Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Strong Privacy-Preserving Multi-Agent Planner

Radek Bumbálek

Supervisor: Michal Štolba, Ph.D.
Field of study: Open Informatics
Subfield: Computer Games and Graphics
May 2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bumbálek** Jméno: **Radek** Osobní číslo: **459128**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Multi-agentní plánovač se silnou ochranou soukromých informací

Název bakalářské práce anglicky:

Strong Privacy-Preserving Multi-Agent Planner

Pokyny pro vypracování:

Cílem práce je implementovat a experimentálně vyhodnotit teoretický koncept multi-agentního plánovače s výpočetně silnou ochranou soukromých informací. Implementace bude založena na existujícím multi-agentním plánovači (PSM Planner) na frameworku pro secure multi-party computation (Sharemind). Úkolem studenta je:
- Nastudovat si relevantní literaturu a pochopit témata multi-agentního plánování a secure multi-party computation.
- Seznámit se s plánovačem PSM, frameworkem Sharemind a s implementací bezpečného průniku konečných automatů v jazyce SecreC.
- Integrovat plánovač PSM a framework Sharemind a implementovat nezbytné úpravy (např. rozhraní).
- Experimentálně vyhodnotit výsledný plánovací systém.

Seznam doporučené literatury:

- J. Tožička, J. Jakubův, A. Komenda. 'Privacy-concerned multiagent planning.' Knowledge and Information Systems 48.3 (2016): 581-618.
- J. Tožička, M. Štolba, and A. Komenda. 'The Limits of Strong Privacy Preserving Multi-Agent Planning.' in Proceedings of the 27th International Conference on Automated Planning
and Scheduling (ICAPS),In Press , 2017.
- D. Bogdanov. 'Sharemind: programmable secure computations with practical applications.' PhD Thesis, 2013.
- R. Guanciale, D. Gurov, and P. Laud. 'Private intersection of regular languages.' in Proceedings of the Twelfth Annual International Conference on Privacy, Security and Trust (PST) IEEE, 2014.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Michal Štolba, Ph.D., katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **15.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Michal Štolba, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgements

Primarily I would like to thank my supervisor Michal Štolba, Ph.D. for his guidance and willingness to help. I would also like to thank Riivo Talviste and Sharemind support team for responding my questions about Sharemind.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instruction for observing the ethical principles in the preparation of university theses.

Prague, 24 May 2019

Abstract

The objective of this thesis is to implement, experimentally evaluate and introduce a privacy-preserving extension of multi-agent planner PSM, using multi-party computation program Sharemind. The document also contains a description of used planning principles, as well as description of PSM planner and Sharemind.

Keywords: planning, planner, multi-agent planner, multi-party computation, privacy-preserving planner

Supervisor: Michal Štolba, Ph.D.

Abstrakt

Cílem této bakalářské práce je vytvořit rozšíření multi-agentního plánovače PSM Planner, které zachovává silnou ochranu soukromých informací. Tato ochrana je zajištěna pomocí programu Sharemind. Práce se zaměřuje kromě samotného popisu řešení i na představení plánovače PSM Planner, jakožto i možností programu Sharemind. Dále práce obsahuje experimentální analýzu řešení.

Klíčová slova: plánování, plánovač, multi-agentní plánovač

Překlad názvu: Multi-agentní plánovač se silnou ochranou soukromých informací

Contents

Figures

Tables

Chapter 1

Introduction

Automated planning ranks among classical artificial intelligence approaches. However, its problem-solving ability offers many practical applications. With a multi-agent approach, the number of possible applications raises. Multi-agent planning is divided into centralized and decentralized versions. The motivation behind the distribution of the planning process is mainly to preserve private information of planning agents. Up to date multi-agent planners prevent an intentional sharing of private information, but an unintentional leakage is often overlooked.

Automated and multi-agent planning with its general problem-solving ability has a high potential for use in many sectors, such as transportation, delivery services, but also creating evacuation plans or even space industry. All these possible usages are demonstrated in benchmark problems.

However for competitive industries, next to the ability to cooperate, there is also a need for preserving private data. On the one side, we want to share information about our possible actions, to enable cooperation and finding the best possible solution. On the other side, we want to hide our weaknesses or patents which might become targetable through cooperation.

The main focus of this work is to implement a strong privacy preserving multi-agent planner, which prevents an unintentional information leakage via an extension to PSM planner [1]. The extension is an implementation of the One-shot-PSM planner proposed theoretically in [2]. This work aims to assure computation safety by using a multi-party computation software Sharemind.

In this thesis, our goal is to describe the solution and present testing results. To assure proper understanding and sufficient knowledge base, the thesis starts with explanations and examples in the field of planning. The second part focuses on the solution itself, as well as experiments. A brief introduction to multi-party computation is also present.

Part I

Theory

Chapter 2

Automated Planning

The most basic description of automated planning is a process, which generates a problem-solving plan. However, this definition is very broad and therefore there are many types of planners. Before we can specify the planner, we need to define the problem in means of automated planning.

2.1 Domains and Problems

In the simplest form we assume there is an entity (agent) in a specific environment (world) where it can achieve some goal. The domain describes the world as a set of states and deterministic actions. An agent is in a state and applying an action will change it. The current state also defines which actions can be used. The state which agent is in, at the beginning of the planning process is called the initial state. The agent wants to achieve a goal state by applying a sequence of actions. If this sequence leads from the initial state to a goal state and it is executable, we call it a plan. The initial state and the goal state is the problem.

A formal definition used in most of the literature follows. We have not included events in this definition hence we are not considering them in problems. This definition otherwise implies the classical planning model.

- Π is a quadruple representing agent's problem: $< S, A, I, G >$
- S is a finite set of states
- A is a set of actions
- $I \in S$ is an initial state
- $G \in S$ is a goal state
- $\gamma : S \times A \to S$ is a transition function

If the domain is defined in this general manner, many different problems can be solved by the same planner. It is called domain-independent planning. Enhancing the planner with some specific information may cause a significant planning speed increase, but it also can restrain some planners from solving the problem. With the need for a specific planner for a specific domain, we use term domain dependent planning. We will focus only on domain-independent planning.

Example 1. *Imagine a navigation problem. Our agent is a car. States are towns. Actions are representing movement on roads between them. The initial state is then the town where we start and the goal is our desired destination.*

2.2 Planning as a state space search

In the initial state, we assume a set of applicable actions. Applying each of them reveals us new reachable states. If we apply actions recursively, we will obtain a graph structure of reachable state space. A graph can contain loops and cycles, which are undesirable. Search algorithms break these cycles and therefore we will refer to these graphs as state space trees. If the goal is in this space, the problem has a solution. The plan is a path in the tree leading towards the goal state leaf.

The planning algorithm then must search this tree to find a solution. Therefore it might appear that a planning problem shrinks to a search tree problem. It might seem that for example, a simple breadth-first search might be sufficient to solve any planning problem, but it would not be efficient. State space trees tend to grow at a very fast pace. The amount of memory or running time necessary to search through each possible state then grows over realistic values.

Example 2. *Imagine a previous navigation problem. With 10 roads leading from each town and 20 towns on our way, the state space tree will have over* 10^{20} *nodes. Even with navigation able to search 1 billion states per second, we would have to wait over 3 thousand years to get our path.*

As seen in **Example 2.**, it is necessary to reduce the state space size, prevent cycling and in general avoid the blind search. Therefore search algorithms are usually customized and enhanced. For classical planning problems where nonoptimal solutions are acceptable, the search algorithm that is used most frequently is Greedy Best-First Search [3]. Another strong enhancement is heuristic functions. A heuristic function evaluates each new state with an estimated cost or a distance from the goal. These estimations help to focus search towards some goal.

2.2.1 Planning as a plan space search

While the state space search uses actions to discover new states until a goal state is found. A plan is generated at the end of the algorithm, after finding the goal as a route to that goal. The plan space search uses the opposite approach. The plan space search instead of discovering new states by applying actions, it uses actions to discover new plans.

Adding, subtracting or changing actions in already created plans refines new plans. All plans create a plan space that is being searched through via these actions. If there is a plan which solves the problem, it is contained in the plan space. However, the plan space is infinite. This is one of the biggest drawbacks which causes plan space search algorithms to be in general slower and less used [3].

Also, a lack of explicit states prevents the usage of some heuristic and control knowledge. Therefore the state space search scales better with bigger problems [4].

The output of a plan state planner might not be a direct sequence of actions. Actions in a plan might be ordered more loosely, while some parts of the plan have to be executed in exact order, others have not. This allows alternative approaches to the plans, such as using a single data structure for storing multiple plans. A convenient structure is, for example, a finite state machine.

2.3 Multi-agent planning

There are several different motivations behind multi-agent planning. Each of them leads towards different problems and therefore different approaches. We distinct cooperative and opponent agents. We define a pair of opponent agents when their goals are different and reaching a goal state for one agent prevents the possibility to reach a goal state for the other agent [5]. Competitive agents often lead more towards game theory and strategical analysis, than planning, therefore we will be focusing on cooperative agents only.

With multiple agents, it is necessary to adjust our problem definition. For multi-agent planning, the most commonly used problem formalization is MA-STRIPS [7]. However, before we define our problems in MA-STRIPS, we have to adjust our previous definition to STRIPS [8].

- Π is a quadruple representing agent's problem: $< P, A, I, G >$
- P is a finite set of atoms (facts, propositions)

States defined earlier are collections of these propositions: $\forall s \in S, s \subseteq P$. Planning algorithms' main tool are actions. Their application expands the state space and thus allows finding goals. In STRIPS formalization actions now does not change whole state, but only some of atoms which define that state. This allows more compact representation, dividing action (a) into its preconditions $(pre(a))$, add effects $(add(a))$ and delete effects $(del(a))$. This is a formally written example for action a which changes state s_x into s_y:

$$s_x, s_y \in S \Rightarrow s_x, s_y \subseteq P, a \in A, a =< pre(a), add(a), del(a) >$$
$$(s_y = a(s_x)) \iff ((pre(a) \subseteq s_x) \wedge (s_y = (s_x \setminus del(a) \wedge s_x \cup add(a))))$$

MA-STRIPS is then just a simple extenstion:

- Φ is set of k agents, $\Phi = \{\varphi_i\}_{i=1}^{k}$
- The multi-agent problem M is a set of local STRIPS problems $M = \{\Pi_i\}_{i=1}^{k}$

2.3.1 Centralized

Example 3. *Imagine a delivery problem. We have two agents, one is collecting packages from senders and the other is then delivering packages to recipients.*

Example 4. *Imagine a traffic problem. We have two trains on one railway track. Each of them is heading towards different stations, but they might share part of their track, where they can not avoid each other.*

Both examples show us multi-agent problems, which can be solved by centralized planning, even though these problems are very different. In **Example 3.** we have two agents that share a common goal. They need to cooperate with their actions in order to achieve it.

In **Example 4.** goals of each agent are different. However, they share an environment. Actions of one influence the possibilities of the other. Therefore it is necessary to plan their plans together to avoid collisions.

Despite differences, if a problem can be solved centrally, we can merge agents with their states and then solve it as a one-agent problem. The initial state then became a combination of initial states of each agent. Even though the action of one agent will not change the current state of the other agent, merging their states creates a new combination. In **Example 4.** different goals can be again merged into one state, which fulfills goals of each agent. In

STRIPS representation, merging states is simple conjunction of atoms.

2.3.2 Distributed

The same problem, which can be solved by centralized planning can be also solved by the distributed version. Even though that problems might not differ (we could even reuse **Example 3.** and **Example 4.**), the procedures are very different. However, distributed planning is usually much more difficult to implement. Moreover, it does not guarantee higher time efficiency.

The reasons behind using distributed multi-agent planning derives from practical use. In real-life problems, we deal with multiple agents having their own goals, and it is often impractical or undesirable to create the plan for all agents centrally [6]. Also, decentralization may be forced by the uniqueness of each agent. Last but not least, we sometimes need to distribute the planning process in order to preserve the privacy of each agent.

In distributed planning, we have to distinguish actions as well as states. Some actions may influence other agents. Therefore it is necessary to share them so other agents could create their plans in accordance with possible actions of the first agent. Both states and actions are defined by atoms. Therefore it is logical to start distinguishing them first, into internal and public.

We call $p \in P$ internal for agent φ_i if other agents can not require p for any of their actions and none of their actions may have p in add or delete effects. If $p \in P$ is not internal then it is public.

- $p \in P_i^{int} \iff (\forall j \neq i)(\forall a \in A_j)(p \notin pre(a) \wedge p \notin add(a) \wedge p \notin del(a))$
- $p \in P^{pub} \iff (\forall i)(p \notin P_i^{int})$

The problem of the agent φ_i is then redefined as:

- $\Pi_i =< P_i^{int} \cup P^{pub}, A_i, I_i, G_i >$

If an action has a public fact in its preconditions, add or delete effects, it is public. Otherwise, it is internal. Public actions cannot be explicitly shared because they contain internal facts as well. Therefore we create a public projection $a^{\triangleright}$ of each public action a. A public projection of action contains only public facts in preconditions, add and delete effects.

- $a \in A_i^{pub} \iff (pre(a) \cup add(a) \cup del(a)) \cap P^{pub} \neq \emptyset$

- $a \in A_i^{int} \iff (pre(a) \cup add(a) \cup del(a)) \cap P^{pub} = \emptyset$
- $a^{\triangleright} =< pre(a) \cap P^{pub}, add(a) \cap P^{pub}, del(a) \cap P^{pub} >$
- $a \in A_i^{proj} \iff (\exists j \neq i)(\exists b \in A_j^{pub})(a = b^{\triangleright})$
- $\Pi_i =< P_i^{int} \cup P^{pub}, A_i^{int} \cup A_i^{pub} \cup A_i^{proj}, I_i, G_i >$

Sharing internal information would be at least ineffective because it would increase the size of the state space and therefore slow the algorithm. And in case of privacy concerned planning, needless sharing of any internal information goes against its principles.

We can now strictly distinguish which atoms and actions to share and which to keep private. By sharing necessary information, it is assured that a state space generated for each agent will contain goal states if they are reachable. However plan of one agent may depend on other agents and therefore it is necessary to ensure compatibility of plans. To do so, we introduce public plans and extensibility.

A plan π_i is a solution to a problem Π_i of an agent φ_i. The plan π_i contains both, internal and non-internal actions. However internal actions cannot influence other agents. We create a public plan $\pi_i^{\triangleright}$ by removing internal actions and creating public projections of public actions used. In order to ensure compatibility we only need to make public plans compatible, which means that by adding private actions of the agent φ_j to $\pi_i^{\triangleright}$, we can create a solution to the problem Π_j of the agent φ_j. Than we call the public plan *j-extensible*.

- π_i is a solution to a problem Π_i of an agent φ_i
- $\pi_i^{\triangleright} = \{a | a \in \pi_i \cap A_i^{proj} \vee (b \in \pi_i \cap A_i^{pub} \wedge a = b^{\triangleright})\}$
- $\pi_i^{\triangleright}$ is *j-extensible* $\iff \exists \pi_j, \pi_j^{\triangleright} = \pi_i^{\triangleright}$

If we find a public plan which is *extensible* for all agents, we found a solution of multi-agent problem M.[2]

Some multi-agent planners generate a few local plans for each agent and then compare their public projections until they find a match. Since security is our main concern, we will make a comparison using secure computation software.

Chapter 3

Secure computation

In the previous chapter, we have described a multi-agent planning problem. In order to implement a secure system, we can not let agents share their plans with others. Even though they need to compare only public projections of their plans, important data may leak during this comparison. Therefore we need a third party software to make a secure comparison of plans to find a global solution suitable for all agents.

There are many approaches to achieve security which means guaranteeing the correctness of the output as well as the privacy of the agents' inputs. Our choice may be influenced by various factors such as speed, scalability or a type of malicious agents. We can divide agents into:

- Honest - Normally working agent.
- Semi-honest (curious, passive) - Agent cooperates and follows given algorithm, however he is trying to obtain additional information about other agents.
- Malicious - Agent does not cooperate. His intentions may be corrupting system with false data or abort it completly.

We consider semi-honest (curious, passive) agents, which would not try to disrupt the whole process but cooperate to gather additional information about other agents. Let's describe the two most significant methods used, homomorphic encryption and multi-party computation.

3.1 Homomorphic encryption

Homomorphic encryption is a specific type of encryption, where encrypted data can be treated in a similar way as original data. We can apply the desired function on encrypted data and after decryption, we will get the same result as if we applied the function on the original input. Therefore the third party does not require decryption key and even if some data would leak, without the key would be worthless.

However classical homomorphic encryption allows only a specific function to be used, based on the encryption approach. Therefore it is impractical. More commonly used is full-homomorphic encryption, which allows almost any function to be applied and get a correct, encrypted solution.

3.2 Multi-party computation

The multi-party computation uses a different approach. Algorithms divide data into parts, which then can be processed individually and the final result is then composed of partial results. As long as some parts of the system remain secure, leaked data will not be sufficient to compose the original input. However, there is no general way of dividing data. It may depend on the type of data, as well as the type of operation.

Multi-party computation is a vast part of cryptography, offering computation safety as well as efficiency. It is often used for example for processing confidential data or data-mining. Therefore there are many solutions and programs offering privacy-preserving property.

Chapter 4

Related work

4.1 Privacy preserving planning

Despite a high amount of planners, privacy specialization is rare. Guy Shani in [13] offers a theoretical approach to algorithms. However, the most important foundation for this planner was [2]. In this paper are described as different approaches as well as their limits. Algorithms are divided into three basic types, each covering two of three main attributes as seen on **Figure 4.1**. Those attributes are *completness*, *Efficiency* and *Strong privacy preserving*. Paper also contains the very important theorem of impossibility, which precludes algorithm having all three attributes at once.

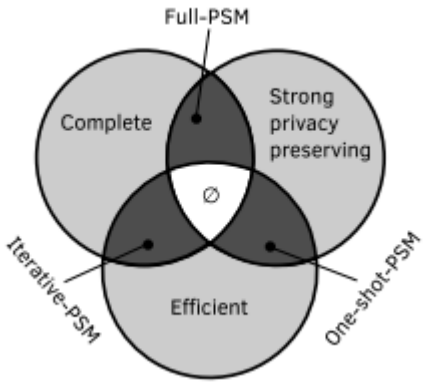

Figure 4.1: Scheme of PSM-based planners properties, from [2]

Brafman describes an algorithm as strongly private if no agent can deduce information beyond the information that can be deduced from its own actions' description, the public projection of other agents' actions, and the public projection of the solution plan, about the existence of a value or a variable that is private to another agent, or about the model of an action [12].

Completeness is perceived as the ability of a planner to always find a solution if the solution exists. Efficiency is an ability to find a solution without

searching through the whole state space. A state space is usually enormous and being able to find the solution without the need to search through each state offers a major speed improvement.

In the paper [2] are considered 3 algorithms. Iterative planning, where agents generate only a small amount of plans and exchange their public projections. If they match, a solution is founded. If not, agents use public projections of other agents to enhance their search. They evaluate public actions used by other agents in order to increase the probability of the appearance of these actions in their next plans. It is considered to be *Complete* and *Efficient*. However not *privacy preserving*, because by each comparison, agents can predict the existence of some private atoms of other agents. The only way how to prevent data leakage is by applying comparison only once.

The remaining two algorithms are similar. It is the Full and the One-shot algorithm. The full version considers searching through the whole state space, generating every plan of every agent. If there is a solution, then it has to be found. A comparison of a set of plans is then applied only at the end of the algorithm and therefore no additional information can not leak. However, searching for all plans is very inefficient. For big problems, there is no guarantee that the program would finish in a reasonable time.

Therefore we decided to implement a One-shot algorithm. Each agent generates only a limited set of plans. The limit may be a number of plans, however since a run time is our other concern, next to privacy-preserving, we use a time limitation. After a specified time period, agents shut down their planning algorithms and gather all plans generated so far. Then plans are compared. We can guarantee *Efficiency* and *Privacy preserving*, but a solution may not be found.

Even though that lack of *Completness* may seem like a big drawback, it is still the better option, since the only other option to keep *Privacy preserving* ability is to search the whole state space. On big problems, it is practically guaranteed that the Full algorithm would not end in a specified time and therefore it would fail. The One-shot gives us at least a chance to actually find a solution.

4.2 PSM planner

The privacy-preserving planner is an extension to PSM planner [14]. PSM means Planning State Machine. It is a specific structure for storing plans, which PSM planner uses. The planning state machine is a finite state machine structure modified for planning purposes. However, in our solution, we omit PSMs and we make use of standard plans.

The PSM planner does not contain a planning algorithm implementation. It preprocesses data from PDDL to STRIPS (MA-STRIPS) format, via a process called grounding. Then it creates threads for each agent as well as a single special thread called broker, which synchronizes agents and processes an output. Each agent runs a fast-downward [15] planner via a script. The PSM planner is implemented in Java. The algorithm in detail is described in chapter 5.

4.2.1 Fast-downward

Fast Downward is a classical planning system based on ideas of heuristic forward search and hierarchical problem decomposition [16]. Fast-downward itself is a state of the art planner. Moreover, it can process landmarks of states in order to enhance the heuristic search. This was specifically useful for the original iterative multiagent planning. Reusing Fast-downward provides a more compact implementation.

4.3 Multi-party computation

As mentioned in chapter 4, multi-party computation is a vital part of secure privacy-preserving planning. Desired operation to compute is a set intersection. Even though this operation is among basic MPC functions and division of threads into independent agents leads to MPC solution, we have decided to use a trusted third party software which guarantees data security.

In [2] is proposed the usage of the privacy-preserving intersection of deterministic finite automata [18]. The presented algorithm can use finite state machines as an input and is implemented in the language secreC 2, executing on the Sharemind platform. We decided to implement a set intersection of plans instead. The reasons were mainly practical. Operating directly with plans allows easier testing. The algorithm itself is simplified, as well as the input. However, we decided to use Sharemind as a solution platform, mainly for its security guarantees.

4.3.1 Sharemind

Sharemind is a novel database and application server that collects data in an encrypted form and uses techniques like homomorphic encryption, secure multi-party computation and hardware isolation to process it without leaking the private inputs even to the machine memory, providing end-to-end encrypted data processing [17].

The solution was implemented on the Sharemind Academic Server environment, which is distributed as a virtual box image, running on Debian. The

Sharemind Academic Server also contains (with a need of a specific license) a CSV-importer, which allows importing data into the database via CSV and XML files.

Sharemind offers variable licenses. The Application Server License runs computations on separate servers. In our case, for testing purposes, we use local separated threads instead of dedicated servers. However, the application works identically. In both cases, scripts are written in SecreC 2 language and the approach is the same, thus functionality on separated servers is guaranteed.

Part II

Solution

Chapter 5

Implementation

The solution consists of the extension of PSM-planner written in Java and the Sharemind set intersection script written in SecreC 2. Moreover, we used bash scripts for tests and evaluation, which will be briefly mentioned in the sixth chapter.

5.1 Planner

The PSM planner run changes dramatically based on the problem which it is solving. Also, multiple simultaneously running threads make description more difficult. We can divide the process into preprocessing of the problem and planning itself as well as threads can be distinguished as Agents and a Broker.

An agent can not communicate with other agents. They are all connected to the broker (centralized communication) who provides information exchange. For a better understanding of changes made a description of the original program follows.

5.1.1 Original PSM planner

The original PSM-iterative planner starts with the preparation of a problem and a domain (viz uml sequence diagram **Figure 5.1**). This preprocess does not only prepare data for the planner but also contains a relaxed planning algorithm and sets initial landmarks. Landmarks are used to evaluate the probability of actions to lead towards the solution. The planner is trying to primarily use marked actions.

Preprocessing does not share private actions and facts directly. However, there is a possibility of privacy information leakage because of public fact sharing during relaxed planning. Determining the amount of information leaked is difficult. Removing completely relaxed planning would need changes in the whole preprocessing. Without the possibility to measure information

leakage, we can not easily adjust the algorithm and finally any changes of the preprocessing algorithm would need completely new testing, which is very time demanding.

Therefore we decided to keep this change for future work and not change preprocessing when implementing our extension. Also, input is strictly determined in [19] and thus will not be changed.

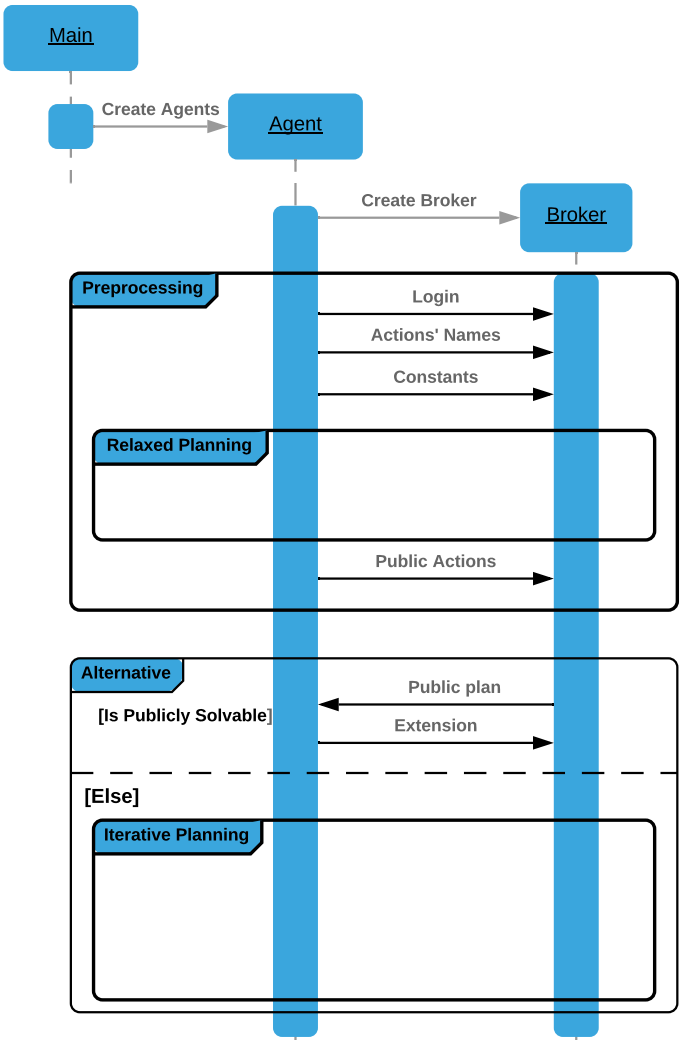

Figure 5.1: Distributed planning algorithm scheme

Preprocessing

The main function needs a location of the factored problem directory. Based on files inside this directory, corresponding agents' threads are created as well as the broker. After connecting each agent to the broker, problems are read.

For purposes of easier composing of the output, the algorithm starts with exchanging original names of actions.

Each agent determines and exchanges constants. Constants are predicates which cannot be changed by actions and thus remain constant during planning. Next step is relaxed planning, depicted by scheme **Figure 5.2**.

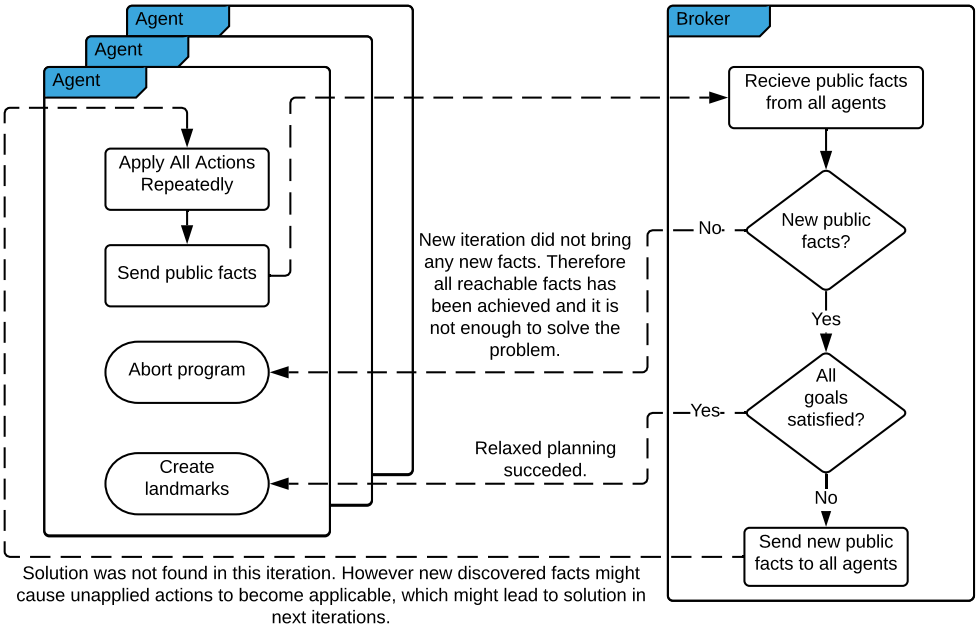

Figure 5.2: Relaxed planning scheme

The agent applies repeatedly all actions as long as new facts are generated. Then it sends those facts, which are public to the broker. If the broker does not acquire new public facts from at least one agent, it means that all possibilities have been searched through and the solution was not found. If it gets some new public facts, compares it with goals to be reached. If all goals are reached, the algorithm ends successfully. If goals are not reached, all public facts are sent back to agents. At least some agents acquire new facts which may lead to new applications of actions, thus to new reachable facts to be revealed.

Relaxed planning is a heuristic approach where planning proceeds without negative effects of actions. As a heuristic, its function is to fasten planning, which is achieved by evaluating facts by landmarks. However relaxed planning fulfills more purposes in this case. It also provides reachability analysis. If goals are not reachable by relaxed planning, the problem does not have a solution and planning ends.

Otherwise, the algorithm continues by grounding and exchanging public projections of actions. Now it is determined whether the goal of each agent can be reached by his own actions only. If so, the broker generates a public plan and agents only create their extensions to that plan. The output is then generated and the program ends. Otherwise, the distributed planning itself takes place.

Planning

After grounding problems into STRIPS, Fast-downward can be used. The broker calls a new iteration and each agent uses FD to generate several plans. Then public projections are created and compared. Each agent checks the extensibility of each plan. If there is a plan extensible by all agents, the program found a solution and generates output.

However, if there is no plan extensible by all agents, planning continues. Non-extensible plans are used to adjust landmarks of public actions used in order to higher the chance of finding an extensible plan in following iterations. The broker then calls the next iteration and the planning process repeats until a solution is found.

5.1.2 One-shot algorithm implementation

As being said earlier, the preprocessing remains the same (viz uml sequence diagram **Figure 5.3**). We have only slightly adjusted an input in order to ease testing. Making a problem and a time limit as main functions' arguments allows us to use bash scripts to solve several problems in an automated sequence. Because the preprocessing is not changed, in some cases planner solves the problem even without calling the One-shot algorithm.

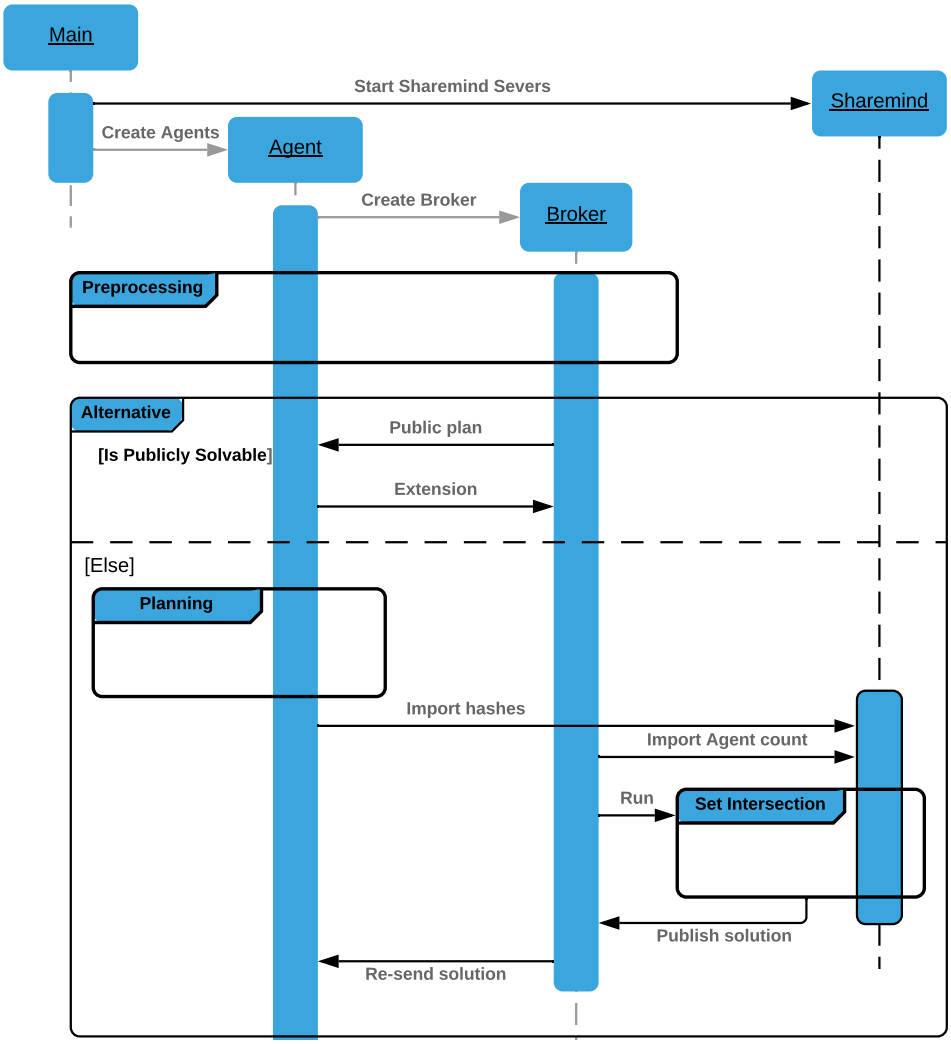

Figure 5.3: One-shot planning algorithm scheme

After grounding One-shot agents call Fast-downward in a loop. Each call returns several plans which are saved for future use. When the time limit expires, all Fast-downward threads are killed. Agents then make public projections of plans they have found. Different plans may share the same public projection. These duplicates are found and deleted. We always compare plans before deletion and the shortest plan is kept, while others are removed.

To justify the next step, we need to explain the Sharemind input first. Sharemind itself does not have a simple input, the language secreC 2 does not allow scripts to be run with additional arguments. Sharemind was implemented for operations over big tables of data, therefore we need to create these tables.

The iterative algorithm makes use of knowledge of how similar plans are. Based on similarity and usage of public actions, landmarks are set. However,

in the One-shot algorithm, only one comparison is made. Any additional information could not be used anyway. Therefore we only need a simple comparison, whether the plans are the same, or not. Therefore we can use encryption on the whole plan. We decided for hashing our public projections with SHA256.

Because public plans are strings of variable lengths usually much longer than is a hash size, it is possible that different plans would be encrypted with the same hash. However, it is extremely unprobable. In test situations we usually have 2-8 agents, each generating roughly 30-100 original public projections. Counting the probability leads to the so-called birthday problem. Rough estimation via Taylors polynomial leads to:

$$p(n, d) \approx 1 - e^{\frac{-n^2}{2d}}$$

Where d is size of hash $d = 2^{256}$ and n is a number of plans. Even if we use a ridiculously huge estimation such as $n = 10^{10}$ we still get the probability of collision:

$$1 - e^{\frac{-10^{20}}{2^{257}}} = 4.31 \times 10^{-68}$$

Hashed plans are then saved into the table database as sets of four 64bit Longs. Hashes are much easier to compare than sets of strings of variable length. Also hashing increase security in case of eavesdropping on the input or the output of the Sharemind script. The table database is created via Sharemind CSV importer. Each agent creates a CSV file containing hashes and XML file with a description of tables to be imported. Then they call the import command.

The broker also creates CSV and XML files with simple information about the number of agents for the Sharemind script. Name conventions are based on the numbering of agents, therefore knowledge of their count is crucial. After all, agents import their hashed plans, the broker calls set intersection script. The script returns a hash of a common plan if there is at least one. The broker then resends it back to agents so everyone could reload the original plan based on the hash and create an output.

For purposes of testing, there is a specific pattern of the output, where all actions (both public and private without distinction) are numbered by their order and saved in a single file. In order to achieve it, agents send their private plans back to the broker which merges them. However, this option is only for testing purposes and thus violates privacy preserving quality. In normal use this function would be turned off (by a special variable).

5.2 Sharemind

Sharemind is operated via built-in commands and created scripts. These scripts as already mentioned are written in the secreC 2 language. This language is rather limited and resembles the C language. The specific part of this language is privacy types, which are annotated with a privacy domain [20].

Private variables are being treated in a very specific way. Each variable is divided into three parts, one for each sharemind servers. Every operation over these variables is done in means of multi-party computation. Declassification can be done only by a declassify expression.

5.2.1 Set intersection

We get a set intersection by a simple comparison each element of a set A with each element of a set B. The set created by comparison of two previous sets is then used for comparing with the next set until the complete intersection is made.

```
pd_shared3p int64[[2]] intersection = loadSet(dataSource, "table0");
for(int64 i = 1; i < agentCount; i++){
    pd_shared3p int64[[2]] tmp
                    = loadSet(dataSource, "table" + arrayToString(i));
    intersection = setIntersection(intersection, tmp);
}
```

Figure 5.4: Set intersection algorithm

Function loadSet (**Figure 5.4**, line 1) is used to load data from the table database into 2 dimensional privacy type array denoted as pd_shared3p <Type> [[n of dimensions]]. Function setIntersection (in detail **Figure 5.5**) makes an intersection of given arguments, by comparing each member of the first argument, with each member of the second argument.

```
template<domain D : shared3p, type T>
D T[[2]] setIntersection(D T[[2]] setA, D T[[2]] setB) {
     uint64 a = size(setA)/4;
     uint64 b = size(setB)/4;
     pd_shared3p int64[[2]] intersection(4, max(a, b));

     uint64 counter = 0;
     for (uint64 i = 0; i < a; i++){
         for (uint64 j = 0; j < b; j++){
             if ( !declassify((bool)(setA[0, i] - setB[0, j]))
               && !declassify((bool)(setA[1, i] - setB[1, j]))
               && !declassify((bool)(setA[2, i] - setB[2, j]))
               && !declassify((bool)(setA[3, i] - setB[3, j])) ){
                 intersection[0:, counter] = setA[0:, i];
                 counter++;
                 break;
             }
         }
     }
     return intersection[0:, 0 : counter];
}
```

Figure 5.5: Set comparison algorithm

Because of the encryption, we can not directly declare whether two numbers are equal. However, we can apply basic arithmetic operations such as subtraction, or retype the variable from a number to a boolean. By a subtraction and retyping is guaranteed that the declassification of the variable will not expose original values.

The output of the script is the first hashed plan in the resultant set or an empty set in case the algorithm will end with an empty intersection. The output is read by the broker and resent to agents. In case of an empty intersection, the broker terminates the program.

Chapter 6

Experiments

Experimental testing is divided into two main parts. Testing of the planner and testing of the sharemind set intersection algorithm. Because of the character of the one-shot algorithm, we can not easily compare its planning speed, because it will always use the maximum time given for planning. However, we can observe efficiency with changing time. Also, we want to measure a time consumption of the sharemind set intersection script. We need to account this time when we start our planner in order to assure that the algorithm will finish before the time limit.

6.1 Planner experiments

Our main concern is planner efficiency in the time limit given in [19], which is 30 minutes. We have prepared testing bash scripts, which are included in planner files (TestScript.sh, ValidationScript.sh, iterativeScript). As noted in chapter 4, sharemind is distributed as a virtual box image. The virtual box allows us to strictly set properties of the virtual machine. We used settings from [19] which allows 4 thread processor and 8GB of RAM.

Firstly we denote domains (viz. **Table 6.1**), where a public plan solution was found and the one-shot algorithm itself was not started.

Domain	Solved
Blocksworld	20/20
Depot	17/20
Driver log	20/20
Logistics	20/20
Sokoban	17/20
Taxi	20/20
Woodworking	20/20

Table 6.1: Problems solved during preprocessing

The one-shot algorithm was used to solve domains: Elevators, Rovers, Satellites, and Zenotravel. We tested the algorithm with three settings, giving

the planning part 100, 300 and 1740 seconds. We assume one-minute time reserve for the preprocessing and the set intersection.

	Elevators			Rovers			Satellites			Zenotravel		
	100	300	1740	100	300	1740	100	300	1740	100	300	1740
1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
2	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
3	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
4	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
5	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
6	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
7	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
8	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
9	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
10	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗
11	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗
12	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
13	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗
14	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
15	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
16	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
17	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
18	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
19	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
20	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table 6.2: Problems solved by One-shot algorithm

Even though the planner did not solve even half of the problems, the efficiency has to be put in the context by comparing it with the original planner. Virtualization and running all agents on a single device may lower efficiency significantly when compared with the results of the CoDmap competition. Therefore we tested the iterative algorithm as well.

We can see that in some domains (**Table 6.2** - Elevators, Zenotravel) our solution offers unconvincing results. However, problem type Rovers and Satellites were solved with noticeable success. In some domains, the planner succeeded even with a small amount of planning time, but efficiency boost with a long time is rather limited. This is caused by the rapidly growing state space during a search. The pace in which the planner is finding new plans is declining rapidly with time.

6.1.1 Validator

CoDmap competition organizers also offer a validation program that checks whether the given plan fulfills all goals and it is executable. However, a special format of the plan is needed. This format contains all actions in one

Domain	Iterative	One Shot	Total
Blocksworld	20	20	20
Depot	17	17	20
Driver log	20	20	20
Elevators	12	1	20
Logistics	20	20	20
Rovers	14	12	20
Satellites	11	11	20
Sokoban	17	17	20
Taxi	20	20	20
Woodworking	20	20	20
Zenotravel	8	0	20
Σ	179 (81.4%)	158 (71.8%)	220 (100%)

Table 6.3: Problems solved by iterative planning

file, with a given order.

Example 5. *Part of solution from Satellite problem 2 (directory p6):*
21: (take_image satellite0 instrument0 star9 infrared1)
22: (turn_to satellite2 star7 star10)
23: (take_image satellite2 instrument4 star7 infrared3)
24: (switch_off satellite1 instrument3)

Therefore plan extensions have to be passed to the broker which orders them and creates an output compatible with the validator. However, this process violates the strong privacy preserving character of the algorithm, therefore it is included only as an optional function.

All plans created by our one-shot algorithm had passed the test, validator marked them as possible and leading to the goal.

6.2 Sharemind experiments

We prepared a program for generating sets of hashes to simulate problems that the Sharemind script will be solving. Because of the character of used data (hashes) artificially manufactured sets are indistinguishable from the original. Therefore we can faithfully imitate real problems in a controlled environment.

We can control nature of the problem by 3 main attributes: Sizes of sets, Number of sets (corresponding to the number of agents) and similarity rate,

which defines percentage loss of plans in each intersection iteration.

Example 6. *With 3 sets of size 100 and the similarity rate 50%, each iteration eliminates 50% of plans. After 1 iteration (comparing the first 2 sets) only 50 plans remain. After the second iteration (the result of the first iteration intersected with the third set) only 25 plans remain.*

We assume 5 sets, because of the average number of agents. We also assume each set holding 200 members. Higher set sizes allow us to better observe similarity rate impact because, with a lower number of members in each set, low similarity rate would cause intersection to contain only one member in a small number of iterations.

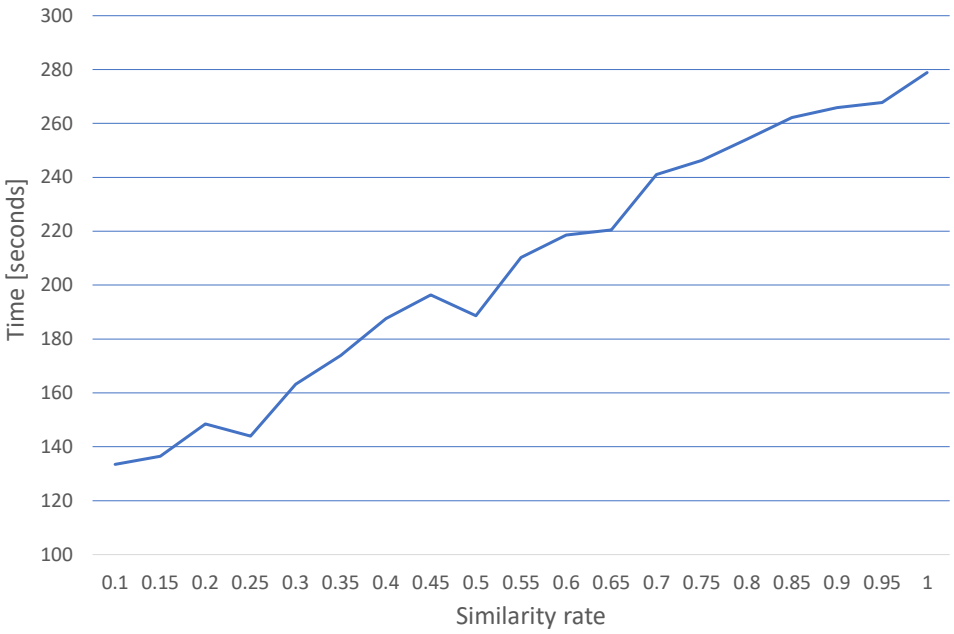

Figure 6.1: Sharemind script time complexity based on similarity of sets

We can see in **Figure 6.1** that the growth is unstable. This is caused by the random shuffling of set members. In some occasions, sets intersect very fast with a comparison of the only the first couple of members. The role of chance is more noticeable when a difference between member count of sets is higher. This difference lowers with higher similarity rate.

Example 7. *With similarity rate 1.0, member count of each set 75, we compare 5 identical sets in 4 iterations. When all set members are present in both sets, we always make* $\sum_{i=1}^{75} i$ *comparisons per iteration, 1560 comparisons in*

total, with no difference between shuffled and unshuffled sets. But with a low similarity rate such as 0.2, we compare 75 with 75 members only in the first iteration. In the second iteration, we will compare only 15 members with 75 and in the third only 3 members of the intersection with 75 members of the fourth set. Based on shuffle, we can make between 6 (1 + 2 + 3) and 222 (73 + 74 + 75) comparisons.

Next, we examined time complexity grow with an increase of set sizes. We were always comparing 5 sets and a similarity rate of 75%. We use a higher similarity rate in order to lower the impact of the random plan shuffle.

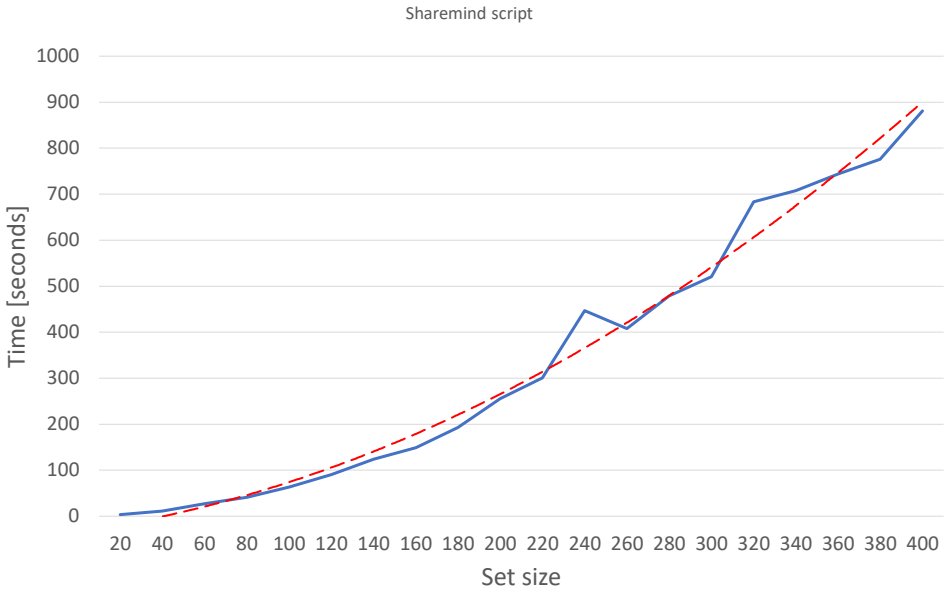

Figure 6.2: Sharemind script time complexity based on set size (blue), and polynomial approximation of rank 2 (red).

The intersection of n sets with m members is done by comparing their $m \times m$ members n times which leads to $O(m^2)$, as observed from the experiment in **Figure 6.2**.

Lastly, we observe time complexity with the growing number of sets. We used to set size 200 and 75% similarity rate.

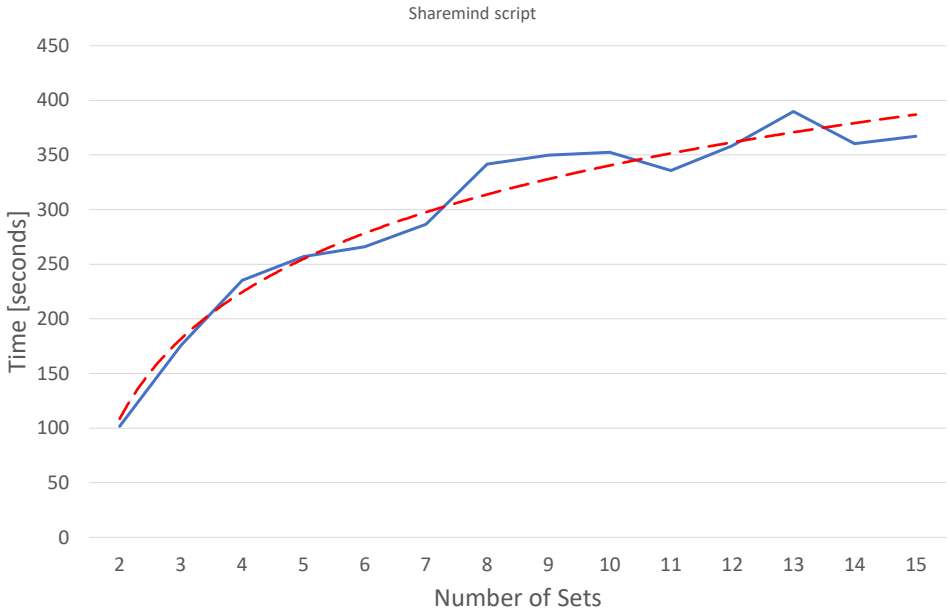

Figure 6.3: Sharemind script time complexity based on number of sets (blue), and logaritmic approximation (red).

As we can see, even with a high amount of agents, time complexity grows slowly, because intersection will get very small after a couple of iterations as seen in **Figure 6.3**.

In conclusion, we are aware, that our proposed set intersection algorithm is not very efficient. Higher efficiency is usually achieved by sorting sets, which allows their comparison in a single run. However, in order to use such a method, we need to not only compare two members but also to determine which of the members is higher. However, we do not want to declassify the actual values of variables.

We can see clearly from experiments, that for normal size of problems (50-120 members in each set) is our solution sufficient. Even a growing number of agents will not increase the time cost significantly. Time complexity starts to be problematic with big sets of plans. However, to find more plans, we have to search for greater state space, where complexity grows even faster.

6.3 Conclusion

Our goal was to implement and test a one-shot algorithm as a PSM planner extension. Implementing the extension has been successful, however for perfect privacy preserving capability, adjusting preprocessing would be necessary. We leave this adjustment for future work.

Our solution has proven to be capable of solving some problems. We have noticed only a slight decline in the efficiency of the planner (9.6% due to **Table 6.3**), however, this is caused by the majority of problems being solved in preprocessing. If we compare only domains where One Shot-PSM take place, the decline is more significant, but this was the expected outcome of strong preserving of private information.

We also experimentally used and tested Sharemind software for secure multi-party computation. Our script proves efficient enough not to disrupt planning potential with restricting planning time too much. For future work we expect a simple set intersection to be exchanged for planning machine intersection, which offers higher potential success.

To sum up, the original goal of creating a PSM extension and Sharemind script has been achieved.

Bibliography

[1] Jan Tožička, Jan Jakubův, Antonín Komenda. *PSM-based Planners Description for CoDMAP 2015 Competition*. CoDMAP 2015, pg.29-32.

[2] Jan Tožička, Michal Štolba, Antonín Komenda. *The Limits of Strong Privacy Preserving Multi-agent Planning*. Published in ICAPS 2017.

[3] Malik Ghallab, Dana Nau, Paolo Traverso. *Automated Planning and Acting*. Published by Cambridge University Press, ISBN: 9781107037274, August 2016.

[4] Malik Ghallab, Dana Nau, Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers. ISBN 1-55860-856-7, May 2004.

[5] Andrea Bonisoli. *Distributed and Multi-Agent Planning: Challenges and Open Issues*. ISSN 1613-0073. 2013.

[6] Mathijs De Weerdt , Adriaan Ter Mors , Cees Witteveen. *Multi-agent planning: An introduction to planning and coordination*. In: Handouts of the European Agent Summer School. 2005.

[7] Ronen I. Brafman, Carmel Domshlak. *From One to Many: Planning for Loosely Coupled Multi-Agent Systems*. In: Proceedings of ICAPS'08, vol 8, pp 28–35. 2008.

[8] Richard E. Fikes, Nils J. Nilsson. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Published in Artificial Intelligence 2 (1971) by North-Holland Publishing Company

[9] Jan Tožička, Jan Jakubův, Antonín Komenda, Michal Pěchouček. *Privacy-concerned multiagent planning*. In Knowledge and Information Systems, Vol 48, pp 581-618. ISSN 0219-1377. 2016.

[10] Ronald Cramer, Ivan Damgard, Ueli Maurer. *General Secure Multi-Party Computationfrom any Linear Secret-Sharing Scheme*. In: Preneel B. (eds) Advances in Cryptology — EUROCRYPT 2000.

[11] Oded Goldreich. *The Foundations of Cryptography - Volume 2*. Chapter 7. ISBN 0-521-83084-2. Published by Cambridge University Press. 2004.

[12] Ronen I. Brafman. *A privacy preserving algorithm for multi-agent planning and search.* In IJCAI, pages 1530–1536, 2015.

[13] Guy Shani. *Advances and Challenges in Privacy Preserving Planning* In IJCAI, pages 5719–5723, 2018.

[14] https://github.com/gree7/psm-planner/tree/master

[15] http://www.fast-downward.org/

[16] Malte Helmert. *The Fast Downward Planning System.* In Journal of Artificial Intelligence Research 26, pp. 191-246. 2006.

[17] https://sharemind-sdk.github.io/

[18] Roberto Guanciale, Dilian Gurov, Peeter Laud. *Private Intersection of Regular Languages.* In Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conferencen, pg. 112–120. IEEE.

[19] http://agents.fel.cvut.cz/codmap/

[20] https://sharemind-sdk.github.io/stdlib/reference/index.html