning problem and discussing modern classical planning algorithms and heuristics. Subsequently, Formal Concept Analysis, which is employed by the proposed heuristic, is introduced together with its formal definitions and a summary of algorithms for constructing a concept (Galois) lattice. The final part of the chapter is addressed to Genetic Algorithms, whose variant was employed for tuning the model. The basic structure of a genetic algorithm and its operators are described.

2.1 Multi-Agent Systems

Multi-Agent Systems (MAS) are a broad field in computer science [10]. The research in this domain is founded on the concept of an abstract agent. It is often referred to as an artificial being that operates autonomously, perceives and adapts to its environment, and pursues defined goals.

The discussed subject focuses on multiple aspects of employing agents for solving problems. These aspects refer to the organisation of relationships between agents [11], their internal architecture [12], communication methods [13], and distributed algorithms for managing them [14], which are the primary focus in this field.

In this section, the terminology of MAS was employed to describe the underlying assumptions, the setting in which the proposed method is applied, and what is the expected outcome. Many video games can be viewed as cases of agent systems, in which characters interacting with a virtual environment are considered as agents. The role of the proposed method is improving observable intelligence of computer-controlled players or agents in general.

The proposed planning method is located in the area of building rational agents as the measure of solution quality depends on the performance of agents accomplishing their objectives. An agent is rational if it seeks for the best outcome, given what information and resources it has. The standard of rationality has strong mathematical foundations, which makes results provided by agents provable.

The dissertation does not discuss agent architectures, ignores communication, simplifies the organisation of a group of agents, and it does not address distributed processing. Challenges related to acquiring and storing knowledge are not taken into account as a game environment is known and appearing problems are defined explicitly. The proposed method focuses on choosing actions by agents, and it can be employed for a single-agent case or a group of cooperating agents controlled by a centralised planning system.

2.2 Automated Planning

Automated Planning is one of the central problems in the field of AI [1]. It is a deliberative process and the component of rational behaviour [2]. Planning is a skill that enables an intelligent agent to prepare actions that lead to a previously specified goal. The process relies on the ability to anticipate future effects of an action. It aims at achieving the best possible outcome. The quality of the process outcome is measured against criteria regarding a problem that is being solved. These criteria are usually expressed as a solution cost, which is optimised. Based on foundations provided by book [2], planning can be defined as follows:

Definition 2.1. “Planning is an abstract process that chooses and organises actions by anticipating their expected outcomes to achieve a defined goal with a minimum cost”.

Although other works may provide their own definitions, differences between them are rather stylistic and refer to the level of detail of declaring the formal assumptions. However, the idea of the planning process remains consistent, e.g. [1]:

Definition 2.2. "(. . .) planning – devising a plan of action to achieve one's goals".

In practice, planning is not always necessary to fulfil objectives. In some cases, a better performance can be achieved by relying on a simple reactive (reflexive) mechanism, which operates without deliberation [15]. It is because the execution of a planning procedure itself is associated with a computational cost. In application areas in which strict limitations are imposed on solution times, planning may become inefficient. However, it is possible to adjust expected solution quality to meet the requirements.

The difficulty of solving a planning problem depends on its properties and adopted assumptions. Many simplifications can be employed to reduce the complexity of a problem significantly. The level of detail of a model describing a problem affects the quality of solutions, and it is adjusted to obtain acceptable results.

The most common practice is expressing a problem in terms of classical planning [16]. In this class of planning problems, the world state is fully observable. Therefore, an initial state is unique and known. Actions are deterministic. They are executed immediately, one at a time and without concurrency. The model describing a problem relies on discrete variables, preferably with a finite number of values.

Temporal planning shares most of the assumptions with classical planning [17]. However, this model considers that actions have a duration, they can temporally overlap, and they can be executed concurrently. The representation of a world state includes information about the current absolute time.

The main feature of probabilistic planning is non-deterministic actions with associated probabilities [18]. Problems in this class are defined on the basis of discrete-time Markov decision processes (MDP). Optionally, partial observability of a world state can be considered. In such case, MPD is replaced by partially observable Markov decision process (POMDP). It also requires employing a mechanism for storing knowledge about a world state, because its variables are not fully observable.

There are many forms of planning. They are associated with types of actions in particular application domains. Some of the examples include navigation planning [19], motion and trajectory planning [20], manipulation planning [21], communication planning [22], planning for information gathering [23], or economic planning [24]. In these cases, planning methods are often adapted to the specificity of a problem.

Each of these forms of planning can be addressed with specific problem representations and specialised tools. In practice, domain-specific approaches are characterised by a high efficiency. The opposite direction is domain-independent planning. It is less efficient than the specialised approach because it does not exploit specifics of a domain. However, it is often less costly to adapt a universal method to a problem instead of solving it anew. Although the proposed approach addresses the commonalities of all forms of planning, it should be considered as domain-specific, because it must be tuned for each problem separately.

The specificity of video games and challenges addressed in this work justify considering planning problems on the level of classical planning. Virtual worlds may be complex, but they are fully observable as the source code is accessible. The duration of actions is often simplified or ignored. The following part of the section formally defines a classical planning problem and describes the principles of solving it.

2.2.1 Classical Planning Problem

Planning is a process that involves combinatorial optimisation [2]. Solving a planning problem instance is finding the cost-optimal state transition path between two nodes in a discrete state-space graph. In such a graph, nodes are system states, and edges are actions. In classical planning, a transition between states is deterministic, and actions are atomic. A path between an initial state and a goal state defines a plan, which is represented as a sequence of actions. The cost of a plan is the sum of the action costs, which are associated with the edges.

Definition 2.3 (State-Space Graph)**.** Let a state space be defined as a directed graph $\mathbb{S} = \langle S, E \rangle$, where S is a set of nodes and E is a set of edges. Each node $s \in S$ is a system state represented by a tuple $s = \langle v_1, v_2, \dots v_n \rangle$ of state variables v. Each edge $e \in E$ is a state transition invoked by an action, which has a cost $c \in \mathbb{R}_{>0}$.

The provided definition of the state space is a mathematical representation that simplifies the discussion regarding planning as a search process. It is assumed that the state space is implicit. It means that a state-space graph is never stored physically. States are visited by executing actions. In practical systems, a typical state-space graph is vast, and it is never fully processed or loaded into the memory.

Definition 2.4 (Planning Task)**.** Let a planning task $t_i \in T$ be a pair $t_i = \langle s_a, s_b \rangle$ of states $s_a, s_b \in S$, where s_a is an initial state, s_b is a goal state, and $s_a \neq s_b$.

For the purpose of the description, the goal in the definition of a planning task is represented by a single state s_b. However, it can be replaced by a set of states S_b, which changes the planning task to finding a transition path between s_a and the nearest state of s_a in S_b.

Definition 2.5 (Plan)**.** A solution to a planning task t_i is a plan p_i. It is defined as a path $p_i = \langle S_i, E_i \rangle$, which is a sequence of states S_i traversed by edges E_i between states s_a and s_b.

The theoretical considerations focus on the cost of traversing the state space. In a practical implementation, a plan will have information which action should be performed by which agent, and in what order.

Definition 2.6 (Plan Cost)**.** The sum of cost values associated with a set of path edges $E_i \in E$ is the cost of a plan p_i, and it is denoted as c_i, Eq. 2.1:

$$c_i = \sum_{e \in E_i} c(e), \tag{2.1}$$

where $c(e)$ is the cost of traversing edge e.

Although minimising the cost of a plan plays an important role, finding any plan is often considered as a challenge. However, pursuing optimality may be necessary for building a high-end computer opponent in video games.

Definition 2.7 (Optimal Plan)**.** A plan p_i^* is optimal if the cost of traversing a path between states in the task t_i is minimal possible (Eq. 2.2). The optimal cost is denoted as c_i^*.

$$p^* \in \{p_i : \arg\min_{p_i}(c_i)\}. \tag{2.2}$$

Solving a planning problem is nontrivial if the cost (or distance) estimate function $\delta : S \times S \to \mathbb{R}_{\geq 0}$ between any two noncontiguous states is unknown, or it is too complex to be computed in a potentially infinite state-space graph. The following sections describe the methods of solving classical planning problems.

2.2.2 State Search Algorithms

State search algorithms traverse a state-space graph to find a path between an initial state and a goal state, which is equivalent to solving a classical planning problem. Such algorithms are the core of planning systems. The next part introduces the most popular algorithms in the discussed area.

Best-First Search

Best-First Search (BFS) algorithms are characterised by exploring graph nodes in a specific order [25]. During the search, a currently visited node is expanded, and its neighbours are generated. The newly generated successors are placed in a priority queue (open list) in which they are sorted according to an evaluation function. In each algorithm iteration, the first node is taken from the queue and expanded so the procedure repeats. Each visited node is added to a closed list to avoid expanding it more than once.

The evaluation function can be expressed as $f(s) = z \cdot c(s) + w \cdot h(s)$, where $c(s)$ is the currently accumulated cost at a state node s, $h(s)$ is an estimated minimal cost of reaching the goal state starting at a state s, and z and w are parameters that can be used to customise the behaviour of a BFS algorithm.

A*

A* is a BFS algorithm with parameters $z = w = 1$ (Appendix, Alg. 9). It performs informed search to find the shortest (cheapest) route between two states in the state space [26]. The search is guided by the heuristic cost estimator $h(\cdot)$ to advance towards the goal. The algorithm returns cost-optimal solutions if $h(\cdot)$ is *admissible*. Admissibility is a property ensuring that the heuristic function never returns an estimated cost that is greater than the real cost of reaching the goal. If the heuristic is consistent (monotonic) then there is no need to visit a state node more than once. The algorithm is optimal and guided by an optimal cost estimator expands a minimum number of states to find an optimal solution.

There are many modifications of A* adapted for particular applications. One of the basic variants is Weighted A*, which scales parameter $w > 1$ to enforce a weak heuristic estimation [27]. The heuristic function becomes inadmissible, but in practice, the method finds solutions faster. There are also specific variants intended for pathfinding, which are covered in the next chapter (Section 3.2).

Dijkstra's Algorithm

Uniform Cost Search (UCS), which is a practical version of Dijkstra's algorithm, can be interpreted as a BFS variant with parameters $z = 1$, $w = 0$ (Appendix, Alg. 8) [28]. It performs an uninformed search by progressively expanding the neighbourhood of an initial state until it finds the goal. The technique is characterised by a low performance, and it is not directly applied in practice. However, A* algorithm guided by an ineffective heuristic estimator can expand the same number of states as UCS. Therefore, UCS can be used as a point of reference to judge the quality of a heuristic.

Enforced Hill-Climbing

Enforced Hill-Climbing (EHC) is similar to a Greedy BFS, in which parameter $z = 0$ and $w = 1$ [29]. The search is solely guided by the heuristic function, and the current cost is ignored. In each iteration, the algorithm uniformly explores the neighbourhood of the current state until it finds a state with a better cost estimation. If a new state is found, the previous one is discarded. Thus, the search can be trapped in a "dead end" and may not return a solution although it exists.

IDA*

Iterative Deepening A* (IDA*) joins the idea of depth-first search and informed search performed by A* [30]. It prioritises successors with a better heuristic estimation while it deepens the search. However, it expands the same state nodes many times, which leads to performance issues. It is characterised by a very low memory usage, which is useful in certain cases, but currently, it is not used by modern planning systems.

2.2.3 Classical Planning Systems

In the literature, the topic of classical planning is dominated by domain-independent planning systems. This trend has its origin in the development of Stanford Research Institute Problem Solver (STRIPS) [31], which is the most recognisable automated planner. The formalism introduced by STRIPS is the foundation of languages that are currently used for describing planning problems [32]. One of the most popular ones is Planning Domain Definition Language (PDDL) [33], which was proposed to standardise planning languages for the need of the first International Planning Competition (IPC). PDDL was also inspired by ADL (Action Description Language) among others. The language has many variants, and extensions, which were adapted for different application areas.

In domain-independent planning, a formal problem domain definition comprises sets of all propositions (Boolean-valued symbols) and actions [16]. A state is represented by a set of propositions (fluents). An action can be executed in a given state if the state satisfies preconditions specified by the action. The effect of an action is defined by operators that modify the current state – they add and delete sets of propositions. A planning problem instance is defined by an initial state and the goal one.

A planning problem description may be enriched by domain-specific knowledge about the structure of actions. In Hierarchical Task Networks (HTNs), actions are expressed as tasks that are decomposed hierarchically [34]. Such task organisation reduces the complexity of solving a planning problem because the goal can be defined as executing a top-level task, which specifies a set of primitive tasks (actions). Also, traversing the state space is faster by using compound actions rather than atomic ones.

The leading planners employ state search [35]. The optimal ones are built around A* algorithm and its variants. The planners that do not ensure optimal solutions use inadmissible heuristics and EHC algorithm for exploring the state space [29]. Formal representation of a planning domain allows performing inference over its structure to reduce its complexity [36]. Actions and paths in the state space that do not lead to the goal can be automatically detected and pruned. Most of the modern domain-independent planning systems are focused on obtaining a heuristic estimate from a propositional representation of a planning problem instance.

The existence of domain-specific planners stems from engineering problems that arise from planning in practice. In video games, which are the focus of this work, the programming paradigm and language are imposed by the engine (e.g., Unity, UnrealEngine, CryEngine), which holds a codebase. PDDL does not give means to fully express all the aspects of game logic or manage its complexity through object-oriented design [37]. Natively implementing a custom game state representation, A* (or EHC) algorithm, and several heuristics is often faster and more efficient than transforming game logic into PDDL and managing the redundant representation. It can be concluded that PDDL can be efficiently used for describing and solving planning problems that were existing in a conceptual form. However, virtual environments, in which planning problems are specified dynamically, usually require a specialised planning subsystem. The presented view is supported by the fact that planners spotted in commercial video games are domain-specific, which is covered in the next chapter (Section 3.3).

This study is dedicated to the practical aspects of planning in video games rather than competing with domain-independent planners. Therefore, the proposed method should be understood as a component that can be integrated with a domain-specific planner. Although the border between the theory of solving planning problems in general and employing planning in practice is not strict, it may determine different methodologies of developing a new planning method. Adapting the proposed heuristic for a domain-independent representation was never one of the goals, and therefore, it is not addressed in this study.

2.2.4 Planning Heuristics

Heuristic estimators are used for improving the performance of search-based deterministic planners by guiding the state search algorithms towards the goal [38]. Automatically obtained estimators represent the leading technique in domain-independent planning.

In the literature, admissible heuristics, which guarantee optimal solutions, are the primary focus. It is because formally defining the quality of an inadmissible heuristic is difficult. The heuristics can be divided into several general groups, which are addressed in the following sections.

Relaxations

Relaxation heuristics are based on the general idea of using a simpler (relaxed) version of the original problem to estimate the cost of solving the complex one. Problem simplification (relaxation) is done by dropping some of the restrictions imposed on available actions in the initial definition of the planning domain. The cost of solving a relaxed problem represents a cost estimation for the original problem. Such a heuristic function is admissible and consistent [38].

One of the automatically obtained relaxed heuristics is the planning graph reachability heuristic, which was introduced by GraphPlan algorithm [39]. The heuristic was later generalised by h^+ [40]. The method ignores delete effects of operators – it simplifies achieving the goal. The heuristic is considered as very informative. However, it is NP-hard to compute, and due to its computational complexity, its inadmissible variants are more frequently applied in practice. The technique is employed by the Fast-Forward planner, which implements one of the leading heuristics [29].

Intuitively, the notion of relaxations can be employed in domain-specific and domain-independent planning as well. Relaxations are also present in other discussed heuristics.

Abstractions

Abstraction heuristics construct abstract problem spaces for a planning problem by dividing it into a number of independent subproblems. The cost of solving each subproblem is calculated as a separate estimate. If the subproblems are independent, then the sum of their cost estimates gives a heuristic function that is admissible and more informative than each of them individually [38].

One of the most recognisable abstraction heuristics is Pattern Databases [41]. The method precomputes and stores in a pattern database the solution costs for all possible subproblems. The database is constructed by searching backwards from the goal and recording the cost, which represents depth in the breadth-first search. It is a very expensive procedure, but it is executed only once before planning starts. During state search, the heuristic estimate is obtained by hash table lookup, which calculates in constant time. The major disadvantage of this approach is a very high memory consumption and the fact that the database has to be recalculated if the goal changes.

It should be noted that obtaining an informative heuristic is difficult, and mapping a planning problem into independent abstract subproblems is not always possible [38].

Landmarks

A landmark can be viewed as a condition that must be satisfied at some state in every plan. It can also refer to a formula or a fact in the propositional representation of a state. Reaching the goal may require achieving the landmarks in a defined order. The landmarks can also be associated with actions (action landmarks). The notion of landmarks provides an intuitive method to locate subgoals and measure the progress of solving a planning task. Therefore, it can be used as an admissible heuristic estimate [38].

An example of a planner that automatically discovers some of the landmarks in the preprocessing phase is LAMA [42]. Its heuristic is inadmissible. The approach may lead to plans that are much longer than the optimal one. Ensuring that landmarks and their order are true in every plan may not be possible in all planning domains.

Critical Paths

The estimate is calculated as the minimum cost of achieving the most expensive subgoal in a defined planning problem [43]. An optimal plan reaching such a subgoal can be understood as a critical path. Its cost is a good approximation if the goal comprises a subgoal that significantly outweighs other ones. Alternatively, the method can be extended to calculating the cost for a subset of subgoals. However, it may become ineffective because the computational complexity is exponential for the number of subgoals. Both variants are admissible [38]. The family of critical paths heuristics generalises the abstraction heuristics and reachability heuristics that underlie the planning graph [43].

Cost Partitioning

Cost partitioning refers to combining heuristics to produce a better one [38]. It is known that the pointwise maximum of two admissible heuristics also gives an admissible estimate. However, it is less informative than their pointwise sum. It is admissible provided that the sum never overestimates the real cost of reaching the goal. Ensemble estimators combining heuristics of different types are state of the art [44].

Space Partitioning

Agents navigating in a virtual environment repeatedly search for the shortest routes. Space partitioning is a standard technique used for accelerating pathfinding [45, 46]. Rather than searching for the path on the most granulated level, space is divided into regions. Each region is a node in an abstract graph. The cost between such nodes is precomputed. Larger regions can nest smaller ones. Such a model is usually utilised for increasing the precision of a heuristic estimating the distance in a non-uniform metric space. Modern pathfinding techniques applied to video games are discussed in the next chapter.

2.3 Formal Concept Analysis

Formal Concept Analysis (FCA) is a subfield of applied mathematics [47]. It had its origin in order and lattice theory [48]. FCA mathematises concepts and concept hierarchies [49]. Concepts refer to the theory of philosophical logics of human thought, and they can be understood as units of thought. Specifically, they can be interpreted as cognitive acts and knowledge units that are potentially independent of language.

A single concept is formed by its extension and intension, which FCA formalises. The extent contains all abstractly defined objects that belong to the concept. The intent is constituted by all attributes that apply to all objects in the extent. An attribute can also be understood as a property or meaning.

The relationship between concepts is defined by subconcept-superconcept relation. In such relation, a concept is a subconcept of a superconcept if the extent of the subconcept is contained in the extent of the superconcept. Equivalently, the intent of the subconcept contains the intent of the superconcept. The relation determines a hierarchy of concepts, which is termed a concept lattice.

A concept lattice is best represented by a diagram. Figure 2.1 depicts an example of such a diagram. The example refers to a hierarchy of video games and their genres. In the diagram, the nodes symbolise concepts, and the edges indicate subconcept-superconcept relations. A horizontal separator divides each node – the upper part lists objects and the lower one attributes. Each node contains game titles representing the extent of a concept. The intent is expressed by a set of genres applying to a preceding set of game titles.

FCA is an interdisciplinary area, and it has a broad range of application domains. It is especially useful in the areas including, but not limited to ontologies [50], linguistics [51], data mining [52], software engineering [53], and economics [54]. This work introduces the application of FCA for classical planning.

2.3.1 Formalisation

The following description introduces basic definitions and the formal foundations of concept lattices based on work [49].

Definition 2.8 (Formal Context)**.** A formal context is a triple $K = \langle G, M, I\rangle$, where G is a set of *objects*, M is a set of *attributes*, and $I \subseteq G \times M$ is the binary relation of *incidence*, which is regarded as a bipartite graph associating objects with attributes.

The definition of a formal concept is formulated using Galois operators, Def. 2.9:

Definition 2.9 (Galois Operators)**.** For sets of objects $G_i \subseteq G$ and attributes $M_j \subseteq M$, Galois operators ($'$) are defined as follows (Eq. 2.3 and Eq. 2.4):

$$G_i' = \{m \in M : \forall_{g \in G_i} I(g, m)\}, \tag{2.3}$$

$$M_j' = \{g \in G : \forall_{m \in M_j} I(g, m)\}, \tag{2.4}$$

where $I(g, m)$ is a predicate denoting that object g has attribute m.

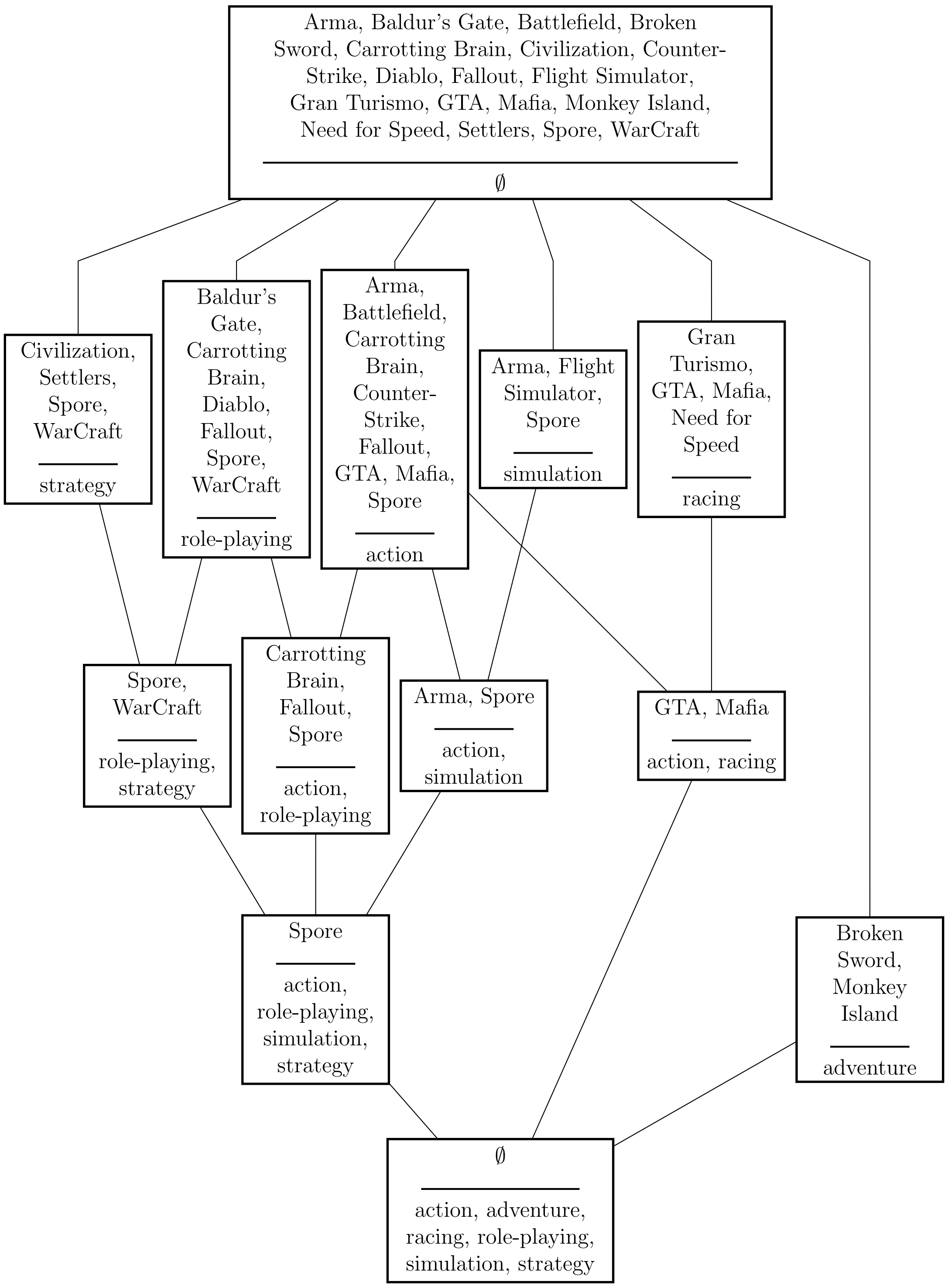
Arma, Baldur's Gate, Battlefield, Broken Sword, Carrotting Brain, Civilization, Counter-Strike, Diablo, Fallout, Flight Simulator, Gran Turismo, GTA, Mafia, Monkey Island, Need for Speed, Settlers, Spore, WarCraft
∅
Civilization, Settlers, Spore, WarCraft
strategy
Baldur's Gate, Carrotting Brain, Diablo, Fallout, Spore, WarCraft
role-playing
Arma, Battlefield, Carrotting Brain, Counter-Strike, Fallout, GTA, Mafia, Spore
action
Arma, Flight Simulator, Spore
simulation
Gran Turismo, GTA, Mafia, Need for Speed
racing
Spore, WarCraft
role-playing, strategy
Carrotting Brain, Fallout, Spore
action, role-playing
Arma, Spore
action, simulation
GTA, Mafia
action, racing
Spore
action, role-playing, simulation, strategy
Broken Sword, Monkey Island
adventure
∅
action, adventure, racing, role-playing, simulation, strategy

FIGURE 2.1: An example of a lattice diagram representing selected video games and genres that describe them

To intuitively understand the meaning of Galois operators, it can be said that G_i' gives a set M_i of attributes possessed by all objects in G_i. By analogy, M_i' gives a set G_i of objects that have all attributes in M_i.

Formal concepts are organized as nodes in the concept lattice of a formal context (Def. 2.10).

Definition 2.10 (Formal Concept)**.** A pair $\langle G_i, M_j \rangle$ is a formal concept of a context K if it satisfies Eq. 2.5:

$$(G_i' = M_j) \wedge (G_i = M_j'), \tag{2.5}$$

where the sets $G_i \subseteq G$ and $M_j \subseteq M$ are called the extent and the intent of a formal concept respectively.

The hierarchy of concepts is defined by the subconcept-superconcept relation, Def. 2.11:

Definition 2.11 (Concept Relation)**.** For two formal concepts $\langle G_i, M_i \rangle$ and $\langle G_j, M_j \rangle$, a partial order operator ($\leq$) is defined by Eq. 2.6:

$$\langle G_i, M_i \rangle \leq \langle G_j, M_j \rangle \iff (G_i \subseteq G_j) \wedge (M_j \subseteq M_i). \tag{2.6}$$

Thus, $\langle G_i, M_i \rangle$ is a subconcept of $\langle G_j, M_j \rangle$ and $\langle G_j, M_j \rangle$ is a superconcept of $\langle G_i, M_i \rangle$. The relation is transitive.

Definition 2.12 (Concept Lattice)**.** The ordered set of all formal concepts of a formal context K is a concept lattice $L(K)$.

The characteristic structure of a concept lattice is defined by the following theorem, Def. 2.13:

Theorem 2.13 (The Basic Theorem on Concept Lattices)**.** If a concept lattice $L(K)$ is complete, then its infimum ($\bigwedge$) and supremum ($\bigvee$) can be described as follows (Eq. 2.7 and Eq. 2.8):

$$\bigwedge_i \langle G_i, M_i \rangle = \langle \bigcap_i G_i, (\bigcap_i G_i)' \rangle, \tag{2.7}$$

$$\bigvee_i \langle G_i, M_i \rangle = \langle (\bigcap_i M_i)', \bigcap_i M_i \rangle. \tag{2.8}$$

Supremum refers to the topmost node in a lattice, and infimum is the lowermost node.

2.3.2 Lattice Construction

Aside from the theoretical foundations of FCA, lattice construction methods are primarily relevant for the proposed approach. Table 2.1 summarises some of the well-known algorithms for generating concept lattices [55]. The table provides information about the computational complexity of the algorithms and whether they operate incrementally. The complexity notation $O(\cdot)$ uses the following symbols:

- K is a formal context comprising objects G, attributes M, and the incidence relation (it expresses which objects have which attributes),
- $L(K)$ is a set of formal concepts of a formal context forming a concept lattice,
- G_i is a set of objects that belongs to one of the formal concepts in a lattice,
- C is the maximal number of candidate sets considered when accessing the formal context [56].

Algorithm	**Complexity**	**Incremental**
AddIntent [57]	$O(\|G\|^2 max(\|G_i'\|)\|L(K)\|)$	✓
Bordat [58]	$O(\|G\|\|M\|^2\|L(K)\|)$	
Chein [59]	$O(\|G\|^3\|M\|\|L(K)\|)$	
Close by One [60]	$O(\|G\|^2\|M\|\|L(K)\|)$	
Mowling [61]	$O(\|G\|\|M\|^2\|L(K)\|)$	✓
Ganter (Next-Closure) [62]	$O(\|G\|^2\|M\|\|L(K)\|)$	
Godin [63]	$O(\|G\|\|L(K)\|)$	✓
Lindig [64]	$O(\|G\|^2\|M\|\|L(K)\|)$	
Norris [65]	$O(\|G\|^2\|M\|\|L(K)\|)$	✓
Nourine [66]	$O((\|G\| + \|M\|)\|G\|\|L(K)\|)$	✓
Titanic [56]	$O(C\|G\|\|M\|\|L(K)\|)$	

Table 2.1: A list of selected algorithms for constructing concept lattices

It should be noted that the complexity measurements may have inconsistent accuracy as some of the authors obtained them empirically and other ones derived them from theoretical assumptions. The algorithms are characterised by various performance depending on the input dataset, and not all construct a diagram graph [67]. For small and medium context, one of the fastest methods is Bordat's algorithm. For larger data, Norris' algorithm is usually a better choice. New lattice construction methods are continuously invented [68, 69].

2.4 Genetic Algorithm

A Genetic Algorithm (GA) is an adaptive metaheuristic that belongs to the class of Evolutionary Algorithms (EAs) [70]. Metaheuristics are algorithms that involve stochastic optimisation in which candidate solutions are gradually improved through random modification [71]. Evolutionary approaches imitate processes existing in nature.

GAs are principally applied to solving complex optimisation problems, which is understood as searching for an optimum of the objective function. In principle, GA does not guarantee to find a global optimum, but it provides near-optimal solutions in a relatively short time [70]. Hence, they are suitable for NP-hard problems which cannot be solved in polynomial time. For such problems, information that could guide the search towards the solution is unavailable, and the solution space is too big to conduct brute-force search.

GA is a metaphor for the biological evolution process in which evolving individuals represent candidate solutions [70]. Solutions carried by individuals are encoded as chromosomes. The objective function of a problem being solved is represented as a fitness function, which is used for evaluating the individuals based on the quality of solutions encoded in their chromosomes.

The algorithm simulates an evolution of a population of individuals. The initial population is usually picked randomly. In each generation, fitness each of individual in the population is calculated. The probability of surviving an individual to the next generation depends on its fitness. Individuals that survived have a chance to become parents of new individuals. During the process, the chromosomes of individuals are the subject of genetic operators, which include recombination and mutation.

The procedure continues to generate new generations until a stop condition is reached. Such a condition may be defined as reaching an arbitrarily provided number of generations, exceeding a specified fitness threshold, prolonging stagnation, or lack of diversity in a population. The routine returns the best solution recorded during its execution. The quality of solutions produced by the algorithm heavily depends on configuration parameters steering its execution.

Genetic algorithms are a popular tool because of their wide range of application. The list is so big that it would be difficult to provide areas in which they are not applied. A comprehensive survey on this topic can be found in book [72].

Although metaheuristics are capable of solving any complex problem, they can hardly compete with automated planning methods in the studied problem domain. Planning methods usually employ efficient algorithms and smart heuristics that exploit information about the expected outcome to advance towards the solution quickly. A high degree of specialisation enables them to use computational resources efficiently. It is the primary requirement for methods operating in the considered field of application. In this study, a genetic algorithm is applied only for tuning parameters of the proposed method, which is done offline – in the phase of preparation for solving planning problems appearing in a specific environment.

2.4.1 Coding

Coding is the method of representing a candidate solution of a problem as a chromosome that is comprised of genes, Eq. 2.9:

$$X_i = \langle x_1, x_2, \dots x_n \rangle, \tag{2.9}$$

where $X_i \in X$ is the chromosome of i-th individual, and x is a gene.

A coding method is usually determined by the specificity of a solved problem. The method should ensure that the genetic operators applied to chromosomes produce valid candidate solutions. One of the popular techniques is converting parameters defining a candidate solution into a binary string. Alternatively, such a string can comprise a set of alphanumeric characters. These are universal methods, but they may require an additional procedure of correcting invalid chromosomes potentially produced by the genetic operators.

Better control over the validity of parameters stored in a chromosome can be gained by preserving their original data types, which usually limits to float, integer, and Boolean. Thus, genetic operators can be designed in the way that they take into account acceptable ranges of the parameters. This technique was used for developing a coding method for the discussed problem.

2.4.2 Algorithm

The basic structure of a genetic algorithm can be described by the following steps (Alg. 1) [73]. In the beginning, the routine initialises a population of a number (*popSize*) of randomly generated individuals (line 1). Next, each individual in population is evaluated – their fitness is calculated (line 2). An individual with maximum fitness is stored (line 3). The algorithm operating in the elitist mode ensures that the population always contains the best individual observed – it replaces the worst one. The optimisation process is done over a defined number of generations (line 4). At the beginning of the loop, a new population is randomly selected by favouring individuals with the highest fitness (line 5). In the next part, the chromosomes of population individuals are crossed over by exchanging parts of the genotype (line 6). Subsequently, the chromosomes are mutated by applying small and random modifications (line 7). Finally, the population is evaluated, and the best individual is saved (lines 8-9). The algorithm returns the best individual that was observed over the generations (line 10). Alternatively, the loop may be stopped as soon as a satisfying solution was found.

```
Alg. 1: GeneticAlgorithm()
 1  var population ← Initialize(popSize)          ▷ generate a set of random individuals
 2  var fitness ← Evaluate(population)             ▷ calculate fitness for each individual
 3  var bestIndividual ← Elitist(population, fitness)   ▷ find an individual with maximum
    fitness
 4  for g ← 0; g < generations; g++ do
 5      population ← Select(population, fitness)
 6      population ← Crossover(population)
 7      population ← Mutate(population)
 8      fitness ← Evaluate(population)
 9      bestIndividual ← Elitist(population, fitness)
10  return bestIndividual
```

There are many variants of the presented algorithm. Some of them have concurrent populations and focus on imitating processes observed in nature realistically. These are not taken into consideration in this study.

In order to apply the algorithm to solving a problem, several steps must be performed. First, a method of representing a candidate solution as a chromosome and calculating fitness must be defined. For the selected representation, the implementation of mutation and crossover operators must be provided. A selection operator can be chosen from the existing ones. A stop condition can be customised.

Evaluation

Calculating the fitness of an individual often requires decoding its chromosome. A fitness function may transform the original value expressing the quality of a candidate solution. The function can be adjusted to favour promising individuals, avoid overly assimilation of individuals in a population, or penalise invalid solutions. These techniques are not employed in this work, and the fitness formula for the solved problem will be provided later.

Selection

Selection is the procedure of choosing individuals that will be crossed over. The probability of selecting an individual as a parent is proportional to its fitness.

A popular selection method is the roulette wheel algorithm. It is implemented as an imaginary roulette in which pockets represent individuals. The size of each roulette pocket is proportional to the corresponding individual's fitness. Usually, the fitness values are normalised. Selecting an individual is equivalent to generating a random number that falls in one of the pockets. The method is used to build a new set of individuals that will replace the previous population.

Alg. 2: TournamentSelection(*population*, *fitness*)

```
1  var $tournSize \leftarrow tournPerc \cdot |population|$
2  var $newPopulation \leftarrow \{\}$
3  while $|newPopulation| < |population|$ do
4      var $tournPopulation \leftarrow population$
5      var $bestIndividual \leftarrow \epsilon$
6      for $i \leftarrow 0;\ i < tournSize;\ i{+}{+}$ do
7          var $individual \leftarrow$ GetRandom($tournPopulation$)
8          Remove($tournPopulation$, $individual$)
9          if GetFitness($bestIndividual$) < GetFitness($individual$) then
10             $bestIndividual \leftarrow individual$
11     Add($newPopulation$, $bestIndividual$)
12 return $newPopulation$
```

A simpler selection method that avoids potential problems with processing fitness values is the tournament algorithm (Alg. 2). The procedure is steered by the tournament percentage parameter (*tournPerc*) – line 1. It is used for calculating the size of a subset of population individuals that are randomly picked for a tournament (lines 6-8). The best individual in a tournament is added to a new population (lines 9-11). The procedure is repeated until the number of individuals in the new population matches the population size (line 3). Eventually, the new population replaces the old one (line 12).

Certain transformations can be applied to the fitness function to steer the development of the population [70]. In this study, they were not employed as the desired results were obtained by using the basic variant of the tournament selection.

Crossover

Once a new population is selected, individuals are paired and crossed over in accordance with a crossover probability (*crossProb*) provided as a parameter (Alg. 3). Children, which represent the crossed chromosomes, replace the parents.

Alg. 3: Crossover(*population*)

```
var $first \leftarrow 0$
var $pair \leftarrow 0$
for $i \leftarrow 0; i < |population|; i{+}{+}$ do
    if $crossProb \geq$ GetRandom(0.0, 1.0) then
        $pair{+}{+}$
        if ($pair$ mod 2) = 0 then
            Cross($population[first]$, $population[i]$)
        else
            $first \leftarrow i$
```

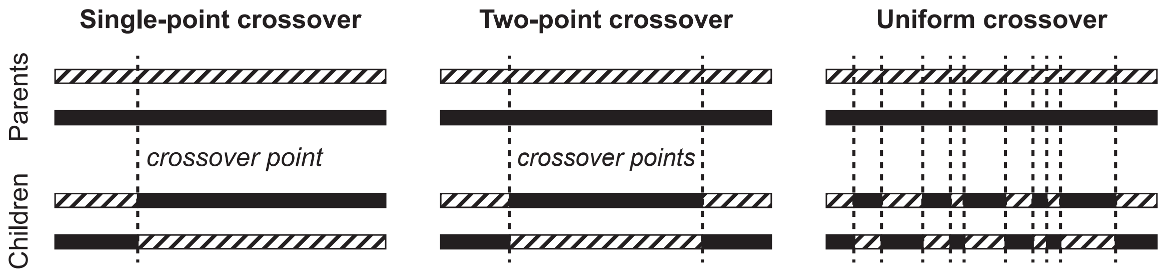

FIGURE 2.2: Crossover techniques

Different crossover techniques can be applied depending on the type of chromosome coding (Fig. 2.2). The single-point crossover resembles the natural process in which two chromosomes are cut at a random point, and then, the resulting parts are exchanged between the parents. In the two-point crossover, two cut points are selected, and all genes between them are swapped between two chromosomes. The uniform crossover method swaps genes with a specified probability (*crossFactor*), which is usually set to 50%. This technique seems to be the most suitable for chromosome coding that was selected for the discussed problem.

Mutation

The idea of the mutation procedure is similar to biological mutation. It involves applying small random changes to genes with a defined probability (*mutProb*). In the binary representation of genes, mutation is implemented as inverting a bit at a random position. For integer and float genes, often used technique is replacing them by random numbers inside specified bounds. However, picking a random value can cause rapid changes, and an alternative approach is adding (or subtracting) a small random number while keeping the result in the bounds. In the proposed approach, the strength of mutation is steered by a mutation factor (*mutFactor*), Eq. 2.10:

$$x(g+1) = x(g) + r \cdot mutFactor, \tag{2.10}$$

where $x(g)$ is a gene in g-th generation, $r \in [-1, 1]$ is a randomly picked number, and $mutFactor > 0$.

Chapter 3

Related Work

The chapter provides a background of AI methods employed in video games and summarises the main areas. It emphasises the importance of classical planning and indicates the areas potentially influenced by the new planning method. The next part of the chapter is dedicated to classical planning methods that are used in video games. It covers pathfinding and general action planning as the proposed method is not strictly limited to one of these problems. Subsequently, approaches that utilise game replays are surveyed. The final section discusses promising research directions and potential advantages of the proposed approach in the context of the current solutions.

3.1 Artificial Intelligence in Video Games

Before the discussion referring to Artificial Intelligence in video games can begin, it should be clearly stated that the academic understanding of AI and the meaning of AI in commercial games may differ. A researcher is interested in solving a complex problem and potentially developing a method that would be meaningful in the field of AI. From the commercial perspective, a video game is a product, and the cost of producing it should match quality that is expected to make it successful. Therefore, the goal of game developers is to ship the perception of intelligence by sacrificing minimal cost, which usually refers to the development time.

In view of such reality, the developers build solutions upon existing AI methods but often employ trivial techniques for simulating human-like behaviours of Non-Player Characters (NPCs) [74]. The methods may include workarounds and cheats to avoid solving complex problems. For instance, the behaviour of NPCs may be predefined and rigidly scripted by the designers. A challenging opponent is achieved by giving NPCs extra resources or access to helpful information that would be unavailable to a human player in the same circumstances. Sometimes the abilities of NPCs must be reduced to balance the competition with human players. That applies to simple cases in which human agility or memory is challenged – cognitive and deliberative abilities of the human brain are not yet surpassed. Because video games focus on good gameplay, building an illusion of intelligence by employing simplistic solutions is acceptable as long as the tricks remain concealed from the players.

However, a continually growing need for believable NPCs and new challenges that are encountered during the development of immense virtual worlds make the developers take advantage of traditional AI methods more often. Currently, the domain of AI in video games is being viewed as a separate research field [75]. The following areas can be identified[1]:

1. NPC behaviour learning – it aims at learning policies (or behaviours) that perform well in a game which is viewed as a reinforcement learning problem [76, 77]. Reinforcement learning techniques and evolutionary approaches are employed for finding the best NPC's policy that maximises the game score [78].

2. **Search and planning** – it focuses on state search problems. Finding a sequence of actions that leads to a goal being a specified game state employs best-first search methods [37, 79]. They are used for pathfinding and solving planning tasks as well [80, 81]. Picking actions that will give the best game score involves Minimax-based game tree search [82, 83].

3. Player modelling – computational models are created and used for identifying types of players and analysing their interactions with the game [84]. It helps game designers to assess how the gameplay affects player experience. Machine learning and data mining methods are employed [85].

4. **Games as AI benchmarks** – problems and tasks existing in games may be used for evaluating and comparing the performance of AI methods [86, 87]. Some of the cases reflect real-world scenarios or at least are characterised by a high degree of analogy. Therefore, methods specialising in solving these problems may also have a practical application outside the domain of games. Planning methods were employed by the winners of StarCraft and Mario AI competitions [88, 89]. The authors of survey [75]

[1]Areas for which planning is particularly relevant are in bold type.

point out that there are no video games as benchmarks or competitions focused specifically on planning techniques.

5. Procedural content generation – it supports the creation of game content by employing automatic and semiautomatic methods, which include evolutionary search and constraint solving [90]. The quality of generated content must be evaluated [91]. Potentially, automated planning can be utilised for testing procedurally generated game levels [75].

6. **Computational narrative** – it is associated with the representational and generational aspects of stories that are told by a game, which can be considered as a form of narrative [92]. Planning methods are employed for automatically creating or maintaining a coherent story plot [93].

7. **Believable agents** – building NPCs that have believable human-like characteristics is a central problem in video games [94]. From the perspective of imitating human behaviours, it is achieved by developing reactive models [95]. However, one of the most important aspects of believability is solving complex tasks that require planning [9].

8. AI-assisted game design – it addresses employing AI methods as tools supporting the game design and development processes [96]. Apart from assisting game content creation, the primary focus of the tools is the development of game mechanics and game rulesets [97]. Potentially, the tools can employ planning methods for validating the gameplay [75].

9. General game AI – it is aimed at building a universal agent that is capable of playing a large variety of games [98]. The research is often associated with the area of Artificial General Intelligence, and it influences many aspects of AI in games [99].

10. **AI in commercial games** – it concentrates on game monetization, which is raised by learning about the target audience and shipping game features that maximise the player experience. Although a contribution to AI is not the primary focus, advanced AI methods can be employed to distinguish and promote the product [100, 101]. Planning features in several successful productions such as F.E.A.R., Killzone 3, or Transformers 3: Fall of Cybertron [9].

To summarise, classical planning methods and their variants are considered as one of the primary tools that influence and contribute to the areas of search and planning, games as AI benchmarks, computational narrative, believable agents, and AI in commercial games.

As a secondary tool, they may appear in the areas of procedural content generation, general game AI, and AI-assisted game design.

3.2 Path Planning

Pathfinding is a special case of classical planning in which a set of actions is limited to the movement of an agent on a game map. In virtual environments, a pathfinding system is commonly implemented as a search-based domain-specific planner optimised for best performance.

In video games, pathfinders are employed for finding the shortest (fastest) route to the target location on a game map [80]. Because virtual worlds are discrete and usually ignore the physical state of agents, motion trajectory planning problems, which are the case in robotic systems, are avoided. Although it would appear that pathfinding is a “solved problem”, there is a range of challenges that are the topic of active research. Some of them are related to moving many agents while avoiding collisions in a limited space or navigating through dynamic environments, which are modified during the game. Another issue is that the scale and complexity of virtual worlds in commercial games grows fast while the capabilities of hardware are often fixed at a certain level (e.g., consoles).

Representing a game map as a discrete grid is a popular method of transforming it into a search graph. In video games, a typical pathfinder is built upon A* algorithm using Euclidean (or Manhattan) distance as an admissible heuristic function [74]. It can be viewed as a baseline approach to pathfinding, and modern systems extend it. The following part introduces these extensions to give an overview of currently used methods although they strictly address **metric spaces** and are not applicable for general action planning, which is the subject of this study.

Manhattan grids with uniform costs usually form metric spaces in which two points can be connected with many optimal paths. Exploring such equivalent solutions needlessly consumes resources. This problem is called symmetry, and it is addressed by JPS (Jump-Point-Search) [102]. The algorithm prunes successors of a node to eliminate the symmetry. The method preserves optimality and completeness.

Grid-based maps are commonly used because they simplify many aspects of the game design. However, they introduce discretization that aligns objects to the grid. It does not allow for building realistic maps with objects of arbitrary shapes and sizes, and agents moving in any direction. This problem can be solved by building a triangular representation of a game map. DCDT (Dynamic Constrained Delaunay Triangulation) is a preprocessing algorithm that isolates obstacles, covers the surface of a map with polygons, and finds paths between sectors for circular objects [103]. It is used together with TRA* (Triangulation Reduction A*), which is a specialised variant of A* operating in a triangulation graph [104].

A heuristic based on Euclidean (or Manhattan) distance is accurate for metric spaces with uniform costs, and it gives a fair approximation for most game maps. However, the topology of a map can be diversified, which reduces the accuracy of the estimator. A more informative heuristic can be obtained by preprocessing static parts of a map. A popular method used for road networks is ALT (A* search, Landmarks and Triangle inequality or also known as a differential heuristic) and its variants [105, 106]. The method relies on landmarks that are arbitrarily selected nodes in a search graph. It builds a look-up table with real distances between regular nodes and the landmarks. The heuristic uses the table to approximate the distance between any two nodes. The heuristic can potentially detect unreachable nodes.

Another method is partitioning the topology of a map and decomposing it into disjoint areas [107]. The method locates entrances between adjacent regions and precomputes optimal distances between pairs of the entrances. It can identify regions that are irrelevant for solving a particular pathfinding problem instance. The family of heuristics that rely on precomputed distances between nodes is often referred to as true distance heuristics [108, 109].

The regions can be organised hierarchically. In that case, the search problem is solved starting from the highest abstraction level, going through intermediate ones, and ending on the ground one. One of the most successful methods is HPA* (Hierarchical Pathfinding A*), which hierarchically decomposes a map into disjoint square sectors [110]. Entrances between adjacent sectors are connected with edges on each abstraction level. The structure ensures that the search procedure never backtracks between hierarchical levels.

A similar idea of hierarchical decomposition is shared by several other algorithms like PRA* (Partial Refinement A*) [111], HAA* (Hierarchical Annotated A*) [112], DHPA* (Dynamic HPA*) [113], or Block A* [114]. PRA* is characterised by a bottom-up composition of adjacent nodes that form a region. It results in a pyramidal representation of the space. HPA* and PRA* can be combined to reduce memory usage [115]. HAA* can be viewed as an extension of HPA*. Its partitioning model takes into account obstacles and terrain types, which is useful information for planning a route for agents characterised by different sizes and capabilities. DHPA* is a variant of HPA* intended for dynamic environments, in which traversable paths may be modified during the game. Block A* precomputes a database of paths between each pair of sectors taking into account any-angle transitions between sectors.

It is clear that using precomputed paths is much faster than computing them in real-time [74]. However, it requires an impractically large amount of free memory, which is the fundamental obstacle to using this technique on a bigger scale. Recent research on this problem resulted in the concept of compressed path databases [116]. It is a lossless compression technique that exploits the property of path coherence, which can be used to avoid storing repeatable transitions that lead to an arbitrary goal through the same area.

As a route for an agent is usually computed in real-time, a pathfinding system must fit within strict time constraints. Waiting for the system to calculate a complete path would look unnatural. An agent should be responsive and start performing an action as soon as possible. This can be achieved by employing real-time search [117]. One of the popular algorithms is LRTA* (Learning Real-Time A*) and its variants [118]. They comprise a planning step, a learning step and an acting step. In the first step, an agent explores his surroundings. In the next one, the results of exploration update its knowledge. Finally, the agent chooses an action. The steps are repeated until the goal location is reached. The main drawback of this approach is an effect called "scrubbing" [119]. It happens during the learning phase when the algorithm enters a local minimum (plateau), which causes states to be revisited repeatedly. It leads to low-quality solutions and irrational behaviour of an agent. This issue is addressed by HCDPS (Hill-Climbing and Dynamic Programming Search), which precomputes a database of subgoals and partitions the search space into reachability regions [119]. The method replaces the learning step with greedy hill climbing. It allows for escaping local minima and eliminating the scrubbing effect.

One of the most challenging problems in the area of pathfinding is managing multiple agents. In a team of cooperating agents, each agent can move to a different target location. If the agents plan independently, their routes can potentially interfere and produce collisions. The difficulty of solving this problem globally by treating a team as a single multi-agent comes from the fact that the number of possible actions is exponential in the number of agents. It leads to a high branching factor in a state-space graph. There are several approaches to solving this problem. The fastest one is computing a path for each agent concurrently and resolving conflicts when they occur [74]. However, this approach may be ineffective if agent cooperation is the priority. Thus, another method of reducing the branching factor is considering agent actions sequentially [120]. In such a case, each agent chooses an action in its turn. There are also attempts to detect independent actions and perform sequential processing only if it is required [121].

Pathfinding is usually separated from general action planning, and these issues are addressed on different levels of granularity. However, in some scenarios, traversability of paths on a game map may depend on objects owned by an agent (e.g., keys to doors). Work [81] addresses inventory-driven pathfinding by proposing InvJPS algorithm as a variant of previously discussed JPS. For the considered scenario, the state representation comprises the location of an agent and a list of items in its inventory. Obtaining an item is treated as an intermediate goal. Unfortunately, the method does not resolve which items are required to reach the target position or estimate the distance to the goal state. Thus, it ensures optimality by performing an uninformed search over the intermediate goals.

3.3 Action Planning

In video games, classical planning methods are employed for performing complex in-game activities such as controlling agent tactics, solving logistics, or managing the player's base [75]. Such activities involve tasks that require deliberate actions leading to the desired goal. For instance, these tasks may include coordinating units, transporting resources, or planning production.

One of the first successful attempts to utilise the idea of planning in a commercial game was presented in work [122]. Planning was implemented for F.E.A.R., a popular first-person shooter (FPS) game. It was used for providing a high degree of realism of combat between the player and a team of enemy agents who were expected to act smart. The proposed planning system, GOAP (Goal-Oriented Action Planning), can be perceived as a domain-specific variant of STRIPS with several improvements. One of them was employing A* algorithm as the actions had assigned costs. Next, a symbolic representation was replaced by a fixed-size array of variables describing to the world state. Thus, preconditions and effects of actions were procedurally computed instead of using standard add/remove lists. Such a design allowed for a more efficient usage of computational resources.

The approach was later extended by LGOAP (Layered GOAP) [123]. It replaced A* with IDA* and introduced a hierarchical organisation in which actions are considered on different levels of granularity. For instance, accomplishing an in-game quest can be placed on a higher level while moving to a position is on a lower one. However, the authors did not consider obtaining a heuristic estimator automatically.

Despite recent advances in the field of automated planning, applying planning to video games is characterised by several challenges that are the subject of active research. An exhaustive planning can be avoided by generating plans from a set of predefined tasks as it was done for steering simple behaviours of a bot in Half-Life game [124]. Existing domain-independent planning systems can be configured for a particular application domain, but they are not specially adapted for real-time planning – they build a complete plan before it is executed [37]. Games that utilise such planning engines model the world as it was static and continually replan whilst its state dynamically changes, and the current plan becomes obsolete.

Another issue is that the planning systems impose a symbolic representation based on first-order logic, which is inadequate for abstracting numerical information [37]. Reasoning with resources in the form of numeric variables is an essential element in many games. A numeric representation is also useful for expressing spatial dependencies and complex relations between objects. These properties often determine the availability and effectiveness of in-game actions. For instance, weapons usually have a specified range, and their actual damage is calculated based on several factors characterising the attacker and the victim. The complications implied by a symbolic representation were a sufficient argument for abandoning it and using a game-specific representation [37].

A similar path was followed by the authors of IW(i) (Iterated Width) algorithm [125]. The method assumes that the game state is characterised by a set of Boolean atoms (features). The representation of the state has the form of a set of variables, which works better with simulators. An atom is defined as a pair comprising a variable and its value. In a given state, an atom is true if its variable has a specified value. IW(i) can be understood as a variant of breadth-first search that prunes a newly generated state if it does not make true a new tuple of at most i atoms. For instance, IW(1) prunes a new state if it does not make a new atom true. This technique can be successfully used for sequentially achieving subgoals as fragments (atoms) of the final goal. It is effective for cases, in which such subgoals can be easily identified. Thus, the method does not perform well in puzzle games, because the progress of solving a problem instance may not be encoded in the representation of the game state.

Performing a state search with a weak heuristic estimate is costly and providing an accurate estimator for a complex problem is difficult. Consequently, some games solve planning problems by focusing on activities that should be performed rather than goals that must be achieved. These activities can be hierarchically organised as tasks by HTN planners such as SHOP2 (Simple Hierarchical Ordered Planner 2) [7]. A hierarchical task network can be viewed as a predefined template for generating plans that match particular situations in a game. Executing such a plan solves a planning problem instance and leads to one of the goals available in the template. In other words, these systems do not allow for arbitrarily selecting the goal state, but they are limited to the ones taken into account by the designers. For instance, the described approach can be successfully applied for constructing buildings. The task can be decomposed into obtaining resources, transporting them to the building site, and engaging builders. However, this approach may become less effective for a dynamic battlefield as it can evolve into a large number of unanticipated situations. Destroying enemy units may require different means and involve unique tasks. Therefore, it makes more sense to solve a problem instance algorithmically, rather than trying to prepare for all possible cases.

One of the important aspects of employing a state search method is selecting a goal state. Such a goal can be provided by specifying values of state variables that match the desired world state, which is a common practice in pathfinding, or declaring facts that must become true, which is equivalent to selecting a subset of symbols in a symbolic representation. Planning is usually conducted in a specific context in a sense that actions or variables that do not directly affect a solved problem are ignored rather than addressing each problem globally. For instance, a pathfinding system and a strategic action planner often use distinct representations of the game state and are placed on different layers of a decision system.

The role of a strategy (tactical) planner is usually taken by the player who controls units by giving them orders (objectives). A human player can be replaced by a game theoretic approach, which involves Minimax search of the game tree [126]. Without going into details, the method is used for finding an action that leads to the best possible game score by considering all actions of each player. For zero-sum games, the basic Minimax algorithm can be enhanced with Alpha-Beta pruning to reduce the branching factor. MCTS (Monte Carlo Tree Search), which is recently the subject of amusement, combines the tree search with Monte Carlo sampling [82, 83]. These methods have been successfully applied to several Atari games and computerised board games such as checkers, Go, or chess. Unfortunately, modern video games are characterised by a branching factor and a depth of the game tree that are several orders of magnitude greater [37]. A regular match may comprise long series of insignificant atomic actions that indirectly affect the game score. Therefore, a more reasonable approach is conducting game tree search for macro actions to reduce the number of processed nodes. A sequence of atomic actions joining two adjacent nodes in a game tree (an initial state and a goal state) could be found using classical planning, which can be significantly accelerated by a heuristic. For instance, a game theoretic method may be used to find the best location for a military base, and a planning method could resolve whether it is possible to build it there, how much time it will take, and what actions should be performed and in what order. Such an approach seems to be theoretically valid, but it is difficult to confirm whether it was successfully applied in practice as the game designers prefer simple architectures. Usually, goals are activated based on a set of predefined criteria [122].

Recent research in the area of classical planning was dedicated to general video game playing, in which the objective is to maximise the score having limited knowledge about the structure of a solved problem. Work [79] compares the performance of game theoretic (Monte Carlo tree search) and classical planning (IW(1) and best-first search) methods for Atari games in the Arcade Learning Environment (ALE) that treats each game as a simulation. The results show that game theoretic approach performs worse than classical planning. The purpose of game theory is modelling a competitor as an intelligent being who seeks to optimise its score, and applying it to a single-player scenario is not well justified. Because planning methods ignore opponent's actions by principle, they can be heuristically guided, which effectively reduces the search space. It should be also noted that arcade games challenge player's agility rather than planning skills. In fact, these games can be solved by applying a reactive approach as their mechanics is based on collecting rewards, rather than achieving complicated goals that would require planning.

Because developing an efficient planning system is difficult in practice, automated planning is used only when necessary, or the benefit of using it is considerable. For simple activities like patrolling, attacking or evading, it is often replaced by a reactive approach such as Behaviour Trees (BTs) [37]. BT is a tree structure that hierarchically organises rigidly predefined agent activities. Each tree leaf holds an implementation of some gaming behaviour. For a given game state, the currently active leaf is selected by traversing the tree in the breadth-first order and evaluating conditions associated with the nodes. The principle of the model is somehow similar to a Finite State Machine (FSM), which was used for the same purpose in the past [74]. These and other reactive architectures are not covered by the subject of this study as they cannot be used for solving complex problems [127].

The survey on general action planning shows that research in this area progresses much slower than the development of pathfinding methods. Work [9] shows that many commercial games stick to well-known and tested architectures such as GOAP or HTN, rather than making an attempt to use state-of-the-art planning heuristics, which were discussed in the previous chapter. One of the reasons is that pathfinding in most games is simply necessary while the illusion of an intelligent NPC can be delivered by a manual effort.

Such an approach is preferred by the game designers since the researchers were unable to fill the gap between academia and practice. Many of the surveyed planning methods rely on a symbolic representation which is seen as inefficient, or they are adapted for simulators that do not expose their source code. Game developers have access to the source code of their game and do not treat it as a black box. They use every piece of designer's knowledge to achieve the best performance. It can be concluded that the research should be aimed at delivering a planning system which is characterised by the efficiency of a domain-specific planner and automation reducing the development time – a practical solution which could compete with planning systems utilizing domain-independent heuristics and rigidly predefined ones as well.

3.4 Processing Game Replays

Originally, replays were invented to reproduce historical matches. In contrast to video recordings, game replays include data that refers to the internal game state. A sequence of in-game actions is usually recorded to a compact format that can be used together with built-in game mechanisms to recreate a match visually. However, it is not complete information, and it is rarely designed for being processed by AI methods. Therefore, researchers often cannot explore their full potential unless they are experimenting with specially designed testbed environments.

Recently, game replays started being considered as a natural source of input data for learning how to play a game [37]. A potential advantage of employing learning is the possibility to build AI system that automatically adapts to a new environment or a new opponent. It can reduce the development time and increase the gaming experience of players. On the other hand, it brings challenges that include providing a training set covering a large space of parameter combinations and testing the system. Quality and reliability standards are set high for a video game as a product; therefore, game developers still prefer manageable solutions characterised by highly predictable performance.

Reinforcement learning has been employed for finding the best policies in FPS and arcade games [76, 77]. In these works, the term of policy has the same interpretation as strategy, which is formally defined in game theory. A strategy can be understood as an algorithm (or a method) of selecting an action for every game state considering that the outcome also depends on other players' actions. The Q-learning algorithm is used for optimising the player's policy and maximising the game score. It is a game theoretic approach that obtains game scores from past games instead of exploring future states of the game tree in a Minimax fashion. The relation between game theory and planning has been outlined in the previous section. The purpose of reinforcement learning is maximising the global game score while classical planning can be used for achieving global or local in-game goals that are not necessarily measurable in terms of score. Although the purpose of these methods differs, they can theoretically coexist in one system, which was explained earlier.

In work [128], game replays were processed to build a real-time Case-Based Reasoning (CBR) planner. D2 (Darmok 2), which is a system developed by the authors, was applied for playing complete matches of StarCraft. It is a popular real-time strategy (RTS) game that evolved into an e-sport. Although the producer of the game has not released the source code, communication with in-game units is possible via network protocol by using BWAPI (Brood War API). D2 framework implements a case-based planning cycle that continuously matches the current game state with one of the previously obtained cases. The database of cases is acquired from human demonstrations that are stored in game replays. Such a demonstration includes a sequence of game states and in-game actions for each time stamp. Each case comprises an initial state, a goal, and a plan that can be executed in the initial state to achieve the goal. States have an object-oriented representation, and plans have the form of Petri nets that can be adjusted to fit the current situation in the game [129]. The approach relies on the ontology of goals provided by a domain expert. The ontology is used to recognise players' subgoals in streams of input demonstrations. The annotation process involves matching observed game states with the subgoals, and it is done automatically based on a set of rigidly predefined rules provided by the expert. Retrieved sequences of actions are plans that are later executed to reach goals specified in the ontology.

Potentially, game replays could be used as an input data for HTN-MAKER to obtain the structure of an HTN automatically [130]. However, there are no traces in the literature whether this approach has been successfully applied to video games. Apart from plan traces, the method would require semantically-annotated tasks and predefined goals. The method assumes that input plans are provided by the domain experts, and it does evaluate their optimality. Therefore, it appears that its benefit is reducing the manual effort of constructing the structure of tasks, but with no guarantees that the model will generate good-quality plans.

3.5 Lessons Learned

Space Partitioning. Space partitioning techniques have been successfully employed by path planning systems to reduce the search space. A space partitioning model stores precomputed information that can be used to increase the precision of a heuristic function and reduce computation effort during runtime. A promising research direction would be testing whether such a model can be successfully built for general action planning by automatically capturing the abstraction of a non-metric state space. Such an approach would accelerate solving planning problems that involve agent cooperation and mix pathfinding tasks together with collecting items or changing the state of a game map.

Domain Representations. The survey shows that domain-independent systems based on a symbolic representation are not suitable for demanding domains such as video games, in which minimising resource consumption is the priority. On the other hand, relying on a game-specific representation reduces the flexibility of a planning method. A significant achievement would be developing an approach characterised by the performance of a domain-specific planner and an automatically obtained heuristic, which is the benefit of using a domain-independent planner. Another important note is that information referring to subgoals may not be directly encoded in the game state representation. Potentially, subgoals could be regarded as regions in a state-space partitioning model, rather than atoms of a state as in the case of IW(i). It would increase the precision of estimating a progression towards the final goal.

Utilising Game Replays. One of the attempts to reuse information obtained from game replays employs a CBR planner. CBR systems follow a cumulative approach, which does not scale well for an increasing number of cases. The method also relies on a predefined ontology of goals, which represents a winning strategy. Such information is usually incomplete, biased by a human factor, and it can quickly become obsolete once a game is rebalanced, which often happens during its development. The performance of the system depends on the quality of game replay annotation. The detected problems motivate researching a search-based planner supported by information obtained from game replays because it utilises generative planning, which is scalable. Rigidly provided information should be limited to the game rules, and a predefined goal structure should be eliminated. The approach should focus on extracting general knowledge that is not sensitive to the quality of plan traces and does not require annotation, which is problematic in most cases.

Although game replays are used by methods based on learning, such an approach cannot be perceived as a starting point for developing a method that addresses the subject of this work. In many cases, game replays play the role of training data that is processed to compensate limited capabilities of simulating future states of the world and using classical forward-chaining planning. It applies to robotic systems, which do not operate in a software simulation, black-box problems, whose complex structure remains concealed, or video games that do not provide full access to the source code. In normal circumstances, game developers integrate an AI method inside the game system to maximise its performance.

Chapter 4

Research Problem

The description in this chapter begins with formulating assumptions referring to classical planning supported by plan traces in video games and formally stating the undertaken research problem. The next part of the chapter summarises the results of early studies, which helped to understand the nature of the problem and localise the main challenges. The initial research resulted in a trivial approach, which is not considered as a version of the proposed method, but conclusions collected during the study provided grounds for developing the core algorithm performing state-space partitioning. The idea of partitioning the state space is discussed subsequently.

4.1 Problem Statement

Before the undertaken problem is formally stated, assumptions referring to utilising plan traces for solving classical planning problems in video games are discussed. The following sections characterise a planning domain in a virtual environment, describe the design of the proposed planning method, formalise input plan traces, formulate the objective of the research, specify how its accomplishment should be evaluated, and identify the main challenges.

4.1.1 Planning Domain

In the proposed approach, a video game is treated as a real-time simulation of a virtual world, which follows a set of game rules. It is assumed that the behaviour of the simulation can be expressed in formal terms of a discrete system. For such a system, all possible states of the virtual world and its permissible transitions are formally defined by the state space as described in Section 2.2.1.

A state s of the world is specified by a tuple $\langle v_1, v_2, \ldots v_n \rangle$ of state variables (Def. 2.3), which refer to game logic data. A transition between two states occurs when an in-game action is executed. Such an action can be invoked by a human-controlled agent or computer-controlled agents. The game rules determine a set of possible actions as well as their cost and effect.

States and transitions are nodes S and edges E in a state-space graph $\mathbb{S}$ respectively (Def. 2.3). Such a graph is implicit and potentially infinite. It is never entirely stored in the memory, and its nodes can be accessed one after another by procedurally computing effects of actions starting from an initial state.

4.1.2 Planning Method

The purpose of a planning method being the subject of this work is finding a path p in $\mathbb{S}$, which represents the solution of a planning problem instance. Such a problem instance is described by a planning task t (Def. 2.4). Such a task specifies the current state s_a of the world, and the goal state s_b, which can be achieved by executing a plan represented by p with cost c (Def. 2.6). Planning tasks are generated by AI-controlled agents to choose the best strategy against the opponent. A strategy is a response to opponent's actions, and it is aimed at maximising the player's score. The strategic layer of AI architecture is responsible for selecting an in-game goal for the controlled agents. It relies on a planning layer to check the feasibility of reaching such a goal, estimate the cost, and obtain a sequence of actions that should be executed. It is assumed that such information can be obtained from the output of the planning method. Plans returned by the method should be optimal (Def. 2.7) to maximise the performance of an AI player.

4.1.3 Plan Traces

The research problem addressed in this work is developing a planning method that utilises plans observed in historical matches. Thus, it is assumed that such a method has access to a set of plan traces, which can be obtained from game replays recorded by internal mechanisms of a video game. Formally, a plan trace is a plan that has the form of a path p in $\mathbb{S}$. The method can operate in two phases. In the offline phase, it can preprocess available input data. Such a process is not constrained by hardware limitations. However, in the online phase, the method should solve planning problems in real-time consuming minimum processor time and memory.

4.1.4 Problem Formulation

The problem can be formally stated as minimising the computational cost of solving a set T of planning tasks by utilising a set P_0 of plan traces provided as input, Eq. 4.1:

$$\min \sum_{t=\langle s_a, s_b\rangle \in T} C\big(\mathbb{S}, P_0, s_a, s_b\big), \tag{4.1}$$

where:

- t is a planning task (Def. 2.4),
- $C(\cdot)$ measures the computational cost of solving a planning task,
- $\mathbb{S}$ is a state-space graph (Def. 2.3),
- P_0 is a set of input plan traces (Def. 2.5).

4.1.5 Efficiency

Minimising the computational cost is equivalent to maximising the efficiency of planning. Assuming constant solution quality, the efficiency of a planning method is affected by its consumption of computational resources. Therefore, function $C(\cdot)$ can measure the following factors:

- the number of unique states visited by a planning algorithm (processor time),
- the maximum size of the priority queue used by a planning algorithm (memory usage),
- the wall-clock execution time (practical performance).

It is worth noting that the first two measures are universal while the third one is domain-specific.

4.1.6 Challenges

The following challenges characterising the proposed approach can be identified:

1. Although the source code of a virtual world is available, the meaning of actions and objects or their importance for particular plans is unknown unless the game designers manually provided the hierarchy of goals, which is not practical for complex environments.
2. Plan traces are just raw sequences of state transitions. They do not hold information about the subgoals nor the actual intention of a human player unless they are annotated. However, annotation should be avoided as it requires a manual effort.
3. In typical environments, the number of possible plans is enormous, and it can be regarded as infinite. Therefore, the method should extract general information from a stream of plan traces instead of accumulating them for future use.

4.2 Early Approach

The initial study was focused on a straightforward case concerning a system with a single agent. It was assumed that a relatively small state space would serve best for exploring early ideas. The outcome of the research is a simple concept of abstracting the state space based on plan traces of an agent. The method relies purely on observation, which means that access to the internal state of the agent is unavailable, and information about the possible states and state transitions in a state-space graph is limited to the observed ones. However, each observed state is fully observable with no uncertainty. This initial concept is not employed by the proposed planning method. Therefore, the following description introduces only the essential points of the approach, rather than providing detailed information, which can be found in work [131].

4.2.1 Modeling the State Space

A general overview on the organisation of a video game employing the discussed approach is pictured in Fig. 4.1. The model includes several modules between which information is passed. The environment module simulates a virtual world, in which planning problems are being solved. The player interacts with the game through interfaces provided by the developers. During play, the game receives player's input as a response to given visual and audial output. The player solves in-game problems and executes his plans by controlling a virtual expert agent. Actions of the controlled agent and system state transitions are tracked and recorded by the observer module, which also holds the repository of plan traces. The traces are collected for a number of simulation runs. The addressed method is located in the reconstructor module. The module builds a network of goals and passes it to the planner module. The planner utilises the received model for planning actions for an AI-controlled agent in the same environment or a similar one.

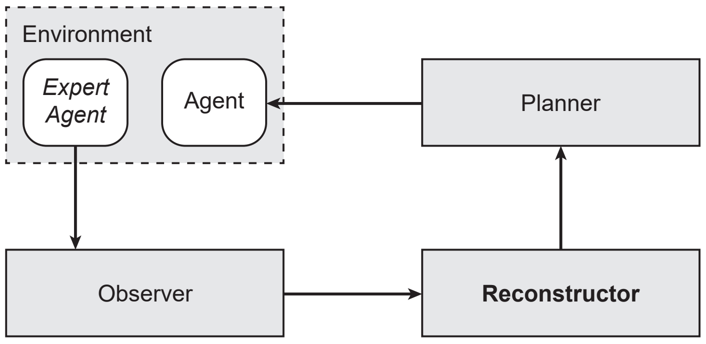

Figure 4.1: The organisation of a video game utilising planning supported by plan traces

The network of goals abstracts the state space, and it has the form of a directed graph. The graph comprises two types of nodes. The first one is a key node that represents a long-term or permanent change applied to the environment. Key nodes are considered as potential goals. The second type is transitive nodes, which are intermediate states between the key nodes. The graph can be used for finding the shortest path to a specified goal. It also provides information regarding the hierarchy of goals – subgoals that must be accomplished in order to reach the final goal.

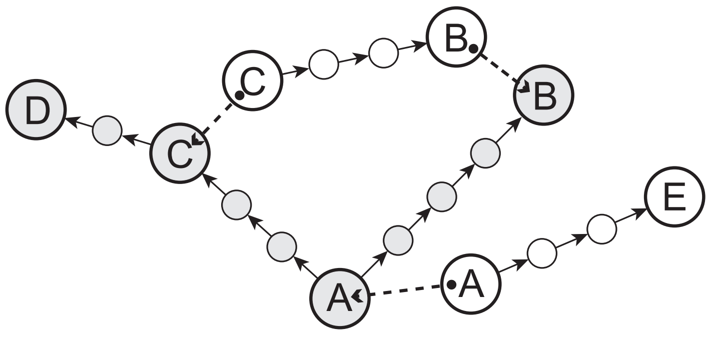

Figure 4.2: Adding new segments to the network of goals

The algorithm of building the model starts with preprocessing plan traces. Each state in a sequence is marked as key or transitive. Then, each sequence is split into many segments in a way that such a segment begins and ends with a key node. Both key nodes are connected with a chain of transitive nodes. In the next step, the segments are assembled into a graph of key nodes. Figure 4.2 visualises the procedure of adding new segments to the graph. In the picture, large circles depict key nodes, and the small ones are transitive nodes. The dashed arrows show attachment points in which equivalent key nodes are matched together. If two segments connect the same pair of key nodes, then the shorter segment replaces the longer one to minimise the distance between the nodes. The order of supplied plan traces does not affect the result. The constructed graph can be expanded incrementally.

4.2.2 Simple Testbed Environment

The properties of the discussed model were verified for a practical case resembling a quest in a video game. Such a quest may comprise a set of partially ordered objectives. For instance, it can be building a ship or a bridge to cross a river. There can be many possible ways to accomplish the quest, but the exact structure of subgoals is unknown. Having that information would reduce planning effort by avoiding the blind search and focusing on relevant subgoals. Therefore, the method attempts to obtain them from plan traces.

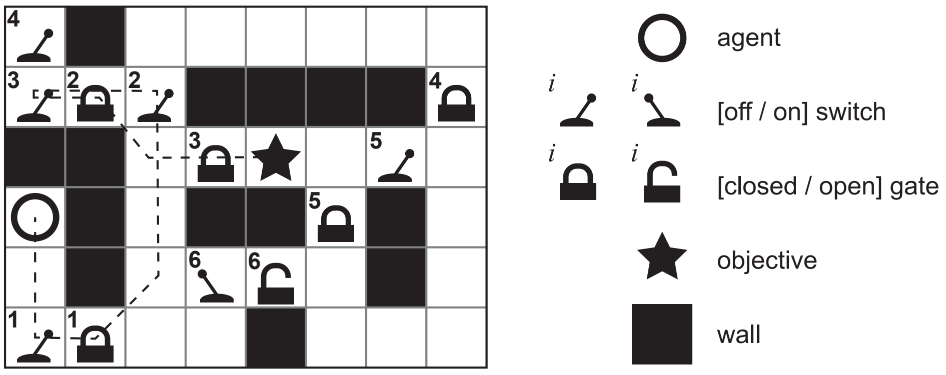

FIGURE 4.3: An example of a game map for a single agent case

The testbed environment was organised as a puzzle game to maintain simplicity and imitate a use case. In the game, the map is represented as a discrete grid. Grid cells are filled with empty space, walls, gates, or switches. Figure 4.3 illustrates an example of a game map. Each switch is connected to a gate, and it can be used by an agent to open the corresponding gate. The goal of an agent is moving towards a target position. Reaching the position may require unlocking a combination of gates in order to unblock the passage. In the illustration, the dashed line indicates the agent's path from his initial position to the objective. The state of the environment is represented as a vector of variables encoding the state of gates (switches) and the location of the agent. A state in which the agent modified one of the objects becomes a key state. Equivalent states were matched based on modified objects.

4.2.3 Experimental Study Results

The experimental study was aimed at examining the properties of the goal network. A set of input plan traces was generated by recording actions of a randomly moving agent. In the experiments, the length of the shortest path to the goal and the size of the graph was measured. The measurements were repeated 10 times and averaged. A chart showing the number of nodes in relation to the size of input data is displayed in Fig. 4.4. It can be observed that the expansion of the model stabilises in time. The decrease of the path length with respect to the number of input plan traces is delineated in Fig. 4.5. It can be concluded that the bigger set of plan traces, the better quality paths can be obtained from the model. Two examples of goal graphs reconstructed for the described game are rendered in Fig. 4.6. The one on the left contains 546 nodes. It was built from 10 input plan traces. The second graph has 2953 nodes, and it used 7000 plan traces. These examples visualise how extensively the structure can grow even for miniature problem cases.

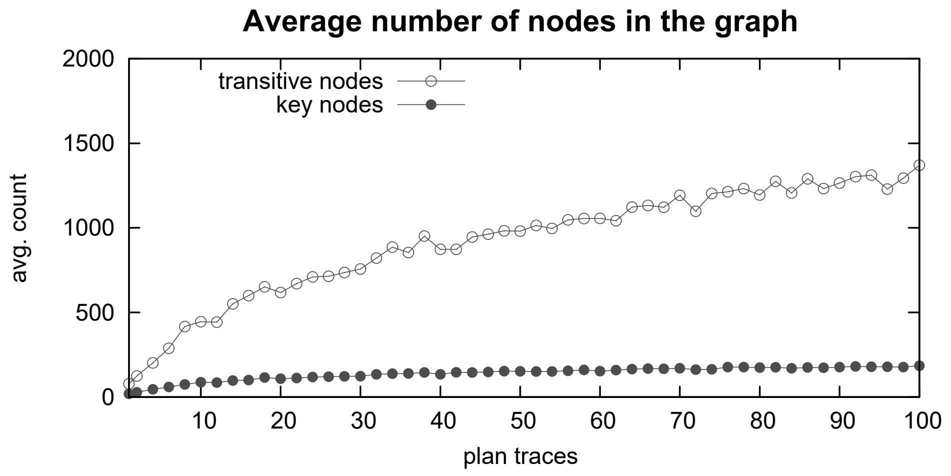

FIGURE 4.4: The number of graph nodes in relation to the number of input plan traces

4.2.4 Lessons Learned

To summarise, the initial study was dedicated to exploring the possibilities of analysing an unannotated set of plan traces and obtaining from it a model of the state space that

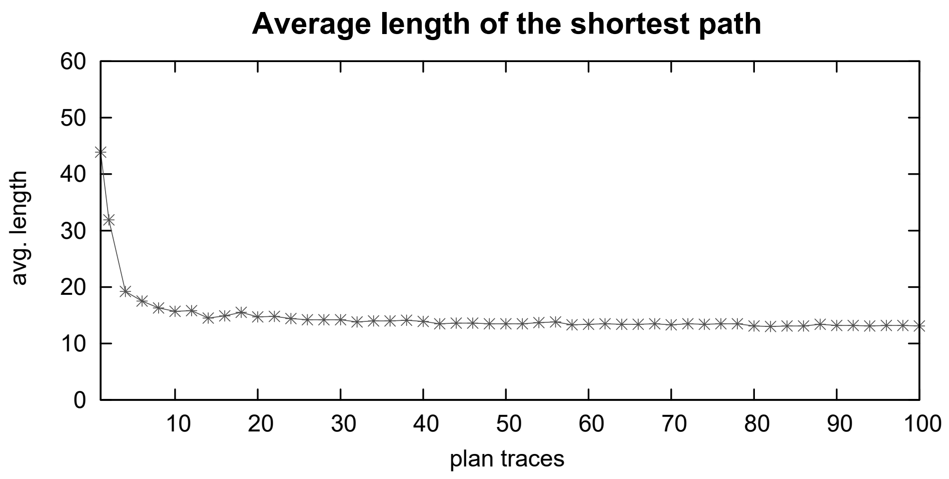

FIGURE 4.5: The length of the shortest path with respect to the number of input plan traces

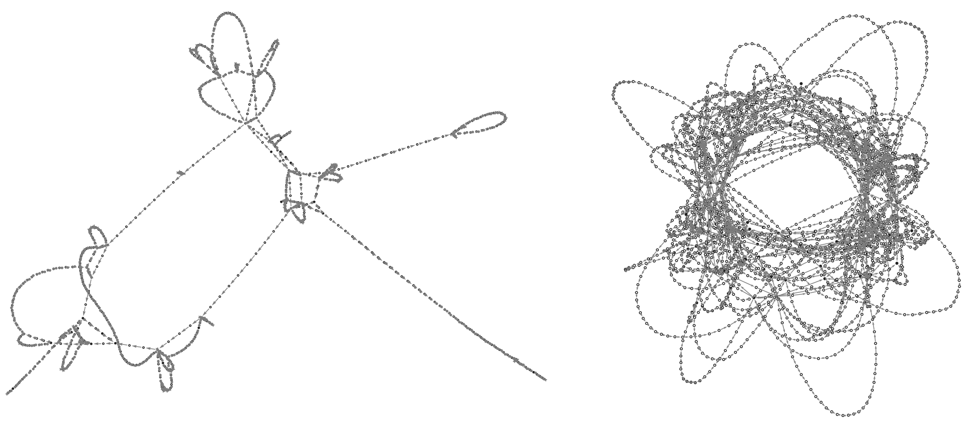

FIGURE 4.6: Examples of goal network graphs built for the game map in Fig. 4.3

could be potentially serviceable for solving complex planning problems. The proposed method automatically reconstructs the structure of plans and goals based on observation of a single agent. The method does not require predefined tasks and goals, and it operates incrementally. It processes input plan traces and merges them into a graph representing an abstraction of the state space. The graph comprises a network of goal nodes that are connected with state transitions. The approach can be considered as a contribution to learning by practice, because it can be used for transferring knowledge from experts like in work [132]. In such an application, a planning system may be supported by suggesting actions that should be executed to reach a selected node in the network of goals. Alternatively, the model may be employed for predicting user's goals based on his past actions.

Although the presented method is characterised by a minimal manual effort, it has a limited range of application in practice. The discussed model does not generalise the state space, which means it cannot be effectively used for solving planning problem instances that were not covered by input plan traces. A large number of distinct states in the input data can potentially lead to an overgrowth of the graph. The model suffers from the lack of state grouping. The representation of states must be organised differently to avoid performance problems in typical environments, which are characterised by vast state-space graphs. Based on the preliminary study results, it can be concluded that the research should be aimed at partitioning a state-space graph. A partitioning model can generalise the structure of the state space by grouping states. However, building such an abstraction from plan traces for planning is a nontrivial problem, and it is the fundamental challenge in this work.

4.3 State-space Partitioning

State-space partitioning refers to dividing a state-space graph into regions. The procedure is applied to reduce the search space. A state-search routine can ignore certain regions of the state space if it is possible to determine whether the regions contain the goal state.

Different types of partitioning can be distinguished (Fig. 4.7). The simplest one divides the state space into a number of non-overlapping regions, and it has a flat structure. In a hierarchical partitioning, regions of the state space are subdivided recursively. Such a structure is described by a tree. Finally, the structure of overlapping regions can be expressed by a concept lattice, which is the idea introduced in this work.

Partitioning of a geometric space is relatively easy. Regions can be contoured manually or calculated by employing linear algebra. The same approach cannot be directly applied to a non-metric state space graph. In such a graph the exact distance between nonconsecutive nodes is usually unknown. The centroid or bounds of a region cannot be determined unless the nodes are iterated. However, it would be impractical because a typical state space is vast and implicit. For virtual simulations, states are expanded by executing actions, and the graph is never explicitly stored in the memory. Thus, it cannot be preprocessed to circumscribe regions and calculate distances between them assuming current hardware capabilities. The lack of a region distance function makes standard hierarchical clustering methods inapplicable [133].

The following chapters address this problem by proposing implicit models of the hierarchical structure of regions in an abstract state-space graph. Such a model is extracted from

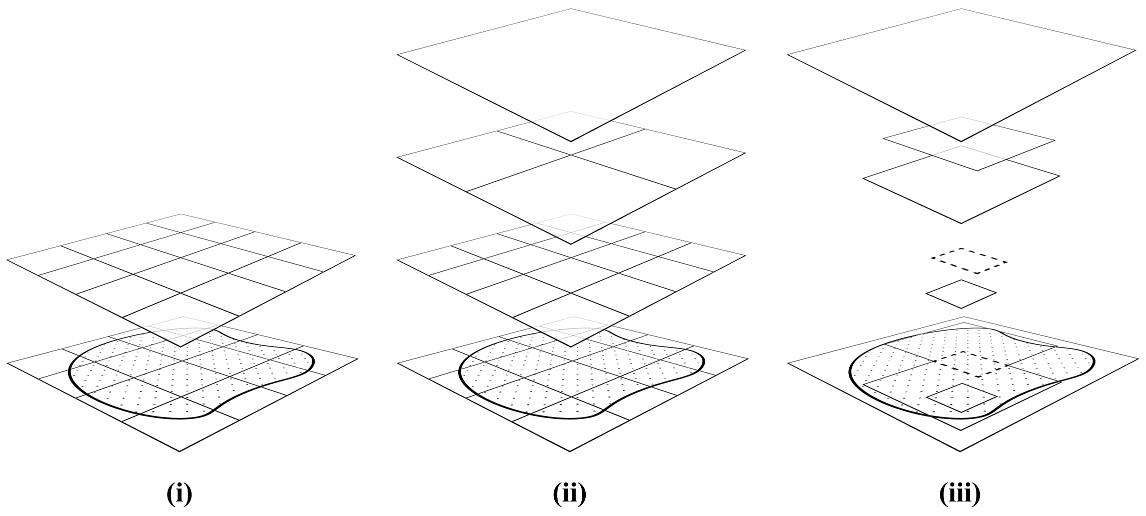

FIGURE 4.7: Types of state-space partitioning: flat (i), hierarchical with non-overlapping regions (ii), hierarchical with overlapping regions (iii)

an input set of plan traces, and it a part of the proposed heuristic estimator employed by search-based planning methods.

Chapter 5

State-space Tree Search Heuristic

In the previous chapter, the main challenges were diagnosed by discussing the results of the early studies. The next part of the research continues the thread of extracting information from plan traces, but it is aimed at developing a new model of the state space that could be effectively utilised for accelerating a planning process by reducing the search space. In the area of classical planning, it can be achieved by estimating the distance to the goal state and directing the search towards promising parts of the state space.

This chapter presents a method introduced in work [134]. The method is a part of the evolution of the proposed approach. The following description begins with the notion of state descriptors representing implicit regions of the state space. Next, the model of a region tree built from plan traces is presented. It is used for partitioning the state space. Subsequently, a new heuristic estimator employing the model is defined. The heuristic was tested in an author's testbed environment designed as a video game.

5.1 Implicit Regions

Lessons learned from the previous part of the studies led to a conclusion that storing individual states in a computer memory would be impractical considering that a typical state space is vast. A better option is focusing on regions of the state space. Such regions can be organised into a hierarchical structure to subdivide the state space and avoid exploring parts of it that do not contain the goal state.

Determining regions and relations between them in an abstract and non-metric state space without using a physical representation of sets of states is a challenging task. In such a state space, the distance between states or regions cannot be procedurally calculated as the distance estimate is unknown by definition. Iterating over nodes in a state-space graph and managing explicit sets of nodes would be impractical. Therefore, state nodes were implicitly grouped into regions using the introduced notion of *state descriptors*. A state descriptor is a predicate that refers to selected features of a state (Def. 5.1). The method of defining regions of the state space may resemble Chu spaces, which generalise the notion of topological space, although the introduced formalism was not inspired by them [135].

Definition 5.1 (State Descriptor). For a set $S_i \subseteq S$, let $d_i \in D$ be a state descriptor defined as a predicate (Boolean function) $d_i : S \rightarrow \{true, false\}$ that determines membership of a state in the set S_i, Eq. 5.1:

$$S_i = \{s \in S : d_i(s)\}. \tag{5.1}$$

The method assumes that the system designer implements and provides a set of state descriptors based on game rules. For instance, a descriptor can return *true* if a particular condition in a state is satisfied. Such a condition may be required for fulfilling one of the possible goals in the game. There are no special requirements for state descriptors, and they can be defined arbitrarily.

5.2 Region Tree

The second attempt to propose a model of the state space resulted in constructing a tree of regions, which are determined by state descriptors. It is a hierarchical model that can be used for partitioning the state space. The model is organised as a structure in which regions hierarchically encompass child regions. Regions can intersect because it would be difficult and impractical to ensure their disjunction for a non-metric state space. Each node in the tree is represented as a set $D_j \subseteq D$ of state descriptors. Such a node implicitly contains a set $S_j \subseteq S$ of states for which each state descriptor $d \in D_j$ is true, Eq. 5.2:

$$S_j = \{s \in S : \forall_{d \in D_j} d(s)\}. \tag{5.2}$$

A set containing two or more descriptors is intuitively interpreted as the intersection of regions covered by each descriptor in the set. If a state satisfies many descriptors, then it falls into a region that is the intersection of regions corresponding to the satisfied descriptors.

Definition 5.2 (State-space Region Tree)**.** Let a pair $h_i = \langle D_i, H_i \rangle$ be a node of a state-space region tree, where:

- D_i is a set of descriptors defining the region of i-th tree node,
- $H_i = \{h_1, h_2, \dots h_n\}$ is a set of child nodes of i-th tree node.

Definition 5.3 (Tree Region Relation)**.** For two tree nodes h_i and h_j, a parent-child relation ($>$) is defined by Eq. 5.3:

$$h_i > h_j \iff D_i \subset D_j. \tag{5.3}$$

Thus, h_i is a parent of h_j and h_j is a child of h_i.

The algorithm of constructing the model is expressed in pseudocode by Alg. 4. It begins by obtaining a list of regions from an input set of plan traces. The routine iterates over each observed state and determines which descriptor sets such a state satisfies (line 2). Each descriptor set represents a region (line 3). Intersections of the regions are added to the list of regions (line 5). In the next step, parent-child relation between the collected regions is determined (line 8). The procedure starts from regions that have the biggest number of children (line 9). A parent region contains a child region if the set of state descriptors defining the parent region is a subset of the one corresponding to the child region (line 11). It should be noted that the more descriptors a region has, the smaller it is. An empty set of descriptors covers the entire state space. Finally, the regions are assembled into a tree (line 12). New regions are recursively attached to the tree by ensuring that they are added once. It is because a region being a product of intersection has more than one possible parent, and potentially, it can be added many times. The outcome is a structure in which any new state, which was not observed previously, can be easily located in the tree.

5.3 Planning Heuristic

The proposed heuristic uses the previously introduced model of a region tree to estimate the distance between states. The method begins by locating the smallest regions in the tree that contain an initial (current) state and the goal state. Then, it calculates the number of parents that the two located regions share – the bigger number, the closer the states are. The number returned by the routine is abstract and does not measure the

Alg. 4: BuildTree(S, D)

```
1  var $DS \leftarrow \{\emptyset\}$                                  ▷ descriptor sets
2  foreach $s \in S$ do
3      $D_s \leftarrow \{d \in D : d(s)\}$                           ▷ a new descriptor set
4      foreach $D_i \in DS$ do
5          $DS \leftarrow DS \cup \{D_i \cap D_s\}$                 ▷ add intersections
6      $DS \leftarrow DS \cup \{D_s\}$
7  var $H \leftarrow \emptyset$                                       ▷ a set of tree nodes
8  while $DS \neq \emptyset$ do
9      $D_s \leftarrow$ the smallest set in $DS$
10     $DS \leftarrow DS \backslash D_s$
11     $H \leftarrow H \cup \langle D_s, \{D_i \in DS : D_s \subset D_i\}\rangle$   ▷ add a region and its child regions
12 return $H$ as a tree
```

cost. It can be intuitively interpreted as the number of regions that have been crossed on the way to the region containing the goal. The number must be transformed to make the heuristic admissible and ensure optimality when used by A* algorithm. First, the number must be inverted, so it decreases when approaching the goal. Next, it should be scaled using the minimal cost of action in a planning problem domain to avoid overestimation. The method assumes that crossing a region requires at least one action. The formula of calculating the heuristic is presented in Def. 5.4:

Definition 5.4 (Tree Region Distance)**.** Let a function $\Delta_H : S \times S \to \mathbb{R}_{\geq 0}$ be a heuristic estimator of the distance between regions in a region tree, Eq 5.4:

$$\Delta_H(s_a, s_b) = c_0 \cdot \Big(\big|H(s_b)\big| - \big|H(s_a) \cap H(s_b)\big|\Big), \tag{5.4}$$

where:

- s_a is an initial state, and s_b is the goal state,
- $H(\cdot)$ is a set of all tree nodes (regions) containing a specified state (Eq. 5.5),
- c_0 is the minimal cost of action,
- $|\cdot|$ is the cardinality of a set.

A set of all tree nodes containing a state s can be obtained using the following formula, Eq. 5.5:

$$H(s) = \{h_i = \langle D_i, H_i\rangle : \forall_{d \in D_i} d(s)\}. \tag{5.5}$$

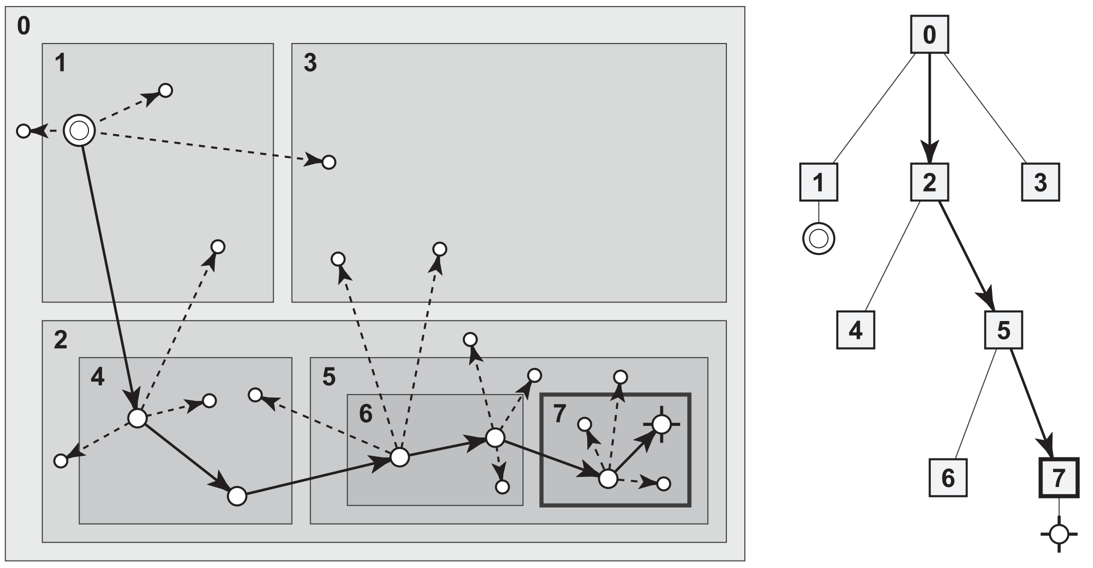

FIGURE 5.1: A visualization of a state search guided by the tree region distance heuristic in a state-space graph partitioned by a region tree

Figure 5.1 visualises a state search guided by the heuristic in a partitioned state space. The initial state is placed in the upper left corner of the picture (1). The goal state is located in the lower right corner (7). The search procedure expands neighbour states in a state-space graph. In the beginning, both states share one parent region, which covers the entire space (0). The graph is explored uniformly until the search enters the rectangle in the lower left part of the picture (2). States in the rectangle share two parent regions with the goal state (0 and 2). It is apparent that the goal state should be expected in the smallest existing region which contains both the current state and the goal state (2). Therefore, entering states outside the region should be avoided if possible. The search continues by narrowing the area in which the goal can be expected (5). In the final phase, the smallest region containing the goal is explored until the state is found (7).

i	$\|D_i\|$	$\Delta_H(s_a, s_b), \quad s_a \in S_i, s_b \in S_7$
0	0	$c_0 \cdot (4-1) = c_0 \cdot 3$
1	1	$c_0 \cdot (4-1) = c_0 \cdot 3$
2	1	$c_0 \cdot (4-2) = c_0 \cdot 2$
3	1	$c_0 \cdot (4-1) = c_0 \cdot 3$
4	2	$c_0 \cdot (4-2) = c_0 \cdot 2$
5	2	$c_0 \cdot (4-3) = c_0 \cdot 1$
6	3	$c_0 \cdot (4-3) = c_0 \cdot 1$
7	3	$c_0 \cdot (4-4) = 0$

TABLE 5.1: Region distances calculated for an example of a region tree in Fig. 5.1

Table 5.1 contains distances calculated regions in Fig. 5.1. Tree nodes (regions) are numbered, and each of them corresponds to a row in the table. The second column shows a minimum number of state descriptors that should be assigned to each node to build such a tree. The last column presents distances calculated for pairs of states, where an initial state s_a is in a set S_i of states covered by node i, and the goal state s_b is placed in node 7.

5.4 Experimental Study

The goal of the experimental study was validating whether the early concept of the proposed heuristic can be successfully applied for reducing the search space for complex planning problems, for which providing a heuristic estimate is nontrivial. Such planning problems may be characterised by implicit subgoals and tasks that require cooperating agents. An example of such a task may be building a bridge that requires at least two agents working together. In the discussed scenario, an implicit subgoal may refer to acquiring tools and resources that are needed for constructing the bridge. Also, the goal of the agents may be defined as crossing the river, which does not imply whether they should use a bridge or a ship. Thus, a testbed environment was designed as a puzzle game to capture a class of planning problems similar to the described one. Input plan traces were obtained from human players in the form of game replays to emulate applying the studied method in practice.

5.4.1 Testbed Environment

The experiments were conducted in a game named Smart Blocks. It was developed as a testbed environment for studying methods that utilise game replays. The organisation of the environment is similar to the one discussed in the previous study. The game mechanics may resemble Sokoban. However, the game introduces additional elements to make planning problems more challenging.

The system imitates a case of a Multi-Agent System (MAS) in which heterogeneous agents must cooperate to solve a problem. The player manipulates the agents in the same manner as a centralised planning method. Whenever a player completes a map in the game, the game replay is submitted to a server. A game replay comprises a sequence of the player's actions, which are used to reproduce simulation steps. The actions are atomic, and the simulation is deterministic.

Game Rules

The goal of the player is to move one of the controlled blocks to the objective. Blocks of different shapes and sizes represent the agents. Their shape and size determine which blocks can stand together in one cell. A block can also change its colour by entering a colour portal. In terms of MAS, it may be interpreted as acquiring a resource by an agent. The path to the objective is blocked by a series of gates, which are unlocked by corresponding switches. A switch is activated if blocks standing on it match a pattern. The pattern is described by the shapes and colours of space occupied by blocks standing together. The last elements playing an important role in the game are energy cells and ground obstacles. Each block move consumes energy. The consumption is greater if a block enters a ground obstacle. The energy supply for the entire team can be slightly replenished by taking an energy cell. The player must reach the objective by fitting in an available reserve of energy. The player's score depends on the final energy reserve.

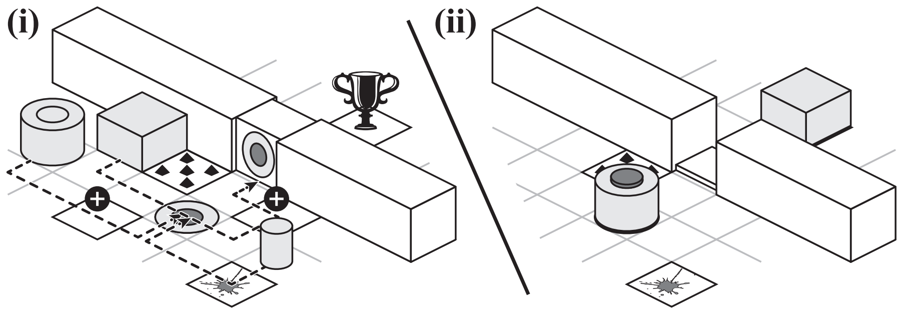

Figure 5.2: An example of a planning problem solved in Smart Blocks

An example of a planning problem that involves agent cooperation is illustrated in Fig. 5.2. It shows a part of a stage that contains three player-controlled agents: a ring, a box, and a small cylinder. Their objective is to reach the artefact located behind the wall (it is marked with a cup). Initially, the path is blocked by the closed gate (i). It can be opened by using the switch. First, the box agent approaches the gate avoiding the ground obstacle and collecting one of the energy cells. Next, the small cylinder goes to the trigger through the colour portal where it changes its colour to the one that is accepted by the trigger. Then, the ring collects the last energy cell, and it ends its move in the same place. As soon as the pattern of the trigger is satisfied, the gate is unlocked. Now, the box agent is free to reach the goal (ii).

The described example demonstrates a scheme of tasks that are present in the game. The game maps were manually designed by joining such subtasks in chains to build challenging planning problems. The maps are characterised by combinatorial problems, for which a heuristic is nontrivial. The initial maps introduce the player to the game rules. In general, the difficulty of subsequent maps increase, but not in every case. The human perception of difficulty and the complexity of computing a solution are not equal. The experiments were conducted for selected game maps as some of them turned out to require an excessive amount of computational resources.

State Descriptors

The set of state descriptors was manually specified based on the designer's knowledge of the game rules. The descriptors identify regions of the state space that may be relevant for solving planning problems in the game environment. The regions group states based on features of the game state, which describe agents and other game objects. A list of predicates corresponding to the state descriptors was generated using templates in Tab. 5.2.

Descriptor Template	**Description**
agent {id} in room {nr}	checks whether a specified agent is inside a defined room
agent {id} on colour portal {nr}	true if a specified agent stands on a defined colour portal
agent {id} has {R|G|B} component	checks whether a colour of a specified agent contains one of the base colours
agent {id} on trigger {nr}	true if a specified agent stands on a defined trigger
agents {id}, {id} ...{id} stand together	valid while a specified list of agents stays on the same field
trigger {nr} is valid	checks whether a pattern of a specified trigger is satisfied
gate {nr} is open	true if a specified gate is open
gate {nr} is held	checks whether one or more agents is standing on a specified gate
goal reached	true if one of the agents reached the golden artefact; groups the goal states

Table 5.2: A list of descriptor templates in Smart Blocks

Plan Traces

Statistics describing the database of plan traces are located in Fig. 5.3. They include the total number of plan traces collected for each game map, the number of actions (steps) that a player made to reach the artefact, and the energy reserve at the moment of solving the map. The information can be used to judge the complexity of the game maps although the difficulty of solving them may be different for a human player and a planning system. Some players were able to find the optimal plans.

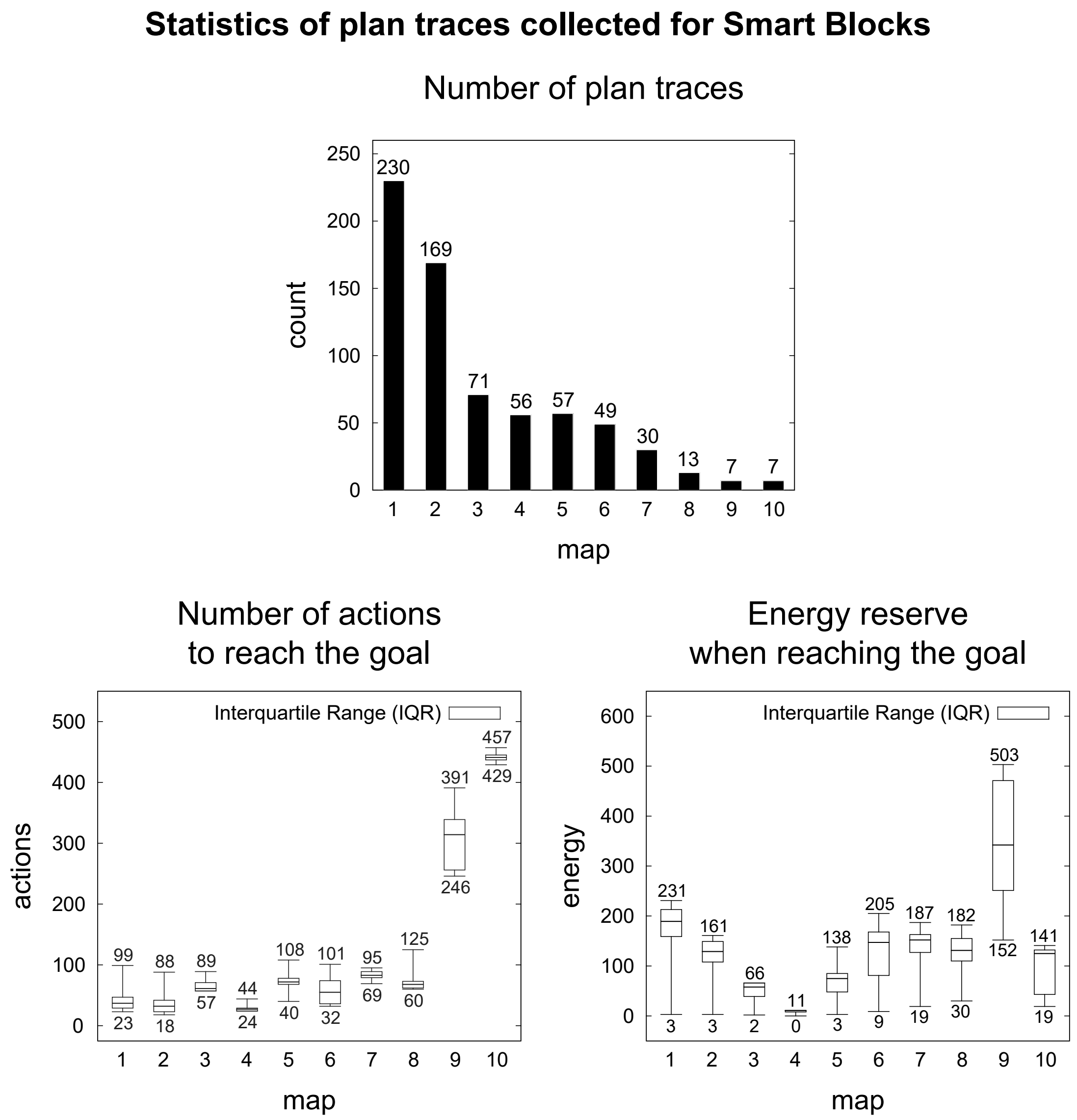

FIGURE 5.3: Statistics describing the database of plan traces collected for Smart Blocks

5.4.2 Experiments

In the first part of the experimental study, statistics of the region tree were studied to ensure that the size of the tree does not imply performance issues. The experiments were conducted for each game map. However, they led to similar conclusions. Therefore, the description focuses on a representative game map, which is characterised by a moderate complexity.

Experiment 5.1. Examining the complexity of the region tree in relation to the number of plan traces.

Parameters: The experiment was conducted for game map 5, for which 57 plan traces were collected from the players (Fig. 5.3).

Course: Region trees were constructed for different percentages of plan traces ranging from 0% to 100% with a step of 10%. The measurements include:

- the number of nodes (regions) in a region tree,
- the number of distinct states obtained from plan traces,
- the depth of a region tree,
- state nonuniformity, which indicates disproportion of the distribution of states in a region tree. It was calculated as a standard deviation of the number of states in each region.

The procedure was repeated 10 times, and the measurements were averaged.

Results: The statistics are presented in Fig. 5.4. The results show that the growth of the region tree slows down as the number of plan traces increases. It can be concluded that only a small number of input plan traces may be sufficient for constructing most of the tree and providing additional data has a minor effect on its structure.

Experiment 5.2. Studying the relation between the number of plan traces and the performance of A* guided by the heuristic (Def. 5.4).

Parameters: The experiment was conducted for game map 5, for which the shortest solution plan has 40 actions, and the optimal energy reserve is 138 (Fig. 5.3).

Course: The heuristic was tested for region trees that were constructed for different percentages of plan traces ranging from 0% to 100% with a step of 10%. The measurements describing the performance of A* guided by the heuristic include:

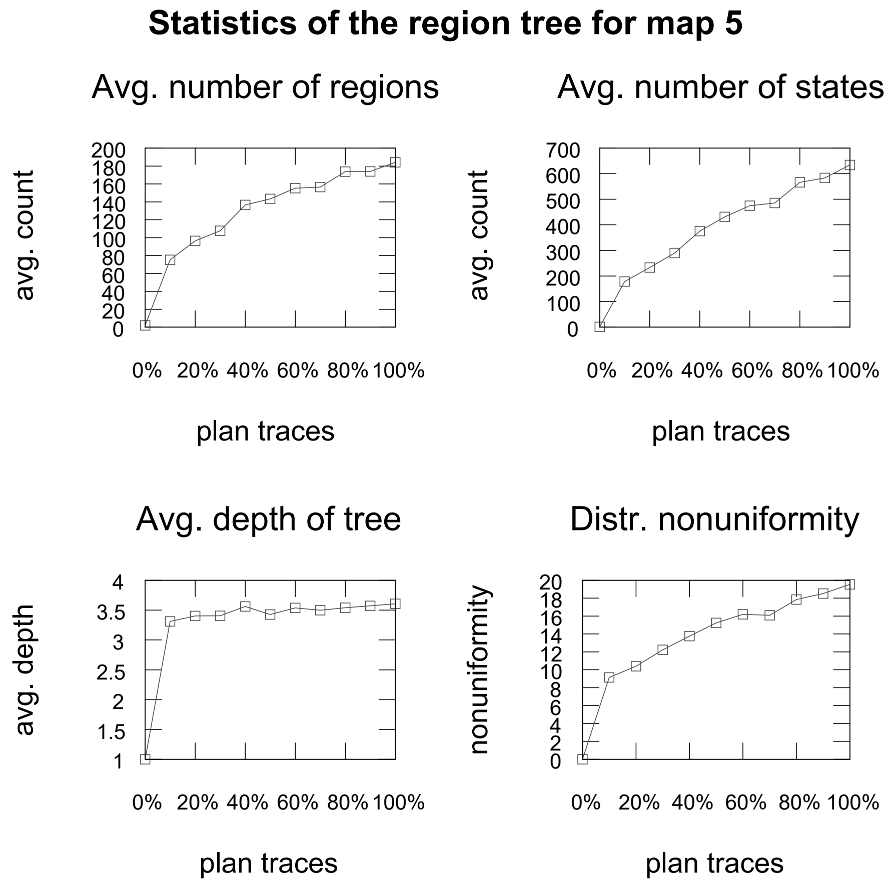

Figure 5.4: Statistics describing constructed region trees in relation to the number of input plan traces used [game map: 5, total plan traces: 57, repetitions: 10]

- the number of iterations in the main loop of the algorithm,
- the number of distinct states visited during the search,
- the maximum size of the priority queue, which indicates a theoretical memory consumption,
- the energy reserve at the moment of reaching the goal state.

The procedure was repeated 10 times, and the measurements were averaged.

Results: The statistics are presented in Fig. 5.5. It can be observed that only a small number of plan traces is sufficient for achieving a notable performance improvement. The quality and the number of plan traces does not affect the quality of solutions returned by A*. However, the more plan traces are provided, the fewer states are visited by the algorithm. The reduction of the number of visited states correlates with the decrease of the size of the priority queue.

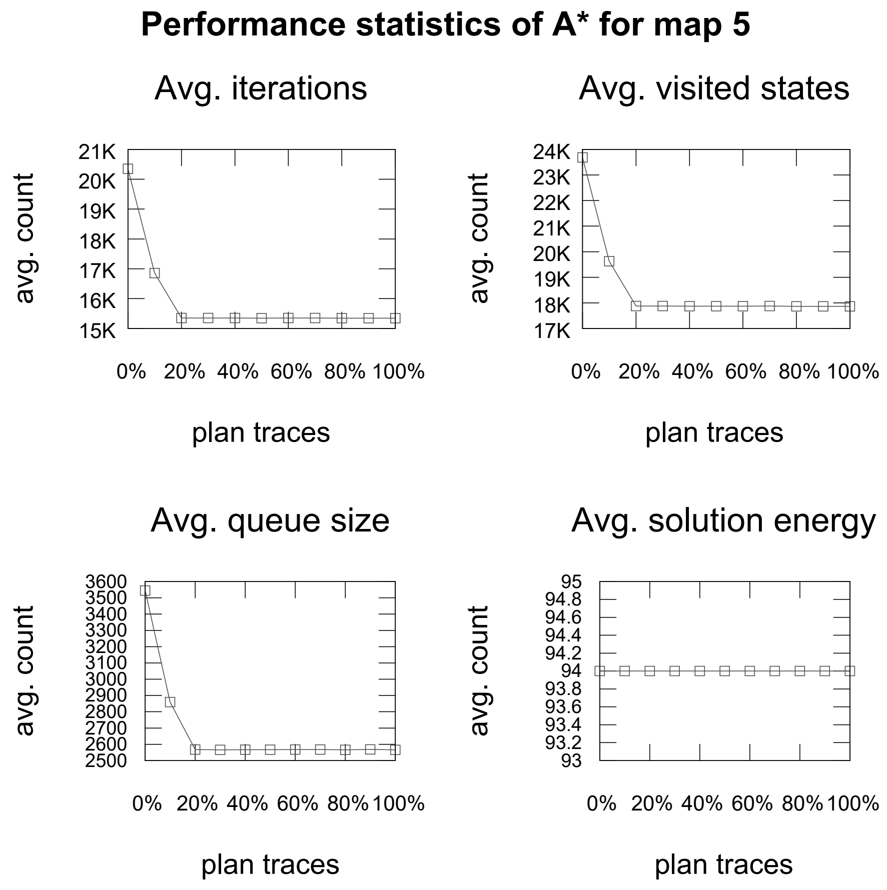

FIGURE 5.5: Performance statistics of A* employing the tree region distance heuristic affected by the number of input plan traces used for constructing a region tree [game map: 5, total plan traces: 57, repetitions: 10]

Experiment 5.3. Comparing heuristic search and uninformed search.

Parameters: In the experiment, A* guided by the proposed heuristic and Dijkstra's algorithm were compared. The tests were conducted for the first six game maps. Unfortunately, performing uninformed state search in the subsequent maps exceeded available computational resources and could not be completed in an acceptable time. However, the encountered problem should not affect the conclusions as these maps are analogous to the previous ones but differ by size. For each of the tested game maps, the heuristic used a region tree built from a total number of plan traces collected (Fig. 5.3).

Course: For each tested game map, the performance of both algorithms was measured using the same parameters as in the previous experiment (Exp. 5.2). The wall-clock execution times were averaged from 10 runs.

Map	**Iterations**		**Visited states**		**Max. queue size**		**Avg. time [ms]**		**Solution length**	**Solution energy**
	Dijkstra	*A**	*Dijkstra*	*A**	*Dijkstra*	*A**	*Dijkstra*	*A**		
1	164	153 **(-6.71%)**	180	163 **(-9.44%)**	13	13 (+0.00%)	0.6	1.3 (+116.67%)	23	231
2	5 271	4 538 **(-13.91%)**	5 800	5 062 **(-12.72%)**	593	593 (+0.00%)	71.3	253.1 (+254.98%)	18	161
3	825 193	747 690 **(-9.39%)**	885 508	806 562 **(-8.92%)**	58 866	58 874 (+0.01%)	12 616.5	13 134.5 (+4.10%)	57	66
4	1 098	440 **(-59.93%)**	1 608	930 **(-42.16%)**	493	493 (+0.00%)	9.3	6.9 **(-25.81%)**	27	11
5	32 609	15 341 **(-52.95%)**	38 025	17 865 **(-53.02%)**	35 45	2 571 **(-27.48%)**	491.7	976.6 (+98.62%)	42	94
6	19 973	12 174 **(-39.05%)**	29 549	19 539 **(-33.88%)**	7 423	7 367 **(-0.75%)**	366.2	993.1 (+171.19%)	53	190

TABLE 5.3: The comparison of Dijkstra and A* employing the introduced tree region distance heuristic

Results: Table 5.3 presents the comparison results. Apart from the performance measurements, the table includes percentage differences to facilitate comparing the algorithms. The last two columns refer to the number of steps in a plan and the energy reserve. For both algorithms, these values are identical, which confirms that the algorithms return solutions of equal quality. The results demonstrate that the heuristic can be effectively used to reduce the search space. However, the wall-clock execution times show that the method runs slower in 5 of 6 cases. This is caused by the fact that visiting a large number of states in Smart Blocks is computationally cheaper than calculating the heuristic for a reduced number of states, which may not be the case for typical game environments.

It should be mentioned that the performance achieved by Dijkstra's algorithm represents the worst-case scenario in which the heuristic function is unavailable or ineffective. It may be an adequate choice for solving planning problems, for which neither the distance to the goal nor the progress can be determined. The described game may be considered as a challenging case because the goal of agents is moving to the target position, and it does not specify any intermediate tasks or subgoals. In a symbolic representation, the goal would be described by only one fact (e.g., `Have(artefact)`). The agents have to interact with each other and the problem cannot be broken down into subproblems for individual agents [136]. The order of gates that must be opened to reach the objective is not exposed to the planning method. It is a realistic assumption because the gate order is a part of the combinatorial problem to be solved.

5.4.3 Lessons Learned

The results demonstrate a visible reduction of the search space delivered by the proposed heuristic for the considered cases. However, the outcome is still unsatisfactory. Although the number of visited states is lowered, the decrease of computation time is not guaranteed. The advantage can be expected for vast state spaces in which the cost of visiting additional states exceeds the cost of computing the distance estimate. The heuristic iterates over the tree and performs multiple intersection operations on descriptor sets, which is costly. The algorithm of calculating the heuristic should be improved.

Another conclusion is that state descriptors should be generated automatically rather than being rigidly specified. The performance of the proposed method relies on them heavily. Automatically optimising them would also improve the flexibility of the approach.

The proposed concept of a region tree demonstrates the potential of partitioning the state space using implicit regions, but it is not an adequate model for representing regions which intersect. In a tree, a node can have only one parent while a region being the product of intersection has many parents. This problem can be solved by employing a concept (Galois) lattice. For this reason, the following chapter abandons the idea of a region tree and addresses the identified issues by developing a lattice-based heuristic.

Chapter 6

State-space Lattice Search Heuristic

The previous part of the research resulted in a promising idea although the described method was characterised by unsatisfactory performance and required several improvements. The concept of state descriptors and a heuristic estimator that relies on the region hierarchy turned out to be valid. However, the studies led to a conclusion that a tree is not an adequate model for representing regions of the state space. The fundamental improvement was replacing a region tree with a region lattice. The idea of the new model is very similar to the previous one, but it allows for estimating the distance between states more accurately.

This chapter presents the final form of the proposed approach. The following description formalises a region lattice and explains the method of constructing it. Subsequently, a new heuristic estimator employing the improved model, and two new state search algorithms, which rely on the properties of the heuristic, are introduced. The complexity and the optimality of the estimator are shown formally. The experimental study demonstrates the characteristics of the method. In the final part of the study, an automatic procedure using a Genetic Algorithm for tuning the structure of a region lattice is examined.

6.1 Region Lattice

The state-space partitioning model introduced in this chapter is constructed from a set of implicit regions, which overlap. A tree cannot accurately express such a structure because a region being the intersection of many regions has many parents while a tree allows for only one. Consequently, a lattice was employed for modelling hierarchical relations between the regions and their intersecting parts. Originally, a concept lattice is a structure that was used for expressing the hierarchy of concepts in formal context [47]. In this work, the lattice is applied to modelling a hierarchical structure of the state space. The following description formulates a state-space lattice by employing terms existing in the domain of state-space search.

6.1.1 Formalisation

In FCA, formal context comprises a set of objects, a set of attributes, and the binary relation of incidence that determines which attributes are possessed by which objects. In reference to the proposed method of state-space partitioning, states are the objects, regions are the attributes, and the incidence relation associates states to regions. It is worth noting that the notion of objects and attributes may be interchanged as the incidence relation remains valid. The regions of the state space are denoted by state descriptors (Def. 5.1).

Definition 6.1 (Formal Context of State Space)**.** Let formal context of a state-space graph $\mathbb{S}$ be a triple $K^{\mathbb{S}} = \langle S, D, I\rangle$, where S is a set of states (objects), D is a set of state descriptors (attributes), and $I \subseteq S \times D$ is the binary relation of incidence, which is regarded as a bipartite graph associating states to regions defined by state descriptors.

For clarity, corresponding Galois operators are provided in Def. 6.2:

Definition 6.2 (Galois Operators for State Space)**.** For a set $S_i \subseteq S$ of states and a set $D_j \subseteq D$ of state descriptor, Galois operators ($'$) are defined as follows (Eq. 6.1 and Eq. 6.2):

$$S_i' = \{d \in D : \forall_{s \in S_i} d(s)\}, \tag{6.1}$$

$$D_j' = \{s \in S : \forall_{d \in D_j} d(s)\}, \tag{6.2}$$

where $d(\cdot)$ is a state descriptor (Def. 5.1).

Definition 6.3 (Formal Concept as Region)**.** A pair $\langle S_i, D_j\rangle$ is a region (formal concept) of a context $K^{\mathbb{S}}$ if it satisfies Eq. 6.3:

$$(S_i' = D_j) \wedge (S_i = D_j'), \tag{6.3}$$

where the sets $S_i \subseteq S$ and $D_j \subseteq D$ are called the extent and the intent of a region (formal concept) respectively.

The hierarchy of regions is defined by the subregion-superregion relation, Def. 6.4:

Definition 6.4 (Lattice Region Relation)**.** For two lattice regions $\langle S_i, D_i\rangle$ and $\langle S_j, D_j\rangle$, a partial order operator ($\leq$) is defined by Eq. 6.4:

$$\langle S_i, D_i\rangle \leq \langle S_j, D_j\rangle \iff (S_i \subseteq S_j) \wedge (D_j \subseteq D_i). \tag{6.4}$$

Thus, $\langle S_i, D_i\rangle$ is a subregion of $\langle S_j, D_j\rangle$ and $\langle S_j, D_j\rangle$ is a superregion of $\langle S_i, D_i\rangle$. The relation is transitive.

Definition 6.5 (State-space Region Lattice)**.** The ordered set of all regions of a context $K^{\mathbb{S}}$ is a concept lattice $L(K^{\mathbb{S}})$.

The basic theorem on concept lattices applies to a state-space lattice (Def. 2.13). Supremum of a state-space lattice covers the entire state space.

6.1.2 Lattice Construction

Components of a context $K_i^{\mathbb{S}} = \langle S_i, D_i, I_i\rangle$ are obtained from plan traces. Let $S_i \subseteq S$ be a set of states observed in a set of plan traces (Def. 2.5). Then, a set D_i of state descriptors can be obtained from Eq. 6.5:

$$D_i = \{d \in D : \exists_{s \in S_i} d(s)\}, \tag{6.5}$$

and a set I_i of incidences is defined by Eq. 6.6:

$$I_i = \{\langle s \in S_i, d \in D_i\rangle : d(s)\}. \tag{6.6}$$

An example of incidences between observed states and state descriptors is presented in Table 6.1.

	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}
s_0				×		×		×	×	×	
s_1		×		×		×		×		×	
s_2	×		×		×		×		×		
s_3		×	×		×		×				
s_4	×			×					×		
s_5											

TABLE 6.1: An example of a table of incidences

Since the properties of a state-space lattice remain unchanged in respect of the original definition of a concept lattice, algorithms for generating formal concepts and constructing

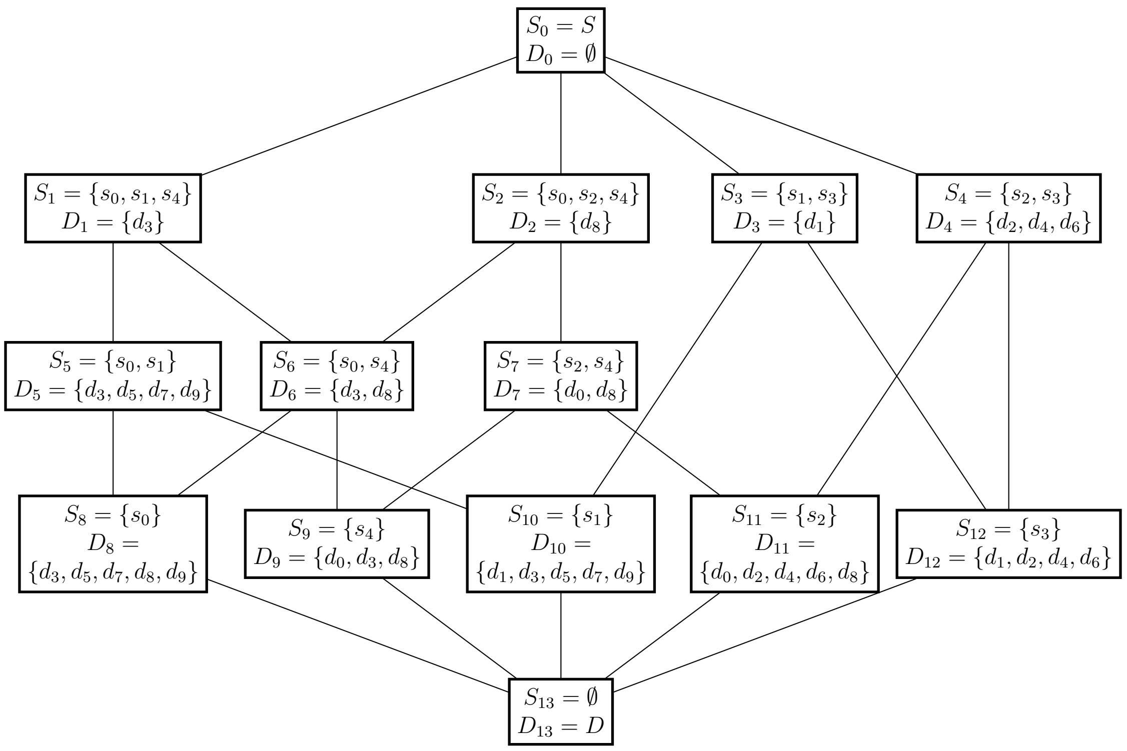

FIGURE 6.1: An example of a region lattice diagram based on input data in Tab. 6.1

concept lattices are the same as the ones used in FCA. The algorithms were discussed in Chapter 3, Section 2.3.2. Such algorithms take as input objects, attributes, and incidences. Compatibility with the discussed assumptions is achieved by replacing the objects and attributes with state and state descriptor symbols respectively. Figure 6.1 illustrates a lattice built using input data from Table 6.1. An example of a state-space graph partitioned by the lattice is visualized by Fig 6.2.

The procedure of building a region lattice may be costly, but it can be significantly accelerated by pruning input observations that are not relevant to the model. Each observed state is associated with a set of state descriptors, which represent regions of the state space. If two states fall into the same set of regions, then one of the states is redundant because it does not contribute to the knowledge about the structure of the state space. In other words, observed states are only used for discovering relations between regions. Therefore, for best performance, observed states should have distinct sets of descriptors, and the repeating ones can be discarded.

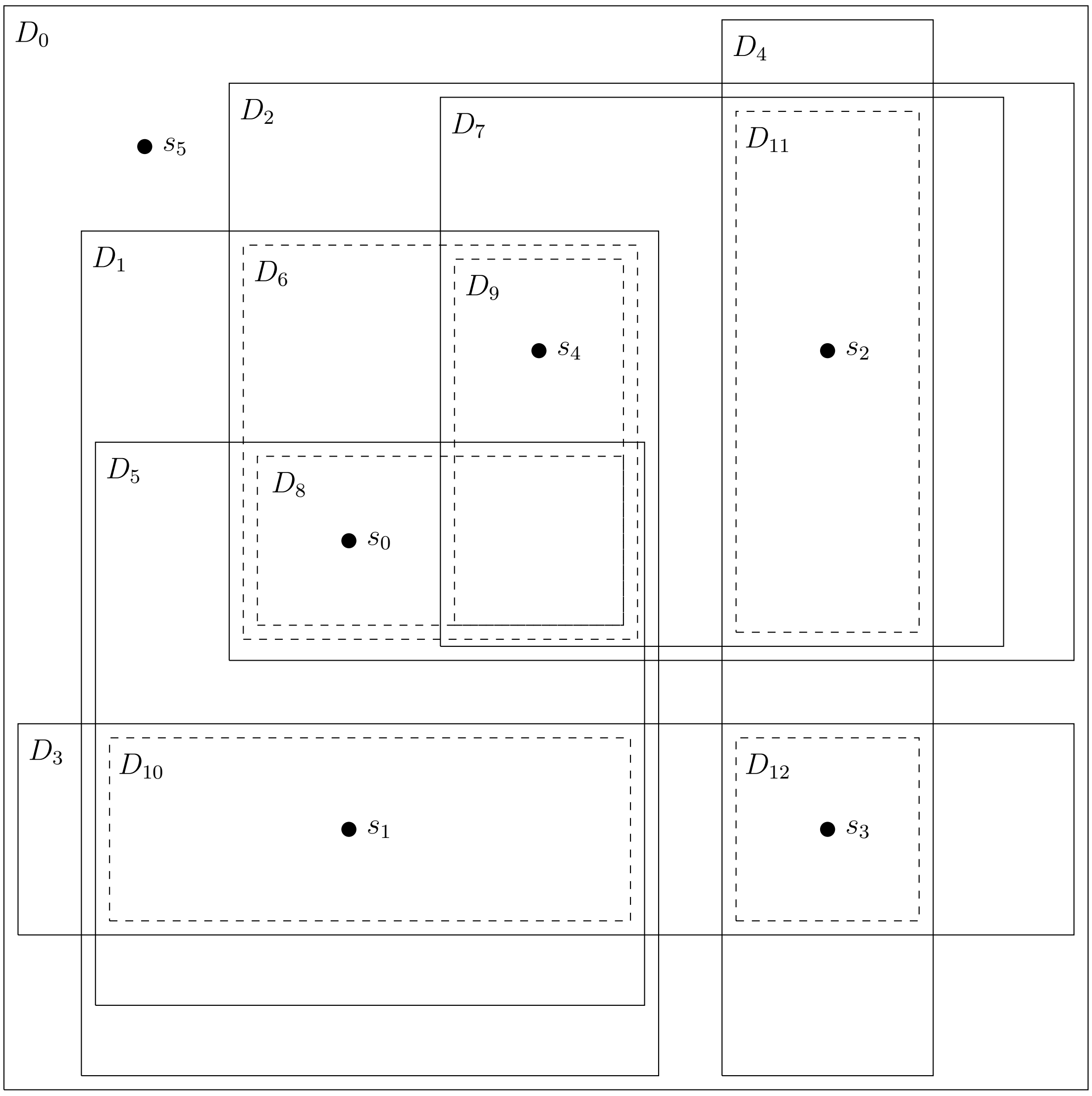

FIGURE 6.2: An example of state-space partitioning based on a region lattice diagram in Fig. 6.1

6.2 Planning Method

The following sections describe the proposed planning method. The description begins with the definition of region distance in a lattice. Next, an algorithmic procedure of calculating the distance is presented. Subsequently, two new state search algorithms, which rely on the properties of the heuristic, are introduced. Finally, the complexity and the optimality of the estimator is discussed.

6.2.1 Planning Heuristic

Each node (concept) in a state-space lattice is a region of the state space. The intent of a concept is defined as a set of descriptors. A region includes states that satisfy all the descriptors in its intent. In other words, the region is the common part of intersection of state sets associated with the descriptors in the intent. If two nodes in a lattice are connected, then the upper one contains the entire region of the lower one. The relation is transitive. The topmost node is the root, and it covers the entire state space. A lattice enables us to locate any state inside its structure, and search for it by narrowing the search space.

$\Delta(s_a, s_b)$	$s_{b=0}$	$s_{b=1}$	$s_{b=2}$	$s_{b=3}$	$s_{b=4}$	$s_{b=5}$
$s_{a=0}$	$\lvert D_8\rvert - \lvert D_8\rvert = 5-5=0$	$\lvert D_{10}\rvert - \lvert D_5\rvert = 5-4=1$	$\lvert D_{11}\rvert - \lvert D_2\rvert = 5-1=4$	$\lvert D_{12}\rvert - \lvert D_0\rvert = 4-0=4$	$\lvert D_9\rvert - \lvert D_6\rvert = 3-2=1$	$\lvert D_0\rvert - \lvert D_0\rvert = 0-0=0$
$s_{a=1}$	$\lvert D_8\rvert - \lvert D_5\rvert = 5-4=1$	$\lvert D_{10}\rvert - \lvert D_{10}\rvert = 5-5=0$	$\lvert D_{11}\rvert - \lvert D_0\rvert = 5-0=5$	$\lvert D_{12}\rvert - \lvert D_3\rvert = 4-1=3$	$\lvert D_9\rvert - \lvert D_1\rvert = 3-1=2$	$\lvert D_0\rvert - \lvert D_0\rvert = 0-0=0$
$s_{a=2}$	$\lvert D_8\rvert - \lvert D_2\rvert = 5-1=4$	$\lvert D_{10}\rvert - \lvert D_0\rvert = 5-0=5$	$\lvert D_{11}\rvert - \lvert D_{11}\rvert = 5-5=0$	$\lvert D_{12}\rvert - \lvert D_4\rvert = 4-3=1$	$\lvert D_9\rvert - \lvert D_2\rvert = 3-1=2$	$\lvert D_0\rvert - \lvert D_0\rvert = 0-0=0$
$s_{a=3}$	$\lvert D_8\rvert - \lvert D_0\rvert = 5-0=5$	$\lvert D_{10}\rvert - \lvert D_3\rvert = 5-1=4$	$\lvert D_{11}\rvert - \lvert D_4\rvert = 5-3=2$	$\lvert D_{12}\rvert - \lvert D_{12}\rvert = 4-4=0$	$\lvert D_9\rvert - \lvert D_0\rvert = 3-0=3$	$\lvert D_0\rvert - \lvert D_0\rvert = 0-0=0$
$s_{a=4}$	$\lvert D_8\rvert - \lvert D_6\rvert = 5-2=3$	$\lvert D_{10}\rvert - \lvert D_1\rvert = 5-1=4$	$\lvert D_{11}\rvert - \lvert D_7\rvert = 5-2=3$	$\lvert D_{12}\rvert - \lvert D_0\rvert = 4-0=4$	$\lvert D_9\rvert - \lvert D_9\rvert = 3-3=0$	$\lvert D_0\rvert - \lvert D_0\rvert = 0-0=0$
$s_{a=5}$	$\lvert D_8\rvert - \lvert D_0\rvert = 5-0=5$	$\lvert D_{10}\rvert - \lvert D_0\rvert = 5-0=5$	$\lvert D_{11}\rvert - \lvert D_0\rvert = 5-0=5$	$\lvert D_{12}\rvert - \lvert D_0\rvert = 4-0=4$	$\lvert D_9\rvert - \lvert D_0\rvert = 3-0=3$	$\lvert D_0\rvert - \lvert D_0\rvert = 0-0=0$

TABLE 6.2: Region distances calculated for an example of partitioning in Fig. 6.2

Definition 6.6 (Lattice Region Distance)**.** Let a function $\Delta : S \times S \to \mathbb{N}_{\geq 0}$ be a heuristic estimator of the distance between regions in a state-space lattice $L(K^{\mathbb{S}})$, Eq 6.7:

$$\Delta(s_a, s_b) = |D_j| - |D_i|, \tag{6.7}$$

where:

- s_a is an initial state, and s_b is the goal state,
- D_i and D_j are intents of partially ordered concepts $\langle S_j, D_j\rangle \leq \langle S_i, D_i\rangle$,
- D_j is the most numerous intent and the smallest region that contains s_b,
- D_i is the most numerous intent that contains s_a, and $D_i \subseteq D_j$. Also, D_i is the smallest region that contains both s_a and s_b,
- $|\cdot|$ is the cardinality of a set.

The estimate calculates the number of regions that must be crossed when moving from an initial state to the goal state. Table 6.2 contains distances calculated for regions in Fig. 6.2. In the table, initial and goal states are placed in rows and columns respectively.

The heuristic measure of the distance between two states in the state space is reduced to the region distance in a state-space lattice (Eq. 6.7). The algorithmic method of calculating the distance is presented by Alg. 5. Below is the list of variables accepted on input by the algorithms:

- *start* – an initial state s_a,
- *goal* – the goal state s_b,
- *Lattice* – a region lattice $L(K^{\mathbb{S}})$.

The result is a non-negative integer number representing the distance estimate.

In the first line of the routine, a complete set of descriptors is obtained from the context of the lattice. In the second one, concepts are populated as lattice nodes. They are in ascending order of their intent size, which is the number of descriptors. This gives a sequence of regions organized from the largest one to the smallest one.

The next part of the algorithm focuses on finding the smallest region that contains the final state (lines 4-12). To do that, a set of descriptors that contain the final state is collected (line 4). The initial region is the topmost node in the lattice, which includes the entire state space (line 5). The loop iterates over the sorted sequence of lattice nodes until the intent of the current node is greater than the number of descriptors that contain the final state (lines 9-10). It is because the final node is updated only if the region of the current node is smaller, and the set of final descriptors is a subset of the current intent (lines 11-12). In line 14, all the lattice nodes that contain the final region are collected using the partial order relation (Eq. 6.4). The sequence of final regions is sorted according to the same rule as the lattice nodes in the second line.

In the final part of the routine, the regions are iterated over to find the smallest region that contains the initial state (lines 16-21). The loop is interrupted as soon as the state falls outside the current region (lines 19-20), because the regions shrink in each subsequent iteration.

Finally, the algorithm returns the difference of intent sizes between the final region and the initial one (line 22). It should be noted that lines 1-14 do not have to be repeated

Alg. 5: GetHDistance(*start*, *goal*, *Lattice*)

```
var $descriptors \leftarrow$ GetAttributes(GetContext($Lattice$))
var $nodes \leftarrow$ OrderedByIntentSize(GetConcepts($Lattice$))
                                        ▷ collect descriptors that contain the final state
var $goalDescriptors \leftarrow \{d \in descriptors : d(goal)\}$
var $goalNode \leftarrow$ GetSupremum($Lattice$)
                                        ▷ find the smallest region that contains the final state
foreach $n \in nodes$ do
    var $D \leftarrow$ GetIntent($n$)
    if $|D| > |goalDescriptors|$ then
        break
    if $|D| > |$GetIntent$(goalNode)| \wedge D \subseteq goalDescriptors$ then
        $goalNode \leftarrow n$
                                        ▷ collect regions that contain the goal region
var $goalRegions \leftarrow$ OrderedByIntentSize($\{n \in nodes : n \leq goalNode\}$)
                                        ▷ find the smallest region that contains the initial state
var $startNode \leftarrow \epsilon$
foreach $n \in goalRegions$ do
    var $D \leftarrow$ GetIntent($n$)
    if $\exists_{d \in D} \neg d(start)$ then
        break
    $startNode \leftarrow n$
return $|$GetIntent$(goalNode)| - |$GetIntent$(startNode)|$
```

if the final state does not change. That part can be precomputed when the state-search algorithm is initialized.

6.2.2 State Search Algorithms

The early concept of the planning heuristic (Region Tree Distance, Def. 5.4), introduced in the previous chapter, was applied to A* algorithm. The distance between regions in a tree was scaled down to make the heuristic admissible. However, this resulted in significantly reducing the accuracy of the estimator. To avoid performance degradation and preserve optimality, two new state search algorithms, which were adapted to the abstract nature of distance in a region lattice, were proposed.

The family of the proposed algorithms, *State-space Lattice Search Heuristics* is abbreviated to SLaSH. The algorithms are variants of BFS (Section 2.2.2). Below is the list of variables accepted by the algorithms on input:

- *start* – an initial state s_a,
- *goal* – the goal state s_b,
- *Lattice* – a region lattice $L(K^{\mathbb{S}})$.

The result is a plan p_i.

The idea of the first algorithm, M-SLaSH (a memory-optimized variant of SLaSH) is simply ignoring states that belong to regions that are uninteresting with respect to the value of the heuristic (Alg. 6). The structure of the algorithm is mostly identical to Uniform-Cost Search (UCS) (Appendix, Alg. 8), which is a Dijkstra's algorithm variant [28]. The routine begins by initializing variables being a part of the original UCS (lines 1-4). Respectively, they include:

- *cost*[] – a cost dictionary that maps a state into the cost of reaching it,
- *previous*[] – a path dictionary for recreating a path from the final state to the initial one,
- *frontier* – a priority queue that sorts states by their cost in ascending order,
- *explored* – a set of previously visited states.

The additional variable is the *heuristic* that represents an estimated distance to the final state (line 5).

The core of the main loop remains unchanged. It populates states from the priority queue until the goal state is reached, or the queue is empty (lines 7-10). Here, states are marked as visited (line 11). Neighbours of each state are expanded in a sub-loop (line 12). The previously visited ones are ignored (lines 13-14). Subsequently, the cost of reaching a state is calculated as the sum of the accumulated cost and the weight associated with the edge between the expanded node and its neighbour in the state-space graph (line 15). If a state has been reached with a higher cost than before, then it is ignored, and the sub-loop continues from its starting point (lines 16-17).

The next part is the heuristic extension of the original algorithm (lines 19-23). Its purpose is to prevent expanding states that are further from the final state than the already visited ones. Thus, if the heuristic distance of the current state is greater than the previously calculated one, then the sub-loop goes back to its beginning, because the state queue contains states that are closer to the goal, so the current state can be ignored. Otherwise,

Alg. 6: M-SLaSH(*start*, *goal*, *Lattice*)

```
1  var $cost[start] \leftarrow 0$                                   ▷ initialize a cost dictionary
2  var $previous[start] \leftarrow \epsilon$                        ▷ initialize a path dictionary
3  var $frontier \leftarrow \{\langle start, cost[start]\rangle\}$  ▷ initialize a min-priority queue
4  var $explored \leftarrow \{\}$                                   ▷ initialize an empty set
5                                                                   ▷ calculate an initial heuristic distance
6  var $heuristic \leftarrow$ GetHDistance($start$, $goal$, $Lattice$)
7  while $frontier \neq \{\}$ do
8      var $node \leftarrow$ Pop($frontier$)                        ▷ remove and take a node with the lowest cost
9      if $node = goal$ then
10         return GetSolution($cost$, $previous$)                   ▷ return the solution cost and path
11     Add($explored$, $node$)
12     foreach $n \in$ GetNeighbours($node$) do
13         if $n \in explored$ then
14             continue
15         var $c \leftarrow cost[node]$ + GetWeight($node$, $n$)
16         if $n \in cost \land c \geq cost[n]$ then
17             continue
18                                                                  ▷ heuristic extension begins
19         var $h \leftarrow$ GetHDistance($n$, $goal$, $Lattice$)
20         if $h > heuristic$ then
21             continue
22         else if $h < heuristic$ then
23             $heuristic \leftarrow h$
24                                                                  ▷ heuristic extension ends
25         if $n \in frontier$ then
26             Remove($frontier$, $n$)
27         $cost[n] \leftarrow c$
28         $previous[n] \leftarrow node$
29         Add($frontier$, $\langle n, cost[n]\rangle$)
30 return $failure$                                                 ▷ solution not found
```

the sub-loop is not interrupted. Additionally, if the heuristic distance is less than the previous one, then the variable is updated (line 23).

The cost determines the order in the priority queue. If a state is already present in the queue, then it is removed (lines 25-26) and later added again to update its position inside the queue according to its new cost. The associated cost and the previous state are set (lines 27-28). In the final part, the current state is added to the priority queue (line 29).

The discussed algorithm will always find the optimal solution if the state-space lattice

Alg. 7: A-SLaSH(*start*, *goal*, *Lattice*)

```
var $cost[start] \leftarrow 0$                              ▷ initialize a cost dictionary
                                                ▷ initialize a dictionary for heuristic values
var $h[start] \leftarrow$ GetHDistance($start$, $goal$, $Lattice$)
var $previous[start] \leftarrow \epsilon$                    ▷ initialize a path dictionary
var $frontier \leftarrow \{\langle start, h[start], cost[start]\rangle\}$  ▷ initialize a min-priority heuristic queue
var $explored \leftarrow \{\}$                                ▷ initialize an empty set
while $frontier \neq \{\}$ do
    var $node \leftarrow$ Pop($frontier$)            ▷ remove and take a node with the lowest cost
    if $node = goal$ then
        return GetSolution($cost$, $previous$)         ▷ return the solution cost and path
    Add($explored$, $node$)
    foreach $n \in$ GetNeighbours($node$) do
        if $n \in explored$ then
            continue
        var $c \leftarrow cost[node]$ + GetWeight($node$, $n$)
        if $n \in cost \wedge c \geq cost[n]$ then
            continue
        if $n \in frontier$ then
            Remove($frontier$, $n$)
        else
            $h[n] \leftarrow$ GetHDistance($n$, $goal$, $Lattice$)
        $cost[n] \leftarrow c$
        $previous[n] \leftarrow node$
        Add($frontier$, $\langle n, h[n], cost[n]\rangle$)
return $failure$                                             ▷ solution not found
```

is strictly comprised of convex regions (Eq. 6.9). Otherwise, the heuristic estimate can be misleading, and the algorithm may not find any solution even if it exists. Therefore, the second algorithm, A-SLaSH was developed. In contrast to M-SLaSH, A-SLaSH does not prevent adding states to the priority queue if their heuristic distance is worse than the previously observed one. Instead, all the states are added to the queue, and they are sorted according to the heuristic estimate. Therefore, the method will always return a solution if it exists. However, optimality is guaranteed only if the lattice regions are convex.

A-SLaSH is presented in Alg. 7. It is based on A* (Appendix, Alg. 9). In the original version of A*, states in the priority queue are sorted according to the sum of a cost value and a heuristic value. The new algorithm uses a double sorting criterion as the region distance does not express the cost and should not be added to the cost value.

The structure of the algorithm is almost identical to M-SLaSH. For each new state, a heuristic estimate is calculated (line 21). The estimated value is then used in the queue to give a higher priority to states that are closer to the final state (line 24). If two states have equal heuristic value, then they are ordered according to their cost, which is the secondary sorting parameter.

To summarize, the second approach is more reliable than the first one, which is applicable to specific cases. However, the second algorithm is characterised by a higher memory consumption, because it stores a larger number of states in the priority queue.

6.2.3 Optimality

During the state search, the proposed lattice region distance is used to prioritize states that are closer to the goal. To guarantee optimality, the heuristic distance to the goal state must be monotonic in relation to the minimal cost of reaching it along the optimal path between an initial state and a final one. Let $p^*(s_a, s_b) = \langle S_k, E_k \rangle$ be an optimal path between any two states s_a and s_b, then monotonicity is described by Eq. 6.8:

$$\forall_{s_i, s_j \in S_k} \Big(\Delta(s_i, s_b) \leq \Delta(s_j, s_b) \iff c\big(p^*(s_i, s_b)\big) \leq c\big(p^*(s_j, s_b)\big) \Big), \tag{6.8}$$

where:

- s_i, s_j are any two states on the path between s_a and s_b,
- $\Delta(\cdot)$ is the lattice region distance (Eq. 6.7),
- $p^*(\cdot)$ is an optimal path between two states,
- $c(\cdot)$ is the cost of a path.

Definition 6.7 (Convex State Set)**.** A set $S_k \in S$ of states is convex if and only if for all two states $s_a, s_b \in S_k$ the cost-optimal path $p^*(s_a, s_b)$ between these two states is comprised of a set of states $S_i \subseteq S_k$:

$$\text{convex}(S_k) \iff \forall_{s_a, s_b \in S_k} \big((p^*(s_a, s_b) = \langle S_i, E_i \rangle) \Rightarrow S_i \subseteq S_k \big). \tag{6.9}$$

A convex region in the state space can be understood as a convex geometrical shape – e.g., a circle, sphere, hypersphere. The dimensionality of such shape depends on the number of state variables in a state. In geometry, a shape is convex if a straight line segment that is drawn between any two points in the shape is completely contained within the shape. In the case of a convex state set, the same rule applies, but the line segment is replaced by the shortest path between two nodes in a state-space graph.

For given assumptions, the heuristic provides optimal plans if it is monotonic:

Monotonicity Proof. Let $L(K)$ be a state-space lattice in which each region is convex:

$$\forall_{\langle S_i, D_i\rangle \in L(K)} \text{convex}(\{s \in S : \forall_{d \in D_i} d(s)\}). \tag{6.10}$$

An initial state and the final one are denoted by s_a and s_b respectively. Thus:

1. The initial heuristic distance is equal to $h_a = \Delta(s_a, s_b) = |D_b| - |D_a|$ (Def. 6.6).
2. Because the final state is constant, the heuristic distance depends on the first argument, which denotes the current state during the search: $h_i = \Delta(s_i, s_b) = |D_b| - |D_i|$.
3. Therefore, the distance can increase only if the algorithm visits a state that belongs to a region whose intent is smaller than the current one: $h_i \leq h_{i+1} \iff |D_i| \geq |D_{i+1}|$.
4. Based on the partial order relation in Def. 6.4, D_i represents a subregion of D_{i+1} $(S_i \subseteq S_{i+1})$.
5. Assuming that D_i covers a convex space S_i, the cost-optimal path between s_i and s_b is always inside S_i, and it will never go through a bigger set S_{i+1}.
6. Based on the foregoing: $h_i \leq h_{i+1} \iff c\big(p^*(s_i, s_b)\big) \leq c\big(p^*(s_{i+1}, s_b)\big)$.
7. Equivalently: $\Delta(s_i, s_b) \leq \Delta(s_{i+1}, s_b) \iff c\big(p^*(s_i, s_b)\big) \leq c\big(p^*(s_{i+1}, s_b)\big)$.

□

For practical problems with a complex structure of the state space, ensuring optimality may be difficult. However, the cost estimate should still give a good approximation. The quality of non-optimal heuristic is an open subject for further research.

6.2.4 Computational Complexity

The computational complexity of the proposed algorithms derives from the complexity of Dijkstra's algorithm. Based on work [28], the worst-case performance of UCS with the optimal implementation of the min-priority queue can be approximated by Eq. 6.11:

$$O(|E| + |S| \log |S|), \tag{6.11}$$

where $|E|$ is the number of edges, and $|S|$ is the number of states in the state-space graph.

The complexity of the discussed approach is affected by the region distance calculation, whose performance depends on the number of concepts in a concept lattice. Assuming that computations regarding the goal state have been executed once in the initial stage of the routine, the worst-case performance of the region distance function is expressed by Eq. 6.12:

$$O(|L(K^{\mathbb{S}})|), \tag{6.12}$$

where $|L(K^{\mathbb{S}})|$ stands for the number of nodes in a lattice.

The worst-case performance of the proposed algorithms combines the complexities of Dijkstra's original algorithm and the region distance function, which is described by Eq 6.13:

$$O\big(|E| + |S|(\log |S| + |L(K^{\mathbb{S}})|)\big). \tag{6.13}$$

It is because the distance to the goal is calculated for each visited state.

The benefit of using the proposed approach is the reduction of the search space, which translates into the decrease of the number of states visited by the algorithm. It depends on the structure of the state space, a set of state descriptors, and the quality of input observations. Therefore, the improvement can be measured through an experimental study for each case separately.

6.2.5 Performance Optimisation

In a practical application, a set of state descriptors can be rigidly defined based on problem-specific knowledge, or it can be automatically obtained from the output of an optimisation method. The first approach may be sufficient in many cases, but potentially less efficient in comparison with the automated optimisation process.

The performance of the proposed planning method highly depends on the quality of the state-space partitioning, which relies on provided state descriptors. An optimal set of state descriptors can be defined formally. For a given set $S_0 \subset S$ of input observations, and a test set $T_0 \subset T$ of planning tasks, such optimal set D^* of state descriptors can be found according to Eq. 6.14:

$$D^* \in \arg\min_{D_i} \sum_{t=\langle s_a, s_b \rangle \in T_0} C\big(\mathbb{S}, L(S_0, D_i, I_i), s_a, s_b\big), \tag{6.14}$$

where:

- t is a planning task (Def. 2.4),
- $C(\cdot)$ is a performance evaluating function for a single execution of a state-search algorithm,
- $\mathbb{S}$ is a state-space graph (Def. 2.3),
- $L(\cdot)$ is a state-space lattice (Def. 6.5),
- $D_i \subseteq D$ is a set of evaluated state descriptors (Def. 5.1) – defined by the system designer or generated automatically,
- I_i is a set of incidences (Eq. 6.6).

Depending on the field of application, the performance criteria for $P(\cdot)$ may differ. In many cases, the evaluation can be simplified to measuring wall-clock execution times. However, to abstract from implementation specificity and focus on principles of the studied algorithms, the number of visited states is used as a point of reference. This measure can be directly translated into the consumption of computational resources.

6.3 Experimental Study

Experiments presented in the previous chapter demonstrate that the proposed concept of a heuristic estimator can be successfully applied to general action planning and solving complex combinatorial problems. Therefore, the following experiments were aimed at validating the improved version of the method and presenting its characteristics. The primary focus of the experimental study was examining an automatic procedure of tuning state descriptors. The procedure employs a Genetic Algorithm.

6.3.1 Testbed Environment

Smart Blocks helped to accomplish a milestone in the method development, but the testbed environment cannot be used to progress with the research. The new environment must meet the following requirements:

1. An environment must provide multiple planning problem instances represented by multiple goals. It is because the purpose of the partitioning model is improving the performance of solving problems dynamically appearing during the game. By design, each map in Smart Blocks has a single goal and a single planning problem.
2. For each planning problem instance, an optimal cost of reaching the goal must be known. It is required for verifying the quality of the heuristic estimator. For Smart Blocks and other puzzle benchmarks, an optimal cost estimator is unknown, and precomputing costs of all paths would be challenging.
3. Plan traces should be generated artificially because collecting them from volunteers each time when the parameters of an environment change would be impossible. Generating plan traces for complex planning domains is nontrivial.
4. A desired feature of an environment is scalability. It is important for conducting experiments that employ a Genetic Algorithm, for which the scale of a problem must be adjusted to fit in time constraints imposed on the study.

In the view of the stated requirements, the new testbed environment models a uniform metric space. A 2D Manhattan space was selected for easy visualisation.

It is worth noting that solving pathfinding problems is simple for specialised methods. However, automatically obtaining a heuristic estimator is still challenging for planning methods that do not make a prior assumption that the state space is metric. It applies to general action planning systems such as domain-independent planners and the proposed method as well.

The following subsections define the state space, discuss the method of constructing a region lattice, and describe the application of a Genetic Algorithm for tuning state descriptors.

Grid Space

The tests were conducted for a 2D grid whose state-space graph can be described by a discrete n-dimensional injective metric space for $n = 2$. A state in such space is defined as a coordinate vector s, Eq. 6.15:

$$s = \langle v_1, v_2 \rangle, \tag{6.15}$$

where v_1, v_2 are state variables, and $v_1, v_2 \in \mathbb{N}_{\geq 0}$. The extent of the state space forms a rectangle, which generalises to an n-cube.

State transition in such a graph is allowed between any two neighbouring coordinates that are inside the space limits and lie within a distance of one unit vector. Diagonal transitions (modifying more than one variable at a time) are not permitted. The cost of transition in such state space is uniform, and it can be estimated using Manhattan distance as the state space forms a hyperconvex metric space. Manhattan distance returns the cost of an optimal path. Such a heuristic dominates all admissible heuristics, which means that A* guided by it visits a minimum number of states.

The performance of the studied algorithms was measured for a series of pathfinding tasks. Each pathfinding task comprises an initial state s_a and a final one s_b, between which a path is searched. Both states are generated by randomly picking values for their state variables, but ensuring that each variable is inside the previously defined limits, and $s_a \neq s_b$. The number of pathfinding tasks generated for each test is specified parametrically.

Lattice Construction

A region lattice is built from two collections – state descriptors and observed states. In a practical application, observed states would be obtained from plan traces. For the sake of simplicity, the observations were artificially generated as random coordinates in the space using the same system as in the case of random states for pathfinding tasks.

In the initial phase of constructing a region lattice, state descriptors and observed states are processed to obtain the relation of incidence. The incidence table and the collections are then passed to the input of a lattice constructing algorithm. Such an algorithm operates on an abstract level of FCA that does not involve any specific information about the state space. Observed states and state descriptors are treated as objects and attributes respectively.

In this study, an algorithm proposed by Bordat in work [58] was selected for building lattices and testing the approach. The pseudocode expressing the algorithm is present in Appendix, Alg. 10. In principle, lattice constructing algorithms generate a complete set of concepts that form a concept lattice, but Bordat's algorithm is one of few that constructs the diagram graph. Based on information in work [67], the algorithm performs well in small and medium contexts with average density. Therefore, it appears to be suitable for conducting research.

Although a lattice is being used here for an untypical application, which is state-space partitioning, the structure of the model remains unchanged. The experiments show only necessary parameters to prove that fact, rather than detailed information, which can be found in works dedicated to studying lattices. The measurements include the number of nodes (concepts) in a lattice and the wall-clock times of an algorithm that builds it. The procedure of building a lattice may be repeated as the wall-clock times are averaged. However, its result is deterministic and does not change if the input remains the same.

Tuning State Descriptors

In the considered state-space graph, regions formed by state descriptors are n-orthotopes (hyperrectangles). For $n = 2$, a region is a rectangle. Such a state descriptor comprises a sequence of orthogonal intervals for each dimension. The intervals encompass certain ranges of coordinates in a state coordinate vector. The descriptor predicate is defined as follows, Eq. 6.16:

$$d\big(s = \langle v_1, v_2, \dots v_i, \dots v_n \rangle\big) = \forall_i v_i \in [a_i, b_i), \tag{6.16}$$

where s is a state, v_i is i-th state variable, a_i and b_i are the bounds of an interval imposed on i-th state variable. An initial set of state descriptors is generated randomly. The generator ensures that the bounds of each interval fall inside the limits of the state space and $a_i \leq b_i$.

State descriptors can be tuned to maximise the performance of the planning method by employing a classic variant of a genetic algorithm, which was introduced in Section 2.4. A state descriptor can be encoded differently depending on specific properties of the state space. The following part of the section discusses a basic and universal chromosome representation.

Let $X_i^D \in X^D$ be a chromosome comprised of a sequence of genes encoding state descriptors, Eq. 6.17:

$$X_i^D = \langle x_1^d, x_2^d, \dots x_k^d \rangle, \tag{6.17}$$

where x^d is a gene encoding a state descriptor, and k is the number of state descriptors in the chromosome. Such a gene stores a sequence of pairs representing limits imposed by its state descriptor, Eq. 6.18:

$$x^d = \big\langle \langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle, \ldots \langle a_i, b_i \rangle, \ldots \langle a_n, b_n \rangle \big\rangle, \tag{6.18}$$

where a_i is the lower bound, and b_i is the upper bound of i-th interval.

Chromosomes of two individuals are uniformly crossed over by exchanging random interval bounds $\langle a_i, b_i \rangle$ between corresponding genes. The interval bounds are mutated using a mutation method described in Section 9. This procedure may cause that the lower bound is a larger number than the upper bound. Whenever this anomaly occurs, the interval is repaired by simply swapping the bounds.

To evaluate an individual, state descriptors in its chromosome and a given set of observed states are used for constructing a region lattice. Fitness function $F(\cdot)$ expresses the performance of a SLaSH algorithm relying on the constructed region lattice. It is calculated as a negated average number of states visited by a SLaSH algorithm for a number of planning tasks, Eq. 6.19:

$$F(X_i^D) = -\sum_{t \in T} \big|S(t, L(S_0, D_i))\big|, \tag{6.19}$$

where:

- X_i^D is the chromosome of i-th individual,
- T is a set of planning tasks,
- S_0 is a set of observed states,
- D_i is a set of descriptors encoded in X_i^D,
- $L(\cdot)$ is a region lattice constructed from S_0 and D_i,
- $S(\cdot)$ is a set of states visited by a SLaSH algorithm,
- $|\cdot|$ is the cardinality of a set.

The number of visited states is negated so the bigger value, the better performance. The performance of a planning method may be assessed differently depending on the field of application. In this particular study, the reduction of the search space was chosen as the primary criterion of evaluation.

6.3.2 Experiments

The initial experiment was conducted to confirm the fact that the presented use of a lattice does not imply any significant performance issues.

Experiment 6.1. Studying the effect of the sizes of a state descriptor set and an observed state set on the construction of a region lattice.

Parameters: The experiment was conducted for a grid of size 100 $\times$ 100. Input sets of state descriptors and observed states were generated randomly.

Course: A region lattice was repeatedly built using different numbers of input state descriptors and observed states. The sizes of both sets were ranging from 5 to 50 with a step of 5. The measurements referring to lattice construction include:

- the size of a lattice expressed in the number of nodes (regions),
- the wall-clock execution time of a lattice construction algorithm.

The procedure was repeated 10 times, and the measurements were averaged. The standard deviation of each averaged sample was calculated.

Results: Figure 6.3 shows the average size of a lattice (and the standard deviation of the averaged sizes) in relation to the numbers of state descriptors and observed states used for constructing the lattice. In the same manner, the wall-clock times of constructing a lattice are presented in Fig. 6.4. It can be concluded that the tested case is characterised by formal context of a medium density, and the growth of a lattice caused by increasing numbers of state descriptors and observed states is not alarming. However, it can be observed that the curve of lattice construction time is nonlinear, and it rises very fast as the size of a lattice grows. It is expected for the computational complexity of Bordat's algorithm, which was employed for building lattices.

As it was shown, an increasing number of input descriptors and states leads to a growth of a lattice. It is worth investigating how this influences the state search. The next experiment was aimed at examining the core characteristics of the heuristic affected by the structure of a lattice. The tests were conducted for a simple setup, so the influence of side factors is minimised, and the features of the algorithms can be easily observed and assessed.

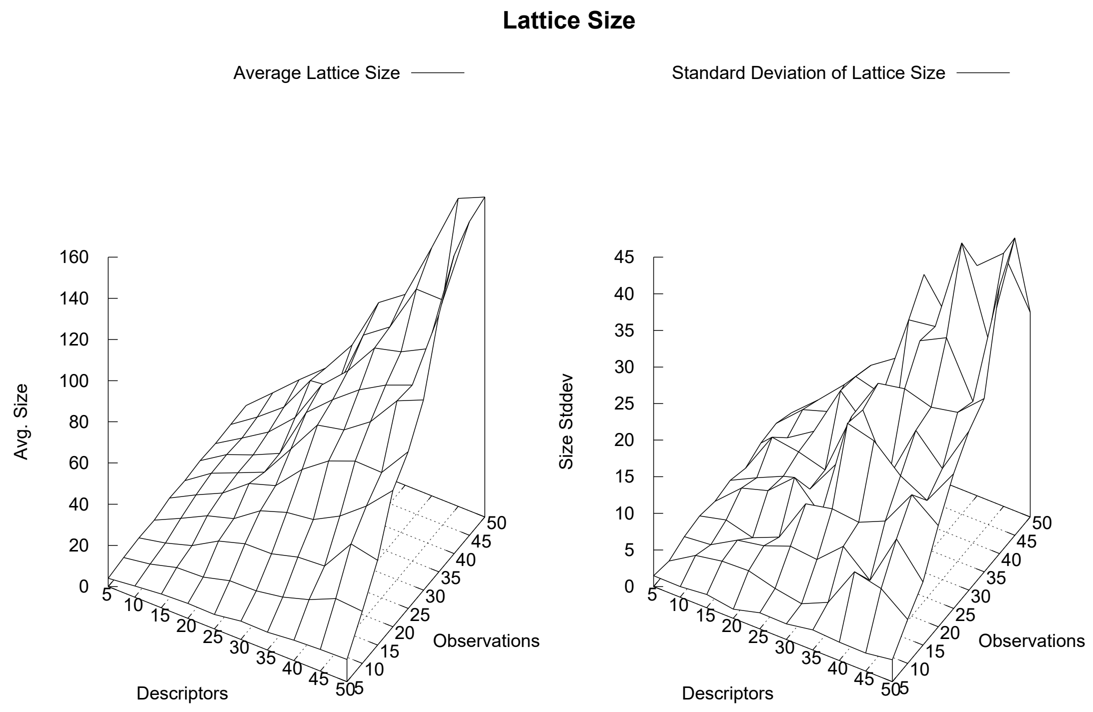

FIGURE 6.3: The effect of different numbers of descriptors and states on the size of a lattice [grid size: 100 × 100, repetitions: 10]

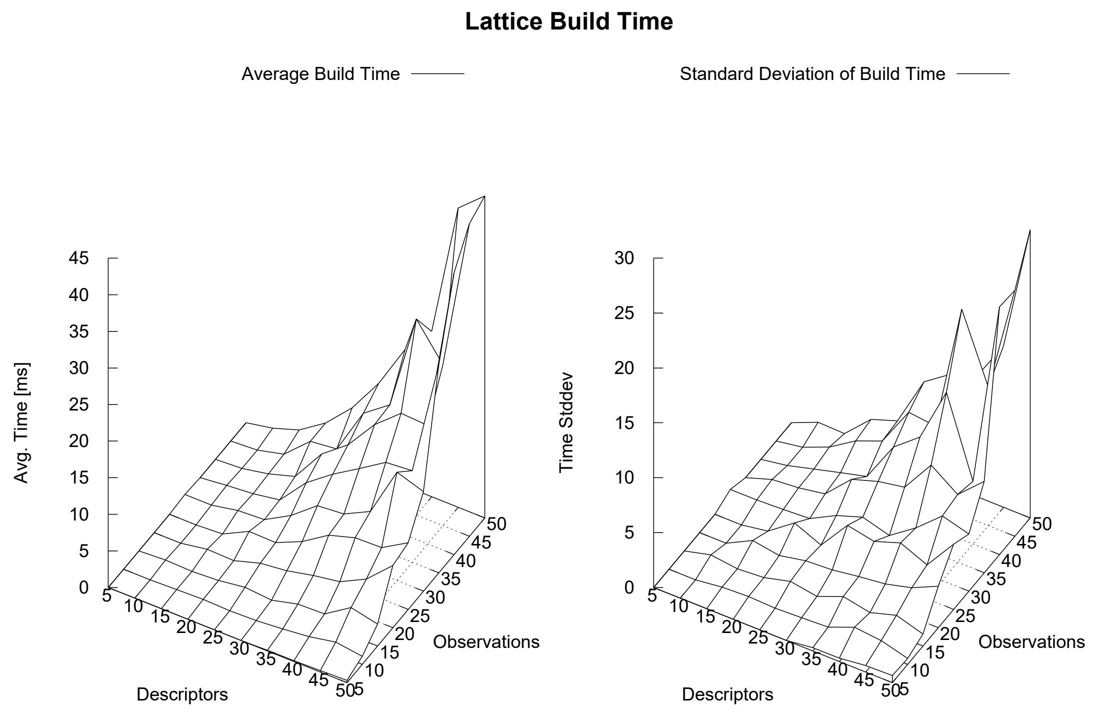

FIGURE 6.4: Lattice construction time affected by different numbers of descriptors and states [grid size: 100 × 100, repetitions: 10]

Experiment 6.2. Studying the effect of the sizes of a state descriptor set and an observed state set on the performance of the heuristic.

Parameters: The experiment uses the same parameters as the previous one (Exp. 6.1). Pathfinding tasks were generated randomly.

Course: A region lattice was built for different numbers of state descriptors and observed states identically as in the previous experiment. For each constructed lattice, both M-SLaSH and A-SLaSH were tested for 100 randomly generated pathfinding tasks. The measurements describing the performance of the algorithms include:

- the number of distinct states visited during the search,
- the wall-clock time of solving a pathfinding task,
- the maximum size of the priority queue,
- a solution error as the difference between the length of an optimal path and the one returned by an algorithm.

The measurements were collected for a number of pathfinding tasks and averaged. The standard deviation of each averaged sample was calculated.

Results: Figures 6.5 and 6.7 show that the number of visited states is decreasing as a lattice grows, and it is true for both algorithms. This phenomenon is explained by the fact that the larger lattice is, the better coverage of the state space it provides by dividing the space into a bigger number of regions. A high density of fragmentation gives better opportunities for narrowing the exploration when searching for a path between two random locations in the state space.

Wall-clock times presented in Fig. 6.6 and 6.8 demonstrate that the additional overhead calculations generated by the proposed distance estimate do not outbalance the benefit of reducing the number of visited states. The reduction of the search space correlates with the speed of solving planning problem instances.

Although the proposed algorithms performed similarly so far, they differ in the sizes of their priority queues. M-SLaSH populates fewer states in the priority queue when the space fragmentation gets denser – Fig. 6.9. However, Figure 6.10 shows the opposite behaviour of A-SLaSH. It is caused by the fact that the second algorithm accumulates states in the queue. States with a more promising heuristic value are prioritised and expanded first. The faster the search progresses towards the goal, the slower states are dequeued by the algorithm.

For this particular type of the state space, both algorithms produce optimal solutions. The solution error remains zero (Fig. 6.11 and Fig. 6.12).

The goal of the subsequent experiment was assessing the quality of the heuristic and demonstrating its essential characteristics by comparing the performance of SLaSH algorithms and the classical ones such as Dijkstra and A*. In the experiment, A* is guided by Manhattan distance, which is an adequate choice for the defined grid. The pseudocodes of the classical algorithms can be found in Appendix (Alg. 8 and Alg. 9 for Dijkstra and A* respectively).

It should be mentioned that the classical algorithms are an important point of reference for studying state search methods. Dijkstra's algorithm represents the worst-case scenario in which a heuristic estimator is unavailable, or it is ineffective. On the other hand, A* guided by an optimal heuristic function can be considered as one of the most efficient state search methods. The experimental study focuses only on universal algorithms that can be successfully employed for general action planning and pathfinding as well. Such algorithms can hardly compete with specialised pathfinders. Therefore, pathfinding methods that exploit favourable features of metric space or precompute paths were not taken into consideration because they are not relevant outside the Euclidean space, which was discussed in Chapter 3.

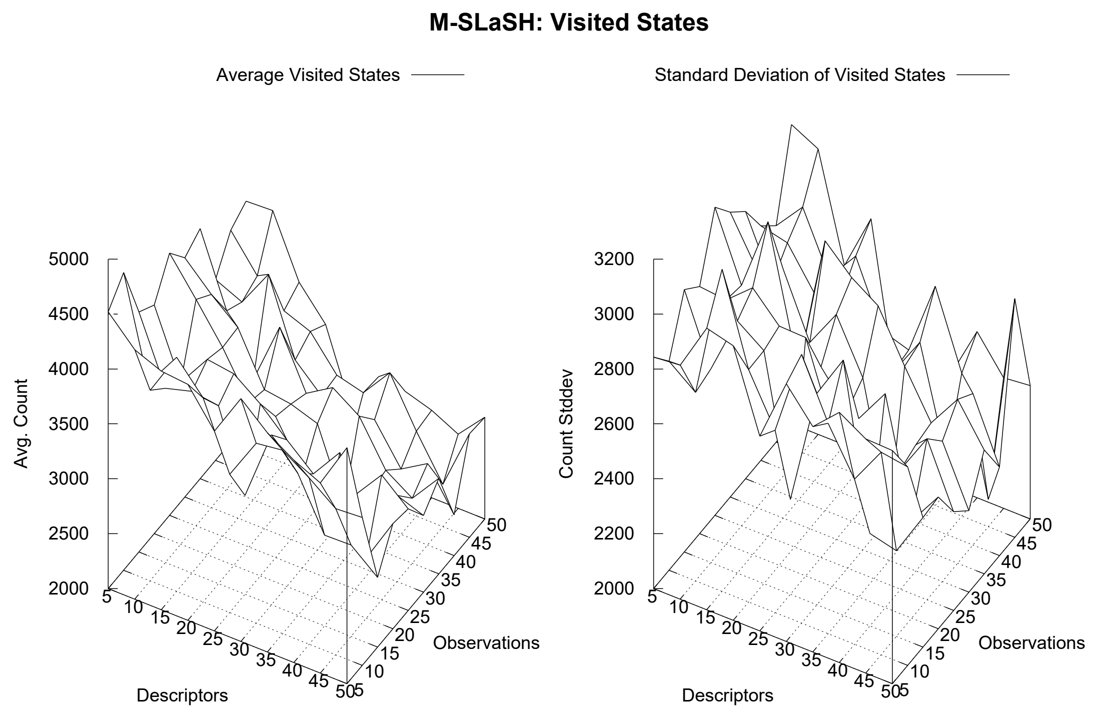

FIGURE 6.5: The impact of the number of input observations and descriptors on the number of states visited by M-SLaSH [grid size: 100 × 100, pathfinding tasks: 100]

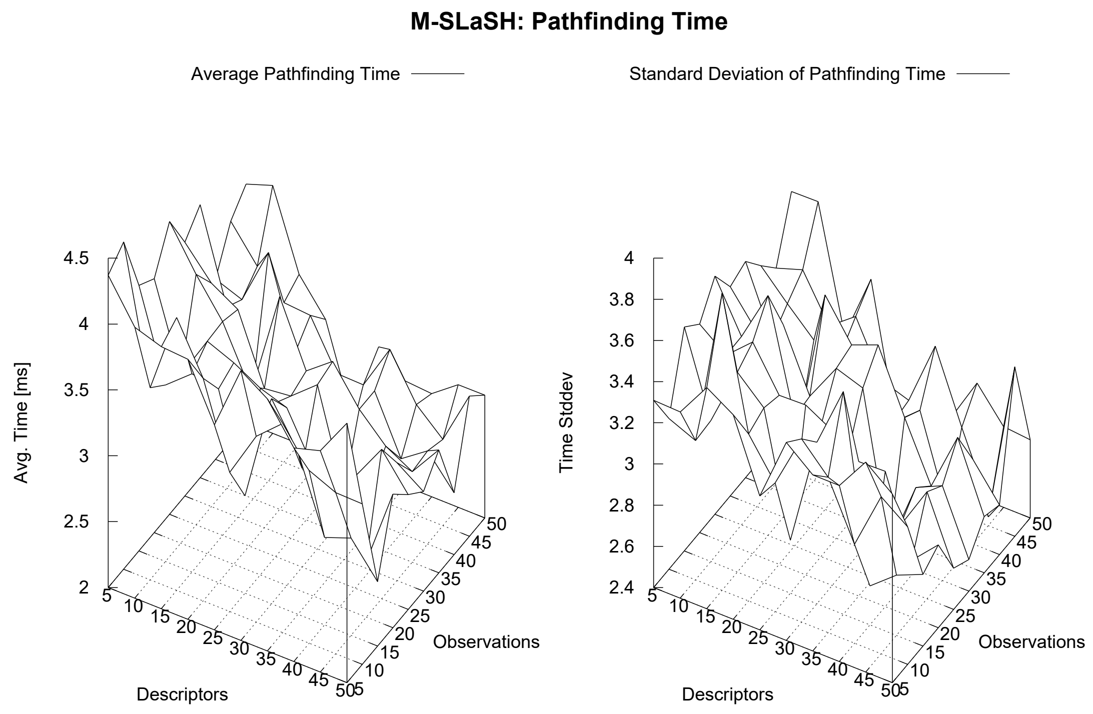

FIGURE 6.6: Pathfinding times of M-SLaSH affected by the number of input observations and descriptors [grid size: 100 × 100, pathfinding tasks: 100]

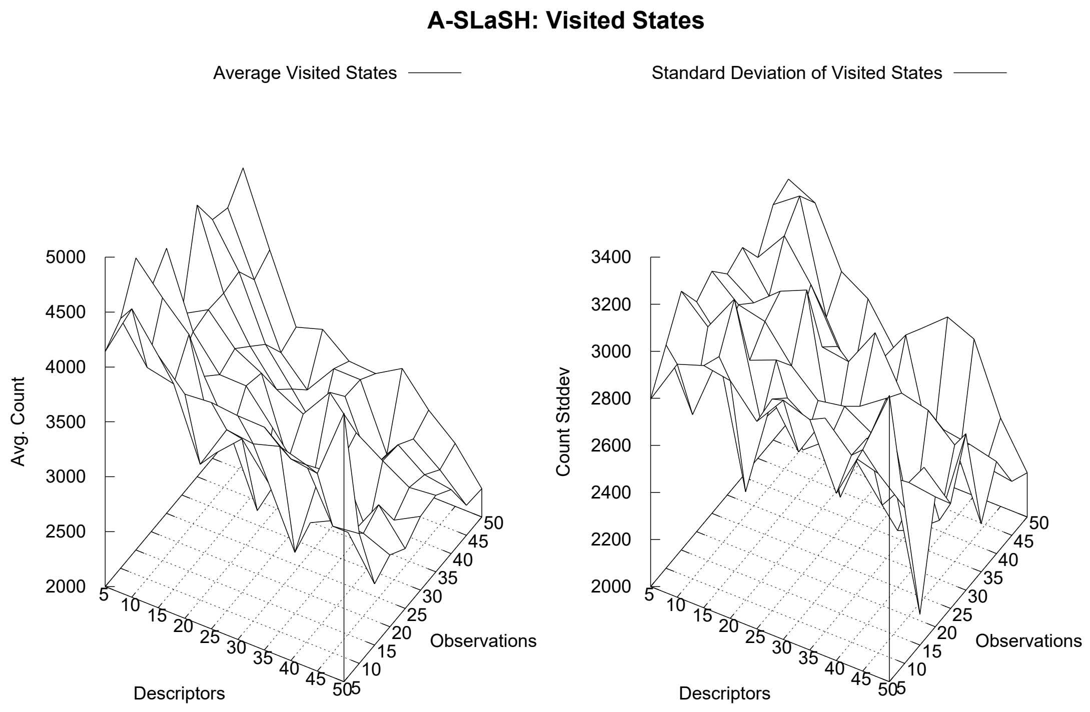

FIGURE 6.7: The impact of the number of input observations and descriptors on the number of states visited by A-SLaSH [grid size: 100 × 100, pathfinding tasks: 100]

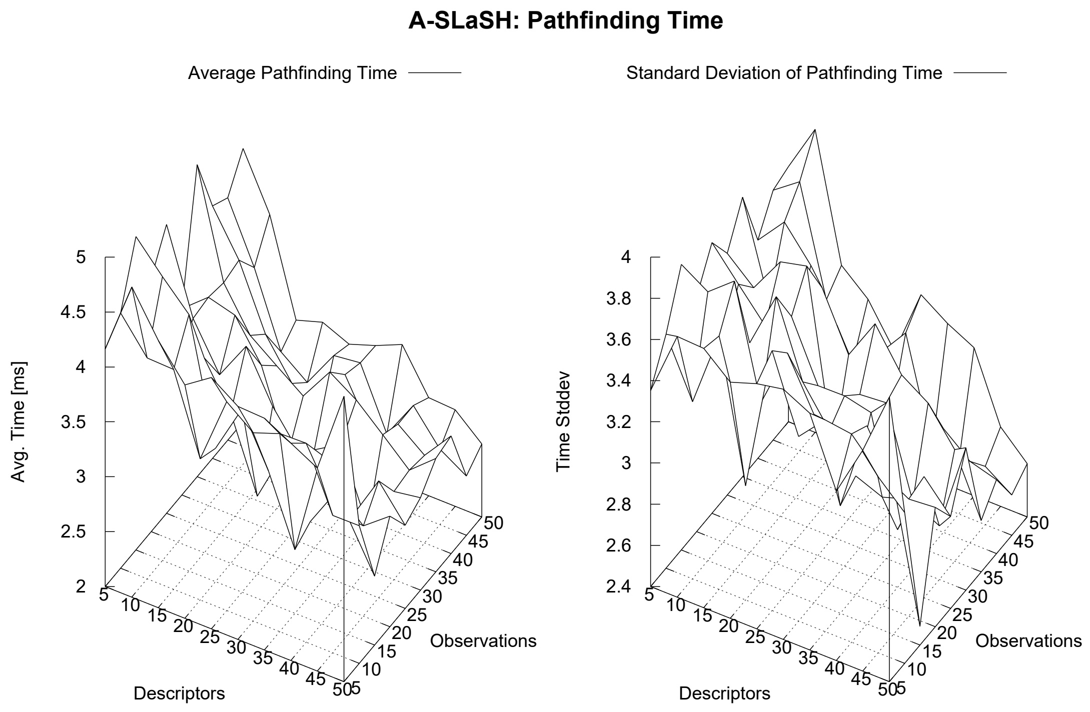

FIGURE 6.8: Pathfinding times of A-SLaSH affected by the number of input observations and descriptors [grid size: 100 × 100, pathfinding tasks: 100]

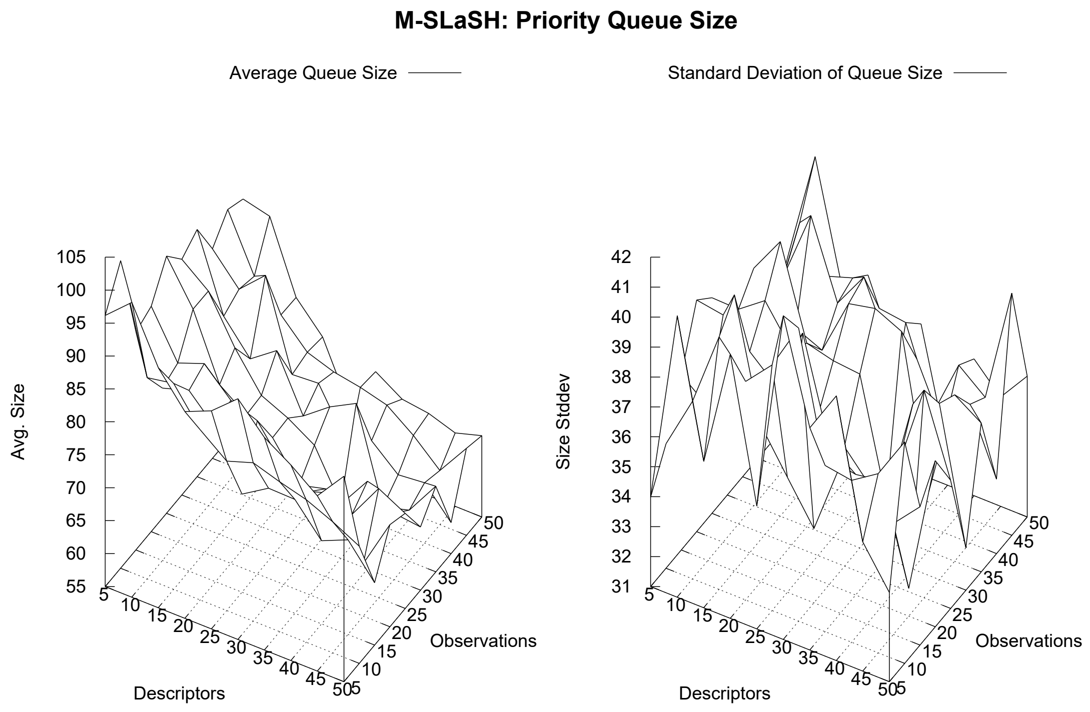

FIGURE 6.9: The effect of the number of input observations and descriptors on the maximum queue size in M-SLaSH [grid size: 100 × 100, pathfinding tasks: 100]

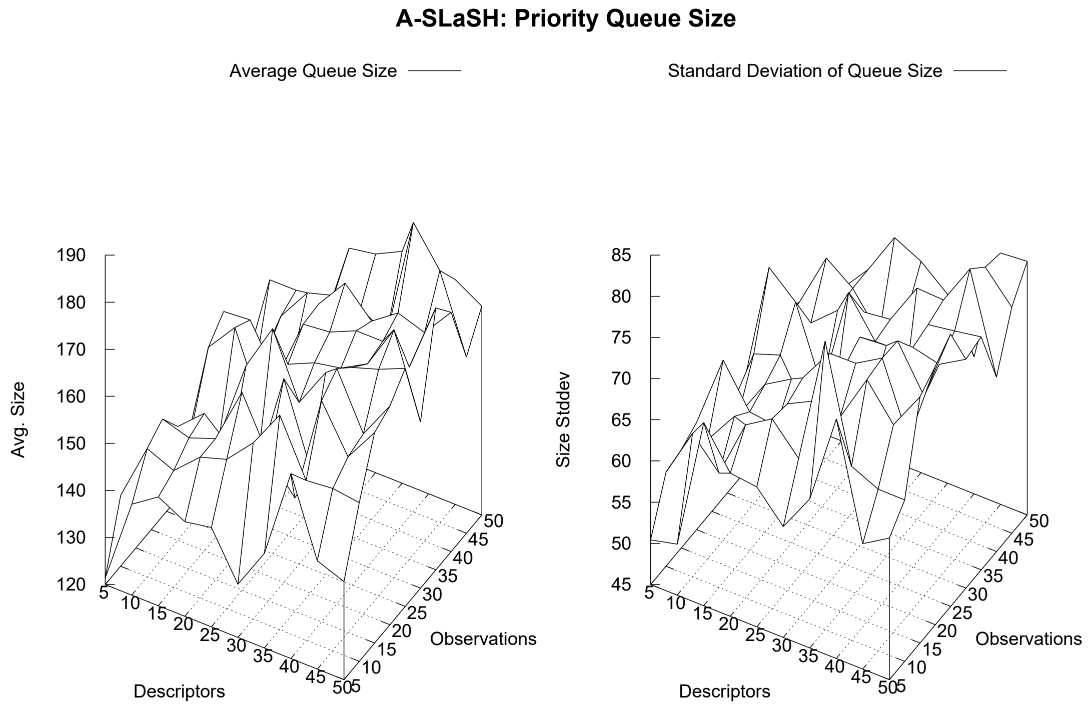

FIGURE 6.10: The effect of the number of input observations and descriptors on the maximum queue size in A-SLaSH [grid size: 100 × 100, pathfinding tasks: 100]

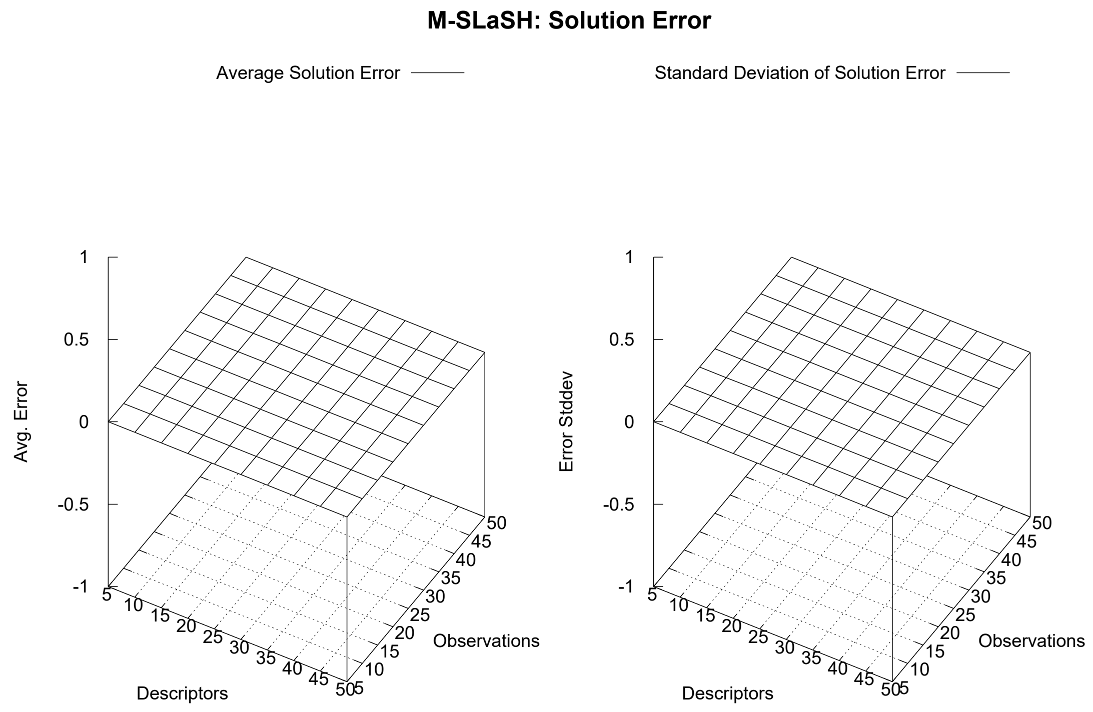

FIGURE 6.11: The solution error of M-SLaSH for different numbers of input observations and descriptors [grid size: 100 × 100, pathfinding tasks: 100]

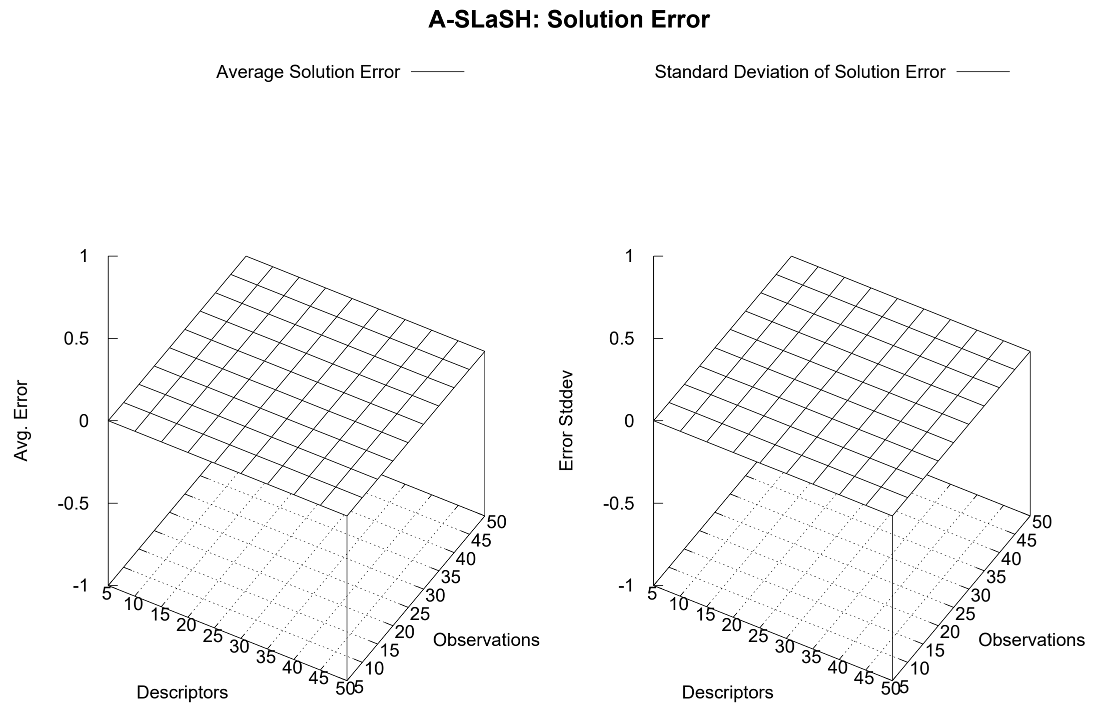

FIGURE 6.12: The solution error of A-SLaSH for different numbers of input observations and descriptors [grid size: 100 × 100, pathfinding tasks: 100]

Experiment 6.3. Comparing the performance of SLaSH and the classical algorithms.

Parameters: The sizes of both state descriptor and observed state sets were set to 50. State descriptors, observed states, and pathfinding tasks were generated randomly.

Course: The algorithms were examined for different grid sizes ranging from 10×10 to 200×200 with a step of 10. For each grid size, each algorithm was tested for 100 pathfinding tasks. The performance of algorithms was measured using the same parameters as in the previous experiment (Exp. 6.2). The procedure was repeated 10 times, and the measurements were averaged.

Results: Figure 6.13 shows the superiority of A* over the remaining algorithms in the number of visited states, which is not surprising since the algorithm is guided by an optimal heuristic estimator. The performance of SLaSH algorithms falls somewhere between Dijkstra and A*. It is a quite good result considering that a region lattice was generated using random input data.

The accompanying measurements of wall-clock times in Fig. 6.14 demonstrate only a slight impact of the overhead computation of the heuristic estimator on the overall performance of SLaSH algorithms. Potentially, the proposed algorithms may be inefficient for small problems, but as the computational cost of visiting a state grows, the reduction of the search space compensates for the additional computations.

While analysing maximum sizes of the priority queues presented by Fig. 6.15, it can be noted that the classical algorithms fall in the middle between SLaSH algorithms. Thus, A-SLaSH is characterised by the biggest theoretical memory consumption and M-SLaSH by the smallest one.

Figure 6.16 confirms that all the compared algorithms return optimal solutions for the considered state-space graph.

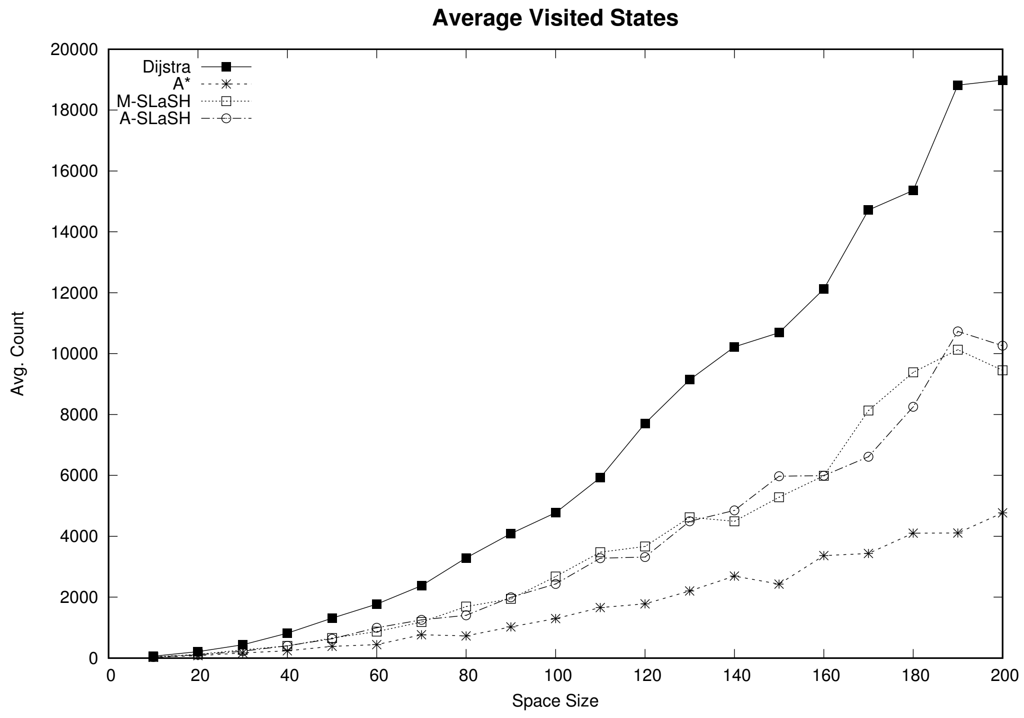

FIGURE 6.13: Comparing the number of states visited by the studied algorithms for an increasing size of the state space [state descriptors: 50, observed states: 50, pathfinding tasks: 100, repetitions: 10]

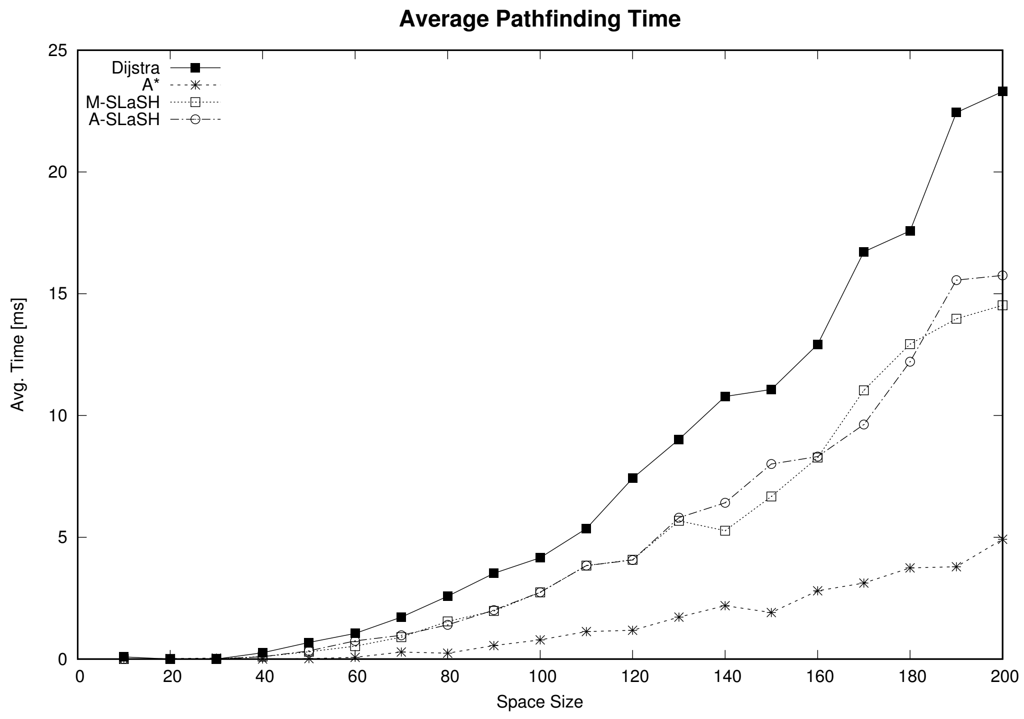

FIGURE 6.14: Comparing pathfinding times of the studied algorithms for an increasing size of the state space [state descriptors: 50, observed states: 50, pathfinding tasks: 100, repetitions: 10]

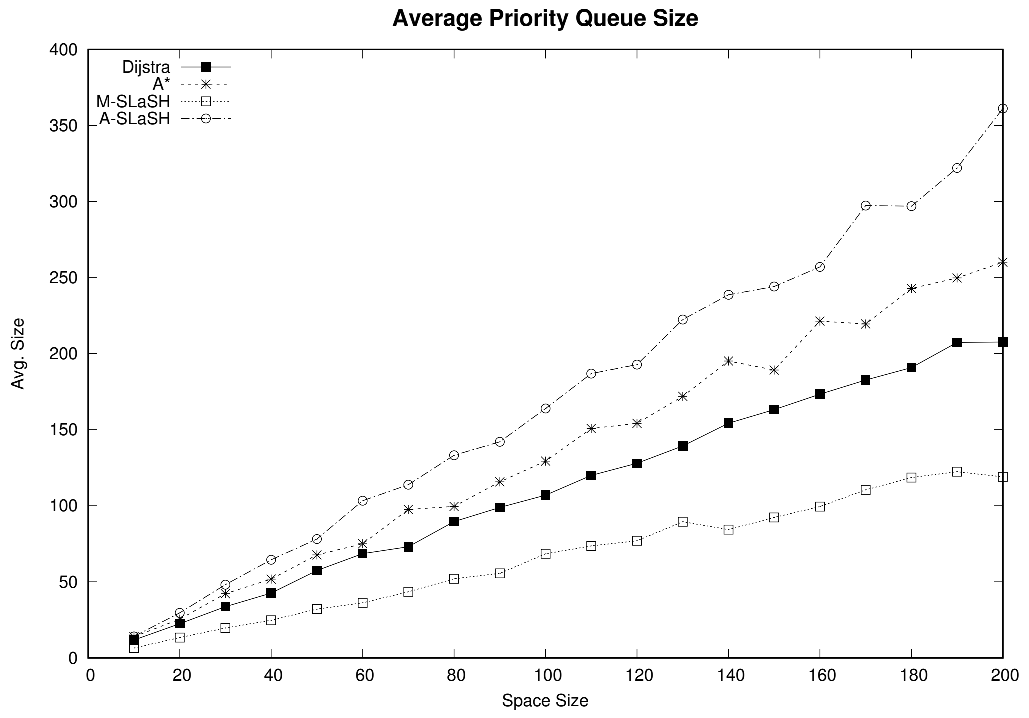

FIGURE 6.15: Comparing the maximum queue size in the studied algorithms for an increasing size of the state space [state descriptors: 50, observed states: 50, pathfinding tasks: 100, repetitions: 10]

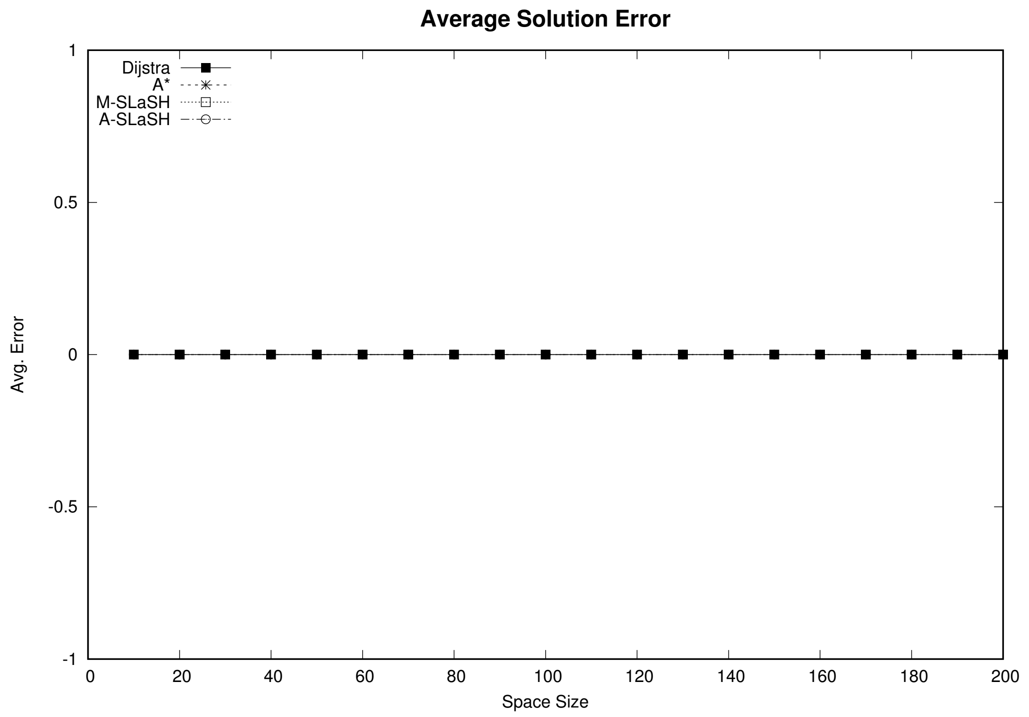

FIGURE 6.16: Comparing the solution error of the studied algorithms for an increasing size of the state space [state descriptors: 50, observed states: 50, pathfinding tasks: 100, repetitions: 10]

In the next experiment, capabilities of automatically adapting the space partitioning model to an arbitrary problem domain and maximising the performance of the proposed algorithms were examined. A region lattice was constructed using a set of state descriptors obtained from the output of a genetic algorithm. The experiment focuses on a single tuning run. The parameters of a genetic algorithm were chosen based on preliminary tests, which are not included in this work. They yielded satisfactory results, though potentially there is still room for improvement. However, the primary objective of this study was aimed at demonstrating the optimisation process itself, rather than finding an optimal set of tuning parameters as each application domain may require dedicated tuning settings.

Experiment 6.4. Examining the efficacy of the automated method of tuning state descriptors.

Parameters: The experiment was conducted for a grid of size 50 × 50. For constructing a lattice, a set of 20 random input states was used. The performance of SLaSH algorithms was measured for 10 random pathfinding tasks. Parameters steering the execution of the genetic algorithm were as follows:

- $popSize = 50$,
- $tournPerc = 10\%$,
- $crossProb = 0.6$,
- $crossFactor = 0.5$,
- $mutProb = 0.1$,
- $mutFactor = 5$,
- $elitist =$ true.

Each individual was carrying a set of 10 state descriptors in its chromosome.

Course: The procedure of state descriptors tuning was conducted for each of SLaSH algorithms independently. The stop condition was set to 2000 generations. In each generation, the following parameters were measured:

- average population fitness,
- the standard deviation of population fitness,

- performance parameters of a SLaSH algorithm relying on a region lattice constructed using state descriptors carried by the best individual observed.

Results: Figures 6.17 and 6.19 demonstrate the progression of population fitness over a number of generations. Based on the standard deviation of population fitness, it can be concluded that diversity inside the population was always preserved. The most dynamic development of the population occurred during the first couple of hundreds of generations. The same effect could be observed in other tuning runs.

An improving performance of SLaSH algorithms relying on state descriptors provided by the best individuals can be observed in Fig. 6.18 and Fig. 6.20. As intended, the number of visited states is decreasing in both cases. It can be noted that M-SLaSH also reduces the maximum size of the priority queue as the tuning progresses.

The final experiment complements the previous one by discussing obtained region lattices and comparing the performance of SLaSH algorithms before and after tuning.

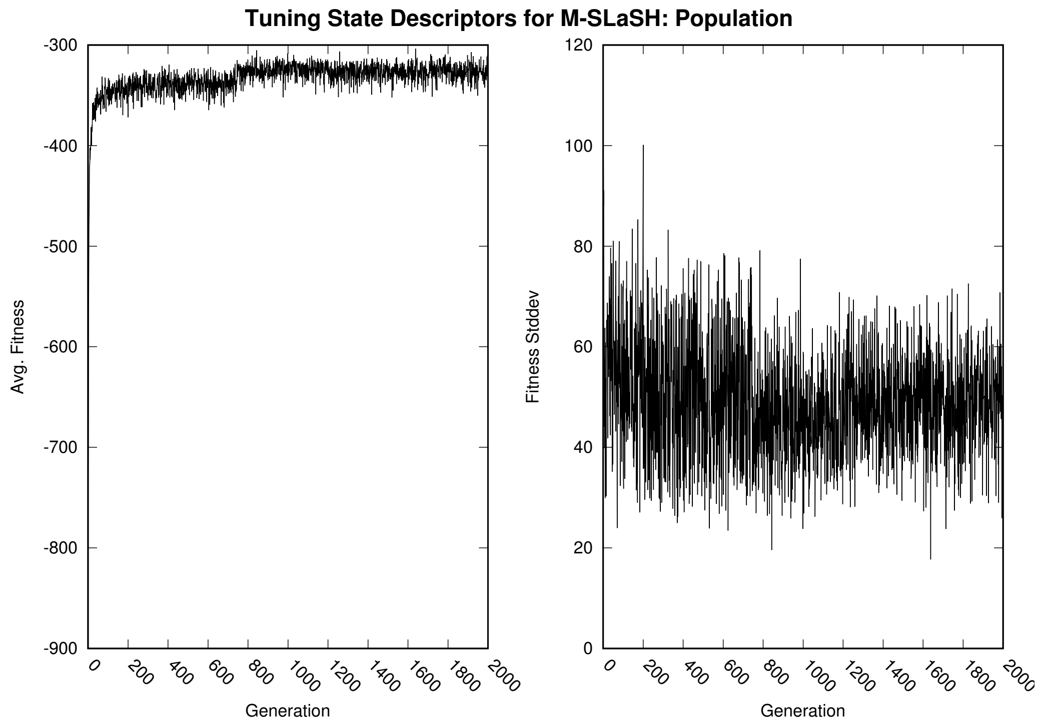

FIGURE 6.17: Statistics of a population tuned over generations by the genetic algorithm optimizing the performance of M-SLaSH [grid size: 50 × 50, *generations* = 2000, *popSize* = 50, *tournPerc* = 10%, *crossProb* = 0.6, *crossFactor* = 0.5, *mutProb* = 0.1, *mutFactor* = 5, *elitist* = true]

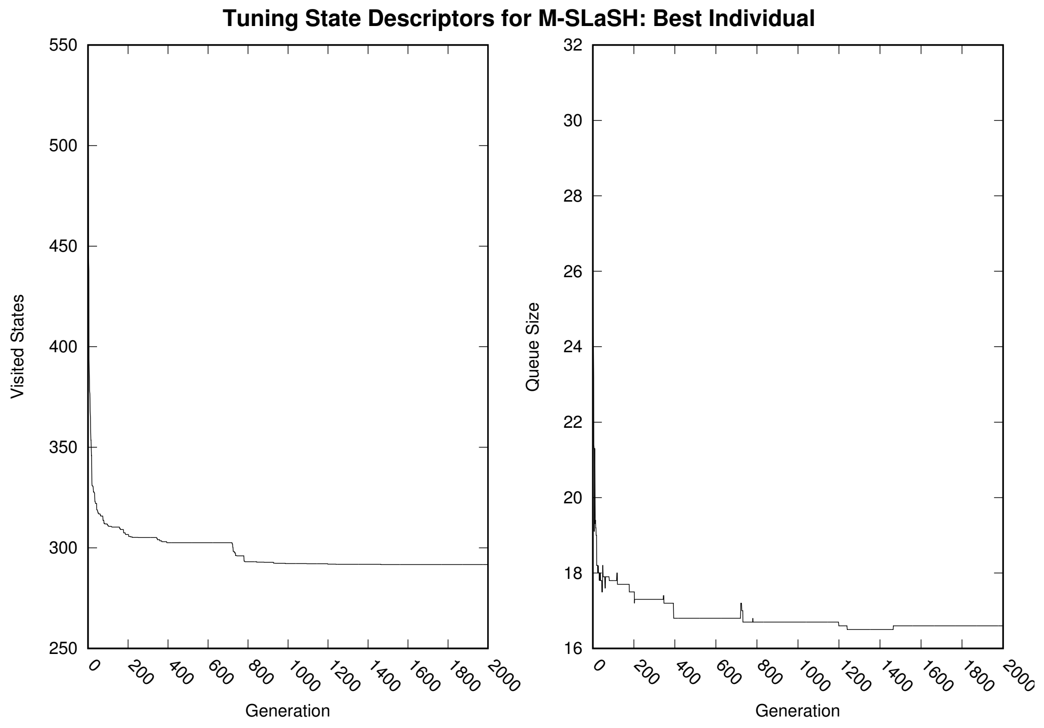

FIGURE 6.18: The performance of M-SLaSH measured for the best individual tuned over generations by the genetic algorithm [grid size: 50 × 50, *generations* = 2000, *popSize* = 50, *tournPerc* = 10%, *crossProb* = 0.6, *crossFactor* = 0.5, *mutProb* = 0.1, *mutFactor* = 5, *elitist* = true]

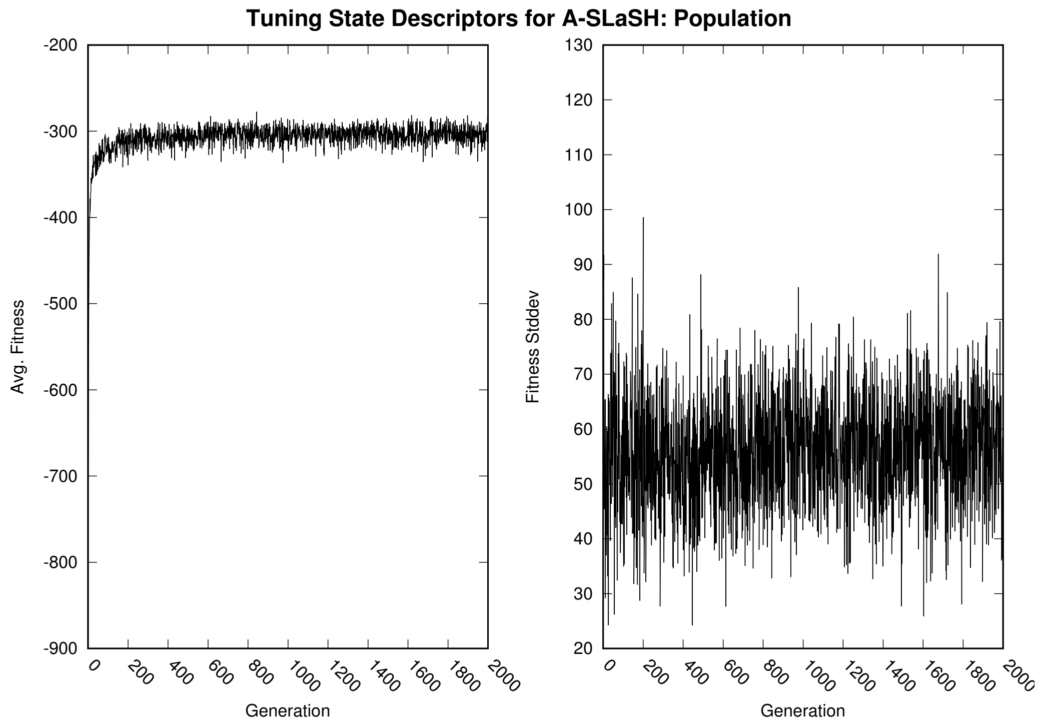

FIGURE 6.19: Statistics of a population tuned over generations by the genetic algorithm optimizing the performance of A-SLaSH [grid size: 50 × 50, *generations* = 2000, *popSize* = 50, *tournPerc* = 10%, *crossProb* = 0.6, *crossFactor* = 0.5, *mutProb* = 0.1, *mutFactor* = 5, *elitist* = true]

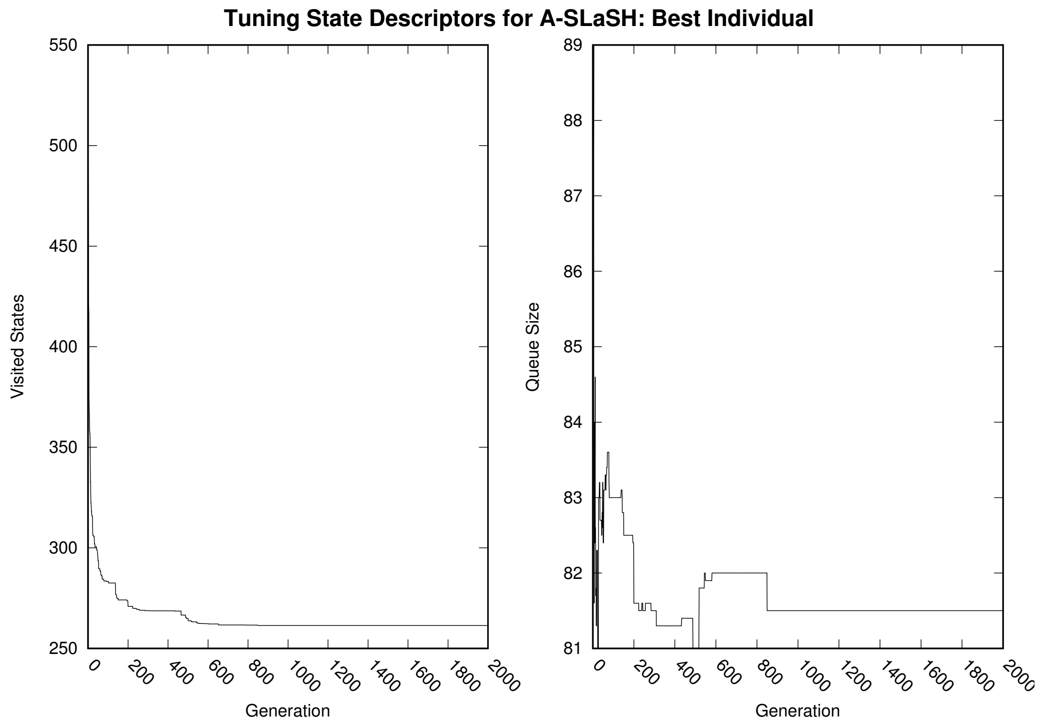

FIGURE 6.20: The performance of A-SLaSH measured for the best individual tuned over generations by the genetic algorithm [grid size: 50 × 50, *generations* = 2000, *popSize* = 50, *tournPerc* = 10%, *crossProb* = 0.6, *crossFactor* = 0.5, *mutProb* = 0.1, *mutFactor* = 5, *elitist* = true]

Experiment 6.5. Comparing the performance of SLaSH before and after tuning state descriptors.

Parameters: The experiment reuses parameters and tuning data collected in the previous experiment (Exp. 6.4). State search algorithms were tested for 10 randomly generated pathfinding tasks on a grid of size 50×50.

Course: The performance of SLaSH algorithms and the classical ones was compared for randomly generated state descriptors and the ones obtained from the output of the genetic algorithm. Performance measurements were averaged from 10 runs.

Results: An overall performance comparison of the examined algorithms is presented in Fig. 6.21. It is apparent that the tuning procedure considerably elevates the efficiency of the proposed algorithms. The proposed heuristic can compete with the optimal one that is employed by A*. For some rare cases, SLaSH algorithms can visit fewer states than A*, which was observed in other tuning runs. It is possible because the space partitioning can eliminate the symmetry problem (explained in Section 3.2) by forming a segment that leads directly to the goal state.

Interesting remarks can be made by studying the partitioning of the state space during the tuning process. Figure 6.22 contains two lattice diagrams. The left one comes from the first generation, in which the population was initialized randomly. The right one refers to the best individual in the final iteration of the tuning process. Digits in each node represent the number of observed states that a particular region covers. Based on these two pictures, it can be concluded that the final lattice is more extensive than the initial one, and the observed states are evenly distributed on each diagram level.

The partitioning of the space is visualised in Fig. 6.23 – the two illustrations correspond to the previously discussed lattice diagrams. In the drawings, the rectangular outlines represent the regions. The grey squares are the observations. Pathfinding tasks are depicted as the black diamonds and triangles connected with dotted lines. It can be discerned that in the final partitioning, the regions are moved onto the observations so the outlines that do not cover grey squares are eliminated. It also gives a perception of a more uniform distribution of the regions – they are not accumulated in one place.

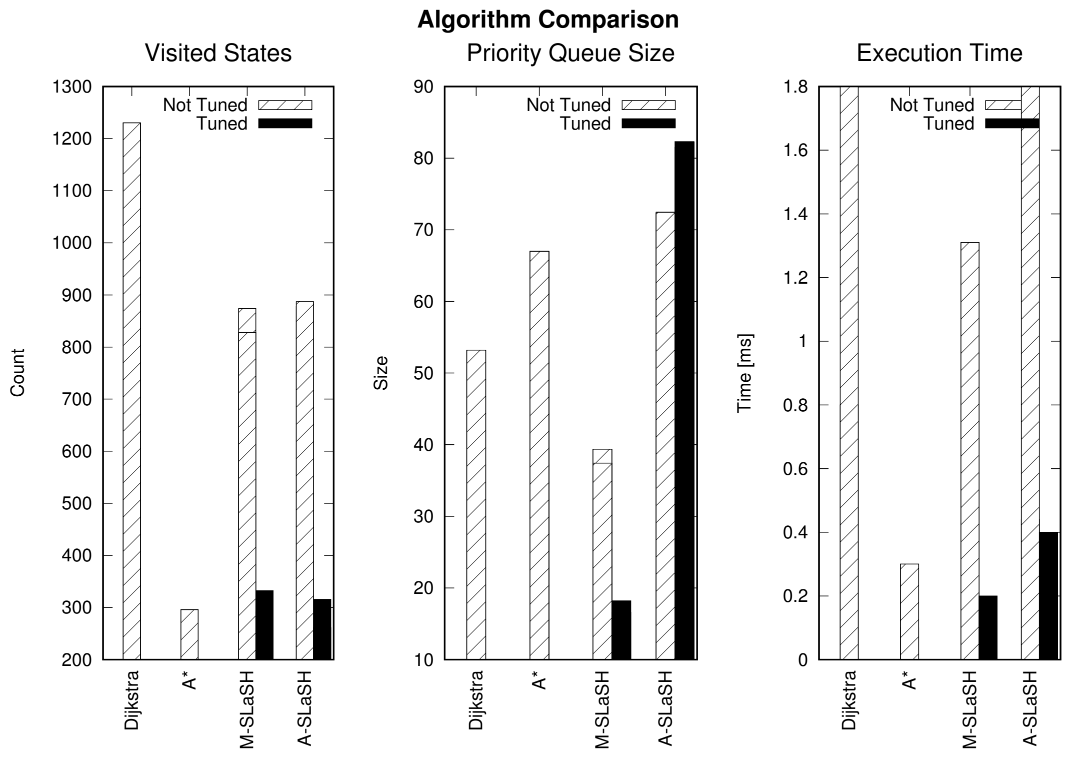

FIGURE 6.21: An overall performance comparison of the studied algorithms before and after tuning [grid size: 50 × 50, *generations* = 2000, *popSize* = 50, *tournPerc* = 10%, *crossProb* = 0.6, *crossFactor* = 0.5, *mutProb* = 0.1, *mutFactor* = 5, *elitist* = true]

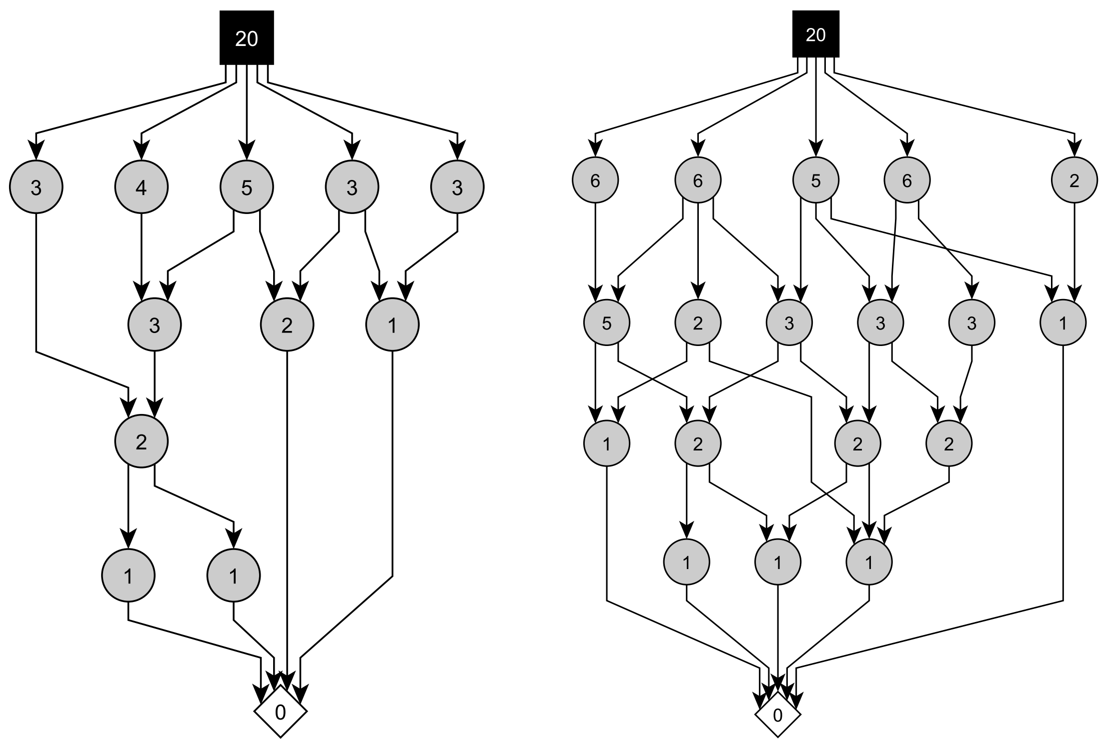

FIGURE 6.22: State-space lattices at the beginning of tuning (left) and at the end of it (right) [grid size: 50 × 50, *generations* = 2000, *popSize* = 50, *tournPerc* = 10%, *crossProb* = 0.6, *crossFactor* = 0.5, *mutProb* = 0.1, *mutFactor* = 5, *elitist* = true]

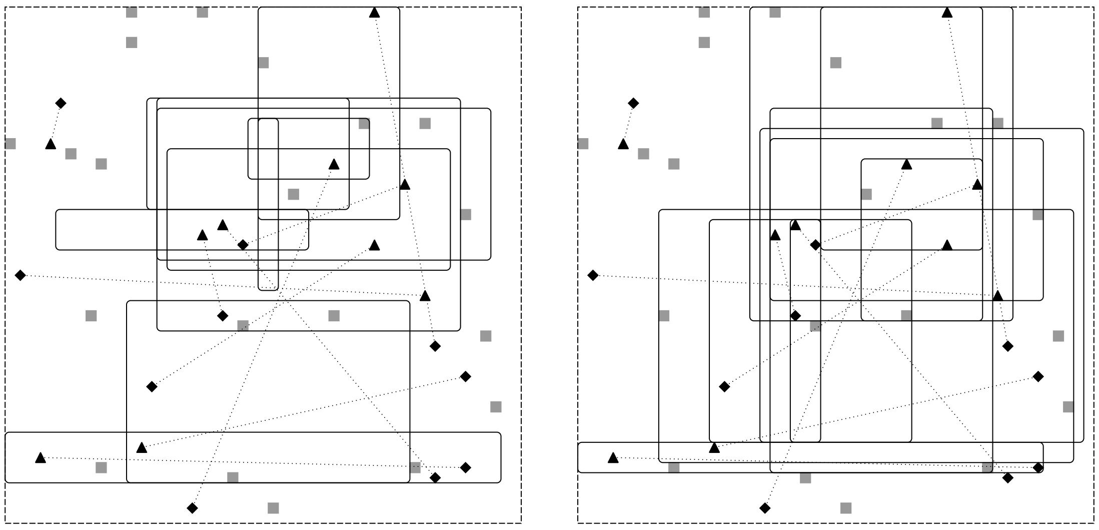

Figure 6.23: Regions of the state-space at the beginning of tuning (left) and at the end of it (right) [grid size: 50 × 50, *generations* = 2000, *popSize* = 50, *tournPerc* = 10%, *crossProb* = 0.6, *crossFactor* = 0.5, *mutProb* = 0.1, *mutFactor* = 5, *elitist* = true]

Chapter 7

Summary

The chapter summarises the researched approach. It outlines the main features of the proposed method. The results of the experimental study are compiled. The impact on planning in video games is discussed. The original contribution of the work is emphasised. In the final part of the chapter, promising development directions of the method are indicated.

7.1 Conclusions

The dissertation explores the subject of supporting the planning process in video games by information extracted from plan traces. Planning problems addressed in this work belong to the class of classical planning and involve combinatorial optimisation. Plan traces contain human player actions, which are represented as sequences of complete-information game states. The developed planning method operates on an algorithmic level of state search in a state-space graph.

The literature review shows that planning in video games can significantly improve the believability of an AI player, but it also brings many performance-related issues. The idea of analysing game replays, which are a potential source of game replays, is a fresh and promising direction. However, processing numerous atomic states in game replays brings similar challenges as the analysis of big data. Thus, mining information useful for planning requires special means.

The dissertation describes the development of the approach which evolved from a simple concept of reusing game replays. The initial studies showed the importance of state grouping and state-space partitioning. Lessons learned helped to develop the model of a region tree partitioning the state space and a heuristic estimator taking advantage of the model. However, the method was characterised by several drawbacks, which were addressed in its next version.

In the final development stage of the proposed approach a novel method of partitioning the state space. It is based on the formal foundations of FCA. It employs a concept (Galois) lattice to model the structure of regions of the state space as a state-space lattice. The routine of building such a lattice takes on input a set of states, which are obtained from plan traces, and state descriptors, which were introduced in this course of studies to define regions of the state space implicitly. State descriptors can be rigidly defined by the system designers or automatically obtained using optimisation methods. In this study, a Genetic Algorithm was employed to automate the procedure.

The extracted model is hierarchical, and it is utilised for estimating the distance between nodes in a state-space graph. The proposed region lattice distance estimate is calculated as the number of regions that must be traversed from the current state to reach the goal state. The main advantage of the estimate is that it is applicable for metric and non-metric spaces as well. Therefore, it can be used for solving complex planning problems. Optimal solutions are expected if state descriptors encompass convex state sets.

The distance estimate is employed by the two proposed state search algorithms – *State-space Lattice Search Heuristics* (SLaSH). The first of them, M-SLaSH is a variant of best-first search that is characterised by a significantly reduced usage of the memory if the estimate is optimal. The second algorithm, A-SLaSH is less restrictive, but it has a typical memory consumption.

In the experimental study, the proposed algorithms were compared with A* and Dijkstra's algorithm. A* was guided by a heuristic function formed on the basis of domain-specific knowledge. It can be considered as one of the most efficient state search methods. On the other hand, Dijkstra's algorithm represents uninformed search. It is the worst-case scenario in which a heuristic function is unavailable, or it is ineffective. The experiments demonstrated that all algorithms achieve better performance than uninformed search. The proposed methods closely competed with A* although they did not rely on domain-specific cost estimate. In the final phase, state descriptors were automatically tuned by a genetic algorithm to optimise the performance of the heuristic.

One of the biggest challenges that were encountered during the development of the approach was the absence of tools aimed at the undertaken research direction. Apart from Smart Blocks, a testbed environment designed for conducting early experiments, there were no games as benchmark environments intended for comparing classical planning methods [75]. The lack of such environments is most likely caused by a high cost of developing a fully fledged game. Traditional planning benchmarks focus on specific manipulation problem instances rather than environments defining a domain of planning problems. Employing the developed method for solving a single problem instance may be seen senseless as a game replay already carries a plan trace, which is a solution to the problem. Therefore, the method is dedicated to games characterised by a large number of possible planning problem instances. In such games, information extracted from a database of game replays can be utilised to support solving planning problems that were not previously observed in the database. Testing the method in a commercial game is one of the future goals.

Another complication that occurred during the study followed from the methodology of conducting experiments involving plan traces as a product of human actions. It is because the performance of the proposed method depends on the quality of the collected data, and there were no benchmark datasets for studying such approach in the area of classical planning. Ideally, the data can be obtained from volunteers participating in an experiment as it was in the case of Smart Blocks. However, the procedure must be repeated if the rules of a testbed environment are modified, which usually happens when obtained results are validated. This time-consuming process can be avoided by producing plan traces artificially. However, it appears to be a non-trivial problem since the research is addressed to solving complex planning problems. Therefore, in the final experiments, the data was generated randomly to test the proposed method against theoretically the worst-case scenario.

Employing the discussed planning method is not restricted to specific game genres as the modern games include mixtures of different mechanics, in which planning can potentially be one of them. The technical aspects of applying the approach in practice are similar to the ones of using a standard domain-specific planning system such as GOAP [122]. The implementation of a video game must allow for simulating future states of the game. Such a design is characterised by encapsulating logic data in an efficient state representation. It is important that recorded game replays must contain a full-information state for each discrete step of a match. A game should run in a simulation mode, so it is possible to obtain state descriptors by using a genetic algorithm automatically. Alternatively, a set of state descriptors can be predefined rigidly.

The impact of the presented work on planning in video games can be understood as opening another door leading to new opportunities in the area of building an intelligent agent. Such an agent can reuse information stored in game replays to accelerate solving classical planning problems. At the same moment, the system gains flexibility as it requires less knowledge that is rigidly predefined by the designers. In the domain of strictly limited computational resources, each performance improvement makes room for addressing more complex problems. This is crucial for agent believability as the difficulty of challenges the agent faces speaks for his intelligence.

The literature review did not result in finding a case of applying FCA to the field of automatic planning as something different than just an analytical tool. This work demonstrates a new and original use of FCA for planning. Although a concept lattice employed for modelling the state space already proved its usefulness, many interesting applications of the model are still unresearched. The most promising development directions are discussed in Section 7.3.

7.2 Contributions

To summarise, the original contribution of this work is:

- Implementing the idea of supporting classical planning by information extracted from **unannotated** plan traces.
- Employing FCA to build a **state-space lattice** as the model of hierarchical partitioning of a state-space graph, which can be constructed from input plan traces **incrementally**.
- Introducing the notion of **state descriptors** for specifying regions of the state space **implicitly**, which avoids storing states in memory.
- Proposing a **novel heuristic** for estimating the distance between states based on a state-space lattice.
- Developing two state search algorithms that utilise the heuristic (State-space Lattice Search Heuristics, **SLaSH**).
- Employing a genetic algorithm to **automatically** tune state descriptors and optimise the performance of SLaSH algorithms.
- Developing a **video game** as a testbed environment and conducting an experimental study finalised with the analysis of the results.

7.3 Future Work

The proposed method of partitioning the state space can potentially contribute to the domain of adversarial games where it can be employed for modelling the opponent. Learning the behaviour of a human adversary can significantly increase the chances of winning the competition. Alternatively, the extracted knowledge can be utilised for imitating human-like behaviours.

A state-space lattice is a model that can be efficiently utilised for locating regions in the state space which were frequently visited by the player based on plan traces. Transitions between the regions observed across a number of games are valuable information for sequential pattern mining [137]. Because of a hierarchical structure of the proposed model, player actions can be viewed at different abstraction levels and generalised. Recognising a pattern that the player is following can be used for identifying his current goal, plan, and strategy [138]. Pattern recognition plays a significant role in predicting future actions of the opponent, which may be referred to as a single player or a cooperating team.

Apart from supporting planning and decision processes, the proposed model can also be a foundation for developing new analytical tools for collecting statistical information about the players as system users. Game replays have the form of long sequences of atomic states and actions that are difficult to interpret. The proposed model can be used to enhance their readability by structuring and annotating them automatically. It can help game designers to understand interactions of the players with the virtual environment and locate potential problems.

Appendix A

Appendix

The supplementary materials contain pseudocodes and brief descriptions of algorithms that have been implemented and used in the experimental study.

1 Dijkstra's Algorithm Pseudocode

The pseudocode of Uniform-Cost Search (UCS), which is a practical variant of Dijkstra's algorithm, is presented in Alg. 8 [28]. The algorithm accepts on input:

- *start* – an initial node,
- *goal* – a final node.

The procedure begins by initializing the following data structures:

- *cost*[] – a cost dictionary that maps a node into the cost of reaching it,
- *previous*[] – a path dictionary for recreating a path from the final node to the initial one,
- *frontier* – a priority queue that sorts nodes by their cost in ascending order,
- *explored* – a set of previously visited nodes.

Alg. 8: Dijkstra(*start*, *goal*)

```
1  var $cost[start] \leftarrow 0$                              ▷ initialize a cost dictionary
2  var $previous[start] \leftarrow \epsilon$                   ▷ initialize a path dictionary
3  var $frontier \leftarrow \{\langle start, cost[start]\rangle\}$   ▷ initialize a min-priority queue
4  var $explored \leftarrow \{\}$                              ▷ initialize an empty set
5  while $frontier \neq \{\}$ do
6      var $node \leftarrow$ Pop($frontier$)                   ▷ remove and take a node with the lowest cost
7      if $node = goal$ then
8          return GetSolution($cost$, $previous$)              ▷ return the solution cost and path
9      Add($explored$, $node$)
10     foreach $n \in$ GetNeighbours($node$) do
11         if $n \in explored$ then
12             continue
13         var $c \leftarrow cost[node]$ + GetWeight($node$, $n$)
14         if $n \in cost \wedge c \geq cost[n]$ then
15             continue
16         if $n \in frontier$ then
17             Remove($frontier$, $n$)
18         $cost[n] \leftarrow c$
19         $previous[n] \leftarrow node$
20         Add($frontier$, $\langle n, c\rangle$)
21 return $failure$                                            ▷ solution not found
```

In the main loop, the algorithm populates nodes from the priority queue until the goal node is found, or the queue is empty (lines 5-8). In the same block, nodes are marked as visited (line 9). Neighbours of each node are expanded in a sub-loop (line 10). However, the previously visited ones are ignored (lines 11-12). Next, the cost of reaching a node is calculated as the sum of the accumulated cost and the weight associated with the edge between the currently expanded node and its neighbour in the graph (line 13). If a node has been reached with a higher cost than before, then it is ignored, and the sub-loop continues from its starting point (lines 14-15). The cost determines the order in the priority queue. If a node is already present in the queue, then it is removed (lines 16-17) and later added again to update its position inside the queue according to its new cost. The associated cost and the previous node are set (lines 18-19). In the final part, the current node is added to the priority queue (line 20).

2 A* Pseudocode

The pseudocode of A* presented in Alg. 9 is almost identical to UCS, which was discussed previously (Alg. 8) [139]. The difference comes from the fact that a heuristic distance to the goal node is estimated for each explored node (lines 3 and 21). The estimated distance is then summed with the cost associated with reaching a node and used for determining the order in the priority queue (line 24).

Alg. 9: AStar(*start*, *goal*)

```
1  var $cost[start] \leftarrow 0$                                  ▷ initialize a cost dictionary
2                                                 ▷ initialize a dictionary for heuristic values
3  var $h[start] \leftarrow$ GetHDistance($start$, $goal$)
4  var $previous[start] \leftarrow \epsilon$                          ▷ initialize a path dictionary
5  var $frontier \leftarrow \{\langle start, cost[start] + h[start]\rangle\}$    ▷ initialize a min-priority queue
6  var $explored \leftarrow \{\}$                                  ▷ initialize an empty set
7  while $frontier \neq \{\}$ do
8      var $node \leftarrow$ Pop($frontier$)          ▷ remove and take a node with the lowest cost
9      if $node = goal$ then
10         return GetSolution($cost$, $previous$)        ▷ return the solution cost and path
11     Add($explored$, $node$)
12     foreach $n \in$ GetNeighbours($node$) do
13         if $n \in explored$ then
14             continue
15         var $c \leftarrow cost[node]$ + GetWeight($node$, $n$)
16         if $n \in cost \wedge c \geq cost[n]$ then
17             continue
18         if $n \in frontier$ then
19             Remove($frontier$, $n$)
20         else
21             $h[n] \leftarrow$ GetHDistance($n$, $goal$)
22         $cost[n] \leftarrow c$
23         $previous[n] \leftarrow node$
24         Add($frontier$, $\langle n, cost[n] + h[n]\rangle$)
25 return $failure$                                            ▷ solution not found
```

3 Bordat's Algorithm Pseudocode

The main routine of Bordat's algorithm is expressed by pseudocode in Alg. 10 [55]. It is a recursive procedure that is started by invoking Bordat($\langle G, G' \rangle$, G', $\emptyset$), where:

- G is a set of all objects,
- G' refers to Galois operation on G, which gives all attributes that are valid for all objects in G,
- $\langle G, G' \rangle$ represents supremum, the topmost node in a lattice.

Alg. 10: Bordat($\langle A, B \rangle$, C, *Lattice*)

```
1  $Lattice \leftarrow Lattice \cup \{\langle A, B \rangle\}$
2  var $LN \leftarrow$ LowerNeighbours($\langle A, B \rangle$)
3  foreach $\langle D, E \rangle \in LN$ do
4      if $C \cap E = B$ then
5          AddSubconcept($\langle A, B \rangle$, $\langle D, E \rangle$)
6          AddSuperconcept($\langle D, E \rangle$, $\langle A, B \rangle$)
7          $C \leftarrow C \cup E$
8          Bordat($\langle D, E \rangle$, $C$, $Lattice$)
9      else
10         $\langle D, E \rangle \leftarrow$ FindNode(Sup($Lattice$), $\langle D, E \rangle$)
11         AddSubconcept($\langle A, B \rangle$, $\langle D, E \rangle$)
12         AddSuperconcept($\langle D, E \rangle$, $\langle A, B \rangle$)
```

In the beginning, the argument $\langle A, B \rangle$ is added to the lattice as a new concept (line 1). Next, the lower neighbours (subconcepts) LN of the argument concept $\langle A, B \rangle$ are populated (line 2). The main loop of the procedure iterates over all concepts in LN (line 3). The intersection between the sets of argument attributes C and lower neighbour attributes E is calculated to check whether the iterated concept $\langle D, E \rangle$ already exists in the structure (line 4). If the result equals the attributes B of the argument concept then the iterated concept $\langle D, E \rangle$ is added to the structure as a new lower node of $\langle A, B \rangle$. Subconcept-superconcept relation between the two concepts is established (lines 5-6). Then, the argument attributes C are enlarged by the lower neighbour attributes E (line 7). The routine is recursively invoked (line 8). If the previously considered condition is not satisfied, then the lower neighbour $\langle D, E \rangle$ already exists in the structure. The iterated concept $\langle D, E \rangle$ must be found to establish the relation with the argument concept $\langle A, B \rangle$

Alg. 11: LowerNeighbours($\langle A, B\rangle$)

```
1  var $LN \leftarrow \emptyset$                                  ▷ $LN$ stores the result
2  var $C \leftarrow B$
3                                  ▷ an infinite loop (interrupted internally)
4  while do
5          ▷ $d_i$ is the first element in $A$ such that $\neg(\{d\}' \subseteq C)$
6      var $d_i \leftarrow$ First($\{d \in A : \neg(\{d\}' \subseteq C)\}$)
7                          ▷ the loop is interrupted if $d_i$ does not exist
8      if $d_i = \epsilon$ then
9          break
10     var $E \leftarrow \{d_i\}$
11     var $F \leftarrow E'$
12                             ▷ loop until $d_i$ is the last element in $A$
13     while $d_i \neq d_n$ do
14                                 ▷ $d_i$ becomes the next element in $A$
15         $d_i \leftarrow d_{i+1}$
16         if $\neg(F \cap \{d_i\}' \subseteq C)$ then
17             $E \leftarrow E \cup \{d_i\}$
18             $F \leftarrow F \cap \{d_i\}'$
19     if $F \cap C = B$ then
20         $LN \leftarrow LN \cup \{\langle E, F\rangle\}$
21     $C \leftarrow C \cup F$
22 return $LN$
```

(lines 11-12). The search begins from the topmost node in the lattice and proceeds to the lowermost one (line 10).

The procedure of populating a list of lower neighbours of a given concept is shown in Alg. 11. The pseudocode comprises a sequence of explicitly defined operations on sets of objects and attributes. They include Galois operations ($'$). Galois operator located on a set of objects produces a set of attributes that are valid for all specified objects. For a set of attributes, the operator gives a set of objects such that each object has the specified attributes.

Alg. 12: FindNode($\langle A, B\rangle$, $\langle D, E\rangle$)

```
1 var $\langle E, F\rangle \leftarrow$ First($\{\langle E, F\rangle \in$ GetLower($\langle A, B\rangle$) $: F \subseteq E\}$)
2 if $F \neq E$ then
3     FindNode($\langle E, F\rangle$, $\langle D, E\rangle$)
4 else
5     return $\langle E, F\rangle$
```

A node existing in the structure can be easily found by iterating over each concept in a lattice. A more efficient method is presented in Alg. 12. The procedure begins searching for a concept $\langle D, E \rangle$ from a starting node $\langle A, B \rangle$ – both are provided as the arguments. The algorithm selects the first lower neighbour $\langle E, F \rangle$ of the starting node $\langle A, B \rangle$ such that the neighbour attributes F are a subset of the argument attributes E (line 1). If the sets of attributes F and E are not equal (line 2), then the routine is invoked recursively and $\langle E, F \rangle$ becomes a new starting node (line 3). Otherwise, the concept $\langle D, E \rangle$ is located and returned (line 5).

Bibliography

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 3 edition, 2009. ISBN 0136042597.

[2] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*, chapter Chapter 1 - Introduction and Overview, pages 1–16. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2004. ISBN 978-1-55860-856-6. doi: 10.1016/B978-155860856-6/50004-1.

[3] R.J. Brachman and H.J. Levesque. *Knowledge Representation and Reasoning*. The Morgan Kaufmann Series in Artificial Intelligence Series. Morgan Kaufmann, 2004. ISBN 9781558609327.

[4] Emil Keyder and Blai Bonet. Heuristics for planning, 2009. URL `http://icaps09.icaps-conference.org/tutorials/tut1.pdf`. Tutorial Presentation, 19th International Conference on Automated Planning and Scheduling.

[5] Daniel Harabor. Beyond A*: Speeding up pathfinding through hierarchical abstraction, 2009. URL `https://harablog.files.wordpress.com/2009/06/beyondastar.pdf`. Tutorial Presentation, NICTA & The Australian National University.

[6] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[7] Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dana S. Nau, Dan Wu, and Fusun Yaman. SHOP2: an HTN planning system. *Computing Research Repository*, abs/1106.4869, 2011.

[8] Phillip. Gowlett, Defence Science, and Technology Organisation (Australia). *Moving forward with Computational Red Teaming*. Joint Operations Division, Defence Science and Technology Organisation Canberra, 2011.

[9] Eric Jacopin. Game ai planning analytics: The case of three first-person shooters, 2014.

[10] A.M. Uhrmacher and D. Weyns. *Multi-Agent Systems: Simulation and Applications*. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, 2009. ISBN 9781420070248.

[11] V. Dignum. *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models: Semantics and Dynamics of Organizational Models*. Information Science Reference, 2009. ISBN 9781605662572.

[12] Rosario Girardi and Adriana Leite. A survey on software agent architectures. *IEEE Intelligent Informatics Bulletin*, 14(1):8–20, 2013.

[13] Tadiou Mamadou Kone, Akira Shimazu, and Tatsuo Nakajima. The state of the art in agent communication languages. *Knowledge and Information Systems*, 2(3): 259–284, 2000. ISSN 0219-1377. doi: 10.1007/PL00013712.

[14] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2008. ISBN 0521899435.

[15] A.J. Champandard. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. NRG Series. New Riders, 2003. ISBN 9781592730049.

[16] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*, chapter Chapter 2 – Representations for Classical Planning, pages 1–16. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2004. ISBN 978-1-55860-856-6. doi: 10.1016/B978-155860856-6/50006-5.

[17] Antonio Garrido and Eva Onaindia. *Artificial Intelligence for Advanced Problem Solving Techniques*, chapter Extending Classical Planning for Time: Research Trends in Optimal and Suboptimal Temporal Planning. IGI Global, 2008. doi: 10.4018/978-1-59904-705-8.ch002.

[18] M. Mausam and Andrey Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool, 2012. doi: 10.2200/S00426ED1V01Y201206AIM017.

[19] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015, 2015. ISSN 1687-7047. doi: 10.1155/2015/736138.

[20] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1):65–100, 2009. ISSN 1573-0409. doi: 10.1007/s10846-009-9383-1.

[21] Caelan Reed Garrett, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Backward-forward search for manipulation planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

[22] Tânia Marques and Michael Rovatsos. Classical planning with communicative actions. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, pages 1744–1745, 2016. doi: 10.3233/978-1-61499-672-9-1744.

[23] Greg Barish and Craig A. Knoblock. Speculative plan execution for information gathering. *Artificial Intelligence*, 172(4):413–453, 2008. ISSN 0004-3702. doi: 10.1016/j.artint.2007.08.002.

[24] Maria Caridi and Andrea Sianesi. Multi-agent systems in production planning and control: An application to the scheduling of mixed-model assembly lines. *International Journal of Production Economics*, 68(1):29–42, 2000. ISSN 0925-5273. doi: 10.1016/S0925-5273(99)00097-3.

[25] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN 0-201-05594-5.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

[27] Rudiger Ebendt and Rolf Drechsler. Weighted a* search – unifying view and application. *Artificial Intelligence*, 173(14):1310 – 1342, 2009. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/j.artint.2009.06.004.

[28] Ariel Felner. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In Daniel Borrajo, Maxim Likhachev, and Carlos Linares Lopez, editors, *Proceedings, The Fourth International Symposium on Combinatorial Search*, pages 47–51. AAAI Press, 2011.

[29] Jorg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[30] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artifical Intelligence*, 27(1):97–109, 1985. doi: 10.1016/0004-3702(85)90084-0.

[31] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620. Morgan Kaufmann Publishers Inc., 1971.

[32] Krystian Jobczyk and Antoni Ligeza. Strips in some temporal-preferential extension. In *International Conference on Artificial Intelligence and Soft Computing*, pages 241–252. Springer Berlin Heidelberg, 2017.

[33] Daniel L. Kovacs. A multi-agent extension of PDDL3.1. In *Proceedings of the 3rd Workshop on the International Planning Competition*, pages 19—-27, 2012.

[34] *Tight Bounds for HTN Planning with Task Insertion*, 2015.

[35] Amanda Jane Coles, Andrew Coles, Angel Garcia Olaya, Sergio Jimenez Celorrio, Carlos Linares Lopez, Scott Sanner, and Sungwook Yoon. A survey of the seventh international planning competition. *AI Magazine*, 33(1), 2012.

[36] N. Lipovetzky. *Structure and Inference in Classical Planning.* LULU Press, 2014. ISBN 9781312466210.

[37] Peter I. Cowling, Michael Buro, Michal Bida, Adi Botea, Bruno Bouzy, Martin V. Butz, Philip Hingston, Hector Munoz-Avila, Dana S. Nau, and Moshe Sipper. Search in real-time video games. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 1–19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. ISBN 978-3-939897-62-0. doi: 10.4230/DFU.Vol6.12191.1.

[38] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Lubos Brim, Stefan Edelkamp, Erik A. Hansen, and Peter Sanders, editors, *Graph Search Engineering*, number 09491 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl, 2010.

[39] Daniel Bryce and Subbarao Kambhampati. A tutorial on planning graph based reachability heuristics. *AI Magazine*, 28(1):47–83, 2007.

[40] Emil Keyder, Jorg Hoffmann, and Patrik Haslum. Semi-relaxed plan heuristics. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, pages 2126–2128. AAAI Press, 2012.

[41] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1007–1012. AAAI Press, 2007. ISBN 978-1-57735-323-2.

[42] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *The Journal of Artificial Intelligence Research*, 39 (1):127–177, 2010. ISSN 1076-9757.

[43] Patrik Haslum, Blai Bonet, and Hector Geffner. New Admissible Heuristics for Domain-Independent Planning. In *National Conference on Artificial Intelligence (AAAI)*, 2005.

[44] Jendrik Seipp and Malte Helmert. Diverse and additive cartesian abstraction heuristics, 2014.

[45] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[46] W. Hewlett. Partially precomputed A*. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):119–128, 2011. ISSN 1943-068X. doi: 10.1109/TCIAIG.2011.2111250.

[47] Rudolf Wille. *Formal Concept Analysis: Foundations and Applications*, chapter Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies, pages 1–33. Springer-Verlag, 2005. ISBN 978-3-540-31881-1. doi: 10.1007/11528784_1.

[48] Garrett Birkhoff. *Lattice Theory*, volume 25 of *American Mathematical Society colloquium publications*. American Mathematical Society, 1940. ISBN 9780821810255.

[49] Rudolf Wille. *Ordered Sets: Proceedings of the NATO Advanced Study Institute held at Banff, Canada, August 28 to September 12, 1981*, chapter Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts, pages 445–470. Springer Netherlands, 1982. ISBN 978-94-009-7798-3. doi: 10.1007/978-94-009-7798-3_15.

[50] Marek Obitko, Václav Snásel, and Jan Smid. Ontology design with formal concept analysis. In Václav Snásel and Radim Belohlávek, editors, *CLA*, volume 110 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[51] Uta Priss. *Linguistic Applications of Formal Concept Analysis*, pages 149–160. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31881-1. doi: 10.1007/11528784_8.

[52] Lotfi Lakhal and Gerd Stumme. *Efficient Mining of Association Rules Based on Formal Concept Analysis*, pages 180–195. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31881-1. doi: 10.1007/11528784_10.

[53] Thomas Tilley, Richard Cole, Peter Becker, and Peter Eklund. *A Survey of Formal Concept Analysis Support for Software Engineering Activities*, pages 250–271. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31881-1. doi: 10.1007/11528784_13.

[54] Rudolf Wille. *Conceptual Knowledge Processing in the Field of Economics*, pages 226–249. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-31881-1. doi: 10.1007/11528784_12.

[55] Sergei O. Kuznetsov and Sergei A. Obiedkov. *Algorithms for the Construction of Concept Lattices and Their Diagram Graphs*, pages 289–300. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-44794-8. doi: 10.1007/3-540-44794-6_24.

[56] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, and L. Lakhal. Fast computation of concept lattices using data mining techniques. In M. Bouzeghoub, M. Klusch, W. Nutt, and U. Sattler, editors, *Proceedings of the 7th International Workshop on Knowledge Representation Meets Databases*, 2000.

[57] Dean van der Merwe, Sergei Obiedkov, and Derrick Kourie. *AddIntent: A New Incremental Algorithm for Constructing Concept Lattices*, pages 372–385. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-24651-0. doi: 10.1007/978-3-540-24651-0_31.

[58] J. P. Bordat. Calcul pratique du treillis de galois d'une correspondance. *Mathematiques et Sciences Humaines*, 96:31–47, 1986.

[59] Michel Chein. Algorithme de recherche des sous-matrices premieres d'une matrice. *Bulletin mathematiques de la Societe des sciences mathematiques de Roumanie*, 1 (13):21–25, 1969.

[60] Sergei O. Kuznetsov. A fast algorithm for computing all intersections of objects in a finite semilattice. *Automatic Documentation and Mathematical Linguistics*, 27(5): 11–21, 1993.

[61] Cornelia E. Dowling. On the irredundant generation of knowledge spaces. *Journal of Mathematical Psychology*, 37(1):49–62, 1993. ISSN 0022-2496. doi: 10.1006/jmps.1993.1003.

[62] Bernhard Ganter. *Two Basic Algorithms in Concept Analysis*, pages 312–340. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11928-6. doi: 10.1007/978-3-642-11928-6_22.

[63] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[64] Christian Lindig. *Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken.* PhD thesis, Technische Universitat Braunschweig, 1999.

[65] Eugene M. Norris. An algorithm for computing the maximal rectangles in a binary relation. *Revue Roumaine de Mathématiques Pures et Appliquées*, 23(2):243–250, 1978.

[66] Lhouari Nourine and Olivier Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, 71(5–6):199–204, 1999. ISSN 0020-0190. doi: 10.1016/S0020-0190(99)00108-8.

[67] Sergei O. Kuznetsov and Sergei Obiedkov. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental and Theoretical Artificial Intelligence*, 14:189–216, 2002.

[68] Caifeng Zou, Jiafu Wan, and Hu Cai. *A Novel Concept Lattice Merging Algorithm Based on Collision Detection*, pages 489–495. Springer International Publishing, 2014. ISBN 978-3-319-13326-3. doi: 10.1007/978-3-319-13326-3_48.

[69] Jirapond Muangprathub. A novel algorithm for building concept lattice. *Applied Mathematical Sciences*, 8(11):507–515, 2014. ISSN 1314-7552. doi: 10.12988/ams.2014.312682.

[70] David E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms.* Kluwer Academic Publishers, 2002. ISBN 1402070985. doi: 10.1007/978-1-4757-3643-4.

[71] Sean Luke. *Essentials of Metaheuristics.* Lulu, second edition, 2013.

[72] Olympia Roeva. *Real-World Applications of Genetic Algorithms.* InTech, 2012. ISBN 978-953-51-0146-8. doi: 10.5772/2674.

[73] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.).* Springer-Verlag, 1996. ISBN 3-540-60676-9.

[74] Steve Rabin. *AI Game Programming Wisdom.* Charles River Media, Inc., 2002. ISBN 1584500778.

[75] G. N. Yannakakis and J. Togelius. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335, 2015. ISSN 1943-068X. doi: 10.1109/TCIAIG.2014.2339221.

[76] Megan Smith, Stephen Lee-Urban, and Hector Munoz-Avila. RETALIATE: learning winning policies in first-person shooter games. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1801–1806, 2007.

[77] M. S. Emigh, E. G. Kriminger, A. J. Brockmeier, J. C. Príncipe, and P. M. Pardalos. Reinforcement learning in video games using nearest neighbor interpolation and metric learning. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):56–66, 2016. ISSN 1943-068X. doi: 10.1109/TCIAIG.2014.2369345.

[78] J. Schrum and R. Miikkulainen. Discovering multimodal behavior in Ms. Pac-Man through evolution of modular neural networks. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):67–81, 2016. ISSN 1943-068X. doi: 10.1109/TCIAIG.2015.2390615.

[79] Nir Lipovetzky, Miquel Ramirez, and Hector Geffner. Classical planning with simulators: Results on the atari video games. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1610–1616. AAAI Press, 2015. ISBN 978-1-57735-738-4.

[80] Adi Botea, Bruno Bouzy, Michael Buro, Christian Bauckhage, and Dana Nau. Pathfinding in Games. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 21–31. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. ISBN 978-3-939897-62-0. doi: 10.4230/DFU.Vol6.12191.21.

[81] Davide Aversa, Sebastian Sardina, and Stavros Vassos. Path planning with inventory-driven jump-point-search. *Computing Research Repository*, abs/1607.00715, 2016.

[82] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810.

[83] Tom Pepels, Mark H. M. Winands, and Marc Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, 2014. doi: 10.1109/TCIAIG.2013.2291577.

[84] Georgios N. Yannakakis, Pieter Spronck, Daniele Loiacono, and Elisabeth André. Player Modeling. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 45–59. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. ISBN 978-3-939897-62-0. doi: 10.4230/DFU.Vol6.12191.45.

[85] C. Bauckhage, A. Drachen, and R. Sifa. Clustering game behavior data. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(3):266–278, 2015. ISSN 1943-068X. doi: 10.1109/TCIAIG.2014.2376982.

[86] J. Togelius. How to run a successful game-based ai competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):95–100, 2016. ISSN 1943-068X. doi: 10.1109/TCIAIG.2014.2365470.

[87] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311, 2013. ISSN 1943-068X. doi: 10.1109/TCIAIG.2013.2286295.

[88] David Churchill and Michael Buro. Build order optimization in starcraft, 2011.

[89] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010. doi: 10.1109/CEC.2010.5586133.

[90] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. Procedural Content Generation: Goals, Challenges and Actionable Steps. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 61–75. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. ISBN 978-3-939897-62-0. doi: 10.4230/DFU.Vol6.12191.61.

[91] J. Roberts and K. Chen. Learning-based procedural content generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1):88–101, 2015. ISSN 1943-068X. doi: 10.1109/TCIAIG.2014.2335273.

[92] B. Kybartas and R. Bidarra. A survey on story generation techniques for authoring computational narratives. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2016. ISSN 1943-068X. doi: 10.1109/TCIAIG.2016.2546063.

[93] A. Ramirez and V. Bulitko. Automated planning and player modeling for interactive storytelling. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):375–386, 2015. ISSN 1943-068X. doi: 10.1109/TCIAIG.2014.2346690.

[94] Fabien Tencé, Cédric Buche, Pierre De Loor, and Olivier Marc. The challenge of believability in video games: Definitions, agents models and imitation learning. *Computing Research Repository*, abs/1009.0451, 2010.

[95] Jacob Schrum, Igor V. Karpov, and Risto Miikkulainen. *Human-Like Combat Behaviour via Multiobjective Neuroevolution*, pages 119–150. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32323-2. doi: 10.1007/978-3-642-32323-2_5.

[96] Geogios N. Yannakakis. Game ai revisited. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 285–292. ACM, 2012. ISBN 978-1-4503-1215-8. doi: 10.1145/2212908.2212954.

[97] M. Shaker, N. Shaker, and J. Togelius. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2013*, pages 72–78, 2013.

[98] John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M. Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General Video Game Playing. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 77–83. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. ISBN 978-3-939897-62-0. doi: 10.4230/DFU.Vol6.12191.77.

[99] Ben Goertzel. Artificial general intelligence: Concept, state of the art, and future prospects. *Journal of Artificial General Intelligence*, 5:1–48, 2014. ISSN 1946-0163. doi: 10.2478/jagi-2014-0001.

[100] J. Orkin. Three States and a Plan: The AI of F.E.A.R. In *Proceedings of the Game Developer's Conference (GDC)*, 2006.

[101] N. R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2197681.

[102] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[103] Marcelo Kallmann, Hanspeter Bieri, and Daniel Thalmann. *Fully Dynamic Constrained Delaunay Triangulations*, pages 241–257. Springer Berlin Heidelberg, 2004. ISBN 978-3-662-07443-5. doi: 10.1007/978-3-662-07443-5_15.

[104] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Proceedings of The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 942–947, 2006.

[105] Meir Goldenberg, Nathan R. Sturtevant, Ariel Felner, and Jonathan Schaeffer. The compressed differential heuristic. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011.

[106] Tristan Cazenave. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games*, pages 27–33, 2006. doi: 10.1109/CIG.2006.311677.

[107] Yngvi Bjornsson and Kari Halldorsson. Improved heuristics for optimal path-finding on game maps. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 9–14, 2006.

[108] Nathan R. Sturtevant, Ariel Felner, Max Barer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 609–614, 2009.

[109] Meir Goldenberg, Ariel Felner, Nathan R. Sturtevant, and Jonathan Schaeffer. Portal-based true-distance heuristics for path finding. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, 2010.

[110] Adi Botea, Martin Muller, and Jonathan Schaeffer. Near optimal hierarchical pathfinding. *Journal of Game Development*, 1:7–28, 2004.

[111] Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings of The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 1392–1397, 2005.

[112] Daniel Harabor and Adi Botea. Hierarchical path planning for multi-size agents in heterogeneous environments. In *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*, pages 258–265, 2008. doi: 10.1109/CIG.2008.5035648.

[113] Alexander William Kring, Alex J. Champandard, and Nick Samarin. DHPA* and SHPA*: Efficient hierarchical pathfinding in dynamic and static game worlds. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[114] Peter Yap, Neil Burch, Robert C. Holte, and Jonathan Schaeffer. Block A*: Database-driven search with applications in any-angle path-planning. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[115] Nathan R. Sturtevant. Memory-efficient abstractions for pathfinding. In *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 31–36, 2007.

[116] Adi Botea. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011.

[117] Vadim Bulitko. Searching for real-time heuristic search algorithms. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search*, pages 121–122, 2016.

[118] Scott Kiesel, Ethan Burns, and Wheeler Ruml. Achieving goals quickly using real-time search: Experimental results in video games. *The Journal of Artificial Intelligence Research*, 54(1):123–158, 2015. ISSN 1076-9757.

[119] Ramon Lawrence and Vadim Bulitko. Database-driven real-time heuristic search in video-game pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(3):227–241, 2013. doi: 10.1109/TCIAIG.2012.2230632.

[120] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[121] Trevor Scott Standley and Richard E. Korf. Complete algorithms for cooperative pathfinding problems. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 668–673, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-118.

[122] Jeff Orkin. Agent architecture considerations for real-time planning in games. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 105–110, 2005.

[123] Giuseppe Maggiore, Carlos Santos, Dino Dini, Frank Peters, Hans Bouwknegt, and Pieter Spronck. LGOAP: adaptive layered planning for real-time videogames. In *Proceedings of the 2013 IEEE Conference on Computational Inteligence in Games*, pages 1–8, 2013. doi: 10.1109/CIG.2013.6633624.

[124] Paweł Wawrzyński, Jarosław Arabas, and Paweł Cichosz. *Predictive Control for Artificial Intelligence in Computer Games*, pages 1137–1148. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69731-2. doi: 10.1007/978-3-540-69731-2_107.

[125] Tomas Geffner and Hector Geffner. Width-based planning for general video-game playing. *Proceedings of 2015 IJCAI Workshop on General Intelligence in Game Playing Agents*, 2015.

[126] Rufus Isaacs. *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*. Courier Dover Publications, 1999. ISBN 0-486-40682-2.

[127] Ben George Weber, Peter A. Mawhorter, Michael Mateas, and Arnav Jhala. Reactive planning idioms for multi-scale game AI. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 115–122, 2010. doi: 10.1109/ITW.2010.5593363.

[128] Ben G. Weber and Santiago Ontañón. Using automated replay annotation for case-based planning in games. In *In ICCBR 2010 workshop on CBR for Computer Games*, pages 15–24, 2010.

[129] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. ISSN 0018-9219. doi: 10.1109/5.24143.

[130] Chad Hogg, Hector Munoz-Avila, and Ugur Kuter. HTN-MAKER: learning htns with minimal additional knowledge engineering required. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 950–956, 2008.

[131] Bartłomiej Józef Dzieńkowski and Urszula Markowska-Kaczmar. *Plan and Goal Structure Reconstruction: An Automated and Incremental Method Based on Observation of a Single Agent*, pages 290–301. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33260-9. doi: 10.1007/978-3-642-33260-9_25.

[132] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *In Proceedings of the 12th International Conference on Machine Learning*, pages 549–557. Morgan Kaufmann, 1995.

[133] Lior Rokach and Oded Maimon. *Clustering Methods*, pages 321–352. Springer US, 2005. ISBN 978-0-387-25465-4. doi: 10.1007/0-387-25465-X_15.

[134] Bartłomiej Józef Dzieńkowski and Urszula Markowska-Kaczmar. A* heuristic based on a hierarchical space model extracted from game replays. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*, volume 8 of *Annals of Computer Science and Information Systems*, pages 21–30. IEEE, 2016. doi: 10.15439/2016F104.

[135] Guo-Qiang Zhang. Chu spaces, concept lattices, and domains. *Electronic Notes in Theoretical Computer Science*, 83:287–302, 2003. ISSN 1571-0661. doi: 10.1016/S1571-0661(03)50016-0.

[136] Matt Crosby, Michael Rovatsos, and Ronald P. A. Petrick. Automated agent decomposition for classical planning. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[137] Charu C. Aggarwal and Jiawei Han. *Frequent Pattern Mining.* Springer Publishing Company, Incorporated, 2014. ISBN 3319078208, 9783319078205.

[138] G. Sukthankar, C. Geib, H.H. Bui, D. Pynadath, and R.P. Goldman. *Plan, Activity, and Intent Recognition: Theory and Practice.* Elsevier Science, 2014. ISBN 9780124017108.

[139] W. Zeng and R. L. Church. Finding shortest paths on real road networks: The case for A*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009. ISSN 1365-8816. doi: 10.1080/13658810801949850.