Karlsruhe Institute of Technology

Cube & Conquer-inspired Techniques for Parallel Automated Planning

Bachelor Thesis of

Jean-Pierre von der Heydt

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

Reviewer: Prof. Dr. Peter Sanders
Advisors: Dominik Schreiber
Tomáš Balyo

1 August 2019 – 31 October 2019

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read and observed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT).*

Karlsruhe, 31 October 2019

. . . . . .

(Jean-Pierre von der Heydt)

Abstract

The field of automated planning is concerned with the automatic generation of plans for different domains of problems. To allow the usage in as many fields as possible, only minimal assumptions are made to the domains. Since the complexity of these problems require a great amount of computation resources, the usage of multi core systems becomes more important. Nevertheless, there exists no clear best strategy to solve the planning problem in parallel. *Cube and Conquer* has proven to be a new and successful approach for solving problems of propositional logic in parallel. In this thesis we will translate the ideas of Cube and Conquer to automated planning.

We focus on the realisation of *cubes* trough nodes in the *State Space Graph* of the planning problem. In doing so, we present techniques to generate the cubes and assign computation time to them. Thereby we hope to approach parallel planning from a new angle. We are able to solve more problems than a comparable sequential algorithm. On hard test cases our algorithm scales well for up to eight cores. On easy test cases or when using more than eight cores we achieve no significant speedup.

Zusammenfassung

Das Forschungsfeld Automated Planning befasst sich mit dem automatisierten Generieren von Plänen für verschiedene Problemdomänen. Dabei werden geringe Voraussetzungen an die Probleme gestellt, um die Anwendung in möglichst vielen Bereichen zu ermöglichen. Da die Komplexität dieser Probleme eine große Menge an Rechenressourcen erfordert, spielt die Verwendung von Mehrkernsystemen eine zunehmende Rolle. Dennoch gibt es keine eindeutig beste Strategie Automated Planning parallel zu lösen. *Cube and Conquer* hat sich als neuer und vielversprechender Ansatz herausgestellt, um Probleme der Aussagenlogik parallel zu lösen. In dieser Arbeit werden wir die Ideen von Cube and Conquer auf Automated Planning übertragen.

Dabei konzentrieren wir uns auf die Umsetzung von *Cubes* als Knoten im *State Space Graph* des Planungsproblems. Wir stellen Techniken vor die Cubes zu generieren und ihnen Rechenzeit zuzuweisen. Dadurch versprechen uns eine neue Richtung einzuschlagen, Planung zu parallelisieren. Es ist uns möglich mehr Probleme als ein vergleichbarer sequentieller Algorithmus zu lösen. Auf schweren Testfällen skaliert unser Algorithmus zufriedenstellend für bis zu acht Kernen. Auf leichten Testfällen oder unter Benutzung von mehr als acht Kernen bleibt ein signifikanter Speedup aus.

Contents

1 Introduction

1.1 Motivation

Consider the following scenario: You are tasked with programming a robot to dip multiple instruments into chemical baths. In addition, you are given different constraints, for example: Some instruments need to be in special baths at specific times and there is a topological order that defines which baths have to be taken first. If the number of baths and instruments is small enough and the constraints are not too complex you can work out a plan by hand. But if the number of baths is bigger than twelve it is more promising to start designing a program that generates the plan for you. Since the criteria are manageable you are able to design a model of the problem after a short while. After giving the command to start generating the plan you go to sleep, confident that your work is done. But the next day your computer does not present you with a plan. Because you forgot about combinatorial explosion, the computer ran out of memory and refused further work. So you modify your program and think about some heuristics that allow you to save some time and memory in various special cases. And indeed, after a few more days of work the computer is able to come up with a plan that dips the instruments in the correct baths at the right time.

A week later, a friend asks you to help him program a solver for the new type of puzzle he invented. It contains a snake that needs to move around a board in a special way. You agree although you are already working on the next project. This time you do not program a robot to dip chemicals but work on a delivery plan for a big trucking company. So you start to wonder if there is some kind of magical algorithm that could do all these things at once: dipping instruments into chemicals, delivering packages and solving puzzles with snakes. But the requirements that must be met by this algorithm seem too strong to be fulfilled properly.

And here automatic planning comes into play. The goal of automated planning is to construct a single algorithm that is able to find plans for different domains of problems. Although such a claim of generality seems to be impossible to achieve, automated planning is already used for various problems.

This does not come without a cost. Most of the time the search spaces for these problems are unbelievably big. And the structure of a planning domain must be very generic in order to maintain generality. But the community around automated planning continuously develops new heuristics and methods to cope with these challenges.

1.2 Contribution

One key piece for practical algorithmics which gained importance over the last couple of years is the efficient usage of multiple cores. There exists no trivial solution on how to parallelize the algorithms used in automated planning. But since the increase of clock speed from processors seems to decrease, usage of multiple cores becomes necessary if an algorithm wants to keep benefiting from hardware improvements [Sut05]. Two important approaches for this are algorithm portfolios and divide and conquer algorithms. Portfolios run different algorithms and configurations on the same problem, hoping that at least one algorithm has the right configuration to quickly solve the problem. Divide and conquer techniques break down the problem and let each core solve a different part of it.

Recently, the concept of cube and conquer sparked our interest. It was shown to be a successful approach for solving difficult SAT problems in parallel. On a very basic level, algorithm portfolios can be described as different solvers working on the same problem while divide and conquer algorithms let multiple instances of the same solver cooperate to work on the same problem. Cube and conquer on the other side runs the same kind of solver on different problems.

We want to build upon the success of Cube and Conquer and apply its ideas to automated planning in order to approach parallel solving from a new direction. In this thesis we look at possibilities to translate SAT solving with Cube and Conquer into the world of automated planning. We identify challenges that arise with our approach and categorize them into three main sections: *finding*, *scheduling* and *solving*. For each section we propose multiple solutions to solve these challenges. We test different combinations against each other to get a better understanding on which attempts are fruitful for planning algorithms. This allows us to judge about which aspects of Cube and Conquer can be useful for automated planing and which aspects require furtherer work in order to be properly applied. For implementation we choose the Aquaplanning framework, since it is easy to adept and expand. Our best configuration is tested against a comparable sequential algorithm from the Aquaplanning framework. We are able to solve up to 15% more problems than the sequential implementation. On hard test cases we achieve an average speed-up of 3.5 using eight cores. However, our algorithm is not able to scale well for more than eight cores. Especially on easy test cases the sequential solver often outperforms all of our parallel solutions.

1.3 Structure of the Thesis

Chapter 2 gives a short overview on the current state of the art SAT solving and important methods to solve the planning problem in parallel. In Chapter 3 we explain how Cube and Conquer is used to solve SAT instances. Afterwards, we define the planning problem and look at algorithms that are commonly used for solving automated planning instances. We explain our parallel approach in Chapter 4 and evaluate the performance in Chapter 5. At last, we conclude our results in Chapter 5 and point out possibilities for future work.

2 Related Work

This chapter gives a brief overview on the current state of SAT solving and parallel automated planning. First, we look at the concept Cube and Conquer (C&C) and how it is used to solve SAT problems. Secondly, we talk about two important approaches for solving planning problems in parallel, algorithm *portfolios* and a divide-and-conquer mechanism by distributing the work via a hash function.

2.1 SAT

The SAT problem asks, whether a given formula of propositional logic is satisfiable (SAT). A formula is satisfiable if at least one assignment of truth values exists that evaluates the formula to true. It is also possible to ask if a given formula is unsatisfiable. In this case no satisfying assignment of truth values must exist. SAT is one of the most important NP-complete problems and many other problems can be reduce to finding a satisfying assignment for a propositional logic formula. Because of this, much research was invested in developing sophisticated heuristics for big SAT instances.

Modern SAT solvers can be categorized into conflict-driven clause learning solvers (CDCL) and lookahead solvers. CDCL solvers are especially successful at solving large but easy SAT instances. On the contrary, lookahead solvers perform better on small but hard problems. This is mainly due to the fact that CDLC solvers have a very local view in terms of heuristics and lookahead solvers have a more global view on the problem [HKWB11]. Popular example for state of the art SAT solvers are Minisat [EB05], Glucose [AS09] or Lingeling [Bie12].

Cube and Conquer The idea behind C&C is to combine lookahead solvers with CDCL solvers for a hybrid approach on SAT solving and was first proposed by [HKWB11]. C&C consists of two phases. In the first phase of C&C, a lookahead solver is used to partition the original problem into many sub-problems. The second phase uses a CDCL solver to solve these sub-problems. The intuition behind this is that the lookahead solver partitions the problem until it become easy enough that the CDCL solver can solve them faster than the lookahead solver. Section 3.2 gives a more detailed explanation of this algorithm.

On hard problems, it was shown that C&C is faster than a pure lookahead or a pure CDCL approach. In addition, C&C can be easily parallelized by distributing the cubes among multiple processors. A breakthrough made possible by C&C was the 2016 solution of the Boolean Pythagorean Triple problem [HKM16]. An implementation of the ideas of C&C is given by the parallel SAT solver Treengeling [Bie12].

2.2 Automated Planning

A solution to an instance of the planning problem is a sequence of actions, called plan, that transforms the initial state of the problem to a desired goal. The set of state can be represented as state space graph, with states as vertices and actions as edges. Finding a plan translates to finding a valid path in the state space graph. Therefore, graph search algorithms play an important role for the field of automated planning. In contrast to SAT, the question whether such a sequence exists in PSPCAE-complete. An important sequential planning system is given by the Fast Downward Planner [Hel06]. We want to give a short overview on existing strategies to solve the planning problem in parallel.

Algorithm Portfolio The idea of using algorithm portfolio for automated planning was first introduced by Robert and Howe [GSV09]. It is based on the observation that no solver or heuristic performs well on every planning domain. In fact, one solver might perform very well on a single planning domain and has difficulties with another whereas for a second solver it is the other way around. An algorithm portfolio tries to pick solvers that complement each other. This means picking solvers that perform well on different planning domains and therefore maximizing overall coverage. When a good set of solvers is chosen, they can either be run sequentially or in parallel. If the portfolio algorithms run sequentially or the number of processors is smaller than the number of solvers, scheduling has to be introduced. Otherwise, the solvers can run in parallel without much synchronization.

The advantage of this approach is that there exists hardly any communication overhead as long the solvers share no information with each other. A disadvantage is that the speedup of this approach is limited by the best sequential solver. It is likely that the portfolio algorithm has a better average performance than any of the used solvers. But on a specific problem, it will never be better than the fastest sequential solver. A more extensive overview on algorithm portfolios is given by [Val12].

A popular example for an algorithm portfolio is Fast Downward Stone Soup (FDSS) [HRK]. FDSS is a sequential portfolio planner that uses different configurations of the Fast Downward Planner [Hel06]. The portfolio makes the assumption that a configuration either solves a problem in a small amount of time or not at all. It uses relatively small time outs to try out a number of different configurations. Another example is ArvandHerd [Val+12]. In contrast to FDSS, it is a parallel planner that runs on multiple cores.

Parallel Search Algorithms In essence, planning consist of finding a path in the state space graph. Thus, parallel planning directly benefits from parallel search algorithms for graphs.

One example for this is Hash Distributed A* (HDA*) [KFB09]. It is a parallel A* algorithm that uses a hash function to distribute the work. Each thread keeps a list of open and closed nodes and has a receive queue for incoming nodes. As long as the receive queue of the thread contains a node, that node is compared against the closed list of the thread. If the closed list does not contain this node, it is added to the open list. When the receive queue is empty, the thread picks a node of the open list and explores it. Each newly explored node gets hashed and send to the corresponding thread depending on the hash value. For a hash function with uniformly distributed hash values, the work is split evenly among the processors.

This approach has been shown to achieve a significant speedup compared to sequential solvers. Additionally, it has access to a larger amount of memory and thus can work with bigger search spaces than a sequential approach. A major disadvantage is the constant sending and receiving of nodes, which results in a communication overhead.

Other parallel search techniques focus on improving load balancing by sending work from working threads to idle ones. An extensive overview to balance the load for a parallel depth first search is given by [San97].

3 Preliminaries

This chapter explains the concepts we uses throughout this thesis. We look more detailed on how C&C is used for solving SAT problems. Afterwards we explain the basics of automated planning. We will define the planning problem and discuss how to solve it with search algorithms. At the end, we describe the bandit problem, which will become useful when presenting some of our algorithms in Section 4.5.

3.1 SAT

In this section we briefly introduce basic terminology for the definition of the SAT problem.

Definition 3.1 (Literal, Clause, Conjunctive Normal Form)**:**
A *literal* is a variable A or a negated variable $\neg A$. A *clause* is a disjunction of literals e.g. $A \vee \neg B$. A formula of propositional logic is in *conjunctive normal form* (CNF) if it is a conjunction of clauses e.g. $(A \vee \neg A) \wedge (B \vee C)$. A formula F is *satisfiable* if there exists an assignment of truth values to the variables in F, such that F reduces to true. F is *unsatisfiable* if no such assignment exists.

Each formula of propositional logic F can be converted into a formula F' such that F is satisfiable if and only F' is satisfiable. The problem of deciding whether a formula of propositional formula in conjunctive normal form is satisfiable is called SAT. SAT is one of the most important NP-complete problems. As such, no polynomial algorithm that decides SAT is known.

3.2 SAT Solving with Cube and Conquer

This section explains what Cube and Conquer is and how it is used to solve SAT. As said in Section 2.1, the idea behind C&C is to combine lookahead solvers with CDCL solvers in two different phases. The sub-problems generated by the lookahead solver in the first phase are called *cubes*. The lookahead solver generates these cubes by building a binary decision tree that assigns some variables a truth value. Each edge in this tree symbolizes an assignment of one variable. The leaves in the decision tree represent the cubes. They can be obtained by walking down the path from the root of the tree and assigning truth values to the variables in the original problem corresponding to the edges. When the CDCL solver solves the cubes in the second phase, we also speak about conquering the cubes.

An outline of the C&C approach can be found in Algorithm 1 and 2 [HKWB11]. Algorithm 1 checks if a given SAT formula in CNF is satisfiable or unsatisfiable. In line 2, it uses Algorithm 2 to generate the cubes. Line 4 then calculates if any cube is satisfiable. This check is done by a CDCL solver and returns a satisfying assignment if one exists. The original problem is satisfiable if and only if any cube is satisfiable. Since this approach is mainly used to test for unsatisfiability, the order in which the cubes are tested by the CDCL solver does hardly matter.

Algorithm 2 is used to generate the cubes. As described above, it generates a decision tree. Each leaf of this tree will either be returned as a cube or discarded because the corresponding sub-problem cannot be satisfied. It takes two parameters: The first parameter describes the original SAT problem P in CNF and the second one is a truth assignment $\mathcal{A}$. The assignments in $\mathcal{A}$ can be understood as the assignments of the current path in the decision tree. By assigning these variables in P, we get a simpler

Algorithm 1: SATSolver

```
   Input: P : SAT problem
   Output: a : assignment of variables
1  C : Set of Cubes
2  C ← GetCubes(P, ∅)                                   // start cubing phase
3  for each c ∈ C do
4  |  a ← isSatisfiable(c)
5  |  if a ≠ ⊥ then                                    // if an assignment is found
6  |  |  return a
7  return ⊥
```

Algorithm 2: GetCubes

```
   Input: P : SAT problem, A : truth assignment
   Output: C : Set of Cubes
1  a : Atom
2  if isTriviallyUnsat(P, A) then
3  |  return ∅
4  else if shouldBeCutOff(P, A) then                    // if cut-off heuristic is triggered
5  |  return {newProblem(P, A)}
6  a ← lookaheadHeuristic(P, A)
7  return GetCubes(P, A ∪ {a}) ∪ GetCubes(P, A ∪ {¬a})
```

formula P'. How easy P' is to solve is not only determined by the size of $\mathcal{A}$. Some formulas can be nearly as hard as the original problem even if $\mathcal{A}$ is big, while others become very easy to solve after only a few variables have been set. The algorithms tries to determine the variables with the biggest effect on the original problem by letting a lookahead solver pick them.

Line 2 checks if P' is already trivially unsatisfiable. This happens if a clause of P' can be simplified to false after assigning the values of $\mathcal{A}$. If this is the case, P' does not have to be checked by the CDCL solver again and we return an empty set. In line 4, a heuristic is used to determine if the cube is already easy enough to be solved by a CDCL solver. This heuristic is called cut-off heuristic. The heuristic is computed based on the number of already assigned variables and the amount of variables they imply. The call in line 5 then returns the formula P'.

If none of the above conditions hold, the solver extends the current path in the decision tree by a new assignment. The heuristic of the lookahead solver in line 6 picks the variable. Afterwards, the algorithm calls itself recursively two times to walk down both new paths in the decision tree. The union of both calls is returned in the end.

3.3 Automated Planning

The research field of automated planning is concerned with solving planning problems. A planning problem is described by an initial state, a set of possible actions and a desired goal. To solve a planning problem, one has to come up with a valid sequence of actions that transfers the initial state into a state that satisfies the goal. In this thesis, we will approach the most basic formulation of a planning problem.

First, we will define some basic building blocks and operations to perform on them. These will be used to formalise the planning problem. Afterwards, we look at basic algorithms to solve it.

Definition 3.2 (Atom and Assignment)**:**
An *atom* is a variable with a value of `true` or `false`. We write $\mathcal{A}$ for a set of atoms. An *assignment* $\sigma\colon \mathcal{A} \to \{\texttt{true}, \texttt{false}\}$ assigns the atoms in $\mathcal{A}$ a value. A *partial assignment* $\tau\colon \mathcal{B} \to \{\texttt{true}, \texttt{false}\}$ with $\mathcal{B} \subseteq \mathcal{A}$ only assigns some atoms in $\mathcal{A}$ a value. An assignment σ *fulfils* a partial assignment τ if

$$\forall x \in \mathcal{B} : \sigma(x) = \tau(x).$$

The *application* of the partial assignment τ on the assignment σ is a new assignment $\hat{\gamma}(\sigma, \tau)\colon \mathcal{A} \to \{\texttt{true}, \texttt{false}\}$ defined by

$$\begin{aligned} \forall x \in \mathcal{A} \setminus \mathcal{B} : \hat{\gamma}(\sigma, \tau)(x) &\coloneqq \sigma(x), \\ \forall x \in \mathcal{B} : \hat{\gamma}(\sigma, \tau)(x) &\coloneqq \tau(x). \end{aligned}$$

Definition 3.3 (State, Action and Goal)**:**
Let $\mathcal{A}$ be a set of atoms. A *state* is an assignment of the atoms in $\mathcal{A}$. An *action* α consists of two partial assignments, the *preconditions* $pre(\alpha)$ and the *effects* $eff(\alpha)$. The action α is *applicable* in a state σ if σ fulfils $pre(\alpha)$. If α is applicable in σ then the *application* of the action α in the state σ is a new state $\gamma(\sigma, \alpha)$ and is defined by

$$\gamma(\sigma, \alpha) \coloneqq \hat{\gamma}(\sigma, eff(\alpha)).$$

A *goal* is a partial assignment of atoms in $\mathcal{A}$. It can also be understood as the set of states that fulfil this partial assignment.

We want to note that, in a similar way, an action α can be *unapplied* on a goal G. The result of unapplying an action on a goal is not a state but a set of states Σ. A state σ is in Σ if and only if α is applicable in σ and the application of α in σ fulfils G. We will see that the set Σ can also be understood as a new Goal G^*, so the result of unapplying an action on a goal is a new goal. First, we determine which actions have the potential to fulfil G. This is the counterpart to checking if an action is applicable in a state. An action α with $eff(\alpha)\colon \mathcal{C} \to \{\texttt{true}, \texttt{false}\}$ and $pre(\alpha)\colon \mathcal{D} \to \{\texttt{true}, \texttt{false}\}$ has the potential to fulfil the Goal $G\colon \mathcal{B} \to \{\texttt{true}, \texttt{false}\}$ if

$$\begin{aligned} \forall a \in \mathcal{C} \cap \mathcal{B} : eff(\alpha)(a) = G(a) \text{ and} \\ \forall a \in (\mathcal{B} \cap \mathcal{D}) \setminus \mathcal{C} : pre(\alpha)(a) = G(a) \text{ and} \\ \mathcal{C} \cap \mathcal{B} \neq \emptyset \end{aligned}$$

The first condition ensures that if an action effects an atom that must be fulfilled in the goal, it must have the correct value. Otherwise no state would exist which application with α can fulfil G, since at least one atom would not have the right value. The second condition ensures that atoms that are uneffected by the action but need to be fulfilled by the precondition and the goal have the same value. The last condition is not necessary to meet our requirements for the potential to fulfil G but we will see that the goals resulting from unapplying an action that does not meet this requirement are not interesting for us.

The predecessor goal $G^*\colon \mathcal{E} \to \{\texttt{true}, \texttt{false}\}$, which is obtained by unapplying α on G is defined by

$$\begin{aligned} \mathcal{E} &\coloneqq (\mathcal{B} \setminus \mathcal{C}) \cup \mathcal{D} \\ \forall a \in (\mathcal{B} \setminus \mathcal{C}) \setminus \mathcal{D} : G^*(a) &\coloneqq G'(a) \\ \forall a \in \mathcal{D} : G^*(a) &\coloneqq pre(\alpha)(a) \end{aligned}$$

As said before G^* can be understood as the set of states Σ. We obtain G^* by removing assignments of the effects from α and adding the assignments of the preconditions from α. The second equation ensures that all other assignments stay the same. The first and last equations remove and add the assignments of the changed atoms. If an action α' meets the first two conditions to be unapplyable in G but not $\mathcal{C}' \cap \mathcal{B} \neq \emptyset$, the Goal G' resulting from unapplying α' on G would just add assignments to G. So the set of states represented by G' is smaller than the set for G. However, this is of no use to solve the planning problem, since it only adds restrictions to the problem and does not introduce different conditions.

Definition 3.4 (Planning Problem)**:**
Let $\mathcal{A}$ be a set of atoms. A planning problem $P = (S, A, \sigma, G)$ is defined by a set of States $S = \{\texttt{true}, \texttt{false}\}^{\mathcal{A}}$, a set of Actions A, an initial State $\sigma \in S$ and a Goal $G \subseteq S$. A plan $p = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ is a sequence of actions. We can apply p on a state by defining the resulting state recursively as

$$\tilde{\gamma}(\sigma, \langle\rangle) \coloneqq \sigma \text{ and}$$
$$\tilde{\gamma}(\sigma, \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle) \coloneqq \tilde{\gamma}(\gamma(\sigma, \alpha_1), \langle \alpha_2, \alpha_3, \ldots, \alpha_n \rangle)$$

This assumes that each action is applicable in the corresponding state. The plan p solves P if $\tilde{\gamma}(\sigma, p)$ fulfils G. We can interpret this as $\tilde{\gamma}(\sigma, p) \in G$.

We will deal with the problem of finding a plan that solves a given planning problem. There are other variants that try to find an optimal, that means the shortest possible, plan or a near optimal plan. The following definition allows us to look at some of the theoretical properties of planning problems.

Definition 3.5 (PlanSAT)**:**
Given a set of atoms $\mathcal{A}$ and a planning problem P, decide whether there is a plan p that solves P.

Each atom of a planning problem P can be assigned two different values. Therefore, if the number of atoms for P is n, then the amount of possible states is 2^n. In addition, the length of a shortest plan for solving P is only bounded by 2^n as well. In fact PlanSAT is PSPACE-complete [Byl94]. Some extended planning models are even undecideable [EHN94]. Hence, there exists no known algorithm for solving the PlanSAT problem in polynomial time. It is not known if PlanSAT is in NP. The natural candidate for a witness, the plan p, can be exponentially long as well.

We can easily transform a planning problem into a graph by mapping states to vertices and applications of actions to edges. We call the resulting graph the state space graph and it has the following form.

Definition 3.6 (State Space Graph)**:**
Let $P = (S, A, \sigma, G)$ be a planning problem. The *state space graph* $G_P = (V, E)$ of P is defined as $V \coloneqq S$ and $(\sigma_1, \sigma_2) \in E$ if there exists $\sigma_1, \sigma_2 \in S$ and $\alpha \in A$ such that α is applicable in σ_1 and $\gamma(\sigma_1, \alpha) = \sigma_2$.

In rare cases, we will also consider a *goal space graph*. It is defined like the state space graph but uses goals as nodes and unapplying actions as edges. The transformation for the state space graph enables us to use graph search algorithms to solve the planning problem. In most cases an uninformed search will get lost in the exponentially big graph. Therefore, we require a sophisticated heuristic which guides us while searching for a plan. Heuristics play an important part in automated planning. In this thesis we are not interested in concrete implementations of heuristics and treat them as black boxes. The only thing that is of importance to us is that a heuristic takes a state of a planning problem and tries to estimate the minimum amount of actions required to reach the goal.

3.4 Search Algorithms

In this section we will clarify some terms considering search algorithms for graphs. This allows us to formulate later algorithms more simple and precise.

Frontier Most search algorithms keep track of which nodes of a graph have been visited and which ones will be explored next. We differentiate between these two types of nodes and call them *closed* and *open*. Closed nodes were already explored by the search algorithm and wont be reconsidered again for exploring. Open nodes were already visited by but not explored yet. We call the data structure that keeps track of both kinds of nodes the *frontier*. The interface of the frontier provides two methods ADD() and TOP(). The first method adds a new node to the set of open nodes from the frontier. If the node was not visited before, the node will be added to the set. If it was already open before, the frontier will discard it to avoid duplicates. If the node was already closed, it will be discarded as well. The second method returns a node from the set of open nodes that will be explored next by the search algorithm. Depending on the exact search strategy, the behaviour of this method will differ. For example, a breadth first search would return the nodes in a FIFO order but a depth first search would return them according to a stack. After the node has been retrieved, it will be marked as closed.

Search Strategies A search strategy determines the order in which a search algorithm explores nodes. Two common examples would be a breath first or depth first search strategy. Another important example for us will be the *best first search* and, a variant thereof, the *greedy best first search*. Both require a heuristic. As said before, a heuristic takes a search node and approximates a lower bound for the distance to the goal. The best first search always takes the node with the overall lowest heuristic value. This is usually implemented via a priority queue that holds all open nodes. The greedy best first search also takes the node with the lowest heuristic value but it only considers immediate children of the currently explored node that were not visited before. If there is no child or all children are closed, it backtracks like a depth first search. We will use both search strategies throughout this thesis.

3.5 One Armed Bandit

In Section 4.5 we will be confronted with the dilemma of choosing between exploration and exploitations. This dilemma can be formulated as a *multiple one armed bandit problem*. In this section we will explain the problem and a known solution to solve it.

For the bandit problem, we imagine multiple casino gambling devices which are commonly called slot machines or one armed bandits. At each point in time exactly one bandit can be played. This will result in a reward. The reward a bandit yields is random but follows some kind of distribution. A solution to the bandit problem aims to find a sequence of bandit plays that maximises the expected reward.

Definition 3.7 (Bandit Problem)**:**
Let n be the number of one armed bandits. The random variables $X_{i,1}, X_{i,2}, \ldots$ describe the bandit with the number $i \leq n$. The variable $X_{i,j}$ represents the reward that the i-th bandit yields if it is played for the j-th time. It is normalised to $[0, 1]$. Each of the variables $X_{i,j}$, $X_{i,k}$ $(j \neq k)$ are independent and identically distributed with an expected value of μ_i. Random variables $X_{i,j}$, $X_{k,l}$ with $i \neq k$ are also independent but not necessarily identically distributed.

This means that bandits have no state and the reward a bandit yields is not dependent on the number of times this bandit got played. Different bandits however can have different reward distributions.

There are known strategies to this problem that try to minimize the *regret*. The best strategy would always play the bandit with the highest average reward. We define the regret $\rho(m)$ of a strategy S after m plays as the expected difference in reward between the best strategy and S

$$\rho(m) := \mu^* m - \sum_{i=1}^{n} \mu_i \mathbb{E}[T_i(m)]$$

where μ^* is the expected reward value of the best bandit and $\mathbb{E}[T_i(m)]$ is the expected number of times bandit i got played after m total plays. The strategy we will use is called Upper Confidence Bound (UCB) and was introduced by [ACF02]. UCB assigns a value v_i to each bandit. This value is only dependent on past reward values of this bandit. The strategy plays each bandit once and after that plays the bandit that has a maximal v_i value with

$$v_i := \bar{r_i} + \sqrt{\frac{2 \ln m}{m_i}}$$

where $\bar{r_i}$ is the average of the past reward values, m is the total number of played bandits and m_i is the amount the i-th bandit got played.

4 Our Approach

In this chapter, we explain our ideas for applying the C&C approach to automated planning. At first, we discuss how to transfer C&C to automated planning. After that, we give a basic overview on the algorithm that we use to implement the C&C approach. It can be divided into three major phases: Finding the cubes, scheduling the cubes and solving them. All these phases will be explained in detail in their respective section 4.4, 4.5 and 4.6.

4.1 Translating the Cube and Conquer Approach to Automated Planning

We decided to model a cube for the planning problem P as a new planning problem P' with a new initial state or a new goal. Additionally, if we find a solution for P' we have to be able to easily construct a solution for P out of it. If we want to find candidates for P', we do a forward search in the state space graph of P and use the found nodes as a new initial state for P'. In a similar way we can do a backward search from the goal of P and generate new goals that we can use for P'.

4.2 Challenges

One of the biggest challenges that arise when modelling cubes this way is that we do not partition the search space. Cubes in context of SAT however partition the problem perfectly into many sub-problems. This has the effect that we will have overlapping search spaces when trying to solve each cube independently. In most cases the search spaces of two different cubes are even identical since the initial state of one cube can be reached by the initial state of the other cube and vice versa. If we repeatedly search trough the same part of the graph again we do not use our computation time efficiently. We approach this issue in the cube finding phase which is explained in Section 4.4.

Another difference between both types of cubes is that the cubes of a planning problem are rarely unsolvable while the cubes of the SAT problem can also be used to prove the unsatisfiability of a problem. In fact, most planners have difficulties to show that a non trivial planning instance has no solution. In order to prove that a formula is unsatifiable all cubes have to be proven unsatisfiable. For SAT disproving all cubes requires roughly as much work as disproving the original problem but for automated planning the work can multiply by the number of cubes. Since the search spaces of our cubes overlap we would just show that the same planning problem has no plan over and over again. Trying to show that a planning problem has no solution by applying the C&C method is hard to justify. Therefore we will only try to prove the existence of a plan for satisfiable planning problems.

In the case of unsatisfiability, all cubes have to be disproven, but if we are concerned with satisfiability we only have to solve one cube. Thus, the order in which we try to solve the cubes matters a lot. Regarding execution time, an optimal strategy would use all its computation time to solve the easiest cube. But of course it is not that easy to determine which cube is fastest to solve in advance. Instead we have split our computation resources between finding the easiest cube and solving the easiest cube. If we use too much of our time looking at all the cubes, trying to find the easiest one, we would be

overtaken by a sequential solver. But if we use all our computation time for solving a cube, the selection of this cube is uneducated and we might try to solve a very bad or even unsolvable cube. We introduce a scheduler in Section 4.5 that tries to balance the computation time between both concepts.

4.3 General Structure of the Algorithm

This section gives an overview on the approach we use to solve the planning problem. We also explain some terms we use throughout the thesis. It is implemented by Algorithm 3 and 4 which apply the C&C method for planning. Some methods of these algorithms are placeholders. We introduce different implementations of these methods in Section 4.4, 4.5 and 4.6 to achieve varying behaviour of the algorithm.

Algorithm 3: PLANNER

```
   Input: P: planning problem
   Output: p: plan
1  C : list of Cubes, Ĉ : list of lists of Cubes, p : plan
2  C ← findCubes(P)                                  // start cubing phase
3  C ← shuffle(C)
4  Ĉ ← split(C)
5  for each l ∈ Ĉ do                                 // start conquering phase
6      p ← startSchedulerThread(l)
7  while p = ⊥ do                                    // while no plan is found
8      wait()
9  return p
```

Algorithm 4: STARTSCHEDULERTHREAD

```
   Input: C: List of Cubes
   Output: p: plan
1  Σ : Scheduler, S : list of cube solver, s : cube solver, p : plan
2  for each c ∈ C do
3      S.add(newSolver(c))
4  Σ ← newScheduler(S)
5  while p = ⊥ do                                    // while no plan is found
6      p ← Σ.scheduleNext()
7  return p
```

Algorithm 3 is a parallel planning algorithm that takes a planning problem and outputs a plan that solves this problem. It does so by splitting the problem into multiple cubes and distributing them among the threads. As described in Section 4.1, cubes represent a new planning problem with either a new initial state or a new goal. In Section 4.4 we will look in detail on how to model these cubes.

The call in line 2 generates the cubes. The equivalent in C&C SAT solving would be the heuristic of a lookahead solver in combination with the cut-off heuristic. The exact amount of cubes is variable but we generally generate a lot more cubes than we have cores. Generating a good set of cubes is a crucial part for a successful C&C approach. We call this aspect of the algorithm the *cubing phase*. Section 4.4 discusses different possibilities for finding a good set.

In line 3 and 4 we shuffle the list of cubes and split them evenly. Each sub list of cubes will then be given to a different thread in line 6. Algorithm 4 describes the behaviour of the threads in more detail. This call is asynchronous and at this point the planner will run in parallel. We call the parallel phase of our algorithm the *conquering phase*. By shuffling the cubes we destroy any ordering that the cubing phase might have induced on the cubes. This distributes the work on all threads more evenly. Additionally, the threads which receive the sub-lists can make no assumptions about the order they receive their cubes in. By doing so we decouple the cubing phase from the conquering phase. This allows us to try different approaches for one phase without too much interference on the other.

Each thread tries to find a solution for one of its given cubes. As soon as one thread finds a solution, p will be overridden by a plan that solves the original problem in line 6. Afterwards all other threads will terminate, the main thread stops waiting in line 8 and then returns the solution of the problem.

Algorithm 4 describes the behaviour of the threads. Each thread receives a list of cubes and tries to find a plan for one of them. It stores a list of *cube solvers* that take turns to solve a cube. They get initialized in line 3. The C&C solver for the SAT problem uses a CDCL solver at this point. We will look at different options for cube solvers in Section 4.6.

Until no plan is found, the variable p is set to $\bot$. The thread then picks one solver according to the scheduling method of the *scheduler* in line 6 which assigns a fixed amount of computation time to it. If the cube solver finds a plan in the given amount of time we return it. Else we repeat the process. The policy after which a solver is picked is of great importance for the performance of our planner. We discuss different approaches in Section 4.5.

4.4 Finding the Cubes

As mentioned in Section 4.1 we want to model our cubes as a new planning problem with a new initial state or goal. In the case that we choose to model them with a new state it is easy to generate cubes by doing a forward search in the state space graph. For each node σ' that was touched by the forward search we know a valid plan that transforms the initial state of the original problem into σ'. Therefore if we find a plan for a problem with σ' as the initial state we can concatenate it with the plan to reach σ' from the original initial state and get a plan for the original problem. If we want to generate cubes with a new goal we do the same thing but as a backward search in the goal space graph.

The C&C algorithm for SAT solving uses a the cut-off heuristic to determine when to stop the cubing phase. This heuristic uses a parameter Θ that is dependent on the number of made decisions and the number of implications through these decisions. However, there is no good analogy for the number of implications while searching in a state space graph. We will fall back to a simpler approach that is not as dynamic. We introduce the parameter N which represents the amount of desired cubes and should be greater than or equal to the number of processors. The cubing phase will continue to search through the state space graph until N cubes are found. If it finds a plan for the original problem while searching for cubes, the conquering phase will be skipped and the plan returned.

The amount of computation resources invested into the cubing phase could also be bounded by time in a similar way. Both methods have their own advantages. Bounding by number of cubes gives more control over how much work each thread receives, while bounding by time allows more control over the ratio of sequential to parallel computation time.

Figure 4.1 illustrates this search. It represents the state of an arbitrary forward search in the state space graph of a planning problem. Depending on the concrete search strategy used the search will look very different. The search starts from the initial state σ and continues towards the goal on the right side. Two different kinds of nodes are known to the search algorithm: closed and open nodes. The closed nodes are symbolized by the thin black lines starting from σ, the open nodes are represented by the

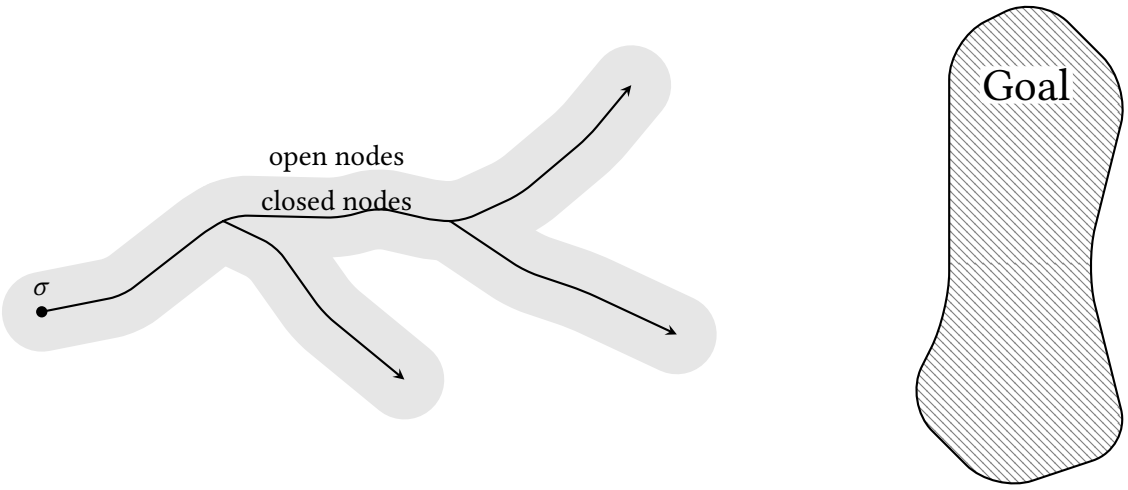

Figure 4.1: Illustration of finding cubes via a search in the state space graph.

grey area around these lines. To generate the cubes we will only use the closed nodes, found during the search. The only exception to this is a breath first search. In this case we use the open nodes for cube generation. The reasons to this will become clearer during this section but in short, the open nodes of a depth first search are too similar and will result in too much redundant work. Section 5.3 includes experimental results comparing the execution time of planners using open or closed nodes.

Breadth First Search If we want to generate a fixed amount of cubes N, the most simple approach would be to do a breadth first search until the layer that will be explored next is bigger than N. Each node in the last layer then becomes a cube. As simple as this approach is, it has some obvious problems. We will use breadth first search as a case study to emphasize important characteristics when finding cubes.

One problem is that our search does not extend deep into the graph. Even if the branching factor of our state space graph is not very big we can expect our layers to growth exponentially in most cases. Therefore we will only visit about $L = \mathcal{O}(\log(N))$ layers. Depending on the concrete planning problem, finding the solution can be significantly easier if only the first few actions of the solution are given. In most cases however, solving the cube wont be significant easier than the original problem.

Another difficulty we have to deal with when using a breath first search is that the resulting cubes can be very similar. The initial states of two different cubes can often be transformed into each other by only a few actions. Especially if the state space graph is undirected, two cubes from the breath first search will only differ by $2 \cdot L$ actions. Should two threads try to solve two cubes that are similar, they will search in similar regions of the graph. Therefore we do not benefit significantly from using multiple threads. For a diverse set of cubes, we try to maximise the minimal number of actions it takes to transform two cubes of the set into each other.

The last problem of the breath first search is that it searches blindly in the graph. Because we use no heuristic, it is likely that we return cubes that are harder to solve then the original planning problem.

We can summarise that a good set of cubes should fulfil two different characteristics. First, the nodes that become the cubes should tend to have a lower heuristic value than the initial node. This ensures that the problems from the cubes are easier to solve than the original problem. And secondly, we want to have a diverse set of cubes such that the search spaces for different cubes have less overlap.

Heuristic Search A simple approach to counter the weaknesses of a breath first search would be to use an informed search with a heuristic h. This means that we either use a best first or greedy best first search strategy together with h. If we want to find N nodes we continue to explore new nodes in the graph until we visited N nodes. All these nodes are then returned as cubes. Unlike in the case of a breath first search we use the closed nodes to return as cubes.

Overall we can expect the nodes of a heuristic search to have a better heuristic value than the nodes of a breath first search and they are therefore easier to solve. However, the search spaces of the cubes will still be very similar. Especially if we use a greedy best first search, the search spaces will be nearly identical. If we use a greedy best first search with the same heuristic and no randomization for finding and solving the cubes, the cube solver will make the same decisions when picking a predecessor as we did when finding the cubes. As a result, the solvers for each cube will search through exactly the same part of the graph. Only if a parent node has a better heuristic value than all of its children the cube finder has to pick a child σ while the solver that starts from σ can pick the parent. This rare case is the only one where the search spaces of the cube solver and cube finder differ.

The problem is not as severe if we use a best first search but the overlap of the search space will still be a significant factor. Especially if the cube finder and the cube solver use the same heuristic, it is probable that the heuristic will guide the cubes in the same direction. To counteract this, we propose three different approaches. We use different heuristics similar to a portfolio approach, introduce randomness to the heuristic and force the cube finder to pick nodes as cubes that are not too close together.

Portfolio A common approach to achieve more diversity is to use a portfolio of different heuristics. As mentioned in Chapter 2, portfolio planners are a fruitful attempt when dealing with a wider range of domains. We try to build upon this success and find a more diverse set of cubes by using a portfolio of heuristics to find cubes.

Algorithm 5: PORTFOLIO

Input: $P = (S, A, \sigma, G)$: planning problem, N : integer
Output: C : list of cubes

```
1 H : list of heuristics, R : set of nodes, C : list of cubes
2 for each h ∈ H do
3     R.addAll(findNodes(P, N/|H|, h))
4 for each v ∈ R do
5     C.add(newCube(v))
6 return C
```

Algorithm 5 implements this idea. The cube finder keeps a set of different heuristics H. For each heuristic $h \in H$ the finder starts a separate search from the initial state. This is done in line 3. The resulting set of nodes gets added to R. Duplicate nodes are ignored. Since all searches stop after $N/|H|$ cubes have been found, the total amount of found cubes in R is at most N. In the end the union of all sets is converted to cubes in line 5 gets returned in line 6.

Random Search Another way to get a diverse set of cubes with good heuristic values is to introduce randomness to the search. We use two different approaches to randomize a heuristic. The first randomizes a depth first search and the second a best first search. Both methods can start multiple searches from the initial node similarly to what a portfolio finder does, to cover different parts of the graph.

The first method looks at a set of nodes $\mathcal{F}$ and tries to pick a random node from it depending on the heuristic values. For each node $v \in \mathcal{F}$, we calculate its heuristic value $h(v)$. Then we assign each node a reward value $r(v)$ according to the following formula:

$$r(v) := h_{\max} - h(v), \text{ with } h_{\max} := \max_{v' \in \mathcal{F}} h(v') + 1$$

The node v will be picked with a probability of

$$p(v) := \frac{r(v)}{\sum_{v' \in \mathcal{C}} r(v')}$$

So nodes with a lower heuristic value have a higher reward value and therefore a higher probability to be picked. This approach is only feasible if the set $\mathcal{F}$ is small. If we add or remove nodes from $\mathcal{F}$, all values $p(v)$ change and it is difficult to update all these values efficiently. Thus, we use this approach as a modified greedy depth first search and the set $\mathcal{F}$ becomes the children of the node that we currently explore.

Algorithm 6: randomDepthFirst

```
   Input: P = (S, A, σ, G) : planning problem, N, D : integer
   Output: C : list of cubes
1  S : stack of nodes, R_g, R_l, F : set of nodes, v, μ : node, C : list of cubes
2  for i = 1 to D do
3      S ←newStack(), S.add(σ)                    // initialize new depth first search
4      R_l ← ∅
5      while not S.isEmpty() and |R_l| ≤ N/D do
6          v ← S.top()
7          F ← v.getUnfinishedChildren()
8          if F.isEmpty() then                          // if no child to explore
9              S.pop()                                                // backtrack
10         else
11             μ ←randomNode(F)                              // explore new node
12             S.push(μ), R_l.add(μ)
13     R_g.addAll(R_l)                    // add local set of nodes to global set
14 for each v ∈ R_g do
15     C.add(newCube(v))
16 return C
```

Algorithm 6 implements this approach. In addition to the parameter N it also uses D, which describes the number of repeated random depth first searches through the graph. Each depth first search uses its own stack S and a local set of closed nodes R_l. They get initialized in line 3 and 4. To ensure that at most N cubes are found, each depth first search stops after it has found N/D cubes. The node v that is currently explored is retrieved from the stack in line 6. The call in line 7 returns a list of all children $\mathcal{F}$ from v that are not closed by the algorithm yet. If this list is empty, v has no feasible children and the algorithm backtracks by popping v from the stack in line 9. Otherwise, a random node μ will be picked from $\mathcal{F}$ in line 11 according to the strategy explained above. The node gets added to the stack and the set of closed nodes in line 12. When the depth first search stops because it found enough cubes, all closed nodes get added to the global set of nodes R_g. These nodes will be converted to cubes and returned in line 15 and 16 after all depth first searches finished.

The second method does not have to update the probabilities of all considered nodes. Instead, if the heuristic value of a node $h(v)$ is calculated, a random real number $r \in [0.9, 1.1]$ is picked. Then the algorithm assigns v the value $h(v) \cdot r$. The node v will be treated as if it had a heuristic value of $h(v) \cdot r$. In contrast to the first method, the values of nodes do not change if new nodes get added or old ones get removed. Because of this, the set $\mathcal{F}$ of considered nodes can be bigger and we combine this method with a best first search. By slightly changing Algorithm 6 we achieve this behavior. The stack S gets exchanged by a priority queue. The call in line 6 returns and removes the node with minimal value. If the node is already closed it gets skipped, otherwise it gets expanded is now considered closed, by adding it to R_l. In line 7, all nodes in $\mathcal{F}$ get added to the queue according to the method above. No method for duplicate detection is applied. The *if-else*-block in line 8 is omitted. We call this algorithm a randomBestFirst search.

Cut-Off Search In order to get a diverse set of cubes we can try to force the cube finder to pick its cubes from different parts of the graph. Algorithm 7 implements this approach. The basic idea behind the algorithm is, that it keeps a set of search nodes we call anchors. If the algorithm visits a node that is too close to an anchor, it stops the search and continues elsewhere. The algorithm will add a new node to the set of anchors after it has searched through a new part of the graph for some time. The new picked anchor then ensures that the algorithm has to stop the search in this part of the graph and find a new one.

Algorithm 7: cutOffSearch

Input: $P = (S, A, \sigma, G)$: planning problem, N : integer
Output: C : list of cubes

```
1  A : list of nodes, F : frontier of nodes, C : list of cubes, v : node, P : list of nodes,
2  F.add(σ)
3  while |C| < N do
4      v ← F.top()                                          // explore new node
5      if cutNodeOff(A, v) then                  // if node is too close to an anchor
6          continue                                  // cut node off by ignoring it
7      else
8          if becomeAnchor(v) then
9              A.add(v)                                 // add v to the list of anchors
10         P ← getPredecessors(v)
11         F.addAll(P)
12         C.add(newCube(v))
13 return C
```

Algorithm 8: cutNodeOff

Input: A : list of nodes, v : node
Output: b : boolean

```
1  for each ω ∈ A do
2      if h.distance(v, ω) ≤ Φ · h.distance(σ, ω) then
3          return true
4  return false
```

This approach is implemented by Algorithm 7 which takes a planning problem P and an integer N as input. It then tries to find N cubes for the problem P. It uses a list of nodes A which work as the set of anchors and a frontier F. After it has initialized the frontier with the initial state of P it continues to explore the graph until it found enough cubes.

In line 4, the frontier provides the node v that should be explored next. If the node is too close to one of the anchors, the call in line 5 will return true. If this is the case the node does not become a cube and we continue to search elsewhere in the graph by picking a new node from F. Since v is a closed node at this point, it will not be reconsidered again whether it becomes a cube. This call is implemented by Algorithm 8.

In the other case, we continue to explore v. The function in line 8 determines if our node should become an anchor. In this case, it gets added to the list of anchors. We use the additional parameter Ψ if we want to determine if a node should become an anchor in line 8. The integer Ψ defines the amount of anchors that are chosen when the algorithm returns. To achieve that, the algorithm picks new anchors at an interval of $i \coloneqq \lceil N/\Psi \rceil$. This means that every i-th node that should become a cube also becomes an anchor. Afterwards, the node will get explored by adding its children to the frontier in line 10 and 11. We also add v to the set of cubes our algorithm will return in the end.

Algorithm 8 works as follows: It uses a heuristic h to approximate the distance between two states and an additional parameter Φ which is a real number between zero and one. Φ decides how close a node has to be to an anchor in order to be cut off. For each anchor ω, it approximates the distance δ between the initial node of the planning problem σ and ω. If the approximated distance between the node that we currently explore v and ω is smaller than $\varepsilon \coloneqq \delta \cdot \Phi$ we cut v off. The idea is that we build an ε-ball around ω and each node that is in this ball will be cut off.

Figure 4.2 illustrates this procedure. It represents the state of Algorithm 7 while searching for cubes in the state space graph. The search started from the initial node σ. A thick black line indicates the path of the search. Most nodes on the path are omitted for clarity. Each node on the path is labelled as finished and picked as a cube. The algorithm already found the two anchors ω_1 and ω_2 in that order and is about to find the third anchor ω_3. The ε-balls around the anchors mark the region of nodes that will be cut off. After an anchor has been picked, all nodes inside of the ε-ball will be ignored by the search and cannot become cubes. We can think about this as cutting the ball out of the graph. Thus, we force the search algorithm to branch off and continue in a new part of the graph. We can see this in the figure. After the algorithm found the anchor ω_1 or ω_2, it had to backtrack until it left the corresponding ε-ball and then had to search around it.

Considering ω_3 the Figure 4.2 visualizes the calculation of the ε-ball. First the distance δ_3 between σ and ω_3 is calculated using the heuristic h. Together with the parameter Φ we calculate ε_3 and can draw the ε_3-ball around ω_3.

4.5 Scheduling the Cubes

We already mentioned in Section 4.2 that scheduling will be an important part of our C&C approach. In our context, scheduling is the process of repeatedly choosing a solver for a cube and assigning a fixed time slice of computation time to the solver. More precisely, this section is concerned with the implementation of line 6 from Algorithm 4. First, we will look at the most basic form of scheduling, a *Round Robin* scheduling algorithm. We can then discuss the weaknesses of this approach and formulate a more informed *Bandit* scheduling algorithm. At last, we will tweak this approach, which results in the *Greedy Bandit* scheduling algorithm.

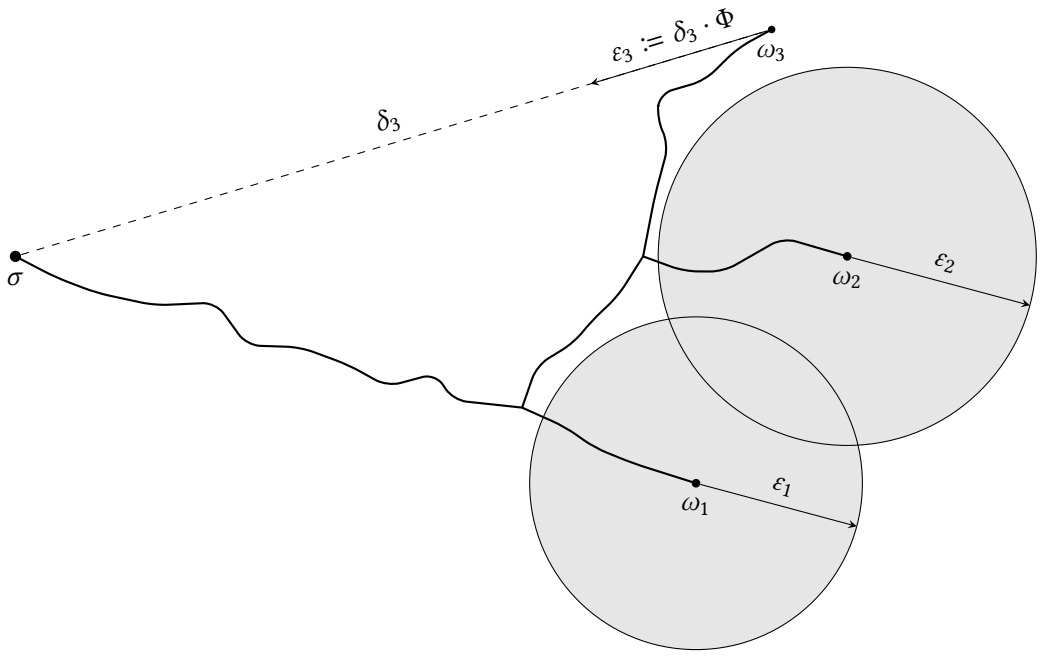

Figure 4.2: Illustration of a cut-off search.

Round Robin A Round Robin scheduling algorithm keeps a circular list of all solvers. It has an internal pointer that initially points to the first solver in its list. If the scheduler should pick a new solver it returns the element its pointer is currently pointing to and moves the pointer to the next element in the list.

The problem with this approach is that the scheduler does not focus the computation time on a single solver. In fact, it distributes the computation time equally across all solvers. This has the effect that if all cubes need an equal amount of time to solve, the scheduler would need an amount of time equals to the number of cubes times the amount of time it needs to solve a single cube to solve the original problem. Therefore, Round Robin performs especially bad if we have a lot of cubes or the cubes are hard to solve.

Bandit The problem we face when choosing a cube solver is an example for the exploration/exploitation dilemma. We already discussed this dilemma in Section 3.5. In this paragraph we want to describe how to apply the UCB strategy for the bandit problem to resolve the dilemma for our scheduling problem.

We will apply the UCB strategy to our scheduling problem in the following way. Cube solvers are equivalent to the bandits and playing a bandit means assigning computation resources to the corresponding solver. The difficult part is to assign a reward value to our solvers that fulfil the conditions of Definition 3.7. A solver which is closer to solving its cube should have a higher reward value than a solver that needs more time to solve it. To approximate the time a solver needs to solve its cube, we use a heuristic. We identify the heuristic value of a solver with the heuristic value of the node that was finished last by that solver. The reward value of a solver should then be dependent on its heuristic value.

We propose the following reward function. It takes the history of all prior heuristic values of this solver L and outputs a reward r between zero and one. Each time a solver uses computation time, we add the updated heuristic value to L. Doing so we can plot these values as points on a two dimensional plane with time on the x-axis and heuristic value on the y-axis. Let $n + 1$ be the size of L. If the i-th

entry ($0 \leq i \leq n$) in L has value y_i then the point representing this entry has the coordinated (x_i, y_i) with $x_i := \delta i$ where δ is the size of the time slice that gets assigned to scheduled solvers. We can then draw the linear regression line $y = \alpha + \beta x$ through these points with

$$\beta := \frac{\sum_{i=0}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=0}^{n}(x_i - \bar{x})^2} \text{ and } \alpha := \bar{y} - \beta\bar{x}$$

where $\bar{x}$ and $\bar{y}$ are the mean values of all x_i or y_i values. If we calculate the intersection with the x-axis, we get an approximation r' for the remaining time our solver needs to finish. We scale this value between zero and one to get r

$$r' := \begin{cases} \min\left(-\frac{\alpha}{\beta},\ B\right) & \text{if } b < 0 \\ B & \text{else} \end{cases} \text{ and } r := \frac{B - r'}{B}$$

where B is a generous upper bound for the total time our solver needs to solve the problem.

Algorithm 9 implements this approach. It will get called repeatedly by the C&C planner to schedule a new solver. It takes no input and outputs a plan if the scheduled solver finds one. The algorithm keeps a list of all solvers S. Each solver $s \in S$ has two additional attributes. The attribute $s.r$ is a list of all reward values of this solver and $s.h$ is a list of all prior heuristic values of the solver. The lists $s.h$ are initialized with one element representing the heuristic value of the initial state for the solver s. The lists $s.r$ are initialized empty. An invariant for the algorithm is that if the list $s.h$ has size n then the list $s.r$ will have size $n + 1$. The integer i is initialized to zero and guarantees that each solver get scheduled at least once.

The check in line 2 schedules each solver once before applying the UCB strategy. This ensures that the heuristic lists of each solver has at least size two and we can calculate the linear regression line to assign a reward value. Afterwards, the UCB strategy in line 6 picks the solver with the highest v_i value. The picked cube solver then tries to solve its cube for a fixed amount of computation time in line 7. We add the new heuristic value of the solver to its list in line 8. After that we can calculate the new reward value for this solver as we described above. In the end, we return p this will be either $\perp$ or a valid plan depending on whether the solver solved the cube.

Algorithm 9: BANDIT

```
   Output: p : plan
1  S : list of solvers, s : solver, p : plan, i : integer
2  if i < S.SIZE() then                    // if not every solver was played at least once
3  |  s ← S.GET(i)
4  |  i ← i + 1
5  else
6  |  s ← arg max_{s'∈S} UCB(s'.r)          // pick a solver after the UCB strategy
7  p ← s.CALCULATEUNTILTIMEOUT()
8  s.h.ADD(s.GETHEURISTICVALUE())
9  s.r.ADD(CALCULATEREWARD(s.h))
10 return p
```

We recall that the random variables $X_{i,1}, X_{i_2}, \ldots$ have to be independent and identically distributed. Our reward function does not fulfil this condition but under the assumption that the heuristic value of a solver will decrease linearly with the amount of computation time it uses, we get close to the desired behaviour. Another problem with our reward function is that the UCB strategy tries to minimise $\rho(m)$ and we do not necessarily benefit from that. Consider the case that we have multiple cube solvers

$i_1, i_2, \dots i_k$ that need an equal amount of time to finish their respective cube. All these cube solvers would have a similar expected reward value $r_{i_1} \approx r_{i_2} \approx \dots \approx r_{i_k}$. If the best solver i^* also has a similar reward value $r^* \approx r_{i_1}$ the regret of the UCB strategy $\rho(m)$ will be the same whether it always picks the best solver or all solvers $i_1, i_2, \dots i_k$ equally often. Only picking the best solver will result in fast solving of the original problem but switching between multiple solvers will result in a much higher execution time. Therefore, a small regret value does not lead to a small computation time. This problem can be boiled down to the reward value of a solver not increasing when it is played often. In fact, the UCB strategy is less likely to pick a solver that has been played often. Additionally, the UCB strategy seems to need a big constant of plays before it plays well. Since playing a solver for a fixed amount of time is very costly, the strategy runs into problems when we have a great number of solvers or a big scheduling interval.

Greedy Bandit The *Greedy Bandit* scheduler is a simplified version of the bandit approach and aims to circumvent its problems. It calculates a reward value in a similar way as the bandit scheduler but does not apply the UCB strategy. Because the reward function does not have to fulfil the requirements of the bandit problem, we have more freedom in defining it.

The bandit scheduler has the problem that it does not take into account how often a scheduler gets played. A very effective way to counteract this is to modify the reward function in the following way. The reward function still takes the list of all prior heuristic values as input but rather than calculating a scaled approximation of the finishing time of the solver it outputs an approximation on the *remaining* finishing time of the solver. Let $y = \alpha + \beta x$ be the same linear regression line as in the bandit approach, δ the size of the scheduling interval, i the amount this bandit got scheduled in the past and B be a generous upper bound for the total solving time. We define our reward value r for the Greedy Bandit scheduler as

$$r := \begin{cases} \min\left(-\frac{\alpha}{\beta} - i\delta,\ B\right) & \text{if } b < 0 \\ B & \text{else} \end{cases}$$

If the linear regression line has a slope greater than zero we set the reward value to B. Otherwise, we calculate the approximated finishing time by intersecting the linear regression line with the x-axis $-\alpha/\beta$ and subtract the amount of time this cube got scheduled $i\delta$ to obtain the approximated remaining calculation time to finish the problem. The scheduler schedules the solver with lowest reward value.

Algorithm 10 implements this approach. It works similar to the bandit scheduler but has some small differences. In addition to the list of all solvers, it also keeps a priority queue with solvers. The solvers will be added to this priority queue after their reward value was calculated for the first time. The solver with the lowest reward value will be at the front of the priority queue.

The first phase of the algorithm works the same as for the bandit scheduler. Each solver gets scheduled once in Line 2 to 4. In the second phase, the scheduler picks the solver with the lowest reward value by removing the top element of the priority queue in line 6. After that the heuristic and reward lists get updated accordingly in line 7 to 9. At the end in line 10 the scheduler will add the scheduled solver to the priority queue with the updated reward value.

The reward function of the greedy bandit scheduler has several advantages over the reward function of the bandit scheduler. First of all, it is easier to calculate. The UCB strategy must update the μ_i values of all solvers after a single solver was scheduled. The greedy bandit scheduler must only update the value of the scheduled solver. This allows the usage of a priority queue which in turn leads to a faster selection of the best solver.

Secondly and more importantly, the new reward function takes into account how often a solver gets scheduled. We recall that the bandit scheduler has problems to focus its computation time on a single solver if multiple solvers have similar finishing times. We can apply this scenario on the greedy bandit

Algorithm 10: greedyBandit

```
   Output: p : plan
1  S : list of solvers, P: min priority queue, s : solver, p : plan, i : integer
2  if i < S.size() then                    // if not every solver was played at least once
3  |   s ← S.get(i)
4  |   i ← i + 1
5  else
6  |   s ← P.popTop()
7  p ← s.calculateUntilTimeout()
8  s.h.add(s.getHeuristicValue())
9  s.r.add(calculateReward(s.h))
10 P.add(s, s.r.getLast())
11 return p
```

scheduler. Since all solvers have similar finishing times they will have similar values $-\alpha/\beta$ for the linear regression line. This value will get reduced by 2δ for the first scheduler that gets scheduled twice and and reduced by 1δ for all other solvers. After that the reward value for the solver that was scheduled two times will be significantly lower than the values of the other solvers. The scheduler will continue to schedule this solver which will in turn decrease the reward value even more. This chain reaction will only stop if the solver gets multiple bad reward values which will increase $-\alpha/\beta$ up to the point where it overtakes $i\delta$. Therefore, the greedy bandit scheduler is better at focusing its computation time in a single solver.

4.6 Solving the Cubes

We already discussed the process of generating and scheduling cubes. In this section we describe how to solve them. For most cases we treat the cube solver as a black box and use the same implementation for it as the default sequential solver. The sequential solver does a forward search in the state space graph of the planning problem. It uses a best first search strategy with a greedy approximation of the FastForward heuristic [HN01] by Marvin Williams. The only exception to this is the randomized cube solver.

Random Solving In Section 4.4 we looked at different methods to randomize the cubing phase of our C&C approach. We can apply the same method to randomize the conquering phase. The first randomization method results in a solver that uses a greedy best first search strategy but picks the next node at random according to the strategy explained in Section 4.4. The second randomization method leads to a solver using a best first search strategy with randomized heuristic values for the nodes of the state space graph. We call these solvers *random depth first solver* and a *random best first solver.*

4.7 Optimisations

This section includes additional techniques we introduce to render our approaches more efficiently.

Sharing Finished Nodes Currently threads share no information between each other to keep the communication overhead low. This has the disadvantage that the search spaces of different threads can easily overlap. We propose the following option as a compromise. In addition to assigning each solver a

cube to solve, we also share the set of finished nodes from the cube finder with each thread. Each cube finder keeps two set of nodes: its own set and the set from the cube finder. If a cube solver needs to check if a node is already finished, it performs a lookup in both sets.

This option cannot be combined with sparsening. Otherwise it is possible that the part of the state space graph containing the goal is not reachable for any cube solver, because it is surrounded by finished nodes from the cubing phase. In this case the algorithm loses completeness.

Sparsening We recall that the C&C approach for SAT solving generated a large set of cubes. A key aspect of this is, that the total amount of work is reduced with a larger amount of cubes. But in our case the amount of work increases or even multiplies with the number of cubes and the scheduler has a harder time finding the easiest cube. Therefore we face the dilemma of keeping the number of cubes low in order to keep the total amount of work low but at the same time having a large and divers set of cubes with good heuristic values. To deal with this dilemma we introduce sparsening.

Under *sparsening* we understand the process of removing cubes from the original set C that is returned at the end of the cubing phase and instead returning the smaller set C' for the conquering phase. This way the set becomes more diverse and the scheduler has an easier time to distinguish between promising and unpromising cube candidates.

We propose two different approaches to remove cubes from C: a deterministic and a random one. Both use the parameter p, which is a real number in the interval $(0, 1]$. The parameter indicates what percentage of the original cubes are removed. In other words $|C| \cdot p = |C'|$. The lower p gets the longer the cube finder has to search for cubes if N stays constant. If $p = 1$ no sparsening is done at all. If $p < 1$, the cube finder has to search for $N \cdot 1/p$ cubes until it can return C'. In the case of deterministic sparsening the cube finder only adds every $1/p$-th cube to C'. Every other cube candidate is simply discarded. By doing so, the gaps between the cubes are evenly spaced. The random approach chooses $p \cdot |C|$ cubes from C at random and returns them. Both approaches add a cube representing the initial state σ to C' at the end so that our algorithm stays complete.

5 Evaluation

This chapter evaluates our contributions from Chapter 4. We explain our test set up and implementation in Section 5.1 and 5.2. Afterwards we compare different approaches and evaluate their performance in Section 5.3.

5.1 Implementation

We implemented our algorithms in Java, building on the Aquaplanning framework[1]. Aquaplanning already provides methods to parse, solve and verify planning problems (in PDDL format). Parsed problems are represented as an internal *Ground Planning Problem* class. To work with this class, the framework also provides methods and objects representing graph structures and search algorithms. Our own algorithms were realized by implementing the *Planner* interface and modifying the existing graph classes to fit our needs. Throughout all algorithms we use a greedy approximation of the FastForward heuristic [HN01] by Marvin Williams.

5.2 Experimental Setup

Our experiments were run on a machine using an AMD Epyc 7551P processor with 32 CPU cores and a clock rate of 2GHz. We had 256GB of DDR4 RAM accessible but never came close to exhausting it.

We used GNU parallel[Tan+11] to run multiple instances of our algorithms at once. The generated data was analysed using Python[2] and PyPlot[3].

We used the following optimisation domains from past international planning competition: agricola, barman, data-network, floortile, hiking, parking, snake, termes, tetris, tidybot, transport, visitall [Pom18]. To generate a manageable set of benchmarks, we ran a sequential solver on several planning domains. Only the test cases that were solved by the sequential solver after at most 5000 seconds are considered for parallel testing. The sequential solver solved 171 out of 231 problems. If we only consider problems solved before 1000 seconds, the sequential solver solved 143 problems. This test set was used to test most parallel solvers. In Section 5.4, we choose the most promising solvers to run on the complete set of test cases. Unless otherwise stated, all parallel solvers were run on four cores for 1000 seconds on the small test set.

5.3 Experimental Results

Scheduler Although we mentioned scheduling methods last in Chapter 4, we will evaluate them first. Taking a look at the different scheduling methods in combination with basic cube find algorithms gives us a good starting point to start from and compare more advanced approaches. Table 5.1 gives an overview on the number of solved test cases for basic parameter combinations. It lists the number of

[1] https://github.com/domschrei/aquaplanning
[2] https://www.python.org/
[3] https://matplotlib.org/api/pyplot_api.html

solved instances for all combinations of the round robin, bandit and greedy bandit scheduler with a BFS or heuristic cube finder. The scheduler was configured with a 10 or 100 milliseconds time slice. The cube finder searched for 40 cubes or 400 cubes.

We can see that the bandit scheduler consistently outperforms the round robin scheduler, which in turn is outperformed by the greedy bandit scheduler. This coincides with our intuition about scheduling in Section 4.5. The greedy bandit scheduler seems to be able to focus its computation time better that the other two schedulers.

Additionally, using a BFS strategy to find cubes is inferior to using a heuristic best first search in most cases. Due to the fact that the impact of the cubing phase is lower when giving it less time, this effect is reduced when using less cubes. When comparing the BFS and heuristic search for 400 cubes, the heuristic approach solves more problems. Although most of the computation resources are invested in the conquering phase, this shows that the cubing phase has a significant impact on the overall performance of the algorithm.

The impact of the time slice has no noticeable impact on the number of solved test cases as long as it is chosen big enough. For a time slice of t seconds and n cubes per thread, every scheduler needs $t \cdot n$ seconds to schedule each cube once. So if T represents the time-out of the algorithm in seconds, t and n should be chosen in a way that $t \cdot n \ll T$. Otherwise, the scheduler does not have enough time to evaluate promising cubes. Increasing the number of cubes from 40 to 400 seems to decrease the performance of the algorithm. The BFS search strategy and the round robin scheduler suffer the most from this. However, this does not mean that we want to keep the number of cubes small for further tests. Choosing a small set of cubes means that the cube solvers solve more similar problems, which makes it difficult to achieve a speedup compared to a sequential solver. With that in mind, we want to note that it is a good property of the heuristic search in combination with the greedy bandit scheduler to not be affected by a change in the number of cubes. We will further investigate on what a good amount of cubes is when we evaluate sparsening techniques for the cubing phase.

Figure 5.1 compares the performance of the sequential solver and a C&C solver, using the greedy bandit scheduler with a time slice of 100 ms and a heuristic best first search cube finding phase, searching for 400 cubes in a cactus plot. For each solver, we sorted the list of execution times to solve test cases independently. The sorted lists are plotted on the graph, with the number of the test case in the sorted list on the x-axis and the execution time in seconds on the y-axis. Test cases that were already solved during the cubing phase are plotted in a different colour. We only plot points for test cases that were solved before 1000 seconds. A vertical lines indicate the time-out for the algorithm.

Two interesting observations can be made. First, the parallel approach has difficulties to keep up with the sequential solver and manages to close the gap for longer execution times. Part of the reason for this is that the test set used to generate this plot favours the sequential solver, since we only ran the parallel algorithm on tests that the sequential already solved. Secondly, most problems that are solved before ten seconds and some problems that are solved after 800 seconds are solved during the cubing phase. This shows that it is hard to bound the amount of time invested in the cubing phase by the number of cubes. On the other hand it is difficult to determine the number of found cubes for a fixed time of cubing.

Figure 5.2 plots the same points of data but does not sort them. Instead, we vertically divide the plot into the different domains in which the points belong to. The ordering of the domains is the same as mentioned in Section 5.1. This way we can see if the parallel approach performs significantly better on a specific domain than the sequential solver. We see that the parallel algorithm performs better on all three solved barman instances. But the sequential algorithm noticeably outperforms the parallel on termes and tetris. We make a clearer distinction of this when evaluating our best approach in Section 5.4.

Further tests use the greedy bandit scheduler by default.

		cube finder			
		BFS		heuristic	
scheduler	time slice	40 c	400 c	40 c	400 c
round robin	10 ms	116	96	118	106
	100 ms	118	100	117	107
bandit	10 ms	131	108	128	123
	100 ms	129	112	126	125
greedy bandit	10 ms	139	123	142	138
	100 ms	140	123	139	142

Table 5.1: Number of solved problems for different parameter combinations

	forward		backward	
time slice	40 c	400 c	40 c	400 c
10 ms	142	138	57	48
100 ms	139	142	62	37

Table 5.2: Comparing number of solved problems between a backward and a forward search for finding cubes

Backward Search In Section 3.3, we mentioned the possibility of a backward search in the goal state graph. In this paragraph, we take a short look at a possible usage of a backward search for our C&C approach. We can use a backward search to find the cubes and a forward search to solve them. In this case, the cubes become problems with a new goal and not with a new state, as it is the case with a forward search. The behaviour of this algorithm can be compared to a bidirectional search.

Table 5.2 compares the performance of our C&C approach using a forward or a backward search in the cube finding phase. We see that the backward search performs exceptionally bad. There are two main reasons for this. First, within the used framework, the generation of nodes in the goal state graph is implemented rather inefficiently. And secondly, the heuristics we used were not intended to estimate distances in a goal state graph. We believe that with further work, the performance can be significantly improved but we do not follow this approach any further in the thesis.

Open and Closed Nodes We recall that we use closed nodes as cube candidates for all cube find algorithms except a BFS. This paragraph compares the usage of open and closed nodes as cubes and justifies this decision.

Table 5.3 lists the number of solved problems for a C&C solver running on the small test set. It compares the usage of open and closed nodes in combination with a BFS or best first search strategy in the cube finding phase. In all cases, a greedy bandit scheduler was used with time slices of 10 or 100 ms. The table shows a slight trend in solving more problems if open nodes are used in combination with a BFS and closed nodes in combination with a heuristic best first search. Although this trend is subtle, we hope that its impact increases with the overall performance of our algorithm.

Intuitively, this can be explained as follows: In the case of a BFS, the children of most closed nodes are closed too. That way, work which was already done by the cube finder is redone by the cube solvers. But if the number of closed children is close to one, like in the case of a depth first search, using closed nodes as cubes allows to explore deeper into the graph before reaching the limit of required cubes. If the average branching factor of a graph is β and the average number of closed children is about one,

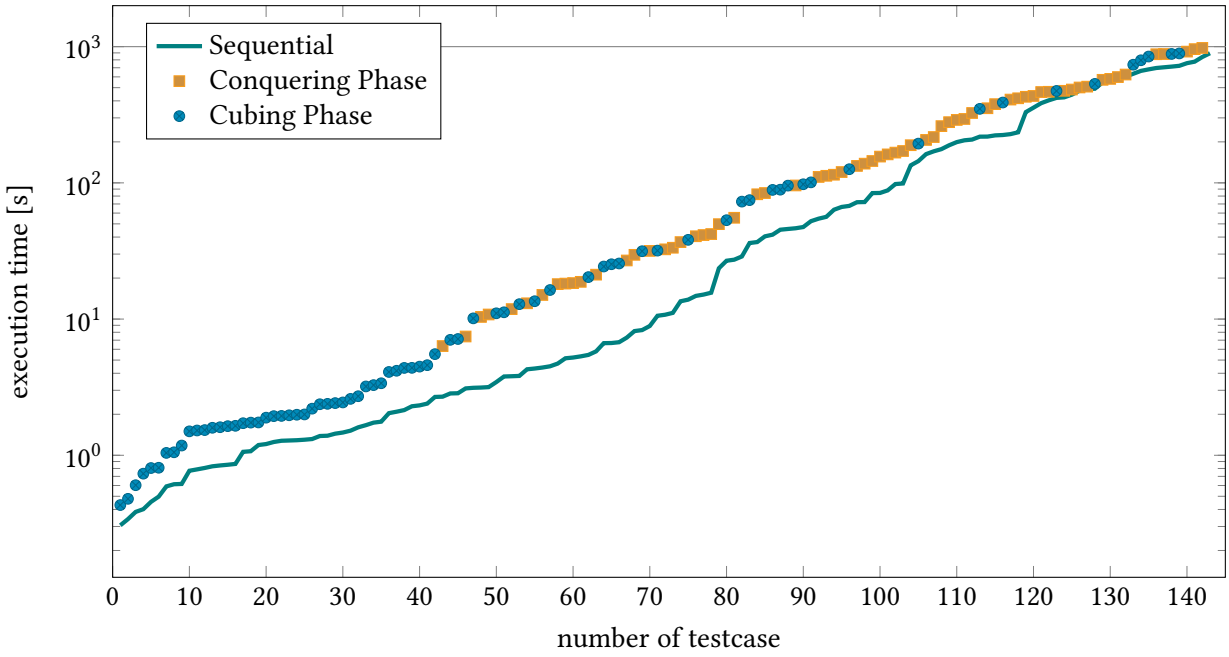

Figure 5.1: Comparing execution time between a sequential and parallel approach with sorted execution times

		open		closed	
cube finder	time slice	40 c	400 c	40 c	400 c
BFS	10 ms	140	123	132	124
	100 ms	139	123	129	122
heuristic best first	10 ms	137	137	142	138
	100 ms	136	135	139	142

Table 5.3: Comparing number of solved problems between open and closed nodes

then a cube find algorithm using closed nodes can explore up to β more nodes than the same algorithm using open nodes. This will result in higher heuristic values for the cubes and more solved problems. As a side effect, most algorithms become easier to implement if closed nodes are used for cubes.

Cut-Off Search This paragraph evaluates the performance of the cut-off cube finder. All tests use time slices of 10 ms and the cube finder was tasked with finding 400 cubes. We recall that in Section 4.4 two new parameters were introduced, Φ and Ψ. Their values are listed in Table 5.4 together with the number of solved problems for this parameter combination.

Overall, this approach performs slightly worse than a normal heuristic best first cube finding phase. Decreasing the parameters yields slightly better results but it does not overtake the heuristic best first approach. The behaviour of the cut-off search can be very chaotic and strongly dependent on the structure of the graph, so this result is neither expected nor astonishing for us. The cut-off cube finder needs a method to estimate the distance between two states. Since normal heuristics only approximate the distance between a state σ and a goal G and not two states σ_1, σ_2, we slightly adjusted our heuristic: The second state σ_2 gets converted into a goal with the exact same atoms $G' = \{\sigma_2\}$. It is possible that a heuristic specifically designed for estimating the distance between two states yield better results.

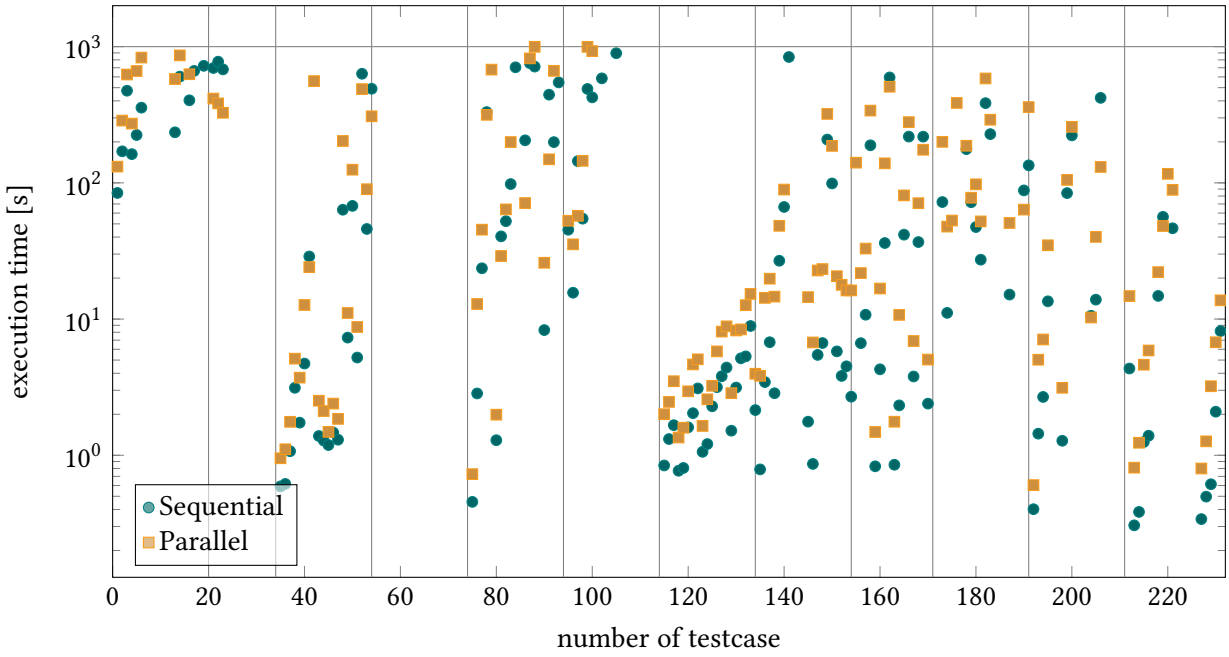

Figure 5.2: Comparing execution time between a sequential and parallel approach with execution times divided by domains

		Distance Factor Φ	
		0.125	0.25
Number of Anchors Ψ	4	140	139
	8	140	135

Table 5.4: Comparing number of solved problems for the cut-off cube finder with different parameters

Random Search Section 4.4 introduced two different approaches to randomize the cube finding phase. Table 5.5 compares these two. It lists the number of solved problems of the C&C solver and either uses a random depth first search or a random best first search. The number of descents indicate how many random searches are started in the cube finding phase. All combinations use a greedy bandit scheduler with a time slice of 10 or 100 ms.

The random best first cube finder solved noticeably more problems than the random depth first approach. Overall, the random depth first cube finder performed even worse than a BFS search strategy. One explanation for this is that the random depth first is more likely to pick a bad node and has trouble to recover after this. Because of the way the probabilities for choosing a node are scaled, the random best first search never chooses nodes with significantly higher heuristic values than other nodes. This is not the case for the random depth first search, since even nodes with very high heuristic values can be picked. Additionally, if the random depth first search chooses to explore a part of the state space graph with high heuristic values, it has a hard time escaping this part. It only considers immediate children of the currently explored node and cannot backtrack unless it reaches a dead end.

descents	time slice	depth first 40 c	depth first 400 c	best first 40 c	best first 400 c
1	10 ms	130	126	148	144
	100 ms	131	123	148	144
5	10 ms	126	117	146	135
	100 ms	127	119	144	134
25	10 ms	122	111	146	132
	100 ms	123	108	144	132

Table 5.5: Comparing number of solved problems for different random searches and different number of descents

	cube finder	
cube Solver	deterministic	random
deterministic	142	144
random	103	108

Table 5.6: Comparing number of solved problems between random and deterministic cube finding and solving

Both approaches suffer under an increase in the number of descents, especially in the cases where the finders searched for 400 cubes. This corresponds to our intuition that a cube finder should search deep into the state space graph in order to generate easy to solve cubes with low heuristic values. Overall, we can say that the C&C approach definitely benefits from a randomized cubing phase. We believe that trying out other randomization techniques has the potential to achieve further increase in performance.

Based on the success of randomizing the cube finding phase, we randomize the conquering phase too. Table 5.6 shows the result of this attempt. It compares deterministic and random cubing and conquering phases against each other. All tests used 100 ms time slices and searched for 400 cubes. We use a random best first search for both solver and finder. Unfortunately, randomizing the conquering phase does not yield the same results. We can see that random cube solver performs significantly worse than a deterministic one.

Sharing finished Nodes As a compromise between communication overhead and search space overlap between threads, we introduced the sharing of finished nodes in Section 4.7. Table 5.7 compares the number of solved problems for various cube finders with or without the sharing of finished states. All cube finders use 400 cubes. Solvers that share finished nodes solve more problems but this is hardly noticeable. It is plausible that the structure of the state space graph is too complex and the amount of shared nodes is too small to have a significant impact on the search space that is explored by the solvers. We do not believe that this attempt has the potential to increase the performance of our C&C approach significantly.

	closed heuristic	open heuristic	closed BFS	open BFS	5 random decents	25 random descents
without sharing	142	137	124	123	133	132
with sharing	144	137	128	126	136	134

Table 5.7: Comparing number of solved problems when sharing finished nodes

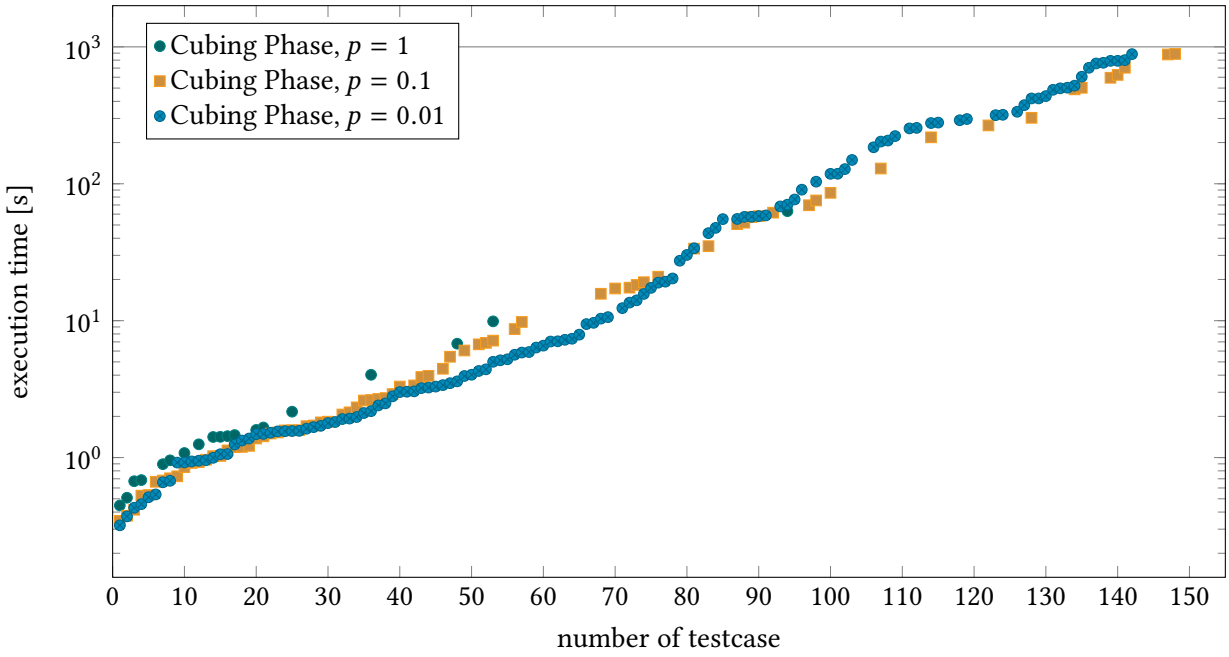

Figure 5.3: Comparing execution time for the random sparsening technique and different values for p

Sparsening of Cubes We introduced two sparsening techniqeus in Section 4.7 in order to fix the dilemma between choosing a small set of cubes while at the same time proving a diverse set of cubes with good heuristic values for the solvers. This paragraph compares the performance of these two approaches. Table 5.8 summarises the number of solved problems for deterministic and random sparsening and different values for the degrees of sparsening p. All tests use a greedy bandit scheduler with a time slice of 100 ms. Overall the deterministic and random approach show a similar performance. The most problems are solved for $p = 0.1$ and searching for 40 cubes. Both sparsening techniques outperform the best first search without it. This shows that sparsening a reasonable approach to fix our dilemma.

Figure 5.3 illustrates an effect that occurs if p is chosen small. It compares the cubing phases of the random sparsening technique for different values for p. Only instances that were solved during the cubing phase are plotted. We see that with a decreasing p the number of solved problems during the conquering phase increases. In fact, for $p = 1, 0.1$ and 0.01 the C&C solver finished 18, 76 and 105 problems out of 171 already in the cubing phase. So the degree of parallelism is drastically decreased if p is too small.

	deterministic		random	
percentage p	40 c	400 c	40 c	400 c
100%	143	146	147	147
10%	150	147	149	146
1%	144	142	143	142

Table 5.8: Comparing number of solved problems for different sparse cube finders

Together with the random best first cube finder these are our best results. It is obvious to combine both. The resulting algorithm uses a random best first search to find 40 cubes and combines it with the random sparsening technique. For $p = 1, 0.1$ and 0.01 this approach solves 148, 148 and 149 problems of the small test set in 1000 seconds. Since this is our most promising candidate we will use it to evaluate the scaling potential of our approach.

5.4 Scaling

In order to evaluate the scalability of our approach we decided to compare a sequential solver to two different configurations of our C&C solver. We run both configurations with a timeout of 1000 seconds on different amount of threads and compare the speedups to the sequential solver. The first configuration is very basic. It uses a heuristic best first search to find 400 cubes and the greedy bandit scheduler with a time slice of 10 ms. The second configuration represents our best approach based on preliminary experiments. It also uses the greedy best first scheduler with a time slice of 10 ms. But for the cubing phase we choose the random best first cube finder searching to find 40 cubes. This is combined with the random sparsening technique and parameter $p = 0.1$.

For this comparison we use the large test set of optimization domains containing 231 test cases in total. After 1000 seconds the sequential solver solves 143 of these.

First Configuration Table 5.9 lists the number of solved test cases and speedups for the first configuration. The second column shows the number of test cases the parallel planner solved after 1000 seconds. The third column shows the number of test cases both, the sequential and parallel planner solved after 1000 seconds. We only calculated the speedup for test cases that both solvers solved before the time-out. Columns four to six contain the average, total and median speedup on these test cases. If L is the list of all speedups then the average speedup is defined as the average of all values in L and the median speedup as the median value in L. The total speedup is defined as the ratio between the sum of all execution times of the sequential solver and the sum of all execution times of the parallel one. The average speedup is calculated as the average of the the speedups The last three columns also show these speedups but only consider hard test cases. We consider a test case hard if the sequential solver only solves it after $T \cdot 10$ seconds, where T is the number of threads. For example, if we use 16 threads, the speedups for the last three columns were calculated only on the test cases that were solved by the sequential approach in between 160 and 1000 seconds and by the parallel solver running on 16 threads before 1000 seconds. This is known as weak scaling. It allows us to take into account that higher amount of computation resources are generally used for bigger problem sizes.

We see speedups of less than one if we use up to 4 threads. Even the solver using 32 threads only achieves an average speedup of 1.17 on all test cases. Considering the hard test cases, we get similar results. Only the average speedup for 8, 16 and 32 threads increases noticeably for the hard test set. This speedup is insignificant but we have to keep in mind that the parallel solver solved up to 19 more problems than the sequential. The problems only solved by the parallel (or sequential) solver are not considered when calculating the speedup. Using hard test cases to calculate the speedup compensates this issue a little bit. However, this only ignores the easiest problems both solvers solved and cannot include the problems only the parallel solver solved.

Finally, we see that our C&C approach has difficulties to scale with higher amounts of threads. The number of solved problems improves for up to eight threads, but after that the increase in speedup and number of solved problems decreases.

Threads	Parallel Solved	Both Solved	speedup all tests Avg.	Tot.	Med.	speedup hard tests Avg.	Tot.	Med.
1	138	131	0.790	0.790	0.912	0.794	0.722	0.887
2	150	139	0.871	0.883	0.937	0.940	0.891	0.931
4	152	142	0.943	0.980	0.962	1.099	0.991	0.975
8	159	142	1.023	1.114	0.965	1.378	1.140	1.019
16	162	142	1.158	1.250	1.026	1.691	1.276	1.178
32	161	142	1.165	1.180	1.026	1.870	1.270	1.174

Table 5.9: Comparing speedup for different numbers of threads, using a heuristic best first search with a greedy bandit scheduler

Figure 5.4 plots the sorted speedups for each test cases. Figure 5.6 plots the same speedups but only considers the hard test cases for the respective amount of threads. We added two vertical lines to indicate a speedup of one and two.

Second Configuration Table 5.10 shows the speedups for the second configuration. First, we notice an improvement in the number of solved problems. This is not surprising due to our experiments in Section 5.3. The parallel approach manages to solve up to 23 more problems than the sequential. Additionally we get significantly higher speedups. On all test cases the parallel approach with 8 threads achieves an average speedup of 1.86 and on the hard test cases of 3.46.

However, the problem that our approach does not scale well with higher number of threads remains. When using more than eight threads we only get diminishing increases of speedup and number of solved problems.

We want to address that the average speedup using only one thread on all test cases is higher than one. The reason for this is that the calculation of the average speedup favours algorithms with high variance in execution time. This is the case for our random approach. If the parallel solver solves the problems half of the time twice as fast as the sequential one and the other half twice as slow, the average speedup will be 1.25.

Figure 5.5 plots the sorted speedups of the second configuration running on all test cases and Figure 5.6 only plots the points on hard test cases (for the respective number of threads). We added three vertical lines to indicate a speedup of two, four and eight. Considering all test cases, we notice that the lowest 70 to 90 speedups stay below one and only the upper quarter is bigger than 2. If we look at the hard test cases we see more promising results. The increase in performance is more noticeable and most points represent a speedup greater than one.

Figure 5.8 shows the execution time of the first and second configuration running on 8 an 32 threads on a cactus plot. The second configuration with 32 threads has the overall best performance. But the configurations converge against it with an increase in execution time. All approaches outperform the sequential solution after the first 80 test cases. Figure 5.9 plots the execution time for configuration one and two running on 32 cores and divides the plot into the respective domains. We see that the parallel approach has better execution times especially on the domains barman, hiking, tetris and tidybot whereas the sequential solver is faster on solving the termes domain.

Threads	Parallel Solved	Both Solved	speedup all tests Avg.	Tot.	Med.	speedup hard tests Avg.	Tot.	Med.
1	145	133	1.354	1.102	0.844	1.939	1.112	1.329
2	153	135	1.369	1.099	0.825	2.101	1.123	1.380
4	158	136	1.430	1.240	0.857	2.319	1.276	1.227
8	164	139	1.864	1.329	1.032	3.460	1.401	1.691
16	166	139	1.911	1.360	1.043	3.960	1.449	2.181
32	161	138	2.052	1.539	1.097	3.717	1.705	2.417

Table 5.10: Comparing speedup for different numbers of threads, using a random best first search with a greedy bandit scheduler

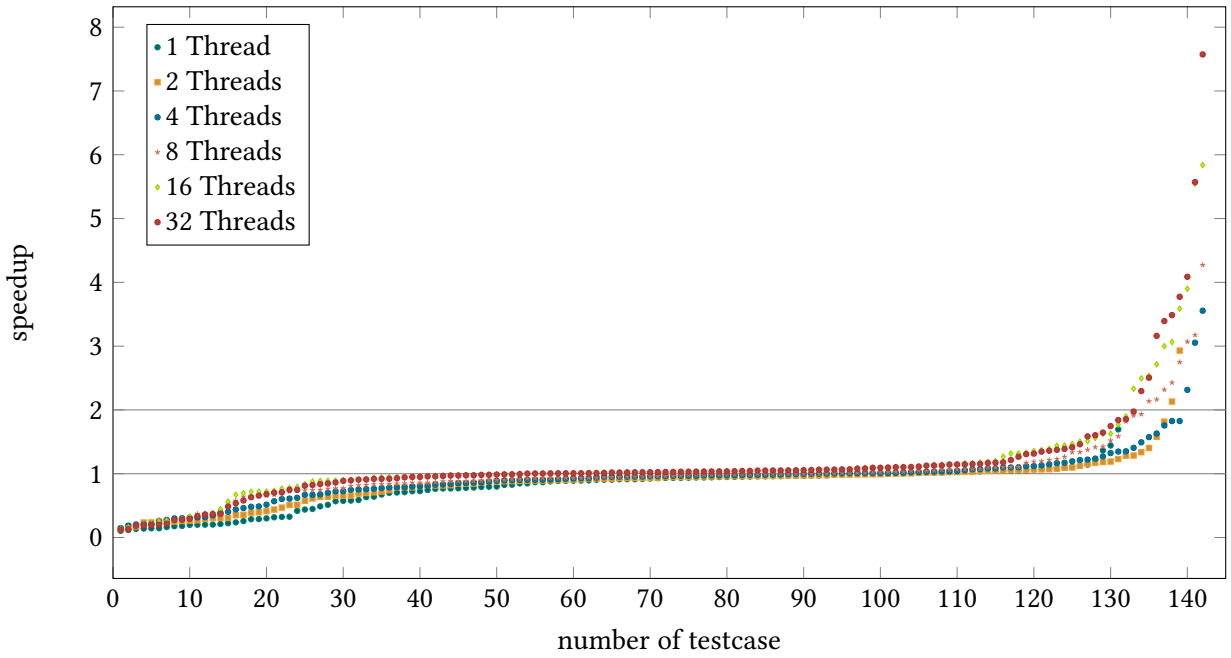

Figure 5.4: Comparing speedup for different numbers of threads, using a heuristic best first search with a greedy bandit scheduler

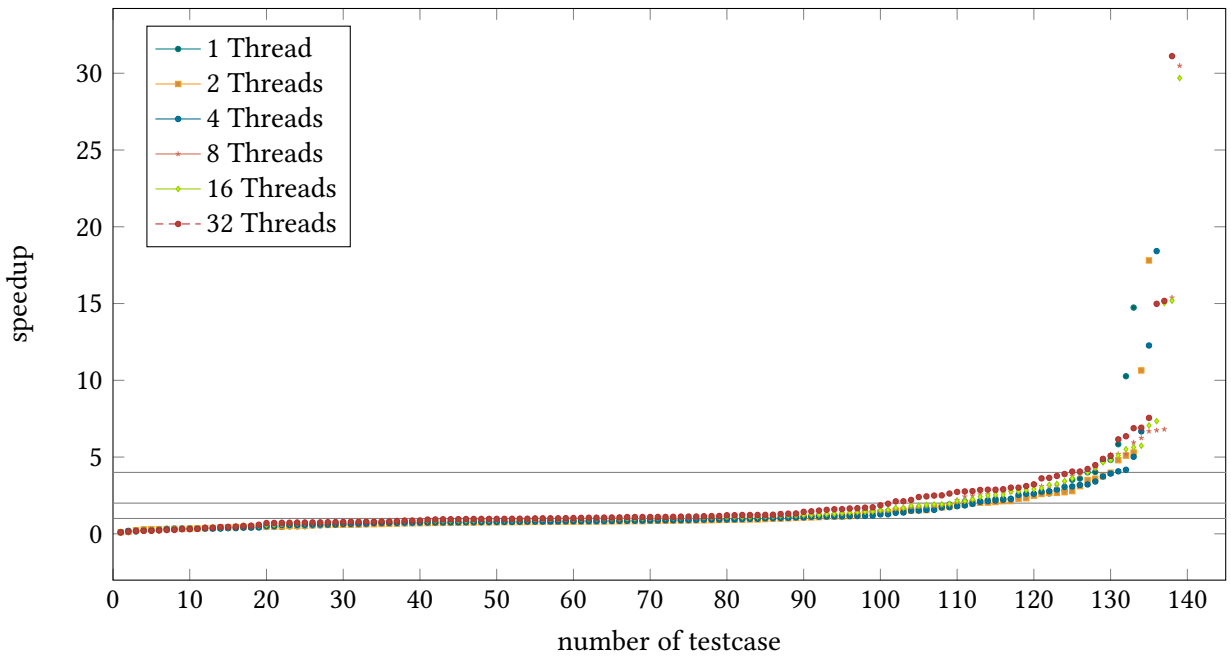

Figure 5.5: Comparing speedup for different numbers of threads, using a random best first search with a greedy bandit scheduler

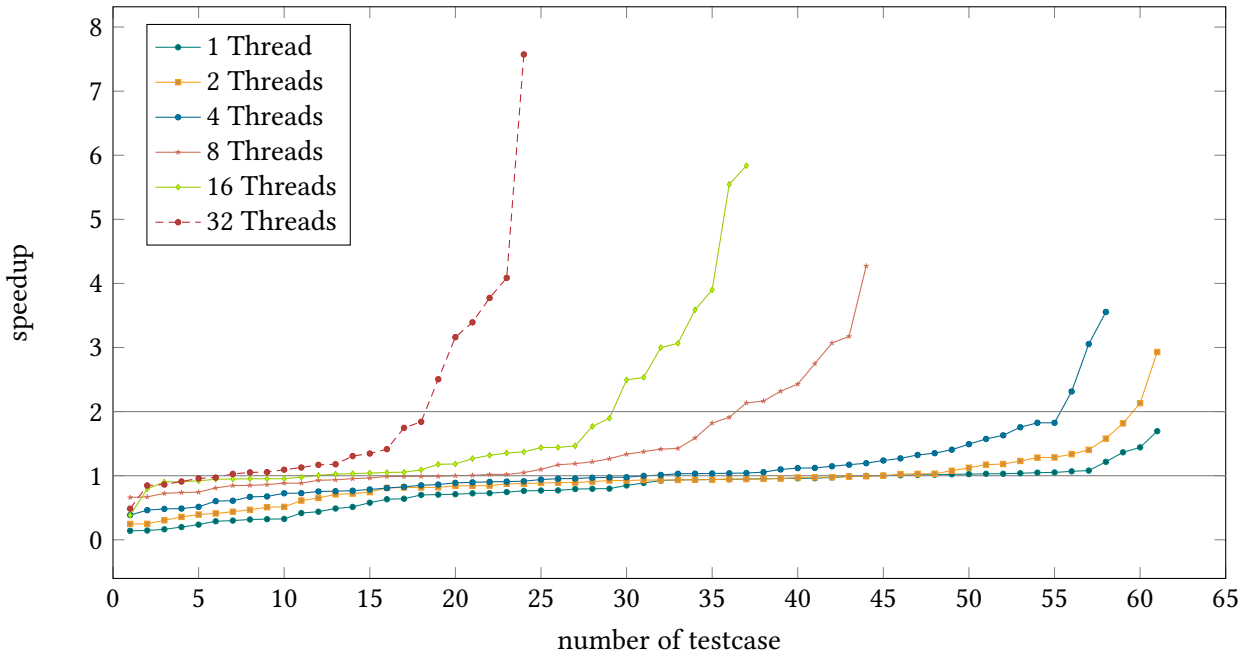

Figure 5.6: Comparing speedup on hard test cases for different numbers of threads, using a heuristic best first search with a greedy bandit scheduler

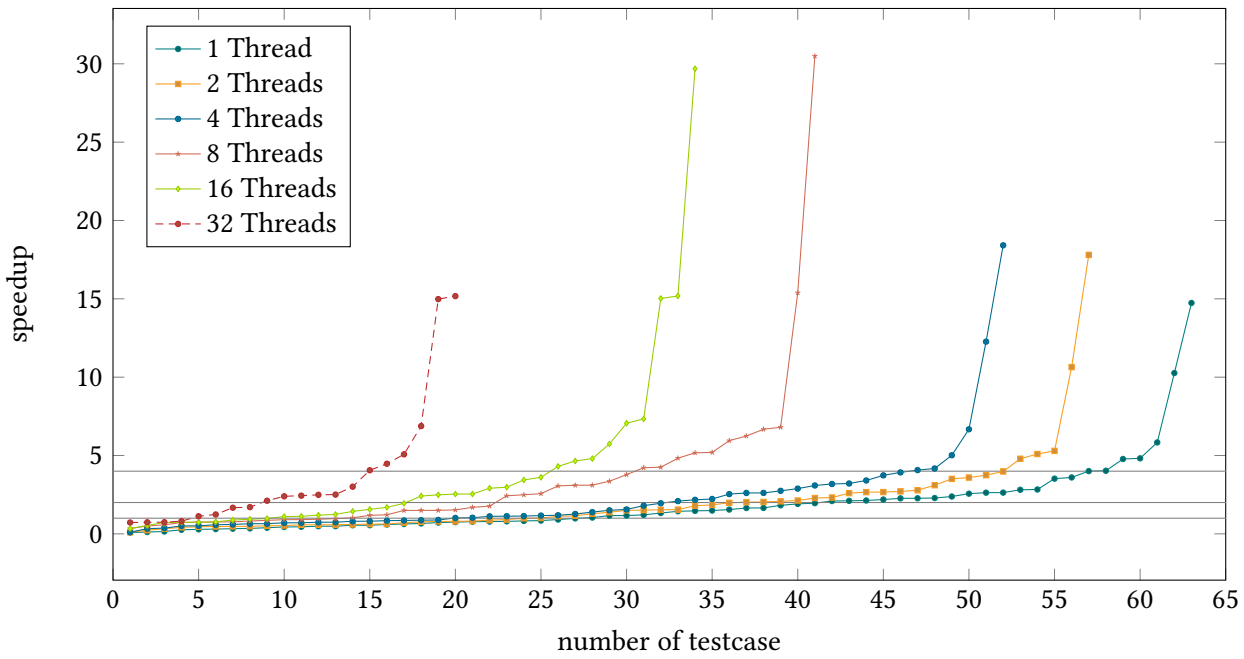

Figure 5.7: Comparing speedup on hard test cases for different numbers of threads, using a random best first search with a greedy bandit scheduler

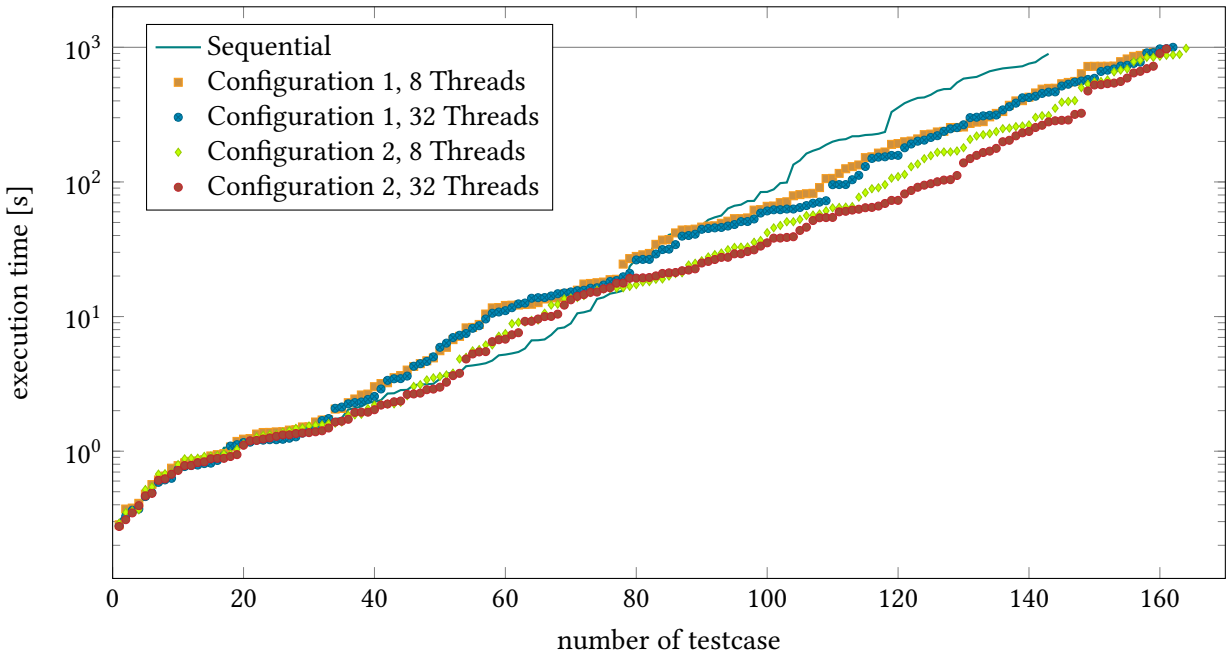

Figure 5.8: Comparing execution time for configuration 1 and 2 and different numbers of threads, using a heuristic best first search with a greedy bandit scheduler

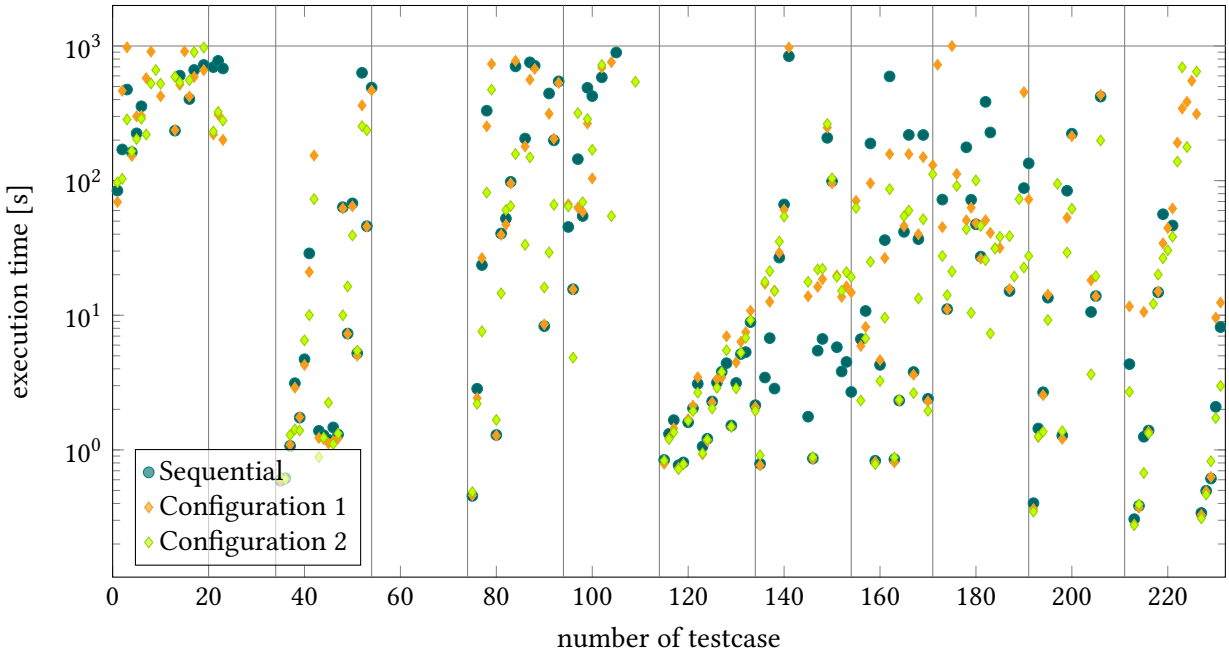

Figure 5.9: Comparing execution of configuration 1 and 2 with 32 threads against a sequential solver with execution times divided by domains

6 Conclusion

In this thesis we applied the concept of Cube and Conquer used in SAT solving, to automated planning.

We showed that the translation of C&C SAT solving to planning faces some difficult challenges, most importantly the partitioning of the search space and the identification of the most promising cubes. These emerge from the difference in structure of the problems but also from varying requirements to the solution. While we were able to find reasonable solutions to the latter, we were not able to fully resolve the first challenge.

We identified characteristics that are necessary to solve the problem of overlapping search spaces and use the computation resources efficiently. This means finding a set of cubes that has good heuristic values while simultaneously covering big portions of the state space graph. These requirements are hard to fulfil due to the complex nature of the planning problem. But we showed that by introducing randomness at different points in our algorithm, the performance can be significantly increased compared to a more basic approach.

For the second challenge we were able to fall back to existing methods that solve the exploration/-exploitation dilemma. By adjusting these methods to fit our needs more precisely, we came up with a robust scheduler that is able to identify the most promising cubes while not overcommitting computation resources into bad decisions.

Combining our most sophisticated approaches resulted in a parallel C&C planner that outperforms the sequential one on hard test cases. While the sequential solver is faster in solving small problems, we achieved an average speed-up of 3.5 on hard test cases using eight cores. In addition, the parallel approach was able to solve up to 15% more problems. We showed that the parallel approach scales well for up to eight threads but yields diminishing returns for higher number of threads.

Although these results are not able to compare with state of the art parallel planners from the international planning competitions [Pom18], our approach leafs much room for improvement. We are sure that the performance of our algorithm can be increased further, especially by modifying the cubing phase. But also other parts have the potential to be optimized. With more research, Cube and Conquer might become a feasible approach for parallel planning, next to algorithm portfolios and distributed search algorithms.

6.1 Future Work

We were not able to provide a satisfying solution to all the challenges we faced, and even some of our successful solutions can still be optimized. This section lists possibilities for further improvements of the performance of Cube and Conquer for parallel planning.

Re-evaluation with more sophisticated planning technologies We only implemented and tested our algorithms in the Aquaplanning framework. This allowed us to experiment with a number of different approaches but restricted our use of more sophisticated planning technologies. We were not able to exploit powerful modern heuristics such as patter databases to find higher quality cubes [CS98]. Because of this limitations we missed out on using global heuristics in the cubing phase and local heuristics

in the conquering phase as it is the case for C&C SAT solving. Another worthwhile addition would be planners that can efficiently determine if a planning problem is unsatisfiable, since this reduces the amount of cubes that the scheduler has to handle.

Diversifying the Cubes We belief that improving the diversity of the cubes yields a great potential for increasing the performance of our algorithm. But it also seems to be the hardest challenge. Introducing randomness to the cubing phase is an successful attempt but has room for improvement. Building upon the cut-off cube finder could result in a stronger notion of diversity. Using a more sophisticated approach to calculate the distance between two states might yield better results. Another possibility would be to introduce randomness to the cut-off finder.

Scheduler The greedy bandit scheduler was shown to be successful in focusing its computation time on promising cubes. But it made the assumption that the heuristic values of cubes decrease linearly with the investment of computation time and calculated the reward values based on that. This could lead to overcommitting to a single cube and discarding other cubes too fast. Not assuming linear but rather an exponential or polynomial scaling of the heuristic values might solve this issue. It can also be useful to weight the heuristic value of the scheduler in such a way that the most recent values have the biggest impact on the reward value. This way the scheduler could detect whether a solver runs into a dead end much faster.

Dynamic method of determining the number of cubes In the cubing phase our algorithm searches for a fixed amount of cubes. The C&C SAT approach however introduces a cut-off heuristic that could dynamically determine when a branch in the search tree would be cut off and added to the cubes. Implementing a cut-off heuristic for C&C with automated planning would allow us to adjust the length of the cubing phase and the amount of found cube more precisely. Additionally, an informed cut-off heuristic could improve the diversity of cubes by identifying important edges in the state space graph and cutting the vertices after these off as cubes.

Sharing Information Between Solvers We introduced the sharing of finished nodes between the cube finder and solver to decrease the overlap of search space. But this approach did not scale well since the amount of shared information was too little. If we do not share information between threads, we can try to share information in between the solvers of a thread. By letting the solvers share the set of finished nodes, we can prevent the search spaces form solvers of the same thread to collide.

Bibliography

[ACF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multiarmed bandit problem”. In: *Machine learning* Volume 47.2-3 (2002), pp. 235–256.

[AS09] Gilles Audemard and Laurent Simon. “Predicting learnt clauses quality in modern SAT solvers”. In: *Twenty-first International Joint Conference on Artificial Intelligence*. 2009.

[Bie12] Armin Biere. “Lingeling and friends entering the SAT Challenge 2012”. In: (2012).

[Byl94] Tom Bylander. “The computational complexity of propositional STRIPS planning”. In: *Artificial Intelligence* Volume 69.1-2 (1994), pp. 165–204.

[CS98] Joseph C Culberson and Jonathan Schaeffer. “Pattern databases”. In: *Computational Intelligence* Volume 14.3 (1998), pp. 318–334.

[EB05] Niklas Eén and Armin Biere. “Effective preprocessing in SAT through variable and clause elimination”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2005, pp. 61–75.

[EHN94] Kutluhan Erol, James Hendler, and Dana S Nau. “HTN planning: Complexity and expressivity”. In: *AAAI*. Vol. 94. 1994, pp. 1123–1128.

[GSV09] Alfonso Gerevini, Alessandro Saetti, and Mauro Vallati. “An automatically configurable portfolio-based planner with macro-actions: PbP”. In: *Nineteenth International Conference on Automated Planning and Scheduling*. 2009.

[Hel06] Malte Helmert. “The fast downward planning system”. In: *Journal of Artificial Intelligence Research* Volume 26 (2006), pp. 191–246.

[HKM16] Marijn JH Heule, Oliver Kullmann, and Victor W Marek. “Solving and verifying the boolean pythagorean triples problem via cube-and-conquer”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2016, pp. 228–245.

[HKWB11] Marijn JH Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. “Cube and conquer: Guiding CDCL SAT solvers by lookaheads”. In: *Haifa Verification Conference*. Springer. 2011, pp. 50–65.

[HN01] Jörg Hoffmann and Bernhard Nebel. “The FF planning system: Fast plan generation through heuristic search”. In: *Journal of Artificial Intelligence Research* Volume 14 (2001), pp. 253–302.

[HRK] Malte Helmert, Gabriele Röger, and Erez Karpas. “Fast downward stone soup: A baseline for building planner portfolios”. In: pp. 28–35.

[KFB09] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. “Scalable, parallel best-first search for optimal sequential planning”. In: *Nineteenth International Conference on Automated Planning and Scheduling*. 2009.

[Pom18] Balyo Pommerening Torralba. “International Planning Competition”. 2018. URL: *https://ipc2018-classical.bitbucket.io* (visited on 10/30/2019).

[San97] Peter Sanders. “Lastverteilungsalgorithmen für parallele Tiefensuche”. German. PhD thesis. 1997. 154 pp. ISBN: 3-18-346310-5. DOI: *10.5445/IR/997*.

[Sut05] Herb Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobb's journal* Volume 30.3 (2005), pp. 202–210.

[Tan+11] Ole Tange et al. “Gnu parallel-the command-line power tool”. In: *The USENIX Magazine* Volume 36.1 (2011), pp. 42–47.

[Val+12] Richard Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer, and Nathan Sturtevant. “Arvandherd: Parallel planning with a portfolio”. In: *Proceedings of the 20th European Conference on Artificial Intelligence*. IOS Press. 2012, pp. 786–791.

[Val12] Mauro Vallati. “A guide to portfolio-based planning”. In: *International Workshop on Multidisciplinary Trends in Artificial Intelligence*. Springer. 2012, pp. 57–68.