Proceedings of the Federated Conference on Computer Science and Information Systems pp. 21–30
DOI: 10.15439/2016F104
ACSIS, Vol. 8. ISSN 2300-5963

A* Heuristic Based on a Hierarchical Space Model Extracted from Game Replays

Bartłomiej Józef Dzieńkowski
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland
Email: bartlomiej.dzienkowski@pwr.edu.pl

Urszula Markowska-Kaczmar
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland
Email: urszula.markowska-kaczmar@pwr.edu.pl

***Abstract*—The paper presents a new method of building a hierarchical model of the state space. The model is extracted fully automatically from game replays that store executed plan traces. It is used by a novel approach for estimating the distance between states in a state-space graph. The estimate is applied in the A* algorithm as a heuristic function to reduce the search space. The method was validated using the game *Smart Blocks*. It is a testbed environment for studying methods that benefit from game replay analysis. The proposed heuristic is dedicated to difficult classical planning problems, for which problem-specific or automated heuristics are difficult to obtain.**

I. Introduction

FOR A long time, AI experts have been developing new methods of imitating intelligent behaviors that can be observed in games. Their goal is to give the player the impression that a computer-controlled unit is an intelligent being. Planning plays an important role in such a task because it enables us to solve complex problems automatically. In classical planning, state search methods are applied to find a sequence of actions between an initial and a goal state. Depending on a particular application, the solution should be optimal. However, it must be the best one that can be provided fitting in a limited computation time, because planning problems in games are solved during play. In this work, we propose a new and promising approach for solving difficult problems in the field of classical planning, which is aimed at application in games. The method introduces a novel technique of extracting and storing information from game replays to increase the performance of planning by providing a new heuristic of estimating the distance to the goal.

Supporting the planning process by information extracted from game replays is a promising direction. This is because, for the majority of games, accumulating the recordings is relatively easy. The data is often used for collecting statistical information that models a player. Among many other benefits, this enables us to analyse players' behaviors and predict their actions.

This work focuses on a particular aspect of the game replay analysis that is a reduction of the search space of a state-space search algorithm. It is assumed that the input data contains traces of plans executed by players. The proposed method processes observed plans to build a general model of the state space. Then, the model is employed in the heuristic of the A* algorithm [1]. It is used for estimating the distance to the goal in a state-space graph. The phase of replay processing is separate. It can be done earlier, so the planning process is not slowed down.

In contrast to the popular planners, our method does not require a STRIPS-like representation of a game state [2]. Instead, it operates on an abstract state-space graph, which is representation-independent. This simplification is significant because providing a symbolic model is time-consuming and difficult in many cases. In comparison with other methods that use plan traces, our approach does not require manual annotations [3]. The proposed method is characterized by a high level of automation. It uses a minimum amount of knowledge about a game and its rules.

The approach is original, and its evaluation requires an adequate testbed environment. The environment should have a nontrivial planning problem. There are relatively few research environments accumulating game replays. Usually, stored replays enable us to reproduce a match visually. However, they do not allow us to access full information about the game state. Adding proper replay storing mechanisms to a complex game is a rather large undertaking. Many of the obstacles can be avoided by developing a new game environment that focuses on the research aspects. Thus, we introduce the *Smart Blocks* game [4]. This light-weight environment enables us to store replays in a simple format, investigate a game state easily, and conduct experiments quickly. It was designed for a range of studies related to planning.

To summarize, the goal of the research was to build a method of accelerating the planning process using information retrieved from the plan traces provided by human players. The original contribution of this paper is:

- the novel method of building a hierarchical model of the state space from game replays,
- the heuristic estimate that relies on the hierarchical space model,
- the new testbed environment designed for analysing actions of players solving difficult planning tasks.

The document is divided into eight sections. At the beginning, references to the related works are provided. Next, the *Smart Blocks* environment is characterized. Subsequently, statistics of the collected replays are presented. In the follow-

978-83-60810-90-3/$25.00

ing section, the algorithm of building a hierarchical model of the state space is introduced. Then, application of the model as a heuristic in A* is thoroughly discussed. In the experimental study, the proposed method and a traditional approach are compared. Finally, features of the approach are summarized, and future development directions are indicated.

II. Related Work

This section sets the proposed approach in the planning context. The term of planning is used differently depending on the application domain [5]. In control theory and especially robotics, planning methods are usually applied for motion and trajectory planning [6]. In video games, the topic often refers to pathfinding algorithms [7]. In this paper, we refer to planning as problem solving. It is a process of choosing and organizing actions by anticipating their outcomes. The process relies on basic concepts like states and actions. Plans come from a decision maker, and they are executed by agents.

It is assumed that actions have deterministic effects, and states are fully known. Therefore, we use the taxonomy of classical planning [8]. Non-classical planning refers to partially observable or stochastic environments.

Planning is applied in many games to solve complex tasks. A survey of the approaches currently used in games can be found in [9]. According to the author, STRIPS, Hierarchical Task Networks (HTN), utility systems, and behavior trees are applied in most cases. However, apart from STRIPS-like planners, the discussed methods are mainly used for modelling the reactive behavior of AI-controlled players. In these cases, the provided model is a plan, and its execution is defined by designers. Referring to them as planning methods, which search for a path to a defined goal state, is misleading. Goal-Oriented Action Planning is closer to the discussed understanding of planning [10]. In general, planning gives better perception of AI players' intelligence, but it is more complex to apply in practice. Therefore, our research is dedicated to increasing the applicability of planning in games.

A popular hierarchical approach is Hierarchical Pathfinding A* (HPA*) [11]. HPA* searches for a path on terrain. A state-space graph is represented by a mesh of nodes on the terrain surface. The improved method HPA* clusters places that lay in the geographical neighborhood (inside rectangles). The method also considers graphs with different levels of abstractions as we do in our tree. However, the cost between groups of states cannot be easily determined if the Euclidean distance between state variables does not reflect the transition cost, which is the case addressed in this work. HPA* solves problems in the geographical space while our method is applicable for any abstract state space.

Analysis of executed plan traces is a subject that is discussed in many fields, e.g., robotics [12], business [13], and games [14], [15]. With a few exceptions, there is not much attention given for supporting state search in classical planning by observed plans [16], [17]. More often the observations are applied to plan recognition [18], [19] or player action prediction [20], [21]. In the work of Wang [3], whose method is old but close to our idea of solving a planning problem, plans are learned from observation. However, the described method is inefficient because plan traces must be examined and annotated by experts manually. In addition, it suffers from STRIPS negative preconditions that are generated without limitations. Our method avoids these problems.

Another approach that is somehow related to our problem is presented in the work of Hogg [17]. The method learns hierarchical planning knowledge to solve tasks in HTN. It takes as input a set of planning states and a set of semantically-annotated tasks. Our approach is different because the knowledge model does not require information about tasks and goals.

III. Smart Blocks

Smart Blocks is a testbed environment. It was designed to study planning methods that can learn from player actions. The project includes two subsystems. The first one is a video game in which a player solves planning problems. The game sends observed solutions to the server, where they are stored. The second one is a simulation. It enables us to reconstruct saved plans and test planning algorithms.

The game can be classified as a single-player logic game. Its mechanics was inspired by *Sokoban* [22]. It was implemented with *Unity3D* [23]. One of the priorities was to make the game attractive to the players because the more they play, the more data is collected. The game offers a visual interface, simple gameplay, and easy online access[1] [4].

A. Game Rules

The gameplay relies on simple box-pushing mechanics. In the game, a player controls a team of agents, each of which is represented by a block. The blocks are characterized by different sizes and shapes. The main goal of the team is to reach a golden artifact by any of the blocks. The task is complicated by the maze of triggers and gates that block the path. The triggers usually require different blocks working together to open a gate. The paint of a block is also important when matching a trigger pattern. The pattern comprises a shape together with a colour that must be satisfied by the blocks standing on the trigger. The paint is obtained from a colour portal, and it mixes with the current colour of a block.

A player has to take into account an energy reserve, which is limited and consumed in each action step. This constraint prevents infinite game duration. Unlike the time constraint, it does not enforce a player to act hastily. Energy consumption depends on the size of a block, and it can be increased by entering a ground obstacle. The energy level can be refilled by an energy cell – it disappears once it is used. A player's score depends on the energy reserve at the moment of level completion.

An example of a planning problem that involves agent cooperation is illustrated in detail by Fig. 1. The example shows a part of a stage that contains three user-controlled agents: a ring, a box, and a small cylinder. Their objective

[1] *Smart Blocks* game is available at http://unity3ddev.net/smartblocks/. The source code can be obtained by contacting the authors.

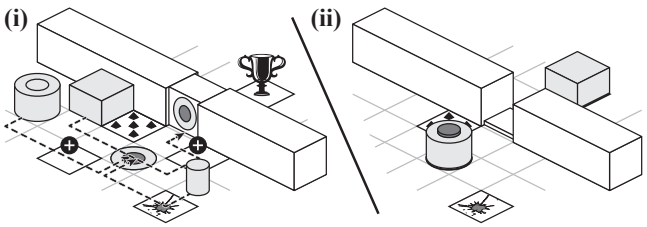

Fig. 1. An example of a planning problem solved in *Smart Blocks*.

is to reach the artifact that is placed behind the wall (the goal is marked with a cup). Initially, the path is blocked by the closed gate **(i)**. It can be opened by using the trigger. 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 field where it changes a paint colour to the one that is accepted by the trigger. Then, the big ring collects the last energy cell, and it ends its move in the same place. As soon as the key of the trigger is satisfied, the gate is opened. Now, the box agent is free to reach the goal **(ii)**. If any of the two agents moves from the trigger, the gate will close (unless the third agent is blocking the gate by standing on it).

The presented scheme shows only one of many possible tasks that can be present on a stage. The game levels were designed to be challenging by joining chains of tasks that must be completed before the goal can be reached.

The first version of the game includes ten levels of different difficulty. Subsequent levels can be accessed without solving the previous ones. However, following the order helps to familiarize with the game rules. In the first level, a player learns how to use a simple cooperation to open a gate and reach the artifact. The next level gives an example of multiple blocks interacting with a single trigger. Later, a player is introduced to paint colours and how they mix. In level 4, a player has to use energy cells for the first time. It is a simpler level, but potentially more challenging for a state search algorithm. Level 5 has an alternate route, which is shorter but more challenging to guess. The subsequent level contains many possible routes, and it is difficult to estimate which one will consume the least energy. Level 7 has a long chain of tasks that have to be executed in a certain order. In the next level, misleading trails are present. The last two levels are characterized by a high complexity and a very large state space.

B. Planning Problem

There are several arguments justifying why the planning problem in *Smart Blocks* is nontrivial. The first one is the difficulty of measuring the distance to the goal. For instance, in *Sokoban*, it is possible to count the number of crates on the spots as smaller tasks [22]. It is a good heuristic for estimating the progress of the goal accomplishment. For our problem, the goal progress cannot be measured easily. One of the blocks must reach the location of the artifact. It is unknown which block, what combination of gates and trigger should be used, or how much energy and how many actions it will take. Sometimes it is not necessary to visit locked rooms in order to reach the artifact. However, it may be required to save some energy. The order of blocks in small corridors is also important because the agents can collide with each other. The world can be modified by agents: temporarily by opening a gate or permanently by collecting an energy cell. Each modification applied to the environment builds a subtree in a state search tree (multiplies the search space).

A significant complication is the presence of energy cells. They introduce edges with negative cost to the state-space graph. This type of a state space requires an algorithm that traverses every edge in the graph to ensure optimality (e.g., Bellman-Ford method) [24]. However, the number of states is usually too large to perform a brute-force search. The first three game levels have neither energy cells nor negative cycles in their state spaces.

The problem stated in *Smart Blocks* is the centralized planning of cooperation of agents with limited resources in a mutable environment [25]. It is located in a group of planning problems in which a heuristic estimate is difficult to provide. An abstract version of this problem can be found in a number of real games. Many analogies are present. However, the specificity of the problem in practical application can differ. In some cases, the problem can be solved offline in the phase of game design. Designers can embed a solution schema of a planning problem in a game rigidly. In this research, we are aiming at cases in which the problems can appear dynamically – for instance, problems invented by players during an online game. Therefore, we minimize the amount of predefined knowledge and try to improve planning performance by relying on the observations.

C. Testbed Environment

The simulation is a C# console program. Its function is to parse the recorded plans, execute planning algorithms, and yield the statistics. It contains the model of game rules and a light-weight representation of a game state. Therefore, it enables us to conduct experiments efficiently.

The fundamental motivation for building the environment was a very small number of planning benchmarks that work with observed plan traces. It is a common practice that game replays store only visual effects of player actions. Therefore, they require a lot of reverse engineering to extract actual game states. Our game stores full-information game states in an easy readable format.

Another argument is complexity. In *Smart Blocks*, planning problems are small and flexible. It means that they are easy to understand by players and simple to scale by the level designer. It is easy to follow the execution of a tested method. In environments like *WarCraft (Wargus)* or *StarCraft (BWAPI)* game rules are complicated, and the game state is large [26], [27]. Their state spaces are vast. For most of the tasks, information about optimal plans is unavailable (or practically incomputable). It is difficult to analyse recorded

TABLE I
REPLAY DATABASE STATISTICS

Level No.		*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Replays		230	169	71	56	57	49	30	13	7	7
Steps	min.	23	18	57	24	40	32	69	60	246	429
	avg.	39.8	34.7	64	27.3	70	58.8	82.7	75	304.6	441.6
	max.	99	88	89	44	108	101	95	125	391	457
Energy	min.	3	3	2	0	3	9	19	30	152	19
	avg.	180.7	121.7	49.8	9.6	70.1	130.3	133.1	124.7	343.7	101.4
	max.	231	161	66	11	138	205	187	182	503	141

plans and evaluate planning algorithms. In our environment, plans can always be compared with optimal solutions or even evaluated visually. It should be noted that *Wargus* and *BWAPI* lack simulation modes (iterative and game-time-independent execution). It is crucial for testing algorithms that traverse a state-space graph.

Finally, our project allows researchers to focus purely on the planning procedure. In another popular environment, *RoboCup*, the mechanics rely on continuous state and physics [28]. It is a good benchmark for robotic applications because it reflects the real world. However, planners usually work with a simplified world model. Providing the model is a challenging task. Our environment abstracts from uncertainty and state discretization issues. These are important problems, but they are located in the conceptual phase of preparing an environment for planning. Our study focuses on traversing the state space, which is discrete by definition.

D. Data

A solution of a stage provided by a player is recorded in a replay file, and then it is submitted to the server. Only complete replays are stored – unfinished or partial solution sequences are discarded. Each recorded and stored solution sequence allows game states to be reproduced fully with no uncertainty. A replay consists of a sequence of steps. Each step is an atomic action committed by an agent. Steps are deterministically ordered – simultaneous actions are not allowed. There are no additional complications behind the described process of collecting data. In *Smart Blocks*, recording the game state is almost as simple as in chess. Actions are animated for the purpose of the presentation, but in fact, they are discrete.

Simple statistics of replays stored in the database are given in Table I. They include for each game level:

- the total number of collected replays,
- a number of action steps to solve a stage,
- an amount of energy at the moment of reaching the final goal.

The statistics provide a good reference for the evaluation of planning algorithms.

It can be seen that the largest number of replays has been collected for the initial levels. It is because the difficulty grows rapidly, and many players resign. The data also gives clues how the levels differ from each other and what is the spread of possible outcomes.

IV. HIERARCHICAL SPACE

To have a better understating of what a hierarchical division of the state space is, we can imagine a city, its district, a residential block placed in there, an apartment in the building, and its room. Searching for a specified room is much easier if we know the name and part of the city, the building address, and the apartment number. The model of our hierarchical space can be perceived as a map that is discovered according to visited places. However, it is not necessary to visit every place to outline a region on the map. The same rule applies to game states and state subspaces.

In formal terms, a hierarchical space is a tree structure. It divides the state space into subspaces that contain groups of states. Subspaces on a higher level are nesting the ones on a lower level. Organizing states as graph nodes inside this kind of structure enables us to traverse and search in the graph more efficiently. For instance, we can reduce the search space by simply skipping whole groups of states that are not leading us to the solution. This procedure can be done on different levels of detail – starting from the most general to the most detailed.

Our approach for building the model relies on *state descriptors*, which is a term introduced in this paper. A state descriptor refers to a selected part or some feature of a state. It can be said that a descriptor partially *describes* a game state. Formally, it is a predicate, and it holds a rule or expression that returns a boolean value depending on whether it is satisfied or not. Descriptors are usually related to intermediate objectives and goals in a game. A descriptor enables us to group a set of states based on a defined criterion. In practice, state descriptors can be added to the implementation easily as boolean functions that accept a state as input. They do not impose formal requirements on problem representation.

In *Smart Blocks*, state descriptors are defined by the designer. They are closely related to the game rules and player's objectives. Therefore, they have a simple and intuitive form. A set of descriptors is generated by descriptor classes – their complete list is presented in Table II. They group states taking into account, for example, the colour of an agent, its position in a room, a gate state, or the goal accomplishment. Each game state can be partially depicted by a subset of descriptors that are satisfied at a given moment. For instance, a state can satisfy a group of three descriptors: {⟨`agent 1 is in room 1`⟩, ⟨`agent 1 is on trigger 3`⟩, ⟨`gate 2 is open`⟩}.

In general, the idea that stands behind state descriptors is to provide a degree of flexibility to the approach. A set of descriptors can be optimized for a problem by machine learning methods. It is the aim of the future study. However, here we use rigidly defined descriptors to focus on the heuristic and provide a proof-of-concept.

Each group of descriptors defines a subspace that covers a part of the state space. The more descriptors work together, the smaller part of the space they cover. Less detailed subspaces nest inside more detailed subspaces. The more general a group of descriptors is, the more children it has. However, it is assumed that state subspaces, covered by descriptors, are not

TABLE II
A LIST OF DESCRIPTOR CLASSES IN SMART BLOCKS

Descriptor Class	Description
agent {id} in room {nr}	informs whether a specified agent is placed in a defined area
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	informs 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 ok	true if one of the agents reached the golden artefact; groups the goal states

required to nest each other fully, but they are allowed to intersect. The method of handling the intersections is provided later in the document.

Theoretical foundations and the algorithm of building a hierarchical model of the state space using state descriptors are presented in the following subsections.

A. Formalization

Below are the theoretical assumptions stated. They are required to discuss optimality of the introduced heuristic.

1) State Space: A classical planning problem can be formulated as finding a path between two arbitrarily specified states in an abstract state space. Let us define a state space as a directed graph $G = \langle S, E \rangle$, 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 of state variables v, Eq. 1:

$$s = \langle v_1, v_2, \ldots \rangle. \tag{1}$$

Each edge $e \in E$ has a transition cost $c \in \mathbb{R}_{>0}$ associated with it. It is assumed that the state space is vast and the cost (or distance) estimate function $\delta : S \times S \to \mathbb{R}_{\geq 0}$ between any two noncontiguous states is unknown or complex. Consequently, state search in the defined space is a nontrivial problem.

2) Plan Observations: A set of executed plans is provided by the system users. An observed plan is a sequence $x_i \in X$ of state transitions, Eq. 2:

$$x_i = \langle s_1, s_2, \ldots, s_n \rangle, \tag{2}$$

where $s_1 \in S$ is an initial state, and $s_n \in S$ is some goal state ($s_1 \neq s_n$). Subsequent state nodes in the sequence are connected by edges.

It is assumed that the observed plans are valid but not cost-optimal. A set X of observations does not cover the entire state space, but it allows us to collect information about its structure.

3) State Descriptor: Let $S_i \subseteq S$ denote a subset of the space states, and $d_i : S \to \{0, 1\}$ is a function that determines membership of a state in this subset, Eq. 3:

$$S_i = \{s \in S : d_i(s)\}. \tag{3}$$

Then, the function $d_i \in D$ is called a state descriptor that classifies and groups states by their selected features. Thus, each state $s_j \in S$ is assigned to a set $D_j \subseteq D$ of descriptors that are valid for s_j, Eq. 4:

$$s_j \Rightarrow D_j = \{d \in D : d(s_j)\}. \tag{4}$$

Based on a set $S_X \subset S$ of states appearing in a set X of plan observations, a set D_X of descriptor sets is extracted, Eq. 5:

$$D_X = \{D_j \subseteq D : s_j \in S_X\}. \tag{5}$$

Descriptor sets enable us to discover a hierarchical structure of the state space, and they play an analogous role as transactions in data-mining.

4) Subspace Tree: State groups separated by descriptor sets are used to build a tree of state subspaces. A state subspace $h_k \in H$ is a pair $h_k = \langle D_k, p_k \rangle$ comprising a set $D_k \subseteq D$ of state descriptors together with an index p_k of a parent subspace. The set D_k determines a set S_k of states that belong to the subspace h_k. The set S_k is the intersection of states grouped by each $d_j \in D_k$, Eq. 6:

$$h_k = \langle D_k, p_k \rangle \Rightarrow S_k = \{s \in S : \left(\forall_{d_j \in D_k} d_j(s)\right) \vee D_k = \emptyset\}. \tag{6}$$

If a subspace h_i is a parent of a subspace h_k, then the set of states covered by the child is a subset of the parent's set, Eq. 7:

$$(p_k = i) \Rightarrow S_k \subset S_i, \tag{7}$$

which is equivalent to Eq. 8:

$$(p_k = i) \Rightarrow D_k \supset D_i. \tag{8}$$

Considering the parent-child relation, it is worth noticing that the implication operator does not have to be applicable in the opposite direction. In other words, a set of states representing a part of the space can be associated with more than one subspace, and thus it has more than one possible parent. For instance, the intersection $S_k = S_i \cap S_j$ of descriptor sets D_i and D_j can be nested by h_i as well as h_j. The selection of a parent subspace is disambiguated algorithmically.

B. Algorithm

The algorithm starts from collecting a set D_X of all possible descriptor groups, which can be found in a set S_X of observed game states. Additionally, the common parts of the groups that intersect are added to D_X. The nesting relations are then stored. Next, a tree structure is assembled by disambiguating the parent-child relations. The result is a tree expressed by a set $H = \{h_1, h_2, \dots\}$ of subspaces, which models a hierarchical structure of the state space.

The basic steps of the algorithm are expressed in pseudocode in Alg. 1. The algorithm accepts a set of game states observed in a replay database as input. A state contains complete information about a temporary situation in a game. Global variables are declared in the context of all methods. In the beginning, unique groups of descriptors are collected (line 2). In the same process, relations between the groups are determined. Next, based on these groups, a hierarchical model of the state space is built (line 8). The sorting in line 7 will be explained later.

Alg. 1: BuildHierarchy ($states$)

```
Data: set of observed states
Result: hierarchical model of state space
1  global descriptorSets ← ∅
2  CollectDescriptorSets ( states )
3  var root ← ⟨ϵ, ϵ⟩
4  global subspaces ← {root}
5  global usedDescSets ← ∅
6  global usedStates ← ∅
7  var descriptorSetsSorted ← Sort ( descriptorSets )
8  foreach descSet ∈ descriptorSetsSorted do
9      BuildSubspaces ( descSet, root )
10 return root
```

At the beginning of the routine, unique groups of descriptors are collected by iterating over every input state (Alg. 2). A descriptor group consists of all the descriptors that are satisfied by a state (line 2). Observed descriptor groups usually overlap (as well as state subspaces). Thus, additional groups are created by intersecting new groups with the already observed ones (line 7), and then stored (line 9). An intersection of two descriptor groups separates a subspace that can be nested by the subspaces of both operands.

Alg. 2: CollectDescriptorSets ($states$)

```
Data: set of observed states
Result: list of descriptor sets
1  foreach s ∈ states do
2      var newDescSet ← DescriptorSetFromState ( s )
3      if InsertDescriptorSet ( newDescSet ) then
4          var intersections ← ∅
5          foreach descSet ∈ descriptorSets do
6              if descSet ≠ newDescSet then
7                  intersections ←
                       intersections ∪ {descSet ∩ newDescSet}
8          foreach descSet ∈ intersections do
9              InsertDescriptorSet ( descSet )
```

It can be concluded that different game states represented by exactly the same groups of descriptors are redundant, and they do not contribute to the model. Therefore, only one state is sufficient for discovering a subspace. The more unique descriptor groups are observed, the richer the structure of the model is.

While the new descriptor groups are added to the list (Alg. 3, line 3), parent-child relations are assigned (line 4). A group of descriptors is a parent of another one if the first one is a subset of the second one. In other words, all descriptors from a parent group can be found in its child. In this relation, a parent will always cover an equal or bigger number of states than its child. The information about subspace nesting will be used in the next step.

Alg. 3: InsertDescriptorSet ($newDescSet$)

```
Data: new descriptor set
Result: stores new descriptor set and assigns parent-child links
1  if newDescSet ∈ descriptorSets then
2      return 0
3  descriptorSets ← descriptorSets ∪ {newDescSet}
4  foreach descSet ∈ descriptorSets do
5      if descSet ⊂ newDescSet then
6          descSet.children ← descSet.children ∪ {newDescSet}
7      else if newDescSet ⊂ descSet then
8          newDescSet.children ←
               newDescSet.children ∪ {descSet}
9  return 1
```

Having prepared such a set of descriptor groups, we can proceed with assembling a data structure that expresses a hierarchy of state subspaces – Alg. 4. Each state subspace is created based on a corresponding descriptor group. State subspaces maintain the same parent-child relation as descriptor groups. At this point, the relation links are copied and disambiguated. A descriptor group may have many possible parents if the group is a product of the intersection operation. However, a subspace in a tree can have only one parent, and it can appear in the structure only once.

In the process of disambiguation, each subspace receives one parent. The process must be performed in a certain order. For instance, if a small subspace is attached to a big one too early, we may lose an opportunity to add an intermediate layer. Therefore, the procedure begins from bigger subspaces that nest the largest number of smaller subspaces. To do so, each time a list of descriptors is sorted descending by the number of unassigned descendants (children, grandchildren, etc.) – line 7 in Alg. 1, and line 6 in Alg. 4.

The recursive procedure in Alg. 4 begins from the root, and it expands the children down to the tree leafs. A subspace can be added to the structure only if it nests (directly or through a descendant) at least one observed state that was not used previously (line 15). Otherwise, the subspace is discarded (line 32). Near the end of the procedure, if there is a subspace that holds a child subspace, and it contains any states at the same time, an additional subspace is created inside the scope of the current subspace to take over these states (line 21).

Alg. 4: BuildSubspaces ($descSet$, $parent$)

```
Data: descriptor set and its parent subspace
Result: subspace tree
1  if descSet ∈ usedDescSets then
2      return 0
3  usedDescSets ← usedDescSets ∪ {descSet}
4  var subspace ← ⟨descSet, parent⟩
5  var anyStateInChild ← 0
6  var descriptorSetsSorted ← Sort ( descSet.children )
7  foreach childDescSet ∈ descriptorSetsSorted do
8      if BuildSubspaces ( childDescSet, subspace ) then
9          anyStateInChild ← 1
10 var anyStateHere ← 0
11 foreach s ∈ descSet.states do
12     if s ∉ usedStates then
13         anyStateHere ← 1
14         break
15 if anyStateInChild ∨ anyStateHere then
16     subspaces ← subspaces ∪ {subspace}
17     parent.children ← parent.children ∪ {subspace}
18     if anyStateHere then
19         var leaf ← subspace
20         if anyStateInChild then
21             leaf ← ⟨descSet, subspace⟩
22             subspaces ← subspaces ∪ {leaf}
23             subspace.children ← subspace.children ∪ {leaf}
24         foreach s ∈ descSet.states do
25             if s ∉ usedStates then
26                 usedStates ← usedStates ∪ {s}
27                 leaf.states ← leaf.states ∪ {s}
28                 s.subspace ← leaf
29                 if IsGoal ( s ) then
30                     leaf.hasGoal ← 1
31     return 1
32 else
33     delete subspace
34     return 0
```

This action ensures that the states are placed only in leaf subspaces. Observed states are assigned to leaf subspaces in line 27. Finally, subspaces that contain goal states are marked (line 29).

The next section explains how the resulting structure can be utilized in planning.

V. Hierarchical Space Based Estimate

In a planning task, the least expensive (shortest) path between an initial state and any state that satisfies the goal is being searched. In our case, the result is a sequence of actions that solves a stage consuming the least amount of energy.

In the considered problem, the best performance can be achieved by applying the A* algorithm [29]. However, it requires a heuristic estimate function to approximate the distance to the goal. Providing such an estimate is not an easy task. In *Smart Blocks*, it is difficult because the progress of solving a stage is hard to measure. The problem structure makes relaxation heuristics futile, which was explained in Section III.B. The goal progress could be calculated using the designer's knowledge about all possible solutions of a level. In this benchmark environment, a solution pattern can be generated, but in real problems, it is an unrealistic assumption.

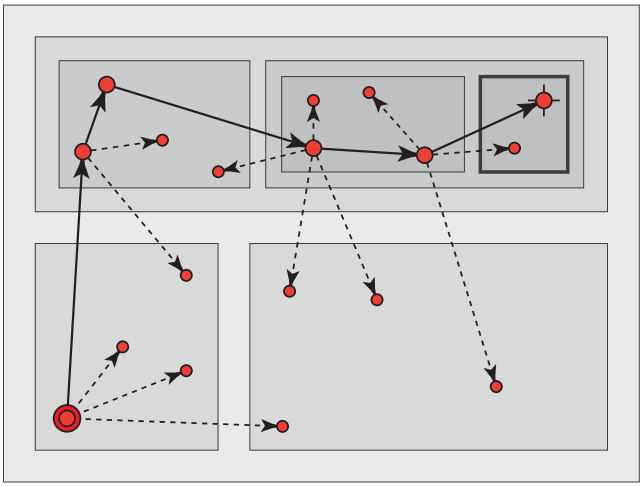

Fig. 2. A visualization of state search supported by the hierarchical model of the state space.

Our idea is to use a hierarchical state-space model to roughly estimate how close a state is to the goal. The distance is calculated based on the number of parent subspaces that a state shares with the nearest goal subspace. The more mutual nodes in a tree they have, the closer to the goal the state is. It is an abstract measure. It supports state search by leading it to more promising regions of the state space.

Fig. 2 provides a visual example of state search that uses information stored in the hierarchical model of the state space. In the picture, the state search begins in the bottom left corner of the figure, and it ends in the top right one. The initial state shares only one parent (the root) with the goal subspace (marked as the bold rectangle). At this point, there are four possible state transitions. One of the subsequent states shares two parents with the goal subspace. This one has a higher priority, and it should be expanded next. The procedure is repeated until the goal subspace is reached.

The formula for calculating the proposed *Hierarchical Space Based Estimate* (HSBE) is defined in Eq. 9:

$$\Delta(s) = 1 - \frac{max_i\Big(\big|parents(s) \cap parents(g_i)\big|\Big)}{d}, \qquad (9)$$

where:

- s is a state,
- g_i is a goal subspace (one of many),
- d is a depth of a hierarchical space tree,
- $parents(\cdot)$ is a function that returns a set of parent subspaces up to the root,
- $max_i(\cdot)$ iterates over all goal subspaces and returns the biggest value.

First, the method finds a parent subspace for a state. It is a leaf subspace with the biggest number of descriptors that match the state. Then, the method collects a set of parent

subspaces of the state (up to the root). The set is intersected with a chain of subspaces from the root to a goal subspace (inclusively). The two sets are intersected to count the number of shared subspaces. The number is divided by the tree depth to normalize the outcome. If there is more than one goal subspace, the most promising one is chosen. The range of $\Delta(\cdot)$ is in $[0, 1)$. The normalized value can be scaled by a minimum transition cost in a state-space graph to ensure that the heuristic never overestimates the cost. Therefore, it is admissible. It should be emphasized that the discussed admissibility is the property of a heuristic that ensures optimality in a state space that does not contain negative cycles, as it was assumed in the formal description.

The main operation of the estimate is tree search, and its computational complexity is logarithmic. A tree structure enables us to quickly collect a chain of nodes between a leaf and the root. In addition, parents of goal subspaces can be stored before the planning phase. Finding a parent subspace for a newly expanded state is linear in the number of leaf subspaces. A significant impact on the computational overhead has the operation of set intersection, which is frequently used. Complexity of this operation depends on the implementation [30]. With some effort, it can be done efficiently.

VI. Experiments

The goal of this study was aimed at checking whether this early concept of a planning method is worth further development. The proposed method is intended for problems in which a heuristic is unavailable, ineffective, or characterized by a high computational complexity [31]. Therefore, Dijkstra's algorithm was chosen as a reference [24]. The algorithm is often used as a benchmark, because it is optimal and represents the worst-case scenario in domain-independent planners [32].

The performance of the methods was measured by:

- the number of iterations in the main loop,
- the number of visited states,
- the maximum size of the state queue (the open set),
- the mean wall-clock execution time.

The results are presented in Table III.

The experiments were conducted on a typical hardware setup: Win 7 x64, Intel i7 @ 3.4 GHz, 8 GB RAM. The wall-clock times were measured for the state search procedure in the main loop. For Dijkstra's algorithm, the measurements were simply averaged from 10 runs. However, HSBE relies on a subspace tree. It can have different sizes, because it can be built from different numbers of plan traces. Thus, the smallest subset of the collected plan traces, which was necessary to obtain full efficiency, was used. Usually, 10% of the set was enough. The subset was chosen randomly, and the procedure was repeated 10 times.

The tests were conducted for the first six game levels. The remaining levels are more complex, their state spaces are larger, and the number of states grows very rapidly. The explosion of states is caused by a bigger number of dynamic objects on the stage. The experiment could not be finished in an acceptable time on the present hardware. However, these levels are somewhat analogous to the preceding ones, and their contribution should not conflict with the initial results.

As expected, the compared methods are equal in the quality of returned solutions. For most of the levels, they provide cost-optimal plans. The exception is level 5 and 6 (compare with the replay statistics in Table I). A slight deviation from the optimal paths is observed. These two levels contain energy cells, which introduce negative-cost transitions to the state-space graph. In this case, solution optimality is not ensured by the algorithms, and the returned solutions may vary depending on the order of states in the queue.

The discussion regarding the performance should begin from comparing the execution times and the number of visited states. In most cases, A*+HSBE is slower than Dijkstra's algorithm. This is caused by the fact that the researched heuristic uses many operations involving complex data structures, while Dijkstra's algorithm is all about adding and removing an item from a queue. On the other hand, the proposed method is characterized by a smaller number of visited states. The execution times include the time of visiting states and the overhead of the method. The more expensive visiting a state is, the lesser part the overhead in the execution time has.

For instance, let us consider level 5. Dijkstra's algorithm is approximately two times faster than A*+HSBE, but it also visits twice as many states. If the cost of visiting a state was tripled, then both algorithms would have almost the same execution time. A*+HSBE is faster if the number of states exceeds this threshold.

The reduction of visited states increases for the larger game levels. The density of the state space division is constant, and it might be too sparse in the simpler ones. If the partition of the state space is low then the heuristic is less informative. On the other hand, if it is too high, then the overhead is considerable.

In the final part of the study, the relation between the number of plan traces, used for building the hierarchical model, and the performance of the proposed method was examined. The experiment was conducted for level 5, and the results are shown in Fig. 3. The subsets of replays in the database were chosen randomly, the process was repeated 10 times, and the measurements were averaged. The quality of the returned solutions remains constant. The reduction of visited states increases as the number of plan traces grows. It appears that a small number of observations is sufficient to discover a large part of the subspace tree. Similar results were observed for the remaining game levels.

Although, the results do not show incontestable superiority of the proposed heuristic over the algorithm used as a benchmark, it should be noted that applying the method in practice may be justified by the reduction of the search space. The profit of employing the introduced approach depends on the computational cost of visiting a state in a particular system. For simple planning problems, the execution time of a complex method can take longer than using a trivial state-search algorithm. *Smart Blocks* as a testbed environment is characterized by a medium-size state space and a very light state so that the experiments could be conducted swiftly.

TABLE III
COMPARISON OF DIJKSTRA AND A*+HSBE (THE LAST TWO COLUMNS CORRESPOND TO BOTH METHODS)

Level No.	Algorithm iterations		Visited states		Max. queue size		Avg. execution 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

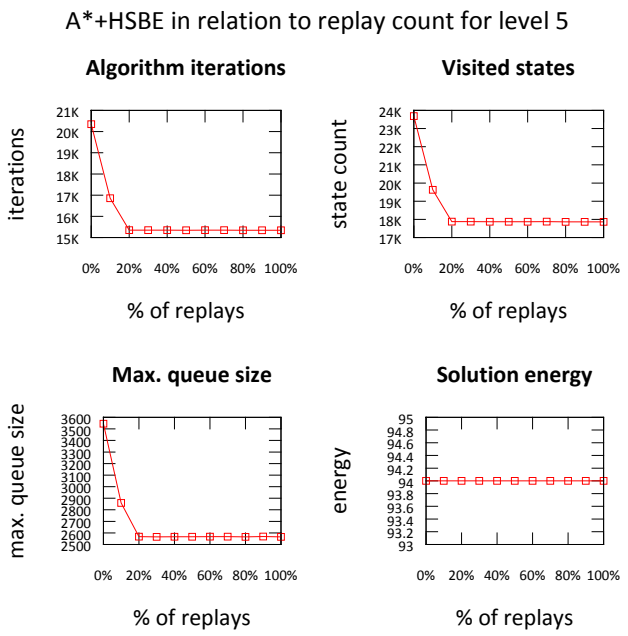

Fig. 3. The set of charts shows the statistics of A*+HSBE for level 5 depending on different numbers of replays used for building a subspace tree.

However, for real planning problems, if the state space is vast and visiting a state is expensive then the method is most certainly worth applying.

VII. CONCLUSIONS

The original contribution of this paper is a new method of building the hierarchical model of the state space from game replays. Information stored in the model represents general knowledge about the structure of the state space. The method accepts executed plan traces as input. The observations are not required to be annotated by experts. The model abstracts from state representation methods. It relies on state descriptors that refer to selected features of a state, and their form is unrestricted. It does not oblige system designers to rewrite a planning problem in PDDL. Instead, they can use the original implementation of a game. It is more convenient and efficient.

The method is used for grouping states hierarchically, and it relies on hyperspace nesting. The proposed heuristic estimates the distance between any two states in the state space to effectively guide the state search towards the goal and reduce the search space. It is intended for difficult cases in which a heuristic estimate of distance in a state-space graph cannot be easily provided and domain-independent heuristics are inefficient.

The proof-of-concept was validated using *Smart Blocks*. It is a testbed environment designed for conducting experiments that involve game replay analysis. The game is modeled as a variant of a multi-agent system. It enables a researcher to focus on classical planning problems. Unlike commercial games, it comes with tools that simplify method testing.

It should be emphasized that the approach is general. Formal assumptions of the method, which include the planning problem definition and the model of the hierarchical state space, are abstract. The same rule applies to the introduced algorithms, because they do not assume any specific properties of the system in which a planning problem is solved. Although the method relies on state descriptors that were implemented according to domain-specific knowledge for this particular study, their formal form is abstract, and it is possible to extract them automatically (see the Future Work section). To summarize, the approach can be applied to any classical planning problem for which input observations are available.

VIII. FUTURE WORK

The presented results mark an important milestone in the development of the approach. There are still many opportunities for improvement, but their examination is a rather large undertaking. This section outlines possible development directions.

In the course of ongoing research, it has been discovered that a concept (Galois) lattice is a more suitable model for expressing relations between state subspaces [33]. In a hierarchical structure described by the lattice, the intersection of state subspaces has many parents, rather than a single one like in the tree structure. Formal foundations of the Formal Concept Analysis (FCA) are consistent with the descriptors introduced in the discussed model of the state space. Thus, a lattice can be built by an algorithm commonly used in FCA [34]. Preliminary study results show a good performance of a heuristic supported by this model. Therefore, a tree will be replaced by a lattice in the subsequent study.

At the current stage of development, issues related to processing a very large replay database have not been taken into account. It is a secondary problem because the process is separated from planning, and its execution time is acceptable. Nonetheless, pruning methods that protect against the overgrowth of the structure should be researched.

Reduction of the search space depends on the quality of the state space division. The algorithm for building a hierarchical space model ensures that the formal assumptions are satisfied. However, the division quality is affected by state grouping, which is handled by state descriptors. Adapting state descriptors automatically can lead to better results. The research is aimed at inventing an adaptive descriptor model that can be tuned by machine learning methods.

REFERENCES

[1] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *J. ACM*, vol. 32, no. 3, pp. 505–536, 1985. doi: 10.1145/3828.3830

[2] R. E. Fikes and N. 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*. Morgan Kaufmann Publishers Inc., 1971, pp. 608–620.

[3] X. Wang, "Learning by observation and practice: An incremental approach for planning operator acquisition," in *Proceedings of the 12th International Conference on Machine Learning*. Morgan Kaufmann, 1995. doi: 10.1.1.36.7719 pp. 549–557.

[4] "Smart Blocks," http://unity3ddev.net/smartblocks, accessed: 2015-11-11.

[5] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006. ISBN 0521862051

[6] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004. ISBN 1558608567

[7] D. M. Bourg and G. Seemann, *AI for game developers*. O'Reilly & Associates Inc., 2004. ISBN 0-596-00555-5

[8] D. Bryce and S. Kambhampati, "A tutorial on planning graph based reachability heuristics," *AI Magazine*, vol. 28, no. 1, pp. 47–83, 2007.

[9] A. J. Champandard, "Planning in games: An overview and lessons learned," http://aigamedev.com/open/review/planning-in-games/, 2013, accessed: 2015-11-11.

[10] J. Orkin, "Applying goal oriented action planning in games," in *AI Game Programming Wisdom 2*. Charles River Media, 2002, pp. 217–229. [Online]. Available: http://web.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf

[11] A. Botea, M. Muller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, vol. 1, pp. 7–28, 2004. doi: 10.1.1.112.314

[12] L. Mosenlechner, N. Demmel, and M. Beetz, "Becoming action-aware through reasoning about logged plan execution traces," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, 2010. doi: 10.1109/IROS.2010.5650989. ISSN 2153-0858 pp. 2231–2236.

[13] I. Hulpuş, M. Fradinho, and C. Hayes, "On-the-Fly Adaptive Planning for Game-Based Learning," in *Case-Based Reasoning. Research and Development*, ser. LNCS, I. Bichindaritz and S. Montani, Eds. Springer, 2010, vol. 6176, pp. 375–389.

[14] S. Sahasrabudhe and H. Munoz-Avila, "Mining cause-effect sequential patterns from action traces," FLAIRS Conference, 2013. [Online]. Available: http://www.cse.lehigh.edu/~munoz/Publications/flairs04.pdf

[15] S. Breining, H. Kriegel, M. Schubert, and A. Züfle, "Action sequence mining," Workshop on Machine Learning and Data Mining in Games, 2013. [Online]. Available: http://www-kd.iai.uni-bonn.de/dmlg11/program.html

[16] N. Nejati, P. Langley, and T. Konik, "Learning hierarchical task networks by observation," in *Proceedings of the 23rd International Conference on Machine Learning*. ACM, 2006. doi: 10.1145/1143844.1143928. ISBN 1-59593-383-2 pp. 665–672.

[17] C. Hogg, H. Munoz-Avila, and U. Kuter, "Learning hierarchical task models from input traces," *Computational Intelligence*, vol. 32, no. 1, pp. 3–48, 2016. doi: 10.1111/coin.12044

[18] H. Xu, B. T. R. Savarimuthu, A. Ghose, E. D. Morrison, Q. Cao, and Y. Shi, "Automatic BDI plan recognition from process execution logs and effect logs." in *Engineering Multi-Agent Systems*, ser. LNCS, M. Cossentino, A. E. Fallah-Seghrouchni, and M. Winikoff, Eds., vol. 8245. Springer, 2013. doi: 10.1007/978-3-642-45343-4_15. ISBN 978-3-642-45342-7 pp. 274–291.

[19] G. Sukthankar and K. Sycara, "Robust and efficient plan recognition for dynamic multi-agent teams," in *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS, vol. 3, 2008. doi: 10.1.1.149.7677. ISBN 978-0-9817381-2-3 pp. 1383–1388.

[20] J. Hsieh, C. Sun, C. Wang, and C. Cheng, "Mining replays of real-time strategy games to learn player strategies," Canadian Artificial Intelligence Conference, 2008. [Online]. Available: http://gamelearninglab.nctu.edu.tw/ctsun/RTS%20player%20modeling.pdf

[21] B. Weber and M. Mateas, "A data mining approach to strategy prediction," IEEE Symposium on Computational Intelligence and Games, 2009. [Online]. Available: http://alumni.soe.ucsc.edu/~bweber/pubs/cig_2009.pdf

[22] A. Botea, M. Muller, and J. Schaeffer, "Using abstraction for planning in Sokoban," in *Proceedings of the 3rd International Conference on Computers and Games*. Springer, 2003. doi: 10.1.1.70.8121 pp. 360–375.

[23] "Unity3D," http://unity3d.com, accessed: 2015-11-11.

[24] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001. ISBN 0070131511

[25] M. Woolridge and M. J. Wooldridge, *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., 2001. ISBN 978-0470519462

[26] "Wargus," http://wargus.sourceforge.net, accessed: 2015-11-11.

[27] "BWAPI," http://bwapi.github.io, accessed: 2015-11-11.

[28] "RoboCup," http://www.robocup.org, accessed: 2015-11-11.

[29] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003. ISBN 0137903952

[30] R. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," in *Combinatorial Pattern Matching*, ser. LNCS, S. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, Eds. Springer, 2004, vol. 3109, pp. 400–408. ISBN 978-3-540-22341-2

[31] J. Seipp, F. Pommerening, G. Roger, and M. Helmert, "Correlation complexity of classical planning domains," in *Proceedings of the 8th Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, 2016, pp. 12–20. [Online]. Available: http://icaps16.icaps-conference.org/proceedings/hsdip16.pdf

[32] R. Liang, "A survey of heuristics for domain-independent planning," *Journal of Software*, vol. 7, pp. 2099–2106, 2012. doi: 10.4304/jsw.7.9.2099-2106

[33] R. Wille, *Formal Concept Analysis: Foundations and Applications*. Springer, 2005, ch. Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies, pp. 1–33. ISBN 978-3-540-31881-1

[34] S. O. Kuznetsov and S. A. Obiedkov, *Algorithms for the Construction of Concept Lattices and Their Diagram Graphs*. Springer, 2001, pp. 289–300. ISBN 978-3-540-44794-8