Karlsruhe Institute of Technology

Parallelizing Graphplan

Bachelor's Thesis of

Patrick Hegemann

at the Department of Informatics
Institute of Theoretical Informatics, Algorithmics II

Advisor: Dr. Tomáš Balyo
Second advisor: Prof. Dr. rer. nat. Peter Sanders

May 2018

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 14.5.2018

.....

(Patrick Hegemann)

Abstract

Automated Planning is a method used in artificial intelligence systems such as autonomous robots or automatic satellite control. While there already are some planning algorithms that make use of multi-core processors, this thesis proposes a parallelized version of the Graphplan algorithm. The proposed algorithm works by transforming plan extraction problems for different horizon lengths into the Boolean Satisfiability Problem (SAT) and solving multiple SAT formulas in parallel. Experiments based on the problem instances provided by the 2014 International Planning Competition show that this method results in a significant parallel speedup and is able to outperform the state-of-the-art SAT-based sequential planner Madagascar in some problem domains. A disadvantage of the proposed algorithm is high memory consumption for some problems with certain configurations.

Zusammenfassung

Automatisierte Planung ist eine Methode, die in Systemen der künstlichen Intelligenz wie beispielsweise autonomen Robotern oder automatischer Satellitensteuerung eingesetzt wird. Während es bereits einige Planungsalgorithmen gibt, die mehrere Rechenkerne von Multi-Core-Prozessoren nutzen, schlägt diese Arbeit eine parallelisierte Variante des Graphplan-Algorithmus vor. Der vorgeschlagene Algorithmus funktioniert indem er die Extrahierung von Plänen aus mehreren Ebenen des Planungsgraphen in das Erfüllbarkeitsproblem der Aussagenlogik (SAT) transformiert und mehrere SAT-Formeln parallel löst. Experimente, die auf den Probleminstanzen der 2014 International Planning Competition basieren, zeigen, dass diese Methode zu einem signifikanten parallelen Speedup führen kann und den state-of-the-art SAT-basierten Planungsalgorithmus Madagascar in einigen Problemdomänen übertrifft. Ein Nachteil des vorgeschlagenen Algorithmus ist der hohe Speicherbedarf für einige Probleme bei gewissen Konfigurationen.

Acknowledgements

I want to thank my supervisors Prof. Dr. Sanders, for providing the necessary resources to accomplish this work, and Dr. Balyo for all our interesting and fun conversations and for always giving good advice. I also thank my family and friends for their support.

Contents

1 Introduction

Automated Planning is a field of Artificial Intelligence (AI) that deals with finding sequences of actions that transform a world state from a given initial state to a desired goal state. This is a key ability for a lot of AI systems such as autonomous robots [1] or the automatic control of satellites [2].

The development of modern processing units is currently going in the direction of adding more cores to processors instead of increasing the clock rate. However, this presents a challenge for software developers as applications must be adapted to run on multi-core processors in order to utilize this additional computational power [3]. Parallel algorithms for many applications, e.g. SAT solving have shown that this effort can result in a substantial speedup compared to sequential algorithms [4].

There are numerous sequential and parallel algorithms for solving planning problems, such as IBaCoP [5], Madagascar [6] and Arvandherd [7]. An approach that hasn't yet been parallelized is that of the Graphplan algorithm first proposed by Blum and Furst in 1997 [8]. Graphplan works by repeatedly expanding a graph structure (the *Planning Graph*) and extracting a plan using this graph. Certain algorithms based on Graphplan use SAT solving in the extraction step which proved to be a very efficient strategy [9, 6].

A mechanism to achieve even better performance is to systematically skip the plan extraction in certain iterations of the algorithm [6]. The idea of using this in a parallel algorithm is promising, because several computationally expensive extraction processes can run on multiple processor cores at the same time. This thesis proposes such a

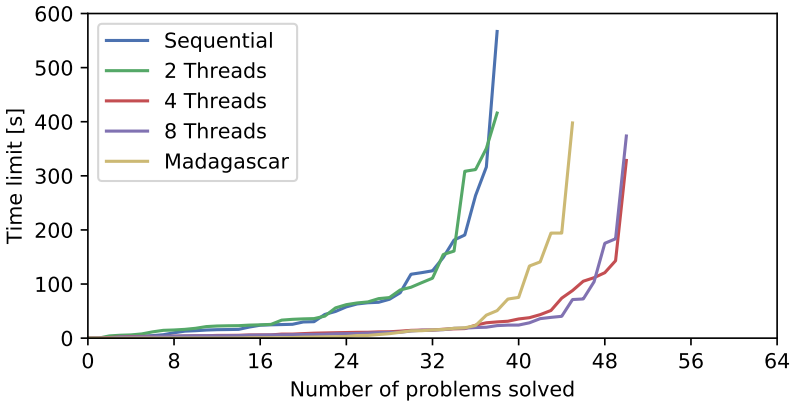


Figure 1.1: Comparing different configurations of the implemented algorithm to the state-of-the-art sequential planner Madagascar

parallel algorithm in order to use the computational power of multi-core processors and thus achieve a further speedup. Figure 1.1 shows a preview of the results regarding parallel speedup.

The thesis starts by introducing the basics of planning and SAT solving in section 2. Section 3 describes the algorithms and techniques that this work is based on. In section 4 a parallelized version of the Graphplan algorithm is proposed. Section 5 shows the implementation of the proposed algorithm and highlights relevant details. Section 6 shows the parallel speedup of the algorithm in different configurations and a comparison to the state-of-the-art SAT-based planner Madagascar. The thesis is concluded in section 7 which will also discuss ways in which the algorithm can be improved further in future work.

2 Background

The definitions in this section are based on [10, 11, 12].

2.1 Planning

Classical planning problems can be defined as follows [10]: A planning problem instance Π is a tuple $(\mathcal{X}, \mathcal{A}, s_I, s_G)$ where

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of multivalued variables with finite domains $dom(x_i)$, i.e. each variable $x \in \mathcal{X}$ has a set of possible values $dom(x)$.
- $\mathcal{A}$ is a set of actions (or operators). Each action $a \in \mathcal{A}$ is a tuple $(pre(a), \mathit{eff}(a))$. Both $pre(a)$ and $\mathit{eff}(a)$ are sets of assignments of variables in $\mathcal{X}$, they are of the form $(x = v)$ where $x \in \mathcal{X}$ and $v \in dom(x)$. $pre(a)$ is the set of preconditions which must hold in order for a to be executed. $\mathit{eff}(a)$ is the set of effects that hold after the execution of a.
- s_I is the initial state, a full assignment of the variables in $\mathcal{X}$.
- s_G is the goal state, a partial assignment of the variables in $\mathcal{X}$.

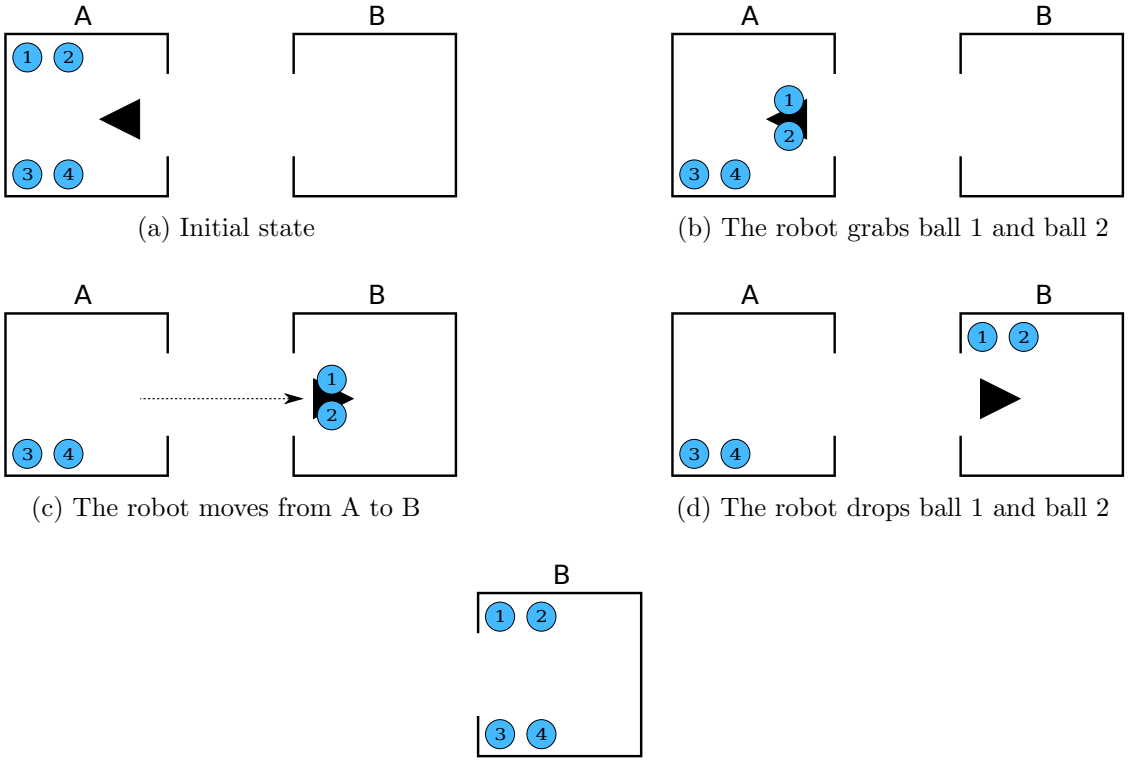
A state is a full assignment of the variables in $\mathcal{X}$, i.e. each variable $x \in \mathcal{X}$ is assigned exactly one value from its domain $dom(x)$. Applying an action $a \in \mathcal{A}$ in state s means changing into a state s' where $\mathit{eff}(a) \subseteq s'$ and the difference between s and s' is minimal. The planning problem for an instance Π is to find a sequence of actions $a_1, a_2, \ldots, a_n$ such that

- All the actions are well-defined within the problem: $\forall i : a_i \in \mathcal{A}$
- Let s_1 be the initial state, and s_{i+1} the state after application of a_i in s_i
- All actions are applicable in their respective state: $\forall i : pre(a_i) \subseteq s_i$
- After application of all actions a_i, the goal is satisfied: $s_G \subseteq s_{n+1}$

The planning problem definition can be varied in numerous ways. Examples for this are cost-optimal planning (plans must be minimal in regard to action costs) and temporal planning (execution time and duration of actions is considered). This thesis will only consider satisficing classical planning problems, i.e. the objective is to find any plan that solves a given problem as defined in this section.

2.1.1 Example: Gripper problem

The Gripper problem is a classical planning domain used in the 1998 AI Planning Systems Competition [13]. It is about a robot with two grippers that needs to transport balls from one room to another. Figure 2.1 considers an example that starts with four balls and the robot in room A, and no balls in room B. As figure 2.1e shows, the goal state is that all four balls are in room B. Note that the goal state doesn't make a statement about the robot's position or the contents of room A, as it is just a partial assignment to the problem variables.



(a) Initial state

(b) The robot grabs ball 1 and ball 2

(c) The robot moves from A to B

(d) The robot drops ball 1 and ball 2

(e) Goal: all four balls are in room B

Figure 2.1: Example of a planning problem from the Gripper domain

This problem instance can be formalized as a tuple $(\mathcal{X}, \mathcal{A}, s_I, s_G)$ as follows: There are variables for the robot's position, for the balls' positions and for the content of each of the robot's grippers:

$$\mathcal{X} = \{\textit{robot-at}, \textit{ball-at}_1, \ldots, \textit{ball-at}_4, \textit{gripper-left}, \textit{gripper-right}\} \tag{2.1}$$

The robot can be in either room A or room B. Each ball can be in room a, room b or in any of the robot's two grippers. In this case it's not important for the ball's domain which gripper the ball is in, because there is a separate variable for the grippers.

A gripper can hold any of the four balls or it can be free.

$$dom(\textit{robot-at}) = \{\textit{room-a}, \textit{room-b}\} \tag{2.2}$$

$$\forall i \in [1,4] : dom(\textit{ball-at}_i) = \{\textit{room-a}, \textit{room-b}, \textit{gripper}\} \tag{2.3}$$

$$dom(\textit{gripper-left}) = dom(\textit{gripper-right}) = \{\textit{ball}_1, \ldots, \textit{ball}_4, \textit{free}\} \tag{2.4}$$

There are three classes of actions in $\mathcal{A}$ that the robot can execute: grabbing a ball, dropping a ball and moving into another room. In this example, the robot can move from room A to room B and vice versa. This means there are actions $move(a, b)$ and $move(b, a)$ which are defined as follows:

$$pre(move(a, b)) = \{\textit{robot-at} = \textit{room-a}\} \tag{2.5}$$

$$\mathit{eff}(move(a, b)) = \{\textit{robot-at} = \textit{room-b}\} \tag{2.6}$$

and similarly for $move(b, a)$. Grabbing or dropping a ball can happen in either room, with either gripper and with any of the balls. In order to grab a ball, the robot has to be in the same room as the ball, and the gripper has to be free. Dropping a ball requires the ball to be in one of the grippers and results in the ball being in the same room as the robot and the gripper being free. The *grab* and *drop* actions are formalized as follows:

$$\forall i \in [1,4], r \in \{\textit{room-a}, \textit{room-b}\}, g \in \{\textit{gripper-left}, \textit{gripper-right}\} :$$

$$pre(grab(\textit{ball}_i, r, g)) = \{\textit{robot-at} = r, \textit{ball-at}_i = r, g = \textit{free}\} \tag{2.7}$$

$$\mathit{eff}(grab(\textit{ball}_i, r, g)) = \{\textit{ball-at}_i = \textit{gripper}, g = \textit{ball}_i\} \tag{2.8}$$

$$pre(drop(\textit{ball}_i, r, g)) = \{\textit{robot-at} = r, \textit{ball-at}_i = \textit{gripper}, g = \textit{ball}_i\} \tag{2.9}$$

$$\mathit{eff}(drop(\textit{ball}_i, r, g)) = \{\textit{ball-at}_i = r, g = \textit{free}\} \tag{2.10}$$

Finally, the initial state and goal state are defined as:

$$\begin{aligned} s_I = \{&\textit{robot-at} = \textit{room-a}, \\ &\textit{ball-at}_1 = \textit{room-a}, \ldots, \textit{ball-at}_4 = \textit{room-a}, \\ &\textit{gripper-left} = \textit{gripper-right} = \textit{free}\} \end{aligned} \tag{2.11}$$

$$s_G = \{\textit{ball-at}_1 = \textit{room-b}, \ldots, \textit{ball-at}_4 = \textit{room-b}\} \tag{2.12}$$

To complete this example, a valid solution to the problem is:

$$\begin{aligned} &grab(\textit{ball}_1, \textit{room-a}, \textit{gripper-left}), grab(\textit{ball}_2, \textit{room-a}, \textit{gripper-right}), \\ &move(a, b), \\ &drop(\textit{ball}_1, \textit{room-b}, \textit{gripper-left}), drop(\textit{ball}_2, \textit{room-b}, \textit{gripper-right}), \\ &move(b, a), \\ &grab(\textit{ball}_3, \textit{room-a}, \textit{gripper-left}), grab(\textit{ball}_4, \textit{room-a}, \textit{gripper-right}), \\ &move(a, b), \\ &drop(\textit{ball}_3, \textit{room-b}, \textit{gripper-left}), drop(\textit{ball}_4, \textit{room-b}, \textit{gripper-right}) \end{aligned}$$

2.2 SAT Solving

A *boolean variable* x is a variable that can have one of two possible values: True (1) or False (0). A *literal* is either a boolean variable (x, *positive literal*) or its negation ($\neg x$, *negative literal*). A *clause* is a disjunction ($\vee$) of literals. The empty clause $\perp$ does not contain any literals and always evaluates to False. A formula in *conjunctive normal form* (CNF) is a conjunction ($\wedge$) of clauses. An example for a formula in CNF is

$$F = (x_1 \vee \neg x_2) \wedge (x_3) \wedge (\neg x_1 \vee x_4) \tag{2.13}$$

All formulas in this thesis will be given in CNF or are easily transformable into CNF.

A *truth assignment* ϕ is a function that assigns a truth value (True or False) to each variable x. This means that either $\phi(x) = True$ or $\phi(x) = False$. An assignment satisfies a positive literal if $\phi(x) = True$, and it satisfies a negative literal if $\phi(x) = False$. A clause is satisfied by an assignment ϕ, if it satisfies any of its literals. Finally, an assignment ϕ satisfies a CNF formula if ϕ satisfies all clauses in it. A formula F is *satisfiable* if there exists an assignment that satisfies F. A formula can have multiple satisfying assignments. As an example, the formula 2.13 is satisfiable with ϕ being a possible solution, where:

$$\phi(x_2) = False \tag{2.14}$$

$$\phi(x_3) = \phi(x_4) = True \tag{2.15}$$

The *Boolean Satisfiability Problem* (SAT) is the problem of determining whether a given formula in propositional logic is satisfiable, and if so, finding a satisfying assignment. A SAT solver is an algorithm that takes a formula (here always in CNF) and outputs whether the formula is satisfiable or unsatisfiable. In case it's satisifable, the SAT solver additionally outputs a truth assignment that satisfies the formula.

3 Related Work

3.1 Incremental SAT Solving with IPASIR

Incremental SAT Solving means consecutively solving several similar but slightly changed formulas. The idea is that the effort spent on the first formula can be re-utilized for subsequent formulas, including e.g. learned clauses and variable scores used for heuristics. This can lead to significant performance boosts [12].

The *Re-entrant Incremental Satisfiability Application Program Interface* (IPASIR) [14] is an interface for Incremental SAT Solving. The function headers that IPASIR provides are shown in listing 3.1. A SAT solver must implement all the functions in order to be used in applications using IPASIR.

Listing 3.1: IPASIR function headers

```
const char* ipasir_signature();
void* ipasir_init();
void ipasir_release(void* solver);
void ipasir_set_terminate(void* solver, void* state,
       int (*terminate)(void* state));
void ipasir_set_learn (void* solver, void* state, int max_length,
       void (*learn)(void* state, int* clause))
void ipasir_add(void* solver, int lit_or_zero);
void ipasir_assume(void* solver, int lit);
int ipasir_solve(void* solver);
int ipasir_val(void* solver, int lit);
int ipasir_failed(void* solver, int lit);
```

By calling `ipasir_init`, an instance of the SAT solver is created and returned as a pointer. This pointer has to be passed as the first parameter to the most other IPASIR functions in order to use this SAT solver instance. `ipasir_set_terminate` sets a termination condition for the SAT solver by passing a callback function that is called by the solver during solving. Calling `ipasir_release` properly releases the given instance and frees its resources.

To manipulate the formula that is to be solved, there are the `ipasir_add` and `ipasir_assume` functions. Adding clauses is done by adding single literals one at a time. Literals are represented by integers: positive literals are positive literals and negative literals are negative integers. To mark the end of a clause, 0 is passed instead of a literal. `ipasir_assume` can be used to make temporary partial assignments, i.e. to prescribe that certain literals must be satisfied in the next solving attempt. `ipasir_solve` will try to solve the current formula with the given assumptions. After one call to this func-

tion, all assumptions previously made with `ipasir_assume` are cleared. `ipasir_solve` either returns that the formula is SAT, UNSAT or that solving has been terminated prematurely. In the case that the formula is SAT, `ipasir_val` can be used to extract the solution, i.e. the truth assignment to each variable. If the literal `lit` is passed to it, either `lit` (literal is $True$), `-lit` (literal is $False$) or `0` (literal is not important) is returned. A full description of all function headers can be found in [14].

Listing 3.2: IPASIR usage example

```
1  // Return value of solve in case the formula is SAT
2  #define SAT 10
3
4  void *solver = ipasir_init();
5
6  // Adding the clauses
7  ipasir_add(solver, -1);
8  ipasir_add(solver, 2);
9  ipasir_add(solver, 0);
10
11 // Assume x1 to be true
12 ipasir_assume(solver, 1);
13 if (ipasir_solve(solver) == SAT) {
14         // Get the value of x2 if formula is SAT
15         int valueOfX2 = ipasir_val(solver, 2);
16         printf("%d\n", valueOfX2);
17 }
18
19 ipasir_release(solver);
```

Listing 3.2 shows an example C code demonstrating the usage of IPASIR. In lines 7-9, one clause is added that makes up the whole formula ($\neg x_1 \vee x_2$, which is equivalent to $x_1 \rightarrow x_2$). More clauses can be added, but this is not done here in order to keep the example concise. Next, the literal x_1 is assumed (line 12) to be true for the next solving attempt which is started in line 13. Since the formula is SAT (also considering the given assumption), lines 15 and 16 will be executed. As described above, `ipasir_val` extracts the value of the literal x_2 from the truth assignment that the solver found. In this case x_2 is satisfied because x_1 was assumed to be true and thus the program will output `2` in the end. Lastly, the solver is released in line 19.

3.2 Planning Problem Formalisms

There are several different formalisms for describing planning problems, e.g. PDDL [15], SAS+ [16] and STRIPS [17]. It is common for planning algorithms to accept PDDL as an input format [6, 18] and to use a technique called *grounding* to translate such problems to SAS+. The reason for this is that the SAS+ format is easier to parse automatically. The algorithm proposed in this thesis will use the grounding algorithm of the Fast Downward system [19] to translate PDDL to SAS+, so that meaningful

comparisons to other planners can be made without implementing a complex PDDL parser from scratch.

3.3 The Graphplan Algorithm

Graphplan [8] is a graph-based approach to planning. The two main mechanisms of the algorithm are expanding the *planning graph* and *plan extraction*. Graphplan works by repeatedly expanding and extracting until a solution is found, or the problem is proven to have no solution.

Definition 3.3.1. A proposition is an assignment of a value to a variable at a given time.

The planning graph is a layered graph with two different kinds of nodes: *propositions* and *actions* (as defined in 2.1). The graph is layered in the sense that it can be divided into proposition layers and action layers, each containing only one kind of node. Proposition layers and action layers are alternating, i.e. nodes in a proposition layer only have edges to the next action layer which in turn only has edges to the next proposition layer. The first layer of the planning graph is a proposition layer called the *initial layer*. It contains one node for each proposition that is true in the initial state of the problem. Figure 3.1 shows part of a planning graph for a simplified version of the gripper problem as described in 2.1.1.

Definition 3.3.2. A *Precondition edge* (p, a) indicates that the proposition p is a precondition of action a. The set of preconditions of an action a is denoted as $pre(a)$.

Definition 3.3.3. A *Positive effect edge* (a, p) indicates that the execution of action a will add proposition p in the next layer. The set of positive effects of an action a is denoted as $\mathit{eff}^+(a)$. The set of actions that has p as a positive effect, i.e. has a positive effect edge to p, is denoted as $act^+(p)$.

Definition 3.3.4. A *Negative effect edge* (a, p) indicates that the execution of action a will delete proposition p in the next layer. The set of negative effects of an action a is denoted as $\mathit{eff}^-(a)$.

Each edge in the planning graph is either a precondition edge, a positive effect edge or a negative effect edge.

Definition 3.3.5. A *no-op action* $a(p)$ of a proposition p is an action with $pre(a) = \{p\}$, $\mathit{eff}^+ = \{p\}$, $\mathit{eff}^- = \emptyset$.

Not all propositions present in a layer are necessarily true in the respective point in time. It rather means that there is a possibility for the proposition to be true. Note that proposition layer P_2 always contains all propositions of P_1. The same is true for actions: An action layer contains all actions that could possibly be executed at that time. To ensure that every layer is always at least as big as previous layers, there is a no-op action n for each proposition p.

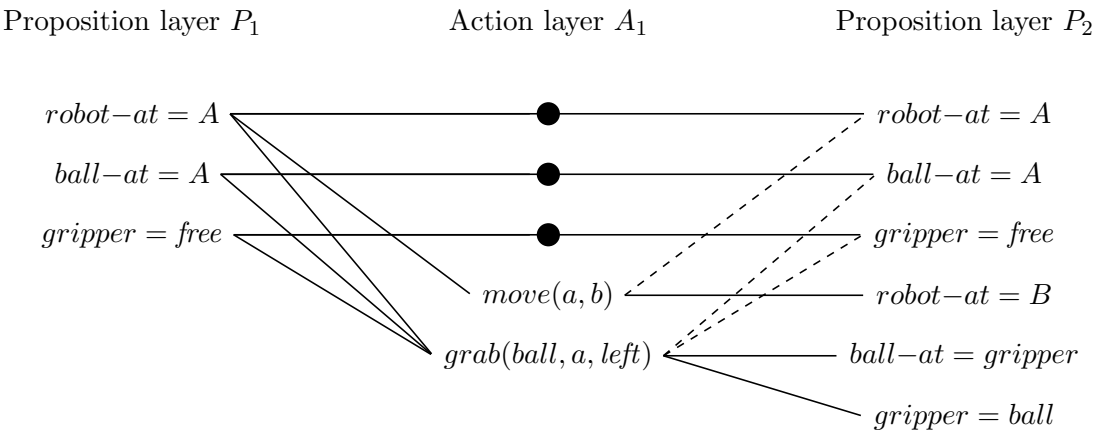


Figure 3.1: Example of a planning graph for a simplified gripper problem with one ball and one gripper. Negative effect edges are represented by dashed lines and no-op actions are represented by dots. Planning graphs can and usually do have more than one action layer.

Pairs of propositions can be mutually exclusive (*mutex*) in a layer, e.g. a robot can't be in room A and room B at the same time ($robot{-}at = A$ and $robot{-}at = B$ are mutually exclusive in every layer). The same is true for actions. The set of propositions that are mutually exclusive to a proposition p in layer P_i is denoted as $\mu P_i(p)$. Likewise, the set of actions that are mutually exclusive to an action a in A_i is denoted as $\mu A_i(a)$.

At the start of the algorithm, the planning graph consists only of the initial layer. The graph is then expanded from its current state. This means that the algorithm determines all possible actions based on the initial propositions and adds them to a new action layer. Consequently, all propositions that are enabled by these actions form the next proposition layer. The expanding algorithm also determines sets of mutually exclusive propositions and actions respectively. The planning graph can reach a *fixed point*. This means that the last layer of the graph doesn't change after expanding, i.e. $P_n = P_{n-1}$, $A_n = A_{n-1}$, $\mu P_n = \mu P_{n-1}$ and $\mu A_n = \mu A_{n-1}$. Initially the graph is expanded multiple times before the first plan extraction, until either all goal propositions are present in the graph or a fixed point has been reached. This is called the fixed-point iteration. In the case that a fixed point is reached here, the problem has no solution.

In the next step, the algorithm tries to extract a plan from the planning graph using a backwards-search. This means that all the goal propositions G are assumed to be true at the last proposition layer of the graph. Based on that, a subset A of the actions in the previous action layer have to be determined so that these actions satisfy all propositions in G and are not mutex with each other (there can be multiple options for sets A). The procedure is repeated using the preconditions of the actions in A as a goal for the previous proposition layer. If the initial layer is reached, a solution is found and the algorithm terminates. If no more set of actions satisfying the given constraints can be found in a given layer, this specific set of goals is added to the so-called *nogood table* and

extraction has to go back one layer again. If this happens at the last layer of the graph, extraction has failed in this iteration. If extraction fails, the fixed-point of the graph has already been reached and the nogood table doesn't grow at the fixed-point layer, then the planning problem has no solution.

Algorithm 1 Graphplan

```
1:  procedure GRAPHPLAN(P_1, Actions, Goal)
2:      ∇ ← ∅, η ← 0                                   ▷ Nogood table and its size
3:      G ← ⟨P_1⟩, i ← 1          ▷ Planning graph and amount of proposition layers
4:
5:      repeat                                          ▷ Initial fixed-point iteration
6:          i ← i + 1
7:          G ← Expand(G)
8:      until (Goal ⊆ P_i ∧ Goal^2 ∩ μP_i = ∅) ∨ FixedPoint(G)
9:                                                      ▷ Abort if goal is not reachable
10:     if Goal ⊈ P_i ∨ Goal^2 ∩ μP_i ≠ ∅ then return failure
11:
12:     Π ← Extract(G, Goal, i)                         ▷ Initial attempt at plan extraction
13:     if FixedPoint(G) then
14:         κ ← i                                       ▷ Remember the fixed-point layer
15:         η ← |∇(κ)|                                  ▷ Keep track of the nogood table's size
16:
17:     while Π = failure do                            ▷ Main loop of the algorithm
18:         i ← i + 1          ▷ Keep expanding and extracting until plan is found
19:         G ← Expand(G)
20:         Π ← Extract(G, Goal, i)
21:         if Π = failure ∧ FixedPoint(G) then
22:             if η = |∇(κ)| then return failure       ▷ Problem has no solution
23:             η ← |∇(κ)|
24:
25:     return Π
```

Algorithm 1 shows the main procedure of Graphplan. Algorithm 2 shows the recursive procedures used for plan extraction [1]. The *Extract* procedure keeps track of the nogood table, and calls *GPSearch* with the desired goal. *GPSearch* tries to find a set of actions that satisfy this goal (lines 18-26). It does so by choosing one action that enables one proposition and that is not mutex to the actions that were already chosen. The algorithm backtracks to line 23, where the selection of actions happens, in case that subsequent calls to *GPSearch* fail. When all propositions of the given goal are satisfied by at least one of the chosen actions, the algorithm will call *Extract* recursively with all preconditions of the chosen actions (lines 13-16).

[1] The pseudo-codes originate from [20].

Algorithm 2 Plan Extraction

1: **procedure** EXTRACT($G, Goal, i$)
2: **if** $i = 1$ **then return** $\langle\rangle$ ▷ Trivial success
3: **if** $Goal \in \nabla(i)$ **then return** *failure* ▷ Goal is in nogood table
4:
5: $\pi_i \leftarrow GPSearch(G, Goal, \emptyset, i)$
6: **if** $\pi_i \neq$ *failure* **then return** π_i ▷ Return plan up to this layer
7:
8: $\nabla(i) \leftarrow \nabla(i) \cup \{Goal\}$ ▷ Add goal to nogood table
9: **return** *failure*
10:
11: **procedure** GPSEARCH($G, Goal, \pi_i, i$)
12: **if** $Goal = \emptyset$ **then**
13: ▷ Call Extract with preconditions of chosen actions
14: $\Pi \leftarrow Extract(G, \{pre(a) \mid a \in \pi_i\}, i - 1)$
15: **if** $\Pi =$ *failure* **then return** *failure*
16: **return** $\Pi \| \pi_i$ ▷ Add chosen actions to plan and return it
17: **else**
18: select any $p \in Goal$
19: ▷ Get actions that satisfy p and are not mutex to chosen actions
20: $resolvers \leftarrow \{a \in A_i \mid a \in act^+(p) \wedge \forall b \in \pi_i : (a, b) \notin \mu A_i\}$
21: **if** $resolvers = \emptyset$ **then return** *failure*
22:
23: non-deterministically choose $a \in resolvers$ ▷ Backtrack here
24:
25: ▷ Recursive call with less goals and additional chosen actions
26: **return** $GPSearch(G, Goal - \mathit{eff}^+(a), \pi_i \cup \{a\}, i)$

As an example, take the graph shown in figure 3.1. When calling *Extract* on proposition layer P_2 and $Goal = \{ball{-}at = gripper\}$, *GPSearch* will be called with the same goal and find that the only suitable action in A_1 is $grab(ball, a, left)$. Thus, it will be chosen and call *GPSearch* again, this time with an empty goal. Since the goal is empty, *GPSearch* will call *Extract* at the previous proposition layer, i.e. P_1, with the preconditions of the chosen action, i.e. $\{robot{-}at = A, ball{-}at = A, gripper = free\}$. Because P_1 is the first layer of the planning graph, *Extract* will return an empty plan ($\langle\rangle$). The second call of *GPSearch* will then add the action $grab(ball, a, left)$ to the first layer of the plan and return it. The first call of *GPSearch* passes that plan through to the first call of *Extract* which returns it back to the main algorithm. Since plan extraction was successful, the algorithm terminates and outputs the plan $\langle\{grab(ball, a, left)\}\rangle$.

3.4 Combining Graphplan and SAT Solving

Using the planning graph generated by expanding as described in 3.3, the plan extraction problem can be transformed into SAT as follows [21]:

For all action layers A_n, let P_m be the proposition layer before A_n and P_{m+1} the proposition layer after A_n. All actions that are taken in A_n require their preconditions to hold in P_m and their effects in P_{m+1}. These constraints can be formalized and added to the SAT formula as [2] [3]:

$$\forall a \in A_n : (\forall p \in pre(a) : a_n \rightarrow p_m) \tag{3.1}$$

$$\wedge (\forall e^+ \in \mathit{eff}^+(a) : a_n \rightarrow e^+_{m+1}) \tag{3.2}$$

$$\wedge (\forall e^- \in \mathit{eff}^-(a) : a_n \rightarrow e^-_{m+1}) \tag{3.3}$$

Furthermore, an action a can't be chosen together with any action b that is mutex with it in A_n:

$$\forall a \in A_n : \forall b \in \mu A_n(a) : \neg a_n \vee \neg b_n \tag{3.4}$$

Additionally, in order for a proposition to hold in a layer P_{m+1}, it has to be enabled by an action in the previous action layer A_n. This can be ensured by the following clauses for each proposition layer P_{m+1}:

$$\forall p \in P_{m+1} : p \rightarrow \bigvee_{a \in act^+(p) \cap A_n} a_n \tag{3.5}$$

Then, the variables representing the goal propositions in the SAT formula are assumed to be true. This ensures that satisfying assignments to this formula are valid solutions to the planning problem at hand. Lastly, if the formula is satisfiable, the satisfying assignment is checked for the values of all the variables representing actions. The actions whose variables are set to true in the assignment will form the plan.

Using this technique instead of the original backwards-search can boost the performance of Graphplan significantly [9, 21].

3.5 Madagascar: State-of-the-Art SAT Planning

Madagascar [6] is a state-of-the-art planner that is based on Graphplan and uses SAT Solving for the extraction step of the algorithm. However, it systematically skips extraction in some iterations. The *horizon lengths* indicate the numbers of the iterations at which an attempt at plan extraction will be made. For example, Graphplan as described in 3.3 uses horizon lenghts of $1, 2, 3, 4, \ldots$. Madagascar provides different configurations

[2] If a proposition p holds in layer P_m this is denoted as p_m, and similarly for actions

[3] An implication $x \rightarrow y$ is logically equivalent to the clause $\neg x \vee y$

for horizon lenghts, e.g. an exponentially growing horizon $(1, 2, 4, 8, 16, \ldots)$. Additionally, it solves multiple SAT formulas for plan extraction from different layers of the planning graph concurrently [4] by assigning each problem a given amount of CPU time. This has proven to be a very efficient technique for plan extraction.

[4] But not physically parallel, i.e. not on multiple processor cores

4 Algorithm Description

4.1 Sequential Algorithm

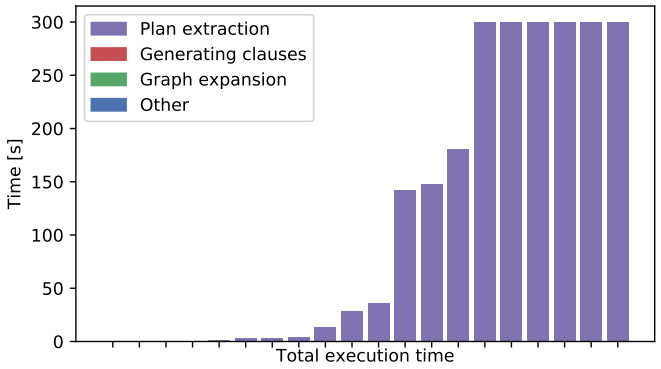
For demonstration purposes, a sequential version of the algorithm proposed in section 4.2 is presented first. It is in principle very similar to the Madagascar planner described in 3.5, as it's a Graphplan-based approach using SAT solving for plan extraction and adjustable horizon lenghts. For the plan extraction, a formula is generated as described in 3.4 and solved using a SAT solver.

The function $h : \mathbb{N} \to \mathbb{N}$ is a strictly increasing function used for calculating the horizon lengths. For example, if $h(1) = 1, h(2) = 2, h(3) = 4, h(4) = 8, \ldots$ then a plan extraction will be done from layers $1, 2, 4, 8, \ldots$. Consequentially, this also means that extraction will be skipped at layers $3, 5, 6, 7$, which can save a lot of runtime as shown by other planners [6]. h can be configured as any strictly increasing function, e.g. a linear function or an exponential function. h^{-1} is the inverse function of h. This is used to determine the iteration of the algorithm where a specific horizon length is used for extraction, in order to decide if further expansion has to happen in the parallel algorithm.

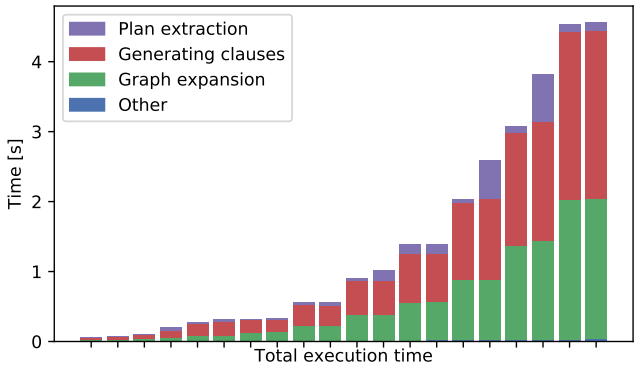
Algorithm 3 Sequential algorithm

```
 1: lastActionLayer ← 0, lastPropositionLayer ← 1
 2:
 3: procedure FindPlan(goal)
 4:     iteration ← 0
 5:     while true do
 6:         while lastActionLayer < h(iteration + 1) do
 7:             ExpandGraph()
 8:         plan ← Extract(goal, lastPropositionLayer)
 9:         if plan ≠ ⊥ then return plan
10:         iteration ← iteration + 1
```

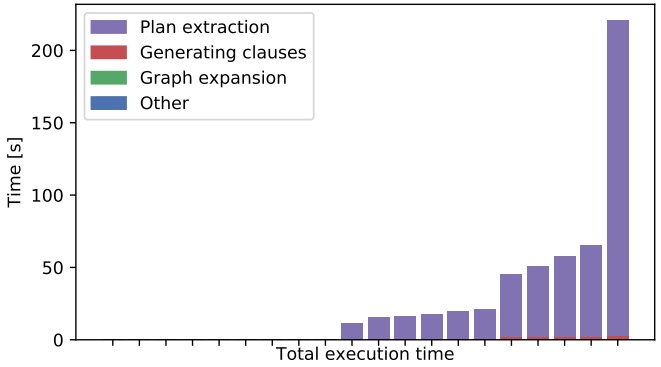
The graphs in figure 4.1 show that the runtime of the sequential algorithm can roughly be divided into time used for graph expansion, generating SAT clauses and plan extraction. Problems of the same domain show similar characteristics in terms of runtime for the respective tasks. The time needed to solve a specific planning problem can be dominated by either of these three tasks. For most problems plan extraction takes the most time, which makes it a target for parallelization. Another approach is to parallelize the graph expansion, especially the determination of proposition and action mutexes. The algorithm proposed in section 4.2 will consider plan extraction in parallel.



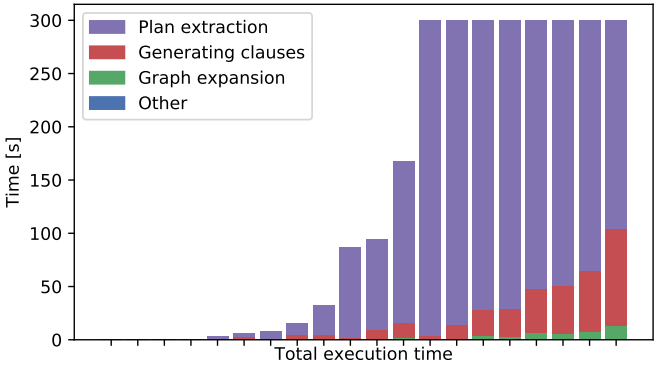
(a) Floortile domain



(b) ChildSnack domain



(c) GED domain



(d) Hiking domain

Figure 4.1: Execution time breakdown of the sequential algorithm on problems of various domains with horizon lenghts of $1, 2, 3, \ldots$ and a timeout of 300s

4.2 Parallel Algorithm

The parallel algorithm proposed here works similar to the sequential algorithm from the previous section. It differs in that plans are extracted from several layers in the planning graph in parallel using a thread pool. However, the algorithm additionally needs to keep track of the extraction progress to coordinate graph expansion. Algorithm 4 describes how this works in pseudocode.

Compared to the sequential version (Algorithm 3), the parallel algorithm declares two more variables at a global scope (i.e. all threads can access them). $lastFailedLayer$ indicates the latest layer in the planning graph from which plan extraction has been attempted and failed. This is updated by the extraction threads after each failed plan extraction and read by the thread running the main loop (lines 6-11)[1] in order to determine whether to expand the graph. This mechanism is necessary because otherwise the planning graph would be expanded endlessly, inevitably causing the machine executing the algorithm to run out of memory. Additionally the algorithm defines a variable for holding a *plan* which is written by the first thread that successfully extracts a plan. As soon as this happens, the algorithm can stop all other extraction threads, terminate, and output *plan*.

Algorithm 4 Parallel algorithm

```
 1: lastFailedLayer ← 0, plan ← ⊥
 2:
 3: procedure ParallelFindPlan(goal, threadCount)
 4:     iteration ← 0
 5:     pool ← InitializeThreadPool(threadCount)
 6:     while plan = ⊥ do
 7:         if iteration ≤ h^{-1}(lastFailedLayer) + threadCount then
 8:             pool.Enqueue(ExtractThread(goal, h(iteration)))
 9:             while lastActionLayer < h(iteration + 1) do
10:                 ExpandGraph()
11:             iteration ← iteration + 1
12:
13: procedure ExtractThread(goal, layer)
14:     p ← Extract(goal, PropLayerAfterActionLayer(layer))
15:     if p ≠ ⊥ then
16:         plan ← p
17:     else
18:         lastFailedLayer ← layer
```

In each iteration of the mainloop, it is checked whether a graph expansion has to be done. This is determined by calculating at which iteration the algorithm should be,

[1] Note that one of the threads is dedicated to running the main loop, i.e. if the algorithm is run with 4 threads that means there will be 3 threads dedicated to extracting plans.

i.e. how far the graph should have been expanded in order for the specified number of threads to be able to extract plans (line 7). If expansion needs to be done, this also means that one thread finished execution and failed an extraction (or didn't have work to begin with, which happens during the first iterations of the algorithm). Thus, a job can be queued to extract a plan from the current last layer (i.e. the horizon function at the current iteration). Then, the planning graph is expanded until the next horizon length is reached.

5 Implementation

The algorithm is implemented using C++ 11 in an object-oriented fashion. It is deliberately made from scratch instead of based on an existing Graphplan implementation. When using such a codebase it can be very time-consuming to understand which parts of the code are prone to race conditions and why that is the case. By implementing the algorithm from scratch, it can be designed with parallelization in mind.

5.1 Planning Graph and Clause Generation

Since actions have the same edges to the same propositions every time they occur in the planning graph, each action's preconditions and effects only have to be stored once. In the implemented planning graph data structure there is an array of lists for action preconditions, positive effects and negative effects respectively. To determine quickly if two propositions p and q are mutex it is important to be able to look up all actions that have p or q as a positive effect. For this purpose there is also a table that maps propositions to their enabling actions.

Proposition mutexes and action mutexes are stored in symmetric matrices of size $|P| \cdot |P|$ and $|A| \cdot |A|$ respectively, with P being the set of all propositions in the problem, and A the set of all actions. The element x_{pq} in the proposition mutex matrix indicates the last layer in which the propositions p and q are mutex. If the element's value is 0 then the two propositions are not mutex in any layer. Action mutexes are implemented the same way.

The propositions and actions enabled in each layer are stored in vectors of linked lists, i.e. one list per layer. For generating clauses for the SAT formula, these data structures have to be read by the respective thread. However, concurrent writing and reading of these data structures is not thread-safe which means that graph expansion and clause generation both need to be declared critical section that must not be executed in parallel. This can become a potential bottleneck in some cases and is in fact a relevant point for the conclusions drawn in section 6.2.

5.2 Thread and SAT Solver Pool

The *SATSolverThreadPool* is a specialized type of Thread Pool. It works similar to a common thread pool in that a set amount of worker threads are created on initialization and work can be queued to be done by those threads. Since these threads will be used for plan extraction using SAT Solving in the algorithm at hand, they need access to a SAT solver. In order to achieve this, the initialization method of the thread pool takes a

function pointer as an additional argument. This function pointer serves as a reference to an initialization method for a SAT Solver and will be called once for each initialized thread. This means that the code using the Thread Pool has control over how the SAT solvers will be created and doesn't have to handle the SAT solvers itself.

5.3 SAT Solver

The SAT solver is an essential part of the algorithm's configuration, since usually most of the runtime is used for SAT solving. Different algorithms and implementations for this can vary substantially in performance, depending on the given problem, i.e. formula. Since this implementation is using the standardized IPASIR, several SAT solvers can be tested using the same interface. To find out which implementation is the most suitable for the task at hand, the following SAT solvers with IPASIR support are evaluated:

- AbcdSAT i17 [22] (First place in the Incremental Library Track of the SAT 2017 competition)
- Glucose [23] version 4 (Second place in the Incremental Library Track of the SAT 2017 competition)
- CryptoMiniSat [24] (First place in the Incremental Library Track of the SAT 2016 competition)
- Lingeling [25]

After a first set of experiments, the solvers Glucose4 and Lingeling showed the best results. A further set of experiments was conducted to compare these two solvers in more detail. In order to test for good scalability, only a range of 32 problems from the IPC14 were chosen for this experiment. Both configurations were run using the parallel algorithm described in section 4.2 with 8 threads. A timeout of 10 minutes was set, which means that an unsolved problem instance will add 10 minutes to the total runtime.

As can be seen in figure 5.1, the configuration with Glucose4 solved one more problem than the one with Lingeling. Furthermore, Glucose4 solved all the problems that Lingeling solved. Comparing only the runtimes on problems solved by either configuration, Glucose4 is more than twice as fast as Lingeling. Thus, Glucose4 will be used for all following benchmarks in this thesis.

5.4 Plan Verifier

Verifying the output of an algorithm makes it easier to detect errors and is usually a lot less complex than the algorithm itself. This is also true for planning algorithms, where generated plans can be checked very quickly. For this purpose, the plan verifier VAL[26] is called after the algorithm terminated and output a plan. The validator outputs information about the total cost of the plan and whether it is valid. Since the proposed algorithm is not optimizing for cost, only the latter is interesting in this case.

SAT Solver	Problems solved	Total runtime	Solved only
Glucose4	16	10507 s	907 s
Lingeling	15	11724 s	2124 s

Table 5.1: Problems solved per SAT solver, total runtime on all tested problems and runtime on problems that were solved by either solver (solved only)

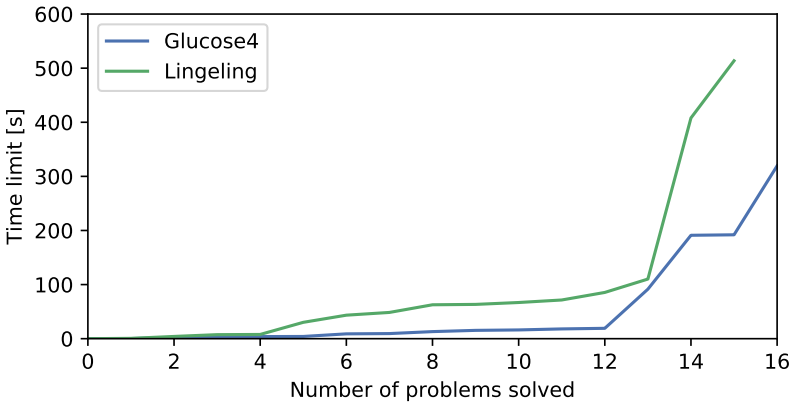


Figure 5.1: Number of problems solved by configurations with Glucose4 and Lingeling

6 Evaluation

In section 6.2, the implementation of the proposed planner (*Parallel Graphplan*) is evaluated in several different configurations on a subset of the problem instances. Cactusplots will be used to visualize and compare the performance of all configurations. In section 6.3, the best configuration will then be compared to the state-of-the-art planner Madagascar (introduced in section 3.5) on a larger set of instances.

6.1 Benchmark Environment

The problems used for evaluation originate from the 2011 and 2014 International Planning Competitions (IPC11 [27], IPC14 [28]). Problems are divided into different tracks and domains. Table 6.2 lists all used problem instances by their domain and name. The problems are given in the standardized PDDL format. The selection of instances is based on the following criteria:

- The problem must be dominated by extraction time rather than time used for graph expansion, so that there can be a reasonable speedup due to parallelization.
- Problems of some domains (such as Barman, Parking, Tidybot) are generally very hard to solve for both Madagascar and Parallel Graphplan. These problems are not part of the experiment comparing the two planners.

The system used to perform the benchmarks has four Intel® Xeon® E5-4640 CPUs (8 cores with 2.4GHz each), i.e. 32 real cores in total, and 512GB of DDR3-1600 RAM. To compile Parallel Graphplan, g++ version 4.8.4 was used.

6.2 Parallel Speedup

The Parallel Graphplan algorithm is evaluated using the following configurations for horizon lengths $h(x)$:

- $h(x) = 4x$
- $h(x) = 8x$
- $h(x) = 1.5^x$
- $h(x) = 1.8^x$

Each configuration is run using the sequential algorithm as well as using the parallel algorithm with 2, 4, 8, 16 and 32 threads respectively. Figure 6.1 shows a cactusplot for problems solved by each configuration (combination of horizon length function and amount of threads). Detailed information about number of instances solved per domain and runtime per problem instance is shown in table 6.1 and table 6.2 respectively.

Figure 6.1 suggests that parallelization is especially efficient when using linear functions for determining horizon lenghts. In that case, configurations with at least 4 threads solved between 16 and 18 instances, where the sequential algorithm could only solve 11 or 12 instances. This effect is less obvious when using more aggressive exponential horizons, but the parallel algorithm still solves more problem instances in less time.

An interesting effect that can be observed here is, that configurations with 16 or 32 threads usually solve less problems or solve them slower than their counterparts with 4 or 8 threads. This can have multiple reasons:

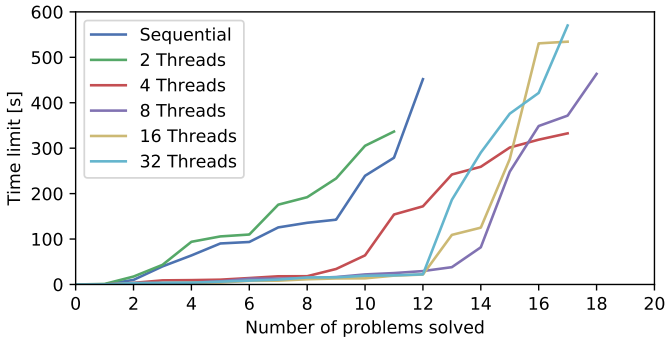
- For very small problem instances, the overhead of creating many threads, initializing SAT solvers and generating clauses can dominate, resulting in a longer overall runtime.
- If more threads are used, the planning graph has to be expanded more which results in a higher memory usage. The system can even run out of memory which happened for domains that contain a lot of actions per layer. This can be observed in particular on the more aggressive exponential horizon lengths.

Another interesting finding is that in most cases not even all SAT solver instances are used in the configurations with 16 or 32 threads, and many solvers are only used once during the whole runtime. This can also be a cause for the longer runtime, as the generation and adding of clauses to a SAT solver takes longer the less clauses it already contains. If SAT solvers are only used once, all clauses have to be added to the SAT solver each time. As pointed out in section 5, this is not done in parallel, since expanding the graph is not a thread-safe operation.

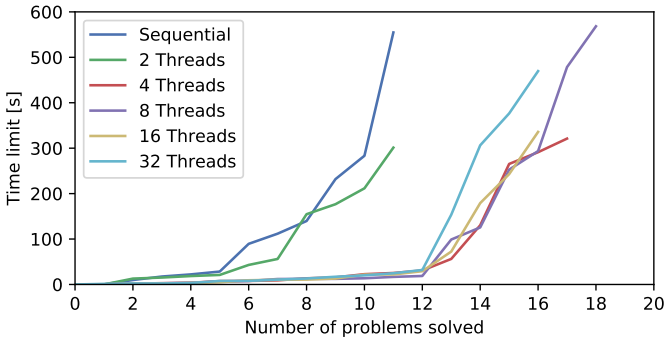
Summarizing the results of this comparison, the best-performing (i.e. solved the most instances) configurations of the Parallel Graphplan algorithm in this environment are:

- Horizon $h(x) = 4x$ with 8 threads (configuration 1)
- Horizon $h(x) = 8x$ with 8 threads (configuration 2)
- Horizon $h(x) = 1.5^x$ with 8 threads (configuration 3)
- Horizon $h(x) = 1.8^x$ with 4 threads (configuration 4)

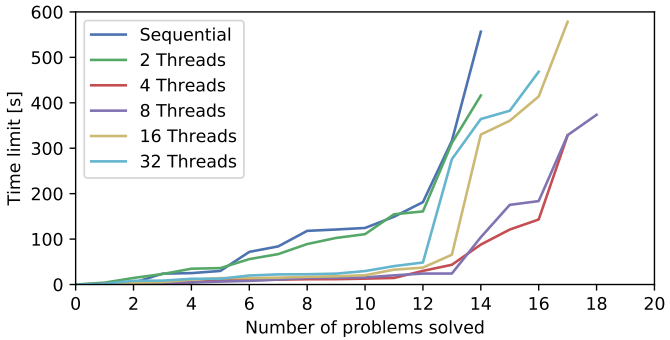
Figure 6.2 shows a first comparison of these configurations to Madagascar as a cactusplot. Tables 6.1 and 6.2 show the amount of solved problems per domain and time required for each instance.



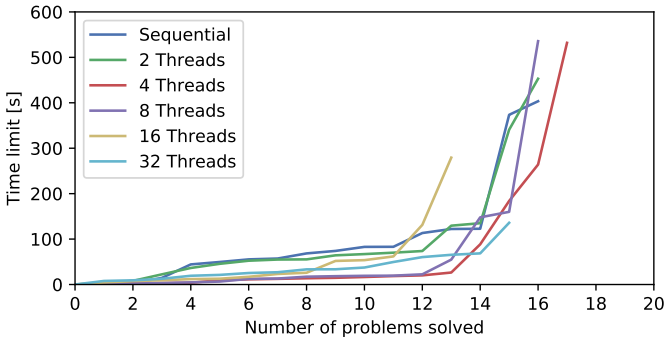
(a) $h(x) = 4x$



(b) $h(x) = 8x$



(c) $h(x) = 1.5^x$



(d) $h(x) = 1.8^x$

Figure 6.1: Amount of solved problems for different horizon functions and thread counts

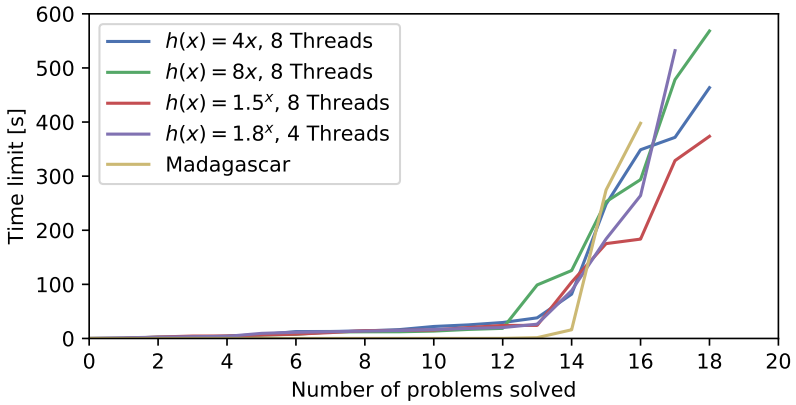


Figure 6.2: Comparison of Madagascar and the best-performing configurations of Parallel Graphplan

Planner	Horizon	Threads	Barman	Floortile	Hiking	Total
Madagascar			**2**	**12**	2	16
Parallel Graphplan	$h(x) = 4x$	8	1	**12**	**5**	**18**
	$h(x) = 8x$	8	1	**12**	**5**	**18**
	$h(x) = 1.5^x$	8	1	**12**	**5**	**18**
	$h(x) = 1.8^x$	4	1	**12**	4	17

Table 6.1: Number of problem instances solved by domain, planner and configuration

Domain	Instance	Parallel Graphplan				Madagascar
		config. 1	config. 2	config. 3	config. 4	
Barman	p435.1	463.52	568.29	328.73	183.92	**1.46**
	p536.1	-	-	-	-	**274.50**
Floortile	p02-6-4-2	4.21	3.78	4.44	3.28	**0.03**
	p02-6-5-2	22.14	12.36	20.17	26.37	**0.15**
	p02-6-5-3	13.86	12.30	14.88	18.58	**0.09**
	p03-6-4-2	2.41	2.45	4.58	3.91	**0.05**
	p03-6-5-2	25.08	18.69	24.07	16.53	**0.11**
	p03-6-5-3	12.73	7.73	7.98	11.43	**0.06**
	p04-5-4-2	1.31	2.51	2.62	2.32	**0.03**
	p04-5-5-2	6.95	6.19	6.35	9.42	**0.09**
	p04-6-5-2	29.48	13.64	15.59	14.74	**0.11**
	p04-6-5-3	12.87	12.00	14.58	12.45	**0.09**
	p05-4-3-2	0.14	0.23	0.14	1.23	**0.01**
	p05-6-5-2	16.34	16.70	10.81	20.12	**0.11**
Hiking	p1-2-7	**81.83**	125.53	175.31	531.97	397.74
	p1-2-8	**248.21**	252.73	373.61	-	-
	p2-2-6	-	-	-	-	-
	p2-2-7	-	-	-	-	-
	p2-3-5	371.68	293.60	**183.63**	264.05	-
	p2-3-6	-	-	-	-	-
	p2-4-4	38.09	98.91	24.17	**13.73**	16.05
	p2-4-5	348.78	478.44	104.11	**88.48**	-

Table 6.2: Runtime in seconds of different configurations on the tested problem instances, compared to the standard configuration of Madagascar. A dash indicates that the problem has not been solved within 600 seconds.

6.3 Comparison to State of the Art

Figure 6.2 shows that the configuration with $h(x) = 1.5^x$ performed best overall. This horizon length will now be used for making a more thorough comparison with the Madagascar planner. All problem instances and the runtimes of the chosen configurations are shown in table 6.3. To get a more detailed view of the parallel speedup of this particular configuration, it is again run in sequential mode and with 2, 4 and 8 threads respectively. Figure 6.3 shows the result of the evaluation as a cactusplot.

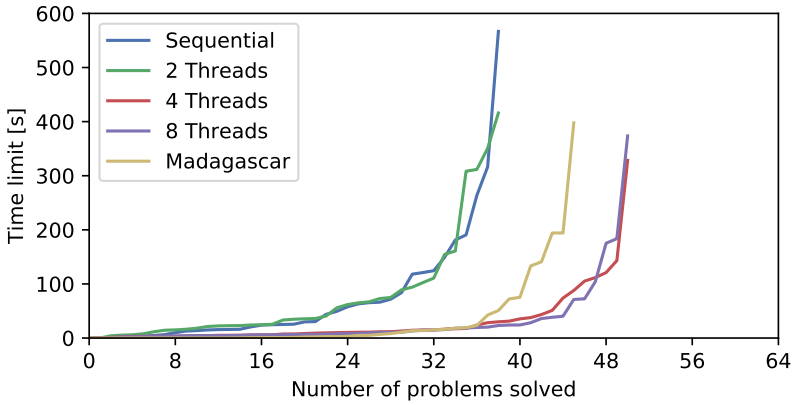


Figure 6.3: Comparison of the chosen Parallel Graphplan configuration with different amounts of threads and Madagascar

Domain	Instance	Parallel Graphplan, $h(x) = 1.5^x$				Madagascar
		Seq.	2 Th.	4 Th.	8 Th.	
Floortile	p02-6-4-2	29.89	34.77	4.34	4.44	**0.03**
	p02-6-5-2	-	-	11.58	20.17	**0.15**
	p02-6-5-3	121.07	160.87	12.91	14.88	**0.09**
	p03-6-4-2	23.88	36.10	5.03	4.58	**0.05**
	p03-6-5-2	-	-	43.43	24.07	**0.11**
	p03-6-5-3	124.47	89.04	10.74	7.98	**0.06**
	p04-5-4-2	3.84	14.41	3.59	2.62	**0.03**
	p04-5-5-2	71.82	102.54	9.16	6.35	**0.09**
	p04-6-5-2	-	415.99	11.76	15.59	**0.11**
	p04-6-5-3	83.89	110.79	9.61	14.58	**0.09**
	p05-4-3-2	0.21	4.13	1.11	0.14	**0.01**
	p05-6-5-2	-	-	30.00	10.81	**0.11**

GED	d-1-8	15.87	24.07	3.11	5.70	**2.08**
	d-2-4	0.04	0.05	0.06	0.20	**0.01**
	d-2-8	49.36	65.03	31.25	40.34	**3.15**
	d-3-4	13.76	18.19	10.37	7.22	**0.61**
	d-4-3	16.21	25.38	7.48	4.72	**1.09**
	d-4-8	63.16	74.94	37.78	38.38	**0.30**
	d-7-5	25.59	24.59	10.84	9.38	**8.26**
	d-7-6	9.94	16.22	2.11	3.22	**0.78**
	d-8-1	14.98	33.29	5.25	10.58	**1.84**
	d-8-2	43.81	94.04	17.66	36.04	**0.72**
	d-8-4	66.29	61.89	23.95	23.17	**0.65**
	d-8-9	190.61	308.40	111.72	72.41	**6.70**
Hiking	p1-2-7	181.31	154.60	**143.11**	175.31	397.74
	p1-2-8	316.30	**311.57**	328.45	373.61	-
	p2-2-6	-	-	-	-	-
	p2-2-7	-	-	-	-	-
	p2-3-5	118.08	**55.94**	120.97	183.63	-
	p2-3-6	-	-	-	-	-
	p2-4-4	25.09	23.04	**14.39**	24.17	16.05
	p2-4-5	149.13	**66.94**	87.90	104.11	-
PegSol	p04	57.81	73.02	51.12	**28.34**	194.17
	p05	-	-	8.36	8.36	**2.66**
	p06	15.64	35.59	3.17	**2.57**	4.79
	p07	-	-	**4.43**	6.08	18.19
	p08	20.53	22.44	**5.91**	13.13	72.25
	p09	263.70	350.61	10.02	**6.78**	42.51
	p10	65.38	40.69	5.09	**3.61**	13.73
	p11	-	-	5.96	**4.22**	23.28
	p12	-	-	17.15	**6.85**	75.30
	p13	6.31	11.56	7.48	**4.32**	140.92
	p14	-	-	28.38	**18.53**	133.32
	p15	30.39	22.83	14.84	**5.50**	13.56
	p16	-	-	35.44	**8.56**	19.12
	p17	566.79	-	16.45	**5.94**	194.19
	p18	-	-	73.93	**19.59**	-

	p19	-	-	105.18	**71.34**	-
	p20	-	-	-	-	-
Thoughtful	b-typed-01	24.62	21.35	14.97	18.14	**5.02**
	b-typed-02	**2.18**	5.35	3.17	2.19	10.82
	b-typed-03	2.56	7.78	2.94	**2.15**	15.21
	b-typed-04	4.67	5.97	5.82	4.42	**2.40**
	b-typed-05	12.92	15.07	**3.91**	10.14	50.82
	p11-6-53	-	-	-	-	-
	p11-6-59	-	-	-	-	-
	t-typed-20	-	-	-	-	-
	t-typed-21	-	-	-	-	-

Table 6.3: Runtime in seconds of Parallel Graphplan on all tested problem instances, using $h(x) = 1.5^x$ in sequential mode and with 2, 4 and 8 threads, compared to the standard configuration of Madagascar. A dash indicates that the problem has not been solved within 600 seconds.

The results in table 6.3 show that Madagascar is superior in the problem domains Floortile and GED, while Parallel Graphplan generally performs better on problems of the Hiking and Peg Solitaire domains. Neither planner is clearly superior on instances of the Thoughtful domain.

7 Conclusion and Future Work

7.1 Conclusion

This thesis proposed and evaluated a Graphplan- and SAT-based planning algorithm that runs on multiple processor cores. The evaluation results showed that parallelization yields a speedup compared to the sequential algorithm and even state-of-the-art SAT-based planning in some problem domains. This speedup is achieved by doing multiple plan extractions in parallel using multiple SAT solvers. However, a disadvantage of the proposed algorithm is the additional memory consumption due to the multiple big SAT formulas that are generated for plan extraction, which renders the algorithm somewhat unsuitable for use with a very high amount of threads.

7.2 Future Work

The algorithms can be optimized further in a number of ways. By using planning-specific heuristics and specialized data structures in the SAT solving algorithm, performance can be boosted significantly and memory consumption can be reduced [6].

A number of other encodings to SAT such as the $\exists$-step [29], $R\exists$-step [30] and $R^2\exists$-step [31] encodings can be used. They allow for a less strict selection of actions in each layer of the planning graph while preserving the correctness of the algorithm. This also means that the generated encodings will generally be shorter and thus less memory-consuming. These strategies can improve the performance significantly for different problem domains as shown in the respective papers. This allows the algorithm to be more fine-tuned for specific applications.

Certain strategies focus on enhancing SAT-based planning approaches for optimal and temporal planning using incremental MaxSAT solving [32] and the Program Evaluation and Review Technique (PERT) [33]. This can also be applied to the algorithm proposed in this thesis. Parallel plan extraction from multiple layers seems especially promising, because a cost-optimal plan doesn't necessarily use the least possible amount of layers in the planning graph. Thus, multiple plans can be extracted in parallel and then be evaluated for optimality which results in an overall speedup.

Another way to reduce memory usage is to only encode a fixed number n of layers into SAT and using assumptions to indicate which layers are represented in the formula. For example, plan extraction can be started with the goal at proposition layer 20, and for $n = 5$ may yield an assignment for the propositions in layer 15. The formula can then be reused to start plan extraction at layer 15 and so on. This way, the same clauses can be reused more heavily which is a big advantage when using incremental SAT solving.

It is also a very dynamic approach, since possible proposition assignments at any layer can be stored in a buffer and be prioritized for extraction using a heuristic. For the same reason it is also rather complex to implement and was omitted in this thesis.

Bibliography

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[2] M. D. Rodriguez-Moreno, D. Borrajo, and D. Meziat, "An ai planning-based tool for scheduling satellite nominal operations," *AI Magazine*, vol. 25, no. 4, p. 9, 2004.

[3] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[4] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2008.

[5] I. Cenamor, T. De La Rosa, and F. Fernández, "Ibacop and ibacop2 planner," *IPC 2014 planner abstracts*, pp. 35–38, 2014.

[6] J. Rintanen, "Madagascar: Scalable planning with sat," *Proceedings of the 8th International Planning Competition (IPC-2014)*, vol. 21, 2014.

[7] R. Valenzano, H. Nakhost, M. Müller, J. Schaeffer, and N. Sturtevant, "Arvandherd: Parallel planning with a portfolio," in *Proceedings of the 20th European Conference on Artificial Intelligence*, pp. 786–791, IOS Press, 2012.

[8] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial intelligence*, vol. 90, no. 1-2, pp. 281–300, 1997.

[9] H. Kautz and B. Selman, "Unifying sat-based and graph-based planning," in *IJCAI*, vol. 99, pp. 318–325, 1999.

[10] T. Balyo and C. Sinz, "Practical sat solving 2017. lecture notes." `https://baldur.iti.kit.edu/sat/files/2017/l02.pdf`. Accessed: 2018-05-03.

[11] T. Balyo and C. Sinz, "Practical sat solving 2017. lecture notes." `https://baldur.iti.kit.edu/sat/files/2017/l01.pdf`. Accessed: 2018-05-02.

[12] S. Gocht, "Incremental sat solving for sat based planning," Master's thesis, Karlsruhe Institute of Technology, 2017.

[13] D. M. McDermott, "The 1998 ai planning systems competition," *AI magazine*, vol. 21, no. 2, p. 35, 2000.

[14] T. Balyo, A. Biere, M. Iser, and C. Sinz, "Sat race 2015," *Artificial Intelligence*, vol. 241, pp. 45–65, 2016.

[15] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl-the planning domain definition language," 1998.

[16] C. Bäckström and B. Nebel, "Complexity results for sas+ planning," *Computational Intelligence*, vol. 11, no. 4, pp. 625–655, 1995.

[17] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[18] J. Hoffmann, "The metric-ff planning system: Translating"ignoring delete lists"to numeric state variables," *Journal of artificial intelligence research*, vol. 20, pp. 291–341, 2003.

[19] M. Helmert, "Concise finite-domain representations for pddl planning tasks," *Artificial Intelligence*, vol. 173, no. 5-6, pp. 503–535, 2009.

[20] R. Barták, "Planning and scheduling. lecture notes." `http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/lecture05eng.pdf`. Accessed: 2018-05-13.

[21] H. Kautz and B. Selman, "Satplan04: Planning as satisfiability," 2006.

[22] J. Chen, "Improving abcdsat by weighted vsids scoring schemes and various simplifications," in *Proceedings of SAT Competition 2017 - Solver and Benchmark Descriptions*, pp. 8–9, 2017.

[23] G. Audemard and L. Simon, "Glucose and syrup in the sat'17," in *Proceedings of SAT Competition 2017 - Solver and Benchmark Descriptions*, pp. 16–17, 2017.

[24] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pp. 244–257, 2009.

[25] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," *Proceedings of SAT competition*, vol. 51, 2013.

[26] R. Howey, D. Long, and M. Fox, "Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl," in *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pp. 294–301, IEEE, 2004.

[27] Á. García-Olaya, S. Jiménez, and C. Linares López, "The 2011 international planning competition," 2011.

[28] M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, S. Sanner, *et al.*, "The 2014 international planning competition: Progress and trends," *Ai Magazine*, vol. 36, no. 3, pp. 90–98, 2015.

[29] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12-13, pp. 1031–1080, 2006.

[30] M. Wehrle and J. Rintanen, "Planning as satisfiability with relaxed ∃-step plans," in *Australasian Joint Conference on Artificial Intelligence*, pp. 244–253, Springer, 2007.

[31] T. Balyo, "Relaxing the relaxed exist-step parallel planning semantics," in *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pp. 865–871, IEEE, 2013.

[32] R. Martins, V. Manquinho, and I. Lynce, "Open-wbo: A modular maxsat solver,," in *Theory and Applications of Satisfiability Testing – SAT 2014* (C. Sinz and U. Egly, eds.), (Cham), pp. 438–445, Springer International Publishing, 2014.

[33] Z. Xing, Y. Chen, and W. Zhang, "An efficient hybrid strategy for temporal planning," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pp. 273–287, Springer, 2006.