Solving Constraint Satisfaction Problems in a Distributed Setting

Master thesis

by

Wouter Menninga
s2315556

Supervisor
Prof. Dr. Alexander Lazovik

Second supervisor
Drs. Michel Medema

Contents

Glossary

BTD Backtracking with Tree-Decomposition.

CELAR Centre d'Electronique de l'Armement.

COP Constraint Optimization Problem.

CSP Constraint Satisfaction Problem.

DCOP Distributed Constraint Optimization Problem.

DCSP Distributed Constraint Satisfaction Problem.

EPS Embarrassingly Parallel Search.

H-TD-WT Heuristic Tree Decomposition Without Triangulation.

JVM Java Virtual Machine.

MAC Maintaining Arc Consistency.

RLFAP Radio Link Frequency Assignment Problem.

SAT Boolean satisfiability problem: the problem of determining whether a given boolean formula can be satisfied.

Abstract

Solving constraint satisfaction problems (CSPs) is an NP-complete problem. Many algorithms exist for solving CSPs, some of which make use of parallelization. However, such algorithms are restricted by the amount of resources that can be available on a single machine. Therefore, this thesis implements a distributed approach for solving a CSP, using the actor model.

In our work, an adapted, distributed version of the BTD algorithm was implemented using the actor model. Experiments using the implemented algorithm show that the solve time for a problem is reduced logarithmically when increasing the number of cores, while it grows linearly when increasing the number of nodes. The original CSP is decomposed into subproblems by means of tree decomposition. Different strategies for deploying these subproblems among the cluster nodes are explored. We found that the deployment strategy has a large impact on the solve time. The best strategy is dependent on the structural properties of the problem being solved. Moreover, we found that taking into account the separator size during the deployment, by deploying nodes in the tree decomposition that share a large separator to the same machine, has a considerable beneficial impact on the solve time of up to 24%. Finally, we show that the method of decomposition has a high impact on the achieved scalability.

Chapter 1

Introduction

Constraint Satisfaction Problems (CSPs) are a class of problems that can be used to model a variety of different problems from a range of differing domains. A CSP consists of a number of variables, each having its own domain of values that can be assigned to it, and constraints, which limit the values that can be assigned to combinations of variables. Solving a CSP consists of finding an assignment of values for the variables, such that all of the constraints are satisfied.

A variation on constraint satisfaction problems are Constraint Optimization Problems (COPs). Optimization problems are similar to satisfaction problems, but introduce a function which assigns a value to a solution. In the optimization variant, the goal is to find an optimal solution, i.e. a solution that either minimizes or maximizes the value assigned by this function.

Solving generic CSPs (and COPs) is an NP-complete problem, which means that the solve time increases exponentially with respect to the input size. There has been a lot of study towards discovering algorithms that increase the performance of solving CSPs. For example, some of this research exploits structural properties of the constraint problem [1], such as the BTD algorithm, introduced by Jégou et al. [2].

Most of the presented algorithms and techniques focus on local performance improvements, and are limited to the resources of a single machine. A means of overcoming this restriction is by using a distributed cluster of multiple machines that work on solving the same problem. Using this distributed approach, the problem is divided over multiple machines and removes the limitation of the resources of a single machine, making it possible to find a solution faster. Some research has also focused on using such a distributed cluster of machines for solving constraint problems, such as by using a work-splitting approach [3]. A less researched area is that of the deployment of the (sub-)problems over the cluster of machines and the effects of the

deployment on the performance of the distributed search.

This thesis aims to add to the existing research, by building and evaluating a solver that combines techniques for distributed solving with the BTD algorithms structural performance improvements and the opportunities that algorithm offers for parallelization. Moreover, it explores the effects of different deployment strategies on the performance of the search process.

The main research question is formulated as:

> *How can a constraint satisfaction problem be solved using a distributed cluster of machines?*

Additionally the following two sub-questions are asked:

> *Which strategy for dividing the constraint satisfaction problem over a distributed cluster of machines minimizes the search time?*
>
> *Is the optimal deployment strategy different for different categories of constraint satisfaction problems?*

This research builds on the Master thesis of R. Kip [4], in which a distributed solver for a specific constraint optimization problem (namely a version of graph coloring) was developed, which was capable of solving such problems when derived from a generated tree decomposition. In the accompanying experiments, different configurations of tree decompositions were tested, in order to asses the performance and scalability behavior of the algorithm. In Section 3.6, a more comprehensive summary of this work is described.

This thesis expands on the aforementioned Master thesis with a more general implementation, which is capable of solving a large range of more general satisfaction and optimization problems. For the experiment, the focus lies more on how this algorithm performs and scales for problems more akin to the problems encountered in the real world and benchmarks, allowing for a more meaningful comparison with other algorithms and methodologies for solving constraint problems.

This thesis is structured as follows. First, a background is provided about CSPs, COPs and related concepts. Then, an overview of the relevant related work is provided. Next, the concepts and design of the distributed algorithm are introduced, as well as the different decomposition methods and deployment strategies. Afterwards, the different types of problems used in the experiments are introduced, and, for each of the experiments, the methodology followed by the presentation and analysis of the results. Finally, we present the conclusion and list the potential for future work.

Chapter 2

Background

In this chapter, the background knowledge is provided for the context of this thesis. An introduction to constraint satisfaction and optimization problems is provided alongside a formal description of those. Moreover, background information regarding decomposition methods and the actor model is given.

First, constraint satisfaction problems are introduced and a formal description is provided. After this, the same is done for constraint optimization problems. Some basics regarding constraint solving are then provided, after which the concepts of the constraint graph and decompositions are introduced. Finally, some information about the actor model is provided as well.

2.1 Constraint Satisfaction Problems

Constraint satisfaction problems are a generic class of problems that can be used to model a variety of different problems from widely differing domains.

Formally, a constraint satisfaction problem can be defined as a triplet $\mathcal{P} = \langle X, D, C \rangle$, where $X = \{x_1, x_2, ... \ x_n\}$ a finite set of variables, $D = \{D_1, D_2, ... \ D_n\}$ a finite set of domains (where D_i contains all values in the domain of variable x_i), and $C = \{C_1, C_2, ... \ C_m\}$, a finite set of constraints. Each constraint C_i is a predicate, $C_i(x_{i_1}, ... \ x_{i_{j_i}})$, taking j_i number of arguments (one for each variable involved in that constraint), and returns *true* if and only if the combination of the assignments of the variables involved in that constraint satisfy that constraint. More formally, a constraint is a tuple $C_i = \langle V_i, R_i \rangle$, where $V_i = \{x_{i_1}, x_{i_2}, ..., x_{i_{j_i}}\}$, is the set of variables involved in that constraint, and R_i is a relationship containing a subset of the Cartesian product of the domains of the variables involved in that constraint: $R_i \subseteq D_{i_1} \times ... \times D_{i_{j_i}}$. This relation represents the combinations of values for the involved variables for which

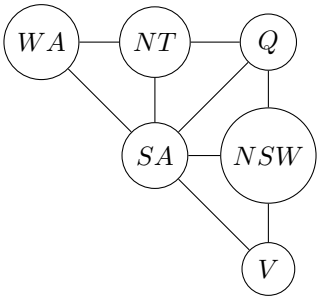

Figure 2.1: A planar graph representing the states of Australia (except Tasmania).

the constraint is *true*.

The problem of solving $\mathcal{P}$ can then be described as finding a set of values from the domains (one for each variable), such that all constraints are satisfied. Formally, this means finding a function $f : X \to \bigcup_{i=1}^{n} D_i$, such that $\forall i, 1 \leq i \leq m$, $\langle f(x_{i_1}), f(x_{i_2}), ..., f(x_{i_j}) \rangle \in R_i$ [2].

Example 1. An uncomplicated example of a problem that can be described as a CSP is that of *cryptharithmetic puzzles*. An example of such a puzzle can be seen in (2.1). The goal of this puzzle is to assign a digit to all the letters in the equation, such that the equation is correct.

In this case, some extra variables are introduced to represent the carry. The variables are then: $S, E, N, D, M, O, R, Y, X_1, X_2, X_3, X_4$. Each of these variables having the domain $[0, 9]$, except for S and M, which have the domain $[1, 9]$.
The constraints are:

$$
\begin{aligned}
&C_1 : (D + E) = Y + 10 \cdot X_1, \\
&C_2 : (N + R + X_1) = E + 10 \cdot X_2, \\
&C_3 : (E + O + N + X_2) = N + 10 \cdot X_3, \\
&C_4 : (S + M + X_3) = O + 10 \cdot X_4, \\
&C_5 : M = X_4
\end{aligned}
$$

$$
\begin{array}{r}
SEND \\
+\quad MORE \\
\hline
MONEY
\end{array}
\tag{2.1}
$$

Example 2. Another example is the *map coloring* problem: given a planar graph G (representing a map with states), and a set of colors K, find an assignment of colors

for the vertices of the graph (i.e. the states), such that no two neighboring vertices have the same color. This problem can be formulated as a CSP in which, for each vertex in the graph, there exists a variable. Initially, the domain for all of the variables consists of all the colors (e.g. $\{red, green, blue\}$). Then, for each edge in the graph, there is a constraint which is satisfied if and only if both variables associated with the nodes at either end of the edge have a different value. Formally, the relationship which describes the possible values for the constraint is $R_i = \{\langle x, y\rangle | x \in D_j, y \in D_k, x \neq y\}$, or concretely $R_i = \{\langle red, green\rangle, \langle red, blue\rangle, \langle green, red\rangle, \langle green, blue\rangle, \langle blue, red\rangle, \langle blue, green\rangle\}$. For brevity, the symbol $\neq$ will be used to denote this relationship. In the case of the map coloring for the graph in Figure 2.1, the set of constraints C would look as follows: $C = \{\langle\{WA, NT\}, \neq\rangle, \langle\{WA, SA\}, \neq\rangle, \langle\{NT, Q\}, \neq\rangle, \langle\{NT, SA\}, \neq\rangle, \langle\{Q, SA\}, \neq\rangle, \langle\{Q, NSW\}, \neq\rangle, \langle\{NSW, SA\}, \neq\rangle, \langle\{NSW, V\}, \neq\rangle\}$.

2.2 Constraint Optimization Problems

Related to constraint satisfaction problems are *constraint optimization problems*. In these problems, the objective is not just to find a solution to the problem, but to find an optimal solution. In order to compare different solutions, an objective function is defined, which can be a cost function or energy function, depending on whether it should be minimized or maximized respectively.

Formally, an optimization problem is defined similar to a satisfaction problem, but adds an objective function, $\phi : D_1 \times ... \times D_n \to \mathbb{R}$. This function assigns an objective value to every assignment in the solution space. The optimal solution to a constraint optimization problem is the solution that either minimizes this score (in case of a cost function) or that maximizes it (in case of an energy function). Formally, given the solution space $S = D_1 \times ... \times D_n$, for the optimal solution s_{opt}, it holds that $\forall_{s \in S} : \phi(s_{opt}) \leq \phi(s)$ in the case of minimization, or $\forall_{s \in S} : \phi(s_{opt}) \geq \phi(s)$ in the case of maximization of the objective function.

In this thesis, we will from now on focus on the case of minimization of the objective function. However, the case of maximization is symmetrical, and all findings can be applied equally to the case of maximization.

2.3 Solving constraint problems

Finding an (optimal) solution to a constraint problem is NP-complete. A common method for solving constraint satisfaction and optimization problems is the use of *backtracking*. In backtracking, values are assigned to variables sequentially and after each assignment the constraints are checked. When one of the constraints fails, the algorithm will backtrack to the last assignment and assign a different value for that variable. Backtracking has a theoretical time complexity which is exponential, namely

$\mathcal{O}(m \cdot d^n)$, where m is the number of constraints, d is the largest domain size and n is the number of variables [5].

Another technique used, often alongside backtracking, is *forward-checking*. In forward-checking, after assigning a value to a variable and checking the constraints, the domains for all neighbouring variables (i.e. all other variables that share a constraint with the assigned variable) are reduced, such that they only contain legal values. If a domain for a variable becomes empty due to this step, then backtracking will occur.

Another widely used technique is Maintaining Arc Consistency (MAC). A variable X is said to be arc consistent with respect to a variable Y if for all values in the domain of X, there exists at least one legal value in the domain of Y. When using MAC, after assigning a value to a variable and performing forward-checking, the domains are made arc consistent. Similar to forward-checking, whenever a domain becomes empty, backtracking occurs. This technique detects failures earlier than forward-checking.

2.4 Constraint graphs

A method of representing a CSP graphically is by using a graph representation of the variables and constraints. This is done in a constraint graph. Multiple graphical models exist to represent the CSP as a graph, each dealing with higher-arity constraints in a different way. For this thesis, we use the *primal constraint graph*.

The primal constraint graph is defined as $G = \langle V, E \rangle$, where V is the set of vertices (the variables) and E the set of edges. The edges are defined as: $E = \{(x_i, x_j) | x_i, x_j \in V \wedge i < j \wedge (\exists c_k \in C : x_i \in Scope(c_k) \wedge x_j \in Scope(c_k))$, where $Scope(constraint)$ is a function that returns all variables that are involved in the passed constraint. In other words, there exists an edge between two variables, if both of those variables occur in the scope of the same constraint.

2.5 Decompositions

Solving a constraint satisfaction problem is known to be NP-hard. Much research has been done towards identifying *tractable classes* of CSPs. In general, the methods for obtaining these tractable classes can be separated in two groups [1], namely those which are tractable due to restrictions on the structure of the problem, and those which are tractable due to restrictions on the constraints. Different polynomially tractable classes of CSPs can be defined based on different structural properties of the constraint problem. Usually, these are graph-theoretic properties of the constraint graph (or constraint hypergraph in case of CSPs with constraints of an arity higher than 2).

It is known that CSPs with constraint (hyper)graphs which are acyclic, can be solved

in polynomial time [6]. The properties of the constraint graph that lead to polynomially tractable classes are all based on some generalization of this acyclicity [7]. Methods exist for converting an arbitrary constraint satisfaction problem into a binary and acyclic problem. Such methods are called *decomposition methods*. An overview and comparison of different CSP decomposition methods was compiled by Gottlob et al. in [7]. Each decomposition method specifies some concept of *width*, which can be seen as a measure of the cyclicity of the constraint graph. All CSPs for which the tree width of the primal graph is bounded by some constant k, are solvable in polynomial time [8, 9].

One such decomposition method is *tree decomposition* [10]. A tree decomposition is a representation of an original graph as a tree, in which every node (called *bags*) can be mapped to a set of variables from the original graph. The following three properties need to hold:

1. Every variable in the original graph needs to appear in one of the bags.
2. If two variables are connected by an edge in the original graph, then they need to appear together in at least one bag.
3. If a variable occurs in two bags in the tree, then every bag on the path between those needs to contain that variable.

Formally, a tree decomposition of a graph $G = \langle X, E \rangle$ is a pair $\langle T, B \rangle$, where T is a tree ($T = \langle I, F \rangle$) and $B = \{B_i \ : \ i \in I\}$ is a family of subsets (called *bags*) of X, mapping each node in the tree to a set of variables from the original graph. Each subset is a node in the tree T and satisfies the following three conditions:

1. $\bigcup_{i \in I} B_i = X$
2. for all edges $\langle x, y \rangle \in E$, there exists $i \in I$, for which $\langle x, y \rangle \subseteq B_i$.
3. for all $x, y, z \in I$, if z is in a path from x to y in tree T, then $B_x \cap B_y \subseteq B_z$.

The *width* of a tree decomposition is the size of the largest bag minus 1.

2.6 Actor Model

The actor model [11], [12], is a model of parallel computation. In this model, so-called *actors* are the primitive unit of concurrency. These actors communicate with one another by sending and receiving messages between them. Based on a message, an actor can make local decisions, send more messages, create additional actors and change its behavior for the next incoming message. An actor can only modify its own state, and therefore, the states of other actors can only be influenced by sending messages.

This higher level of abstraction relieves developers from the need to manage lower level concepts, such as threads and explicit locking. Moreover, the design of the actor model is location transparent, such that the distribution of an application consisting of actors is not fixed or hard-coded, but can be dynamic or configured at runtime.

Chapter 3

Related Work

A signification amount of study has gone into the field of constraint satisfaction and optimization. In this section, some of the work relevant to this thesis is discussed. The focus lies mainly on related work in which the distributed or parallelization aspect plays a significant role.

3.1 BTD

One of the approaches to solving constraint satisfaction problems leverages the structural properties of the decomposition. One such algorithm is Backtracking with Tree-Decomposition (BTD), introduced by Jégou et al. [2]. In BTD, a new way is introduced to provide bounds on the theoretical complexity of a CSP. Conceptually, in BTD, the method of backtracking search is guided on its choice of variables by an ordering on the variables based on the structure of the tree decomposition of the constraint graph.

The ordering on the variables, combined with the structural guarantees provided by the tree decomposition, allows for the use of so-called *"structural nogoods"*, which is the same as regular nogoods (i.e. an assignment of some of the variables which cannot be extended to a complete solution [13]). However, in this case, it can be derived from structural properties. These structural nogoods are used in order to prune parts of the search-tree for which it is known that no consistent solution exists.

Moreover, *"structural goods"* are defined, which are assignments of variables for which it is known that they can be extended to a solution for an identified remaining part of the problem. Again, these are detected using structural properties of the problem. The structural goods are used to skip parts of the search-tree (also called *forward-jumping*), whenever it is known that a consistent solution exists.

By analysis, it is shown that the theoretical time complexity is $\mathcal{O}(n.s^2.m.\log(d^s).d^{w+1})$, where n is the number of variables, m is the number of constraints, d is the maximum domain size, s is the size of the largest separator and w is the tree width. The space complexity is shown to be $\mathcal{O}(n.s.d^s)$, where s is the maximum size of minimal separators of the tree decomposition.

After obtaining a tree decomposition, BTD explores the search space using a *compatible order* based on this tree decomposition. In other words, each bag in the tree decomposition is visited in a way compatible with this order. Within a bag, the enumeration of the variables in the search space is not restricted by this order.

When a consistent instantiation is found for all the variables in a bag (B_i), the search goes on with the variables of the first son of this bag (B_{i+1}), if such a son exists. At that moment, the algorithm checks if the assignment for the variables in the separator of those two bags (i.e. variables which exist in both of the bags, $\mathcal{A}[B_i \cap B_{i+1}]$), has been recorded as a good or a nogood. In case of a nogood, the search continues for a different instantiation of B_i. In case of a good, the enumeration skips a part of the search space by means of "forward-jumping". In this case, the enumeration continues with the first variable after the $Desc(B_{i+1})$. In case it has not been recorded as either a good or nogood, then $\mathcal{A}[B_i \cap B_{i+1}]$ must be extended towards a consistent solution, and the enumeration continues with the variables of B_i.

In their experiment, they compared implementations of BTD (coupled with forward-checking and MAC respectively) to non-BTD implementations. They also compared an implementation of BTD with goods and nogoods to an implementation of BTD without good and nogood recording. An experiment was done for both generated CSPs and real-world instances. For the generated CSPs, the performance of all the implementations was tested for classical random CSPs with networks with a tree width that is not necessarily small, in order to validate that BTD does not perform worse on such problems. Then they continue testing with structured random CSPs (with a small tree width), for which it is expected that BTD can perform better due to the structural properties of the tree decomposition.

They observed that for classical random CSPs, the implementations of BTD with and without good and nogood recording achieved very similar results. Moreover, they observed that, for the classical random CSPs, the BTD implementations achieved similar results to the non-BTD versions and in some cases performs slightly better.

For the case of structured random CSPs it was observed that the BTD implementation with good and nogood recording performed significantly better. Moreover, the implementations of BTD were significantly faster than the non-BTD versions.

Finally, the results of the comparison for real-world instances is that the BTD version receives either better or similar results.

The BTD algorithm serves as the basis of the algorithm used in this thesis. However,

some adaptations are made. In particular, the algorithm has been adapted to also work for optimization problems, whereas the original paper focuses exclusively on satisfaction problems. Moreover, our implementation exploits opportunities in the algorithm for parallelization (and, as an extension of that, also to make it distributed).

3.2 Embarrassingly parallel search

One approach to solve CSPs in a (massively) parallel way has been proposed by Régin et al. [3, 14]. In this approach, called Embarrassingly Parallel Search (EPS), the search space, i.e. the domains of the variables, is split into a huge number of disjoint parts. This is done by means of a depth-bounded depth-first search (i.e. a depth-first search that never visits nodes located at a depth greater than a given value). The depth-bound is increased until the right number of subproblems is generated. Every worker then receives a list of these disjoint parts, which effectively assigns a different portion of the search space to every worker. Each worker can then explore its assigned search space independently and without communication with the other workers.

It is shown that when a sufficiently large number of subproblems is generated (30 times the number of workers), and if these subproblems are not trivially detected as being inconsistent (using the propagation mechanism of a solver), then the division of the work will be balanced.

Since embarrassingly parallel search splits the domains, the time-complexity of solving CSPs using backtracking can be reduced from $\mathcal{O}(e.d^n)$ to $\mathcal{O}((e.d/w)^n)$, where w is the number of workers.

While the original work by Régin et al. [3] only focuses on the case of parallel solving, this technique may also be used for solving in a distributed manner. This approach was tested by Malapert et al. [14]. They found that the EPS algorithm can be adapted with reasonable effort to a distributed version, where the jobs for the subproblems are submitted to a batch scheduler. However, it increases the overhead as a new worker is created for each subproblem. Additionally, some early work [15] is also being undertaken in the development of a portable implementation of EPS which can be used on different machine architectures. In the future, this work can be used to solve constraint satisfaction problems in a distributed manner. In this approach, Akka is used in combination with an existing solver, namely Choco solver.

While between our work and EPS, the idea of dividing the problem into smaller subproblems and using workers to solve these problems in parallel is similar, one major difference is that our work looks at the structural properties in order to divide the problem into smaller subproblems. Due to the use of these structural properties, the size of the total search space (i.e. the product of the domain sizes of the variables in the subproblems) is smaller than the size of the search space of the undivided problem. This means that solving the subproblems, even when done so in a non-

parallel way, is expected to provide a performance gain over solving the full problem – a key motivation for the BTD algorithm. In the case of EPS, such structural properties are not leveraged and the search space itself gets split into a large number of smaller parts, where the sum of the search space of all these parts adds up exactly to the search space size of the undivided problem.

3.3 Multi-agent search

Another approach for solving CSPs in a distributed way is that of *multi-agent search*. Some of the early work applying this method is provided by Clearwater et al. [16]. In this approach, there are numerous problem solving agents. Each agent is capable of solving the problem on its own and every agent may use a different technique to solve the problem. All the agents work on a copy of the complete problem, and may also communicate with each other.

In an experiment, Clearwater et al. [16] showed that a number of cooperating computational agents were able to solve a set of cryptarithmetic problems with a super-linear speedup. The agents in the experiment wrote 'hints' to a central 'blackboard'. In the case of the cryptarithmetic problems, the hints were lists of letter-to-digit assignments that added up correctly modulo 10 (in order to account for the carry-over) for at least one of the columns.

One of the assumptions made by this approach is that the hint-blackboard access is cost-free. However, once the number of agents grows to a size that no longer fits on a single machine, this blackboard access will have to go over a network for most of the agents, which makes it relatively expensive, in turn limiting the scalability of this approach.

The conceptual ideas from multi-agent search were used in the SAT community. An example of where this was used is ManySAT [17]. Instead of sharing 'hints' on a blackboard, the agents share so-called *nogoods*, which are parts of the search space where a solution cannot exist.

The work from this thesis is similar in approach with respect to there being numerous communicating agents that are working to solve the problem. However, unlike with multi-agent search, there is no central blackboard (the good- and nogood stores are not centralized, but local to each agent). Moreover, the agents do not have a copy of the complete problem, but instead work on a dedicated subproblem.

3.4 Distributed constraint satisfaction and optimization

In the literature, extensive research has been performed into Distributed Constraint Satisfaction Problems (DCSPs) and Distributed Constraint Optimization Problems (DCOPs). A formalized description of a DCSP was first provided by Yokoo et al. in [18]. A DCSP is a constraint satisfaction problem where the variables are distributed among a set of automated agents. The agents are capable of solving subproblems and can communicate with each other in order to arrive at a global solution.

An example of an application of DCSP is that of tracking in sensor networks [19] and distributed meeting scheduling [20].

A crucial difference between DCSPs and conventional CSPs (also called *centralized CSP*s) is that in a DCSP, no single agent has access to the complete problem. Each agent has some variables, and constraints exist between variables assigned to different agents. Communication between agents may be necessary for an agent to learn about the existence of some of those constraints.

It should be noted that there is a distinction between the case of solving a DCSP (a distributed problem) and solving a centralized CSP using distributed techniques. The latter is the focus of this thesis.

This distinction notwithstanding, it may still be the case that algorithms or methods used for solving DCSPs are also applicable and useful for solving centralized CSP in a distributed fashion and vice versa. However, much of the research into DCSP focuses on a distributed problem, while one of the major challenges in solving a centralized CSPs is finding a means of decomposing or distributing that centralized problem [21].

3.5 Hypertree Decompositions

A hypergraph is a graph where edges, called hyperedges, can connect more than 2 vertices. Formally, a hypergraph $\mathcal{H}$ is a tuple (V, E), where V is a set of vertices and E a set of hyperedges. A hypertree of a hypergraph $\mathcal{H}$ is a triple $\langle T, \chi, \lambda \rangle$, where $T = (V_T, E_T)$ is a rooted tree, and χ and λ are labeling functions which associate each vertex $p \in V_T$ to a set of variables ($\chi(p) \subseteq var(\mathcal{H})$) and a set of edges ($\lambda(p) \subseteq edges(\mathcal{H})$) respectively. The width of a hypertree is the maximum number of hyperedges in any of its nodes, i.e. $max(\forall_{p \in V_T} : |\lambda(p)|)$.

Hypertree decomposition is the process of converting a hypergraph into a hypertree. Computing a hypertree decomposition with a width less than or equal to some constant k for a given hypergraph can be done in polynomial time [22].

In 2017, Liu et al. [23] proposed a new approach for mapping constraint networks

on multi-core processors by means of hypertree decomposition. In their proposed procedure, the hypergraph of the original constraint network is decomposed into a hypertree with the number of tree nodes equal to the number of available CPU cores. Then each sub-problem on each node of the decomposition tree is solved in parallel, which results in a decomposition tree where each node consists of the tuples that are the outcomes of solving these sub-problems. To arrive at a global solution, directional arc-consistency is performed along a topological ordering of the decomposition tree. Subsequently, all solutions of the subproblems are combined along the topological ordering, starting from the root node.

The approach of this thesis is similar in the sense of using a tree decomposition, although this thesis does not use the hypertree decomposition. However, unlike the work from Liu et al, where the solutions of all the subproblems are computed and then merged to find a global consistent solution, in our work there is an ordering (guided by the tree structure) which determines which nodes can be solved in parallel.

3.6 Distributed COP solver

Earlier work has been undertaken by R. Kip [4] to develop a distributed version of the BTD algorithm. In his work, the BTD algorithm was adapted for solving COPs and the actor model was used to introduce parallelism and the option for distributed execution. For answering the question of how to deploy the actors over a cluster of machines, different deployment strategies were implemented and compared: a random, workload-based and a structure-based deployment strategy.

An implementation was developed where a tree decomposition could be generated with specific structural properties, after which a constraint graph was derived from this tree decomposition. The derived constraint graph then served as the basis for an instance of the graph coloring problem, which could then be solved using the distributed BTD algorithm. The implementation and experiments showed that the BTD algorithm can be successfully distributed using the actor model.

The scalability of the distributed implementation was assessed experimentally and it was found that increasing the number of cluster nodes (with the same total number of CPU cores) seemed to linearly decrease the performance, while increasing the amount of CPU cores (with the same total number of cluster nodes) increased the performance logarithmically. Additionally, it was found that the deployment strategy used can have a considerable influence on the performance of up to 30 percent when compared to a random deployment. The success of a deployment strategy was shown to be highly dependent on the structure of the tree decomposition, and hence different problems benefit from a different deployment strategy. Overall, the weight-based deployment had the most stable performance. In short, this deployment strategy assigns weights to the bags of the decomposition based on the depth in the tree. Bags positioned at a lower level in the tree (closer to the leaves) get a higher weight assigned than bags

higher up in the tree. The deployment strategy then tries to evenly divide the bags over the nodes based on this weight.

This thesis builds on top of the concepts and ideas from the work by R. Kip. However, instead of generating a tree decomposition and then deriving a constraint optimization problem from this tree, this thesis implements several algorithms for computing the actual tree decomposition of an input problem. This allows for a more realistic assessment of the performance of the algorithm when given a tree decomposition computed with a state of the art decomposition technique. Additionally, this approach makes the implemented distributed solver from this thesis more generic and extendable, because other CSPs and COPs can easily be added to the set of supported problems. Lastly, this thesis continues the research into the optimal deployment strategy by implementing multiple new deployment strategies.

Chapter 4

Design & Implementation

In this chapter the design of the developed constraint solver is presented. This chapter focuses mostly on the higher-level design of the solver. Some considerations and details regarding the implementation are also provided.

4.1 Distributed BTD algorithm

Solving large and complicated CSPs and COPs on a single machine is constrained by the limited and finite resources of this single machine. Distributing, by scaling the solving process to multiple machines, offers an opportunity for increasing the amount of resources available.

The developed distributed BTD algorithm from this thesis is based on the BTD-algorithm from Jégou et al. [2] and uses a network of actors, which are distributed over a cluster of machines in a network. Each actor gets assigned to a subproblem, and is responsible for solving this subproblem by means of local computation and communication with the other actors. Conceptually, in the adapted BTD-algorithm, each actor in this network (also called *tree node actor*) receives an assignment from its parent, and then tries to find a solution for its local subproblem that extends this assignment. After finding a solution to the local subproblem, the tree node actor will send an assignment for all the variables in the *separator* (i.e. all variables which are in both the parent and child node) to all of its children, which will then in turn perform the same solving process. Once a solution is found by an actor that represents a leaf node, a consistent path has been found and the solution is send back to the parent. A parent that has received a solution from all its children will merge these solutions and, in turn, send this merged solution to its own parent. This way, the solutions get merged all the way up the tree, until it reaches the root node, where it gets merged into a global solution. If any of the children cannot extend a given assignment, then

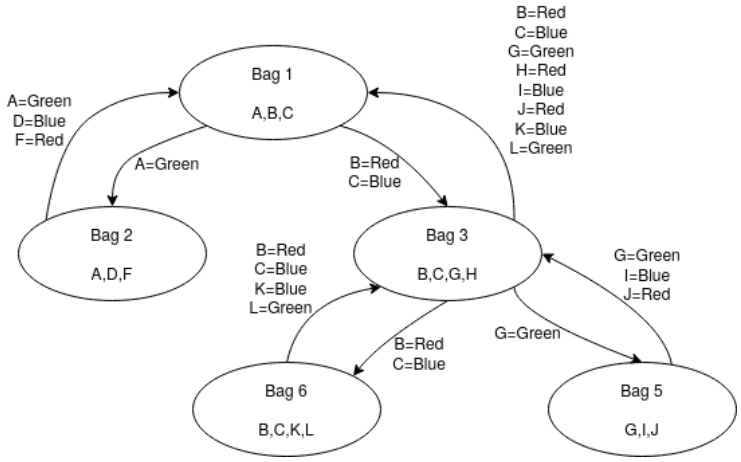

Figure 4.1: The assignments and (sub-problem) solutions that are passed during the solve process.

the parent tree node actor will search for another solution to its local subproblem, after which the process of sending the separator assignment to the children repeats again. This continues, until no more local solutions are found, after which the tree node actor will inform its parent that the given assignment could not be extended to a solution.

For solving the local subproblem, each actor uses an instance of an existing solver, namely Choco solver[1] [24]. This is an open-source library for modeling and solving constraint satisfaction and optimization problems. A variety of techniques is implemented in this solver. However, no means of exploring the solution space in parallel is provided by Choco solver itself.

The search process is kicked off in the top-level actor and works as follows. A tree node actor receives a `Solve`-message with a partial assignment which should be extended to a complete solution. This partial assignment is a mapping from variables to concrete values for the variables in the separator of child bag and parent bag. The actor then uses the local Choco solver to set the assigned variables and to iterate over all the solutions for its local subproblem. For each solution, it sends a `Solve`-message to the child actors, containing again the assignment for the variables in the separator. When it concerns a leaf node (i.e. a tree node actor without children), this means a path with solutions has been found all the way down from the root node. The solution of the leaf node will be send back to the parent actor.

After sending a `Solve`-message, the actor will wait for the solutions of all children. If one or more of the children was unable to extend the assignment to a complete solution, then the actor will continue with the next local solution. If all children

[1] `https://choco-solver.org`, version 4.10.6 was used

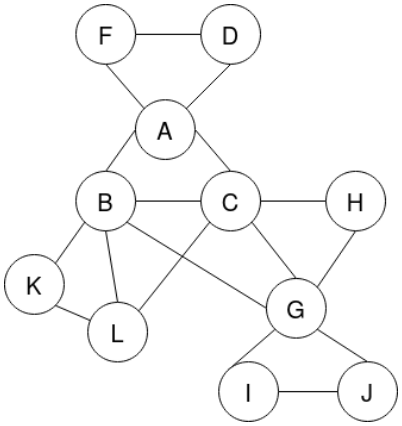

Figure 4.2: An example of a constraint graph of a random graph coloring problem

were able to extend the assignment to a complete solution, then those solutions are merged together and send to the parent. This process of merging solutions when all children are able to extend the local solution, is repeated up the tree, such that these solutions of the subproblems are aggregated together to form the solution to the complete problem in the root node. An illustration of the messages being send during this process can be seen in Figure 4.1. The arrows pointing downwards are the `Solve`-messages being sent from parent to child and contain the assignment in the separator (i.e. the values for the variables which are present in both the parent and the child). These values are the outcome of the local solve process using the Choco solver. The arrows pointing upwards represent the responses from the children that are send to the parent, after they have found a local solution that could in turn be extended by their children. These responses contain the merged solution. The root node merges the solutions of its children into a global, complete solution. In the case of the example, the complete solution can be seen in Figure 4.3.

The actors also implement a good store and nogood store. Whenever a tree node actor receives a `Solve`-message, these local stores are first checked. When an assignment is registered as a good or nogood, the solution can immediately be send back to the parent.

Dividing the CSP or COP into subproblems is not trivial. The developed algorithm uses a tree decomposition of the (primal) constraint graph to achieve this. The resulting tree decomposition, which consists of nodes containing the subproblems, is then used for deploying the actors, where every node in the tree is assigned to an actor. The problem is then solved using the BTD-algorithm, which has been adapted for use in the actor network and for solving constraint optimization problems. In order to obtain a tree decomposition with the subproblems, the implemented solution first creates a (primal) constraint graph from the model of the problem. This is done by creating a graph where the vertices represent the variables and two vertices

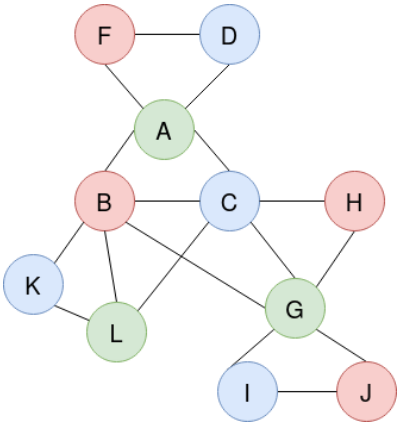

Figure 4.3: A example of a solution of the graph coloring problem in Figure 4.2.

are connected by an edge whenever a constraint exists where both these vertices are involved in. For example, in Figure 4.2, the constraint graph of an instance of a graph coloring problem can be seen. In this graph the vertices represent the variables (in this case labeled with a letter) and the edges represent the constraints (in this case inequality constraints).

After the construction of the constraint graph, a decomposition algorithm is used to compute a tree decomposition of this graph. Finding a tree decomposition of minimum width is NP-complete. Therefore, in order to find a decomposition within polynomial time, the algorithm used does not guarantee a minimum width. A suite of different algorithms which try to approximate this minimum, or otherwise good tree decompositions, has been implemented and are described in Section 4.3.

On top of performance improvements due to leveraging structural properties of the constraint graph by using the tree decomposition, additional performance can be gained by introducing parallelization into the algorithm. This parallelization occurs when, after finding a local solution, `Solve`-messages are sent to the children nodes in parallel. This leads to parallel execution of the distributed solve process, because the child nodes all run as a separate actor in the actor network. This parallelization step occurs each time the solve process moves down a level, for a maximum parallelism bounded by the amount of leaf nodes.

4.2 Constraint Optimization Problems

If the problem being solved is an optimization problem, the algorithm functions slightly differently.

There are two general approaches for implementing optimization:

1. Each tree node finds an optimal solution for the given separator assignment. An assignment is send to a (child) tree node, which will then find an optimal solution. This optimal solution is the solution for the sub-tree rooted in that tree node.

2. Each tree node finds an optimal solution, which costs less than a provided `maxCost` parameter. When propagating the search to the children, the `maxCost` for those children is reduced, to take into account the cost of the local solution that is being extended.

An advantage to the second approach is that it allows aborting a search, once the cost becomes larger than this `maxCost` parameter. However, we may have to search a particular separator assignment more than once if, later on, a search is done with a larger value for `maxCost`, which happens when a (grand)parent is exploring a less optimal local solution, which may potentially be extendable to a more optimal global solution. The second approach was implemented in the developed distributed solver.

In the case of an optimization problem, the implemented BTD algorithm behaves slightly differently then for satisfaction problems. First, a constraint gets added to the Choco model for the local subproblem, which restricts valid solutions to have a cost no greater than the `maxCost` parameter. Secondly, the tree node actor keeps track of the best (optimal) extended solution found up to that point for the given separator assignment. After finding a solution to the local subproblem, a `Solve`-message gets send to the child actors with the `maxCost` parameter reduced by the cost of the local solution (except for the separators).

4.2.1 Objective function

In order to allow for decomposition of the problem and optimization of the performance of the solver, a number of restrictions are applied to the objective function.

The first property that should hold for the objective function is that it is arbitrarily *decomposable*. We define this, similar to Modi et al. [25], as the objective function being decomposable, for any partitioning of the variables, into a sum of local objective functions ϕ_I, such that:

1. Each ϕ_I is defined for one of the partitions of variables (called I)

2. Each partition of variables has a ϕ_i defined for it

3. Each ϕ_I maps assignments to a natural number ($\mathbb{N}_0$).

4. An aggregation operator exists, such that for a complete assignment, the total value is the aggregation of the sub-values. This is typically a sum-operator.

Formally, the partitions are a subset of the set of all variables, i.e. $I = \{x_i, ..., x_j\}$ and

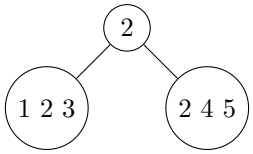

Figure 4.4: A (partial) tree decomposition

$I \subseteq X$ and $\phi_I : D_i \times ... \times D_j \to \mathbb{N}_0$.

A second property that should hold for the objective function is that of '*distributivity*'. In the context of the objective function, we mean with this that when the cost values of two sub-problem solutions are equal, then this implies that those sub-problem solutions can substitute each other in a larger problem, without the total costs changing due to this substitution.

Formally, we can define this as the following criteria:

$$\phi_I(\mathcal{X}) = \phi_I(\mathcal{Y}) \implies \forall_{J \supset I} \forall_{\mathcal{A} \in S[J \setminus I]} : \phi_J(\mathcal{A} \cup \mathcal{X}) = \phi_J(\mathcal{A} \cup \mathcal{Y}).$$

In the equation above, S is the complete solution space (i.e. $D_1 \times ... \times D_n$) and the notation $S[Z]$ refers to the solution space of only the variables in Z. The criteria mandates that whenever, for a local objective function, the costs values of two assignments are the same, this implies that for every local objective function that concerns a larger partition containing the original, both of those assignments can be substituted for one another in the larger assignment.

This *distributivity* property does not hold for all constraint problems. Take for example the optimization problem of finding the chromatic number for a graph whose (partial) tree decomposition can be seen in Figure 4.4. In this instance, the left hand child may return $\{1{=}\texttt{red}, 2{=}\texttt{green}, 3{=}\texttt{blue}\}$ as the optimal solution. A bag may have more than one solution which has the same value for the cost function, so the right child may have as its optimal solutions: $\{2{=}\texttt{green}, 4{=}\texttt{yellow}, 5{=}\texttt{cyan}\}$ and $\{2{=}\texttt{green}, 4{=}\texttt{red}, 5{=}\texttt{blue}\}$. While both of these solutions are equally optimal locally, it can be observed that the global solution would be more optimal if the second local solution for this right child bag was used, as that would limit the number of colors used to 3 instead of 5. Specifically, this problem occurs because the cost function for finding the chromatic number is not distributive: $\phi_{\{1,2,3\}}(< \texttt{red}, \texttt{green}, \texttt{blue} >) + \phi_{\{2,4,5\}}(< \texttt{green}, \texttt{red}, \texttt{blue} >) \neq \phi_{\{1,2,3,4,5\}}(< \texttt{red}, \texttt{green}, \texttt{blue}, \texttt{red}, \texttt{blue} >)$.

From a theoretical point of view, it is possible to allow such non-distributive cost functions. However, this would require the implemented solution to have the child bags send all the optimal solutions to the parent bag. The parent bag would then have to evaluate the cost function for all possible combinations of the solutions returned by the children, in order to find the optimal of the combined solution. To limit the complexity of the algorithm, this was not done for this thesis.

The final restriction on the cost function is that it is assumed to be monotonically increasing, i.e. $\forall_X \forall_Y : X \subseteq Y \implies \forall_{\mathcal{A} \epsilon S} : \phi_X(\mathcal{A}[X]) \leq \phi_Y(\mathcal{A}[Y])$. This implies that when a partial solution is extended by another assignment, the cost function of this assignment cannot decrease. This assumption is required due to an implemented optimization, where the solver will stop exploring a solution once the cost exceeds the cost of a previously found complete solution.

4.2.2 Good recording and Nogood recording

For the good stores and nogood stores to work correctly in the case of optimization problems, some changes are applied to them. First, nogoods are stored along with the maximum cost for which no solution could be found. This is done, because it is possible that at a later moment in the search, a higher and less restrictive maximum cost parameter gets passed, for which a solution could potentially be found.

On top of this modification, the good store has also been adapted to store the most optimal solution with the given assignment. When consulting the good store, an extra check is done to see if the solution retrieved from the good store does not exceed the maximum cost parameter. If it does not, then this stored solution can be send to the parent, otherwise the good is ignored and the actor continues with the local choco solve process as if no good was found.

4.3 Tree Decomposition

In order for the distributed implementation of the BTD algorithm to leverage the structural properties of the constraint satisfaction problem, a tree decomposition first needs to be obtained. In this section, the different methods that are used for obtaining a tree decomposition are elaborated on.

4.3.1 Jdrasil

Jdrasil[2] is a Java library for computing tree decompositions [26]. It supports both exact and heuristic methods for computing tree decompositions.

Jdrasil uses some heuristics to determine how a tree decomposition for a given graph will be computed. Specifically, for graphs with less than 1000 vertices, Jdrasil will compute an exact tree decomposition, i.e. the resulting tree decomposition will have a minimum width.

In case of more than 1000 vertices, Jdrasil uses a stochastic greedy permutation heuristic. This heuristic eliminates a vertex v which minimizes some function $\gamma(v)$. In case of a tie, this will be decided randomly. Six different functions γ are implemented

[2] Available on https://maxbannach.github.io/Jdrasil/

in the heuristic, they are described in [27]. As the ties are decided randomly, different runs will produce different results, which allows for performing a stochastic search by using the heuristic multiple times and using the best result. Jdrasil performs ten thousand iterations of this algorithm, while different functions γ are used in different runs.

4.3.2 H-TD-WT

As finding an optimal tree decomposition is an NP-complete problem, Jdrasil is often not fast enough in finding a decomposition. Even in cases where it uses heuristics instead of computing an exact decomposition, it still takes relatively long due to the large number of iterations used in the stochastic algorithm. Moreover, Jdrasil focuses on reducing the width of a tree decomposition, while indications exist that the best tree decomposition for constraint solving, and for the BTD algorithm in particular, is not necessarily the one with the smallest tree-width [28] [29]. The size of the largest separator seems to be an important factor for the solving efficiency using the BTD algorithm [28].

In 2015, Jégou et al. introduced a new algorithmic framework for graph decomposition, called Heuristic Tree Decomposition Without Triangulation (H-TD-WT) [30]. It is based on traversal of the graph and uses properties of the separators and the associated connected components. Moreover, it allows for the implementation of several heuristics deciding on the choice of the next cluster. These heuristics can guide the structure of the final tree decomposition, using criteria like separator size, cluster size and connectedness of the clusters.

The algorithm starts with computing a first cluster, for which several methodologies exist. The implementation in this thesis uses the maximal clique of the constraint graph as the first cluster. The implementation of the algorithm used for finding the maximal clique is based on the implementation provided by Samudrala and Moult [31], which is based on the clique-finding algorithm design developed by Bron and Kerbosch [32].

The H-TD-WT algorithm keeps track of a list of already visited vertices. At the start of the algorithm, this list is initialized to be the first cluster. Additionally, the algorithm also keeps track of the connected components one gets when removing the already visited vertices from the graph. After establishing the first cluster, the algorithm uses a heuristic for finding the next cluster. Several heuristics are described, each of which works similar: the next cluster starts from a subset of the visited vertices, that neighbor one of the connected components, and then iteratively, the neighbors of those vertices which are a part of the connected component get added to the set of vertices that make up the next cluster. This 'expansion' of the cluster continues until some criteria is met. In the original paper [30], four heuristics are described for choosing the next cluster:

1. H_1-TD-WT, which minimizes locally the size of the next cluster.

2. H_2-TD-WT, which imposes a restriction on the choice of the next cluster, where this next cluster has to be a single connected component.

3. H_3-TD-WT, tries to identify and separate independent parts of the graph.

4. H_4-TD-WT, builds on top of H_3, but additionally aims to limit the size of the separators in the tree decomposition by imposing an upper-bound on the maximum size of the separators for the clusters which are added as a next cluster.

After the criteria is met, or whenever the entire connected component gets consumed, the next cluster is known and gets added to the list of already visited vertices. This process then repeats until all of the connected components have been assigned to a cluster. This results in a list of all the clusters. A tree decomposition is then computed, similar to [33], by computing a maximum spanning tree of a graph whose vertices are the clusters and edges link two clusters that share at least one vertex. The edges are labeled with the sizes of these intersections (i.e. the separator sizes). The resulting maximum spanning tree is then the tree decomposition. In the implementation of this thesis, Prim's algorithm [34–36] is used for computing this spanning tree, which has a time complexity of $\mathcal{O}(n^3)$.

In order to evaluate the different heuristics, Jégou et al. [30] performed experiments to test how well the different heuristics perform with respect to solving a set of constraint satisfaction problems within a fixed amount of time. From their experiments, the heuristic H_4 performs the best.

The H-TD-WT algorithm is implemented and incorporated into the distributed solver, together with heuristics H_1, H_3 and H_4. For the first cluster, the maximal clique of the constraint graph is used, after which one of the heuristics is used to find the remaining clusters.

Besides the heuristics for determining the first and next cluster, the choice of the root node of the tree decomposition from all the clusters can have a high impact on the quality of the decomposition. The implementation from this thesis chooses the root cluster that maximizes the following ratio: $\frac{b_i}{|B_i|-1}$, where b_i is the number of constraints in the cluster and $|B_i|$ is the number of variables in the cluster. This method for choosing the root node was introduced by Jégou et al. in [37].

4.4 Deployment Strategies

In order to solve the CSP or COP in a distributed manner, the sub-problems need to be divided over all the machines in the cluster. In the case of the distributed BTD-algorithm, this means deploying the bags of the tree decomposition over the different machines. Since there is communication between the search processes that are assigned to the bags, and the workload generated by the different bags is not

uniform, it is expected that the manner in which the bags are divided will affect the performance and in particular the time needed to solve the problem. Several different strategies for deploying the bags have been designed and implemented in the distributed BTD-solver, in order to compare different deployment strategies. For the design of the deployment strategies, the assumption was made that all of the machines in the cluster are homogeneous, i.e. all of them possess the same type of CPU, amount of memory and network capabilities.

Formally, a deployment strategy produces a set of partitions $\mathcal{P}$, one for each machine in the cluster, of the tree decomposition $\langle \mathcal{T}, \mathcal{B} \rangle$, such that $\bigcup_{Q \in \mathcal{P}} Q = \mathcal{B}$, and $\forall_{Q,R \in \mathcal{P}} : Q \neq R \implies Q \cap R = \varnothing$. In other words, all bags in the tree decomposition are inside of a partition and there exist no two partitions that have bags in common. Note that there is no requirement for nodes in any of the partitions to be connected by an edge in the tree decomposition.

The following different deployment strategies have been implemented:

- *Random deployment*, which randomly assigns bags to the different nodes.
- *Branch deployment*, which aims to deploy long uninterrupted chains of bags to the same node.
- *Search-space-size weighted deployment*, which aims to equally divide the search space over the different nodes.
- *Separator-based deployment*, which aims to prevent large separators from spanning between two different nodes.

It is worth observing that deployment strategies can be roughly divided into two groups with different approaches: those that try to minimize communication overhead by placing bags that are expected to communicate on the same node (separator-based and branch deployment), and those that try to distribute the workload that a bag is expected to generate in such a way as to maximize the potential for parallelism (such as search-space-size weighted deployment).

In the next sections, a more detailed motivation as well as a formal description for each deployment strategy will be given.

4.4.1 Random deployment

The random deployment serves as a baseline to compare with. For the communication aspect, it is expected that this deployment performs poorly, as many of the bags that communicate will not be on the same node.

This deployment strategy creates equal-size partitions. For each partition, bags are randomly chosen from a set of all remaining bags, until the partition has reached the

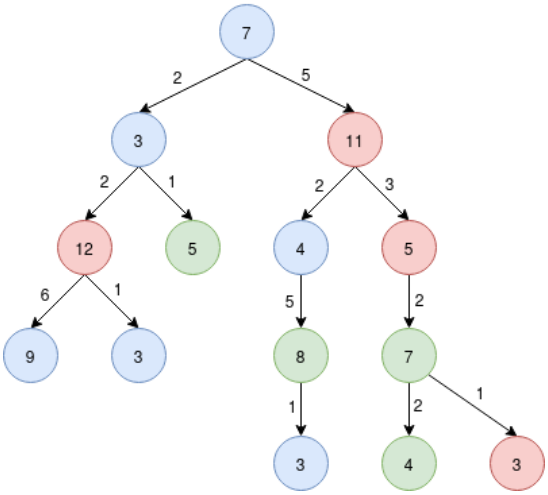

Figure 4.5: An example of a random deployment.

maximum size of $(\frac{\text{nr of bags}}{\text{nr of nodes}})$, after which the process repeats for the next partition. The remaining bags, in case $\frac{\text{nr of bags}}{\text{nr of nodes}}$ does not divide to a whole number, get added to the last partition.

Figure 4.5 shows an example of a random deployment. Colors represent the distribution over the cluster nodes, numbers in the nodes represent number of variables and numbers along the edges represent the separator size

4.4.2 Branch deployment

The distributed BTD algorithm is parallel in the branches of the tree decomposition. On any path from a bag in this decomposition towards the root, there will be only one bag that is, at that moment, computing the solution to a subproblem, while all the bags above it are waiting for child solutions. Moreover, communication happens along the branches, i.e. a bag communicates only with its children and its parent.

Therefore, to minimize communication between cluster nodes and simultaneously maximize parallelism on the different cluster nodes, one strategy is to deploy the longest connected branches of bags.

This deployment works by keeping track of the nodes that have been visited. The algorithm starts at a leaf node, assigns that leaf node to a partition and then iteratively does the same to the parent, until either the root node or a node that is already visited is reached. This process is repeated for all the leaf nodes.

In the end, this leads to a set of partitions, which are all linked bags without any branching. Some of these will be significantly longer than other ones. The partitions

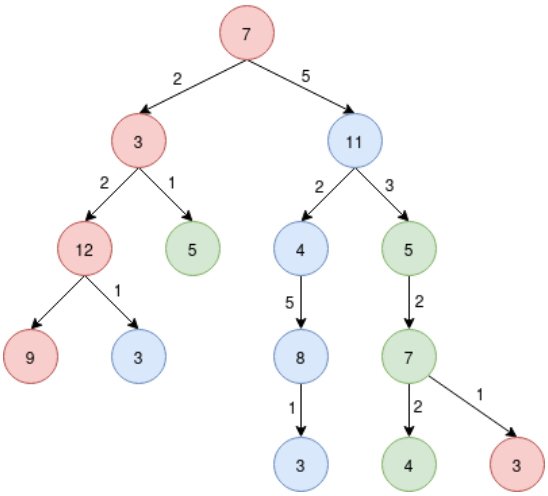

Figure 4.6: An example of a branch deployment.

are then evenly divided over the cluster nodes based on the number of nodes in the partition. Figure 4.6 shows an example of a branch deployment.

4.4.3 Search space size weighted deployment

For a weighted deployment, a weight is assigned to each vertex in the tree decomposition, using a function of the bag: $\phi(B)$. The deployment then creates partitions by assigning vertices to the partition with the lowest total weight, while traversing the tree in a breadth-first manner.

The size of the search space is the number of possible candidate solutions, i.e. the product of all the sizes of the domains of variables in a bag. Therefore, for the search-space-size weighted deployment, the weight function is this product of domain sizes.

When a bag consists of more variables or the variables in a bag have a larger domain, then this means the search space for that bag is expected to be larger. Consequently, it is expected that it will take more time to find a solution. Therefore, this search-space-size weighted deployment aims to find a division where an equal amount of search space is assigned to each node. Figure 4.7 shows an example of a search space size weighted deployment.

4.4.4 Separator-based deployment

As shown by Jégou et al. in [2], the time complexity of BTD is $\mathcal{O}(n \cdot s^2 \cdot m \cdot \log(d^s) \cdot d^{w^+ + 1})$, where s is the size of the largest separator (i.e. the largest intersection between

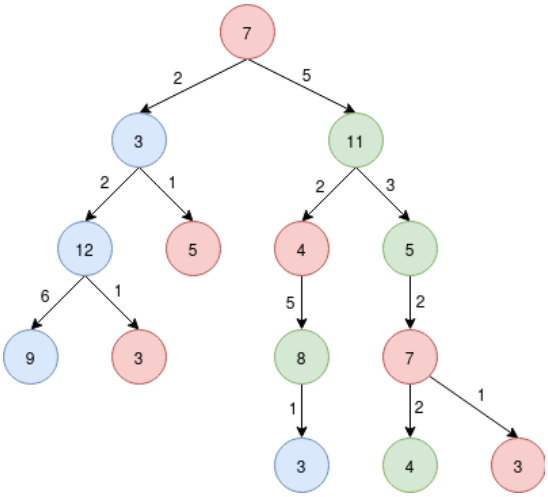

Figure 4.7: An example of a search space size weighted deployment.

bag B_i and its son B_j). In this separator-based deployment, the goal is to divide the bags over the nodes in such a way that pairs of bags with a large separator are always assigned to the same node, such that communication between these two bags does not span a network.

This deployment strategy is parameterized with a parameter S, which indicates the largest allowed separator size before two bags will be deployed to the same node.

The algorithm first generates a tree of bags from the tree decomposition, where all pairs of bags in the original tree decomposition with a separator larger than S are joined in this generated tree. After this, the generated tree is passed to one of the other deployment strategies, which divides the trees vertices over the different nodes in the cluster. In Figure 4.8 an illustration is provided of this process. The process of joining bags with a separator larger than S is visible as the differences between the top-left and top-right decompositions. After this join process, one of the other deployment strategies gets applied to the joined decomposition, the result of which can be seen in the bottom-left. After this step, the result is used to deploy the bags of the original (unjoined) decomposition, which can be seen in the bottom-right.

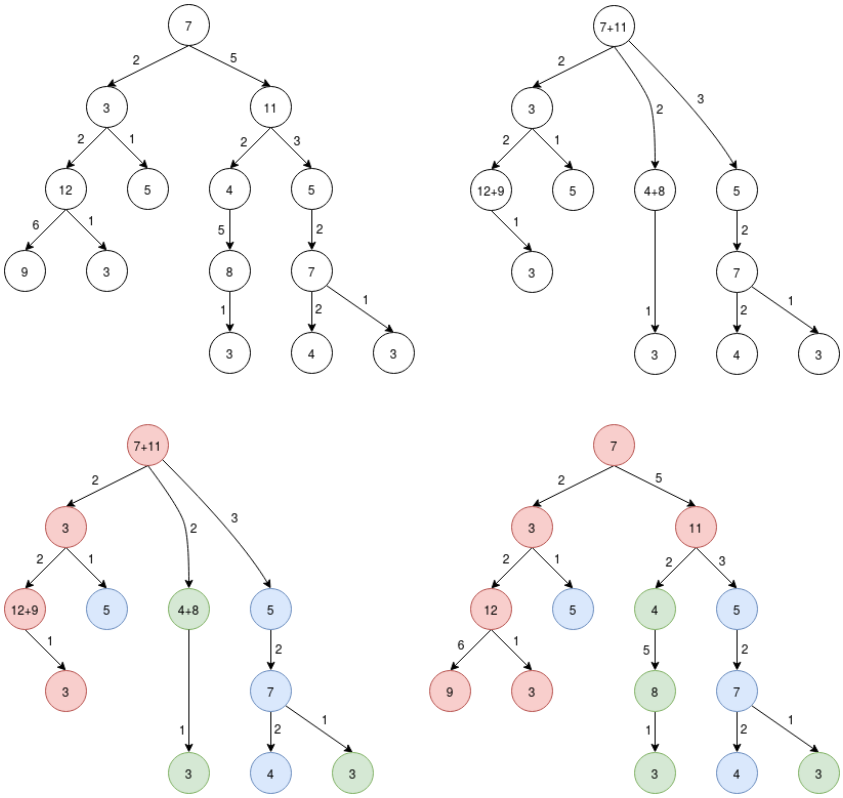

Figure 4.8: LT: the original tree decomposition. RT: after joining bags with separator > S, in this case $S = 4$. LB: after applying branch deployment to joined decomposition. RB: end result after applying the deployment to the original decomposition

4.5 Implementation

This section briefly describes some implementation details of the algorithm.

4.5.1 Deployment process

The first, and therefore top-level, actor that is created in the actor system is the `ClusterNodeActor`. This actor is responsible for joining the other nodes to form the cluster. Moreover, this actor keeps track of the other members of the cluster and whom of these is the leader. Every `ClusterNodeActor` spawns a `TreeNodeManagerActor` to manage all the tree nodes on that cluster node. Additionally, the leader `ClusterNodeActor` kicks off the deployment by sending a message to the `TreeNodeManagerActor` on the same machine.

Each machine in the cluster has one active `TreeNodeManagerActor`, which keeps track of all the `TreeNodeActors` (i.e. the bags in the tree decomposition) deployed on a machine. It is also responsible for kicking off the deployment phase (by spawning a new `TreeDeploymentActor`) when receiving a message from the leader `ClusterNodeActor` and for spawning new `TreeNodeActors` when a message is received from a `TreeDeploymentActor` during the deployment phase.

The `TreeDeploymentActor` is spawned after the leader `ClusterNode` has established that all the members have joined the cluster and is responsible for computing the tree decomposition and then deploying all the tree nodes (i.e. the bags) to the different machines in the cluster. This is done by sending messages to the respective `TreeNodeManagerActor`s. Because all the tree nodes need to have a reference to the actors that represent the children tree nodes, the spawning process starts at the leaf nodes of the tree decomposition and then deploys the tree nodes level by level until the root node is reached.

4.5.2 Algorithm pseudo-code

For each bag in the tree decomposition, a `TreeNodeActor` is created and deployed. These actors together run the distributed BTD algorithm. For reference, pseudocode for this actor can found in Listing 4.1.

At the start of the algorithm the `GlobalSolutionActor` sends a `Solve`-message to the top-level tree node actor. When receiving a `Solve`-message from the parent tree node actor, a check is first done against the good- and nogood-store. In case the assignment is already present in this store, then an answer can be sent back to the parent immediately. When the assignment has not been seen before and is therefore not present in the store, the tree node actor will start with setting the assignment passed by the parent in the Choco solver model of the local subproblem for the bag assigned to that actor (line 16 in the pseudo code). After setting the assignment, the solve process of the local Choco solver will be invoked. This occurs in a `while`-loop,

which finds new local solutions using Choco solver, until either a solution could be extended (in the case of a satisfaction problem), or (in the case of an optimization problem) whenever there are no more local solutions.

If the Choco solver returns no solution, then for the case of an optimization problem, the best found extended solution is recorded and returned to the parent (lines 20-22). When no solution was found at all for an optimization problem and in the case of a satisfaction problem, the assignment is recorded as a nogood and this outcome is sent back to the parent (lines 24-25).

For every solution found by the Choco solver, the tree node actor will send a `Solve`-message to the children tree node actors, asking them to extend this solution, and wait for the reply (lines 28-31). These child actors will run in parallel. When all the child actors have returned a reply with a solution, then this means the local solution could be extended, and the merged assignments from all the child solutions are stored in the good-store and send to the parent (lines 33-37). If at least one of the child actors could not extend the local solution, then an abort message is send to all the other child actors, as this means the local solution cannot be extended. For brevity, this part has been left out of the pseudocode.

In the case of a constraint satisfaction problem, once a solution was extended by all of the child actors, the process stops after sending this merged solution to the parent and storing it in the good-store. When a local solution cannot be extended, then the local Choco solver is invoked again to find the next local solution. This process continues until no more local solutions are found. In the satisfaction case, this means the assignment could not be extended to a solution. It will then be stored in the nogood-store and send back to the parent.

As opposed to constraint satisfaction, in the case of an optimization problem, after receiving the child solutions, the solve process of the local Choco solver is invoked again to find the next local solution. Meanwhile, the tree node actor keeps track of the best extended solution found up to that point (lines 38-39). When the local Choco solver cannot find any more solutions, this best found solution is returned to the parent tree node actor (lines 20-22).

```
Input: variables, the variables belonging to this bag
Input: children, the children tree node actors

nogoods = ∅
goods = ∅
variables =
optimal_solution = ∅

Receive 𝒜 from parent                    □ 𝒜 is the assignment in the separator
if 𝒜 ∈ nogoods:
  Send ∅ to parent                         □ inform parent there is no solution
else if 𝒜 ∈ goods:
  solution = goods[𝒜]
  Send solution to parent                        □ send solution back to parent
else:
  set assignment in choco model
  while True:
    local_solution = findSolution()                 □ invoke local choco solver
    if local_solution = ∅:
      if problem is optimization problem AND optimal_solution ≠ ∅:
        goods += <𝒜, optimal_solution>
        Send optimal_solution to parent
      else:
        nogoods += 𝒜
        ∅ → parent
      break
    else:
      for each child in children:
        separator = local_solution[variables ∩ child.variables]
        separator → child
      𝒞𝒮 ← wait for child solutions              □ children execute in parallel
      if ∅ ∉ 𝒞𝒮:
        extended_solution = ⋃_{S∈𝒞𝒮} S ∪ local_solution
        if problem is a satisfaction problem:
          goods += <𝒜, extended_solution>
          Send extended_solution to parent
          break
        else if ϕ(extended_solution) < ϕ(optimal_solution):      □ ϕ is cost function
          optimal_solution = extended_solution
```

Listing 4.1: Pseudo code for tree node actor

Chapter 5

Experiments & Analysis

This chapter presents the methodology, results and analysis of the experiments. First, the different types of problems which were tested are described. After this, for each experiment, the setup is discussed, followed by the presentation and analysis of the results.

5.1 Constraint problems

In this section, a brief description will be provided on the different constraint satisfaction problems which are used in the benchmarks. The problems can be divided into two categories, namely structured and unstructured.

5.1.1 Random Graph Coloring

For this instance, an adapted version of the graph coloring problem is used. In the traditional graph coloring problem, a connected graph has to be colored with a finite number of colors in such a way that no two vertices connected by an edge have the same color. In the optimization case, traditionally a solution is scored based on the number of different colors that are used. However, such a cost function would not satisfy the assumption of *distributivity* required by our implementation, as explained in Section 4.2.1. For this reason, the cost function used in this thesis assigns distinct numbers to each color and then computes the sum of those numbers as the cost.

In the *Random Graph Coloring* problem, a random graph is created by creating N vertices, linked in a long chain where each vertex is connected to the next one, in order to guarantee that the graph consists of a single connected component. Every node has a probability of being connected to any of the other nodes. This probability

is configurable using the `connectedness` (`cn`) parameter, having a value between 0 and 100. A value of 100 means a complete graph and 0 leads to a linked chain of vertices with no further edges.

5.1.2 Random Geometric Graph Coloring

A *random geometric graph* is constructed by randomly placing nodes in a metric space, and then connecting two nodes when they are within a certain range. One of the properties of random geometric graphs is that they display a high amount of community structure [38], meaning that they consist of densely connected clusters which are sparsely connected in between.

As one of the constraint optimization problems used in the benchmark, a graph-coloring problem will be used, for which the underlying graph is a random geometric graph. For graph coloring, the constraint graph is exactly the same as the graph that is being colored. Consequently, due to the properties which allow the random geometric graph to be decomposed, this will mean that a more favorable tree decomposition can be found. The random geometric graphs used in this thesis are based on the 2D space and euclidean distance. The parameters for this problem are: the number of nodes (`n`), the size of the grid (`g`), the range (`r`) and the number of colors used (`c`).

It is also worth mentioning that the graph coloring problem for random geometric graphs is closely related to the frequency assignment problem [39] (p. 109), which is the problem described in Section 5.1.4.

5.1.3 Tree-shaped graph coloring

In order to test the implementation, a specific constraint optimization problem is used, which is highly suited for the BTD algorithm due to its structure (consisting of many branches) offering a significant opportunity for parallelism. This is the *tree-based random graph coloring* problem: a graph coloring problem where the underlying graph is a tree of clusters, as can be seen in Figure 5.1. Each cluster is a small random graph, similar to the one from Section 5.1.1. The problem is parameterized in such a way that different values can be provided for the cluster size (`cs`), number of children for all the nodes (`b`), the cluster connectedness (`cc`) (i.e. the probability of two vertices within a random cluster to be connected by an edge), the separator size (`s`) and the depth of the tree (`d`).

In the first experiment, this problem is used in order to evaluate the performance of the different decomposition methods. All four implemented decomposition algorithms (i.e. `h1tdwt`, `h3tdwt`, `h4tdwt` and `jdrasil`) were compared to each other. In the case of the clusters being complete graphs (i.e. with a value of 100 for the connectedness), a 'perfect' tree decomposition can be derived from the constraint graph, as can be seen in the bottom in Figure 5.1. A decomposition method for deriving this 'perfect' tree decomposition was implemented, in order to compare the performance of this

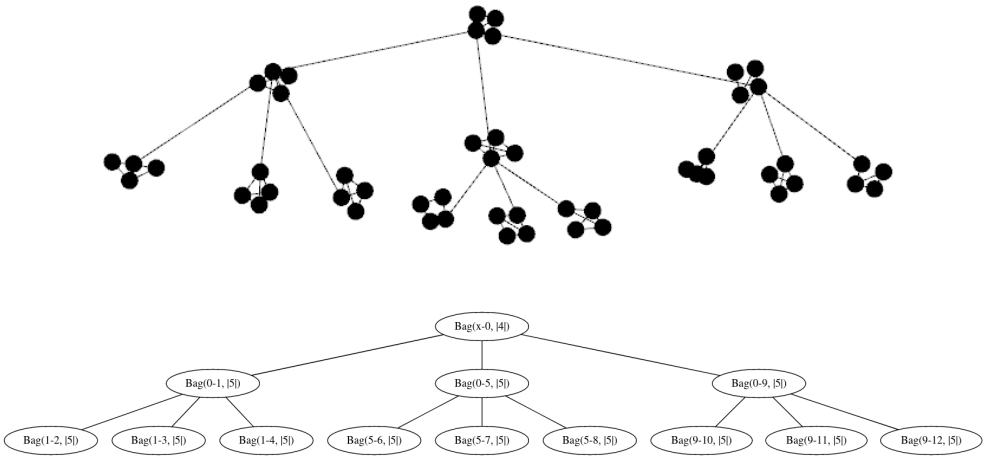

Figure 5.1: Tree-based constraint graph (top) and associated tree decomposition (bottom)

ideal tree decomposition with the other deployment methods. In an ideal scenario, when the clusters in the tree are fully connected, a decomposition algorithm is able to arrive at the result of this 'perfect' tree decomposition, which provides the lowest overhead and maximum opportunity for parallelization. For smaller values for the connectedness, the random graphs that form the clusters in the tree, in many cases, will contain some structure that can be exploited by a tree decomposition method, allowing for a potentially more optimal decomposition.

5.1.4 RLFAP

The Radio Link Frequency Assignment Problem (RLFAP) is a set of constraint optimization problems made available by the French "*Centre d'Electronique de l'Armement*" (CELAR)[1].

The RLFAP problem has as its objective to assign frequencies to a set of radio links between pairs of locations in a way that avoids radio interference. Each radio link is a variable in the COP and the domain of the variable represents the set of all frequencies that can be assigned to that link. The constraints that should hold are described by the equation:

$$|F_1 - F_2| > k_{12}$$

This equation mandates that the difference in the frequencies assigned to two radio links which are close to one another, is larger than some pre-defined constant k_{12}. A

[1] instances can be retrieved from `https://miat.inrae.fr/schiex/rlfap.shtml`

list of pairs of radio links with the associated k_{12} constants is provided as a part of the problem description.

In order to facilitate addition of new radio links later on, a choice of frequencies is preferred which leaves room for later additions. Therefore, several different criteria can be added to the satisfaction problem described above, turning it into an optimization problem. Two criteria are typically considered:

1. Minimization of the maximum frequency used: in this case one of the frequencies higher than the maximum used can be used for new radio links

2. Minimization of the number of frequencies used: in this case the unused frequencies can be considered for new radio links

Due to the assumption of *distributivity* (explained in Section 4.2.1), the second one cannot be used in the distributed solver from this thesis. For this reason, we will be using the first optimization criteria.

5.2 Environment

The experiments from this thesis are performed on the Peregrine[2] cluster, ran by the Center of Information Technology of the University of Groningen. The machines used in the experiments from this thesis have the following specifications per node:

- 24 cores at $2,5$ GHz (2x Intel Xeon E5 2680v3 CPUs)
- 128 GiB memory
- 1 TiB of internal disk space

Additionally, the internal network is also relevant to mention, because it is used to communicate between different nodes in the cluster. The Peregrine cluster makes use of a 56 Gbps non-blocking Infiniband network.

The Peregrine cluster runs CentOS 7 as its operating system. The distributed solver from this thesis has been developed in version 2.13.1 of the Scala programming language, using version 2.6.12 of the Akka typed framework for the actors. For solving the local sub-problems version 4.10.6 of the Choco solver was used. During the experiments we use the JVM from OpenJDK, more specifically version 11.0.2.

[2] `https://wiki.hpc.rug.nl/peregrine/introduction/cluster_description`

5.3 Benchmarking decomposition methods

The goal of the first experiment is to find out which decomposition method performs the best on a set of instances of constraint satisfaction and optimization problems described in Section 5.1. Primarily, we focus on finding the decomposition method that solves the largest number of instances, and only look at the solve time as a secondary quality metric. The decomposition methods described in Section 4.3 are compared in this experiment. For the H_4-TD-WT method, two variations for the threshold for the separator size (namely 5 and 15) are tested. Concretely, the following methods are compared:

- Jdrasil
- H_1-TD-WT
- H_3-TD-WT
- H_4S_5-TD-WT
- H_4S_{15}-TD-WT

5.3.1 Setup

The set of problems which are used is a selection of instances without structure (the random graphs), instances with structure (geometric random graphs and RLFAP) and instances with a tree-like structure (tree-clustered graphs). These instances are run on a setup with 1 node having 8 CPU cores. The following instances are used:

- Random Graph Coloring
  - RG-N21-C10-CC3
  - RG-N25-C10-CC3
- Radio Link Frequency Assignment Problem:
  - Scene 05
  - Scene 11
- Random Geometric Graph Coloring
  - RGG-N79-G750-R80-C7
  - RGG-N1000-G2675-R50-C7
  - RGG-N2000-G8000-R35-C10

- Tree-shaped Random Graph Coloring
  - TRG-B2-D3-CS6-CC100-C6-S2
  - TRG-B2-D4-CS7-CC33-C6-S2

To give an idea about the size of the problems described above, Table 5.1 below provides the number of variables, constraints and the size of the domains of the variables.

Instance	**# variables**	**# constraints**	**domain size**
RG-N21-C10-CC3	21	23	10
RG-N25-C10-CC3	25	29	10
RLFAPSAT-05	400	2598	7 - 49
RLFAPSAT-11	680	4103	7 - 49
RGG-N79-G750-R80-C7	79	122	7
RGG-N1000-G2675-R50-C7	1000	1099	7
RGG-N2000-G8000-R35-C10	2000	2005	10
TRG-B2-D3-CS6-CC100-C6-S2	42	117	6
TRG-B2-D4-CS7-CC33-C6-S2	105	192	6
TRG-B5-D4-CS4-CC100-C4-S3	624	1401	4

Table 5.1: Different metrics of the problem instances

5.3.2 Results

Table 5.2 lists the results from the experiment. A cross in one of the cells indicates that the problem was not solved within 25 minutes, which is the maximum time per job available to us on the Peregrine cluster.

Instance	**Jdrasil**	$\mathbf{H}_1$	$\mathbf{H}_3$	$\mathbf{H}_4\mathbf{S}_5$	$\mathbf{H}_4\mathbf{S}_{15}$
RG-N21-C10-CC3	1010s	0s	141s	106s	114s
RG-N25-C10-CC3	28s	×	×	×	×
RLFAPSAT-05	×	×	×	2s	×
RLFAPSAT-11	×	11s	35s	40s	36s
RGG-N79-G750-R80-C7	×	276s	×	×	×
RGG-N1000-G2675-R50-C7	×	254s	×	222s	328s
RGG-N2000-G8000-R35-C10	×	87s	76s	96s	130s
TRG-B2-D3-CS6-CC100-C6-S2	×	86s	×	84s	90s
TRG-B2-D4-CS7-CC33-C6-S2	109s	132s	361s	25s	231s
TRG-B5-D4-CS4-CC100-C4-S3	110s	×	×	×	×
# Solved	4	7	4	7	6

Table 5.2: Results for decomposition method experiment

In Table 5.2, the results can be seen when trying to solve the instances using the tree decompositions from the different decomposition methods that were implemented. Jdrasil solves the lowest number of instances (only 4), even though it does manage to solve one of the unstructured instances and one of the tree-shaped instances (`RG-N25-C10-CC3` and `TRG-B5-D4-CS4-CC100-C4-S3`), which were not solved using any of the other decomposition methods.

H_1-TD-WT and H_4S_5-TD-WT solve the highest number of tested instances. For the remaining experiments, we are focusing on solving structured instances, and therefore we decide to use H_4S_5-TD-WT as the decomposition method for these experiments, because the average solve time for the structured instances that are solved by both H_1-TD-WT and H_4S_5-TD-WT is faster for H_4S_5-TD-WT (114.0 and 93.4 for H_1-TD-WT and H_4S_5-TD-WT respectively).

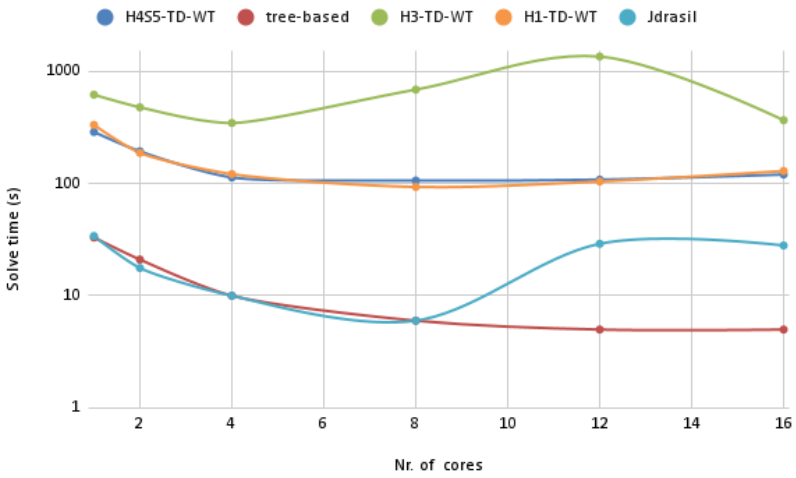

Figure 5.2: Solve times when using different methods for tree decomposition with an increasing number of cores on a single machine of the `TRG-B5-D4-CS5-CC100-C5-S3` instance (logarithmic scale).

In order to show the performance of the different decomposition methods when the number of cores increases, an experiment was done where an instance of the tree-shaped random graph coloring, `TRG-B5-D4-CS5-CC100-C5-S3`, was solved with all decomposition methods. This instance was also solved using the *tree-based* decomposition method, which is the optimal tree decomposition where every cluster in the constraint graph matches with exactly one bag in the tree decomposition. Using this setup, we can compare the performance of the different decomposition methods with an optimal tree decomposition. The results can be seen in Figure 5.2. For this instance, the performance of Jdrasil is actually quite close to that of the tree-based decomposition. The performance of H_4 and H_1 is quite similar, with both performing

around a factor 10 worse than the tree-based decomposition. H_3 performs the worst on this instance: with 8 cores or more the solve time is over a 100 times larger. This is a good illustration of how large the impact of the decomposition method is on the total solve time.

5.4 Benchmarking deployment strategies

The goal of the second experiment is to compare the performance of the different deployment strategies.

5.4.1 Setup

In order to measure and compare the different deployment strategies, an experiment is done where a set of instances is ran initially on a single node with many cores. Then, this is repeated on configurations where the cores are scaled more horizontally, up to a configuration of many nodes with one core each. The total number of cores in the cluster varies in each of the experiments and is based on the maximum observed number of cores where adding more cores no longer decreases the solve time. As an example, in the case of an instance that shows improvements in the solve time up to 16 cores, this would concretely mean the following configurations:

- 1 node with 16 cores each
- 2 nodes with 8 cores each
- 4 nodes with 4 cores each
- 8 nodes with 2 cores each
- 16 nodes with 1 core each

The goal of the experiment is to measure the overhead of running the solving process with the cores spread over multiple machines, and the expected associated networking overhead, for each of the deployment strategies. It is expected that some deployment strategies perform better than others. For this experiment, the decomposition method chosen in Section 5.3 is used (i.e. H_4S_5-TD-WT), unless mentioned otherwise.

5.4.2 Results

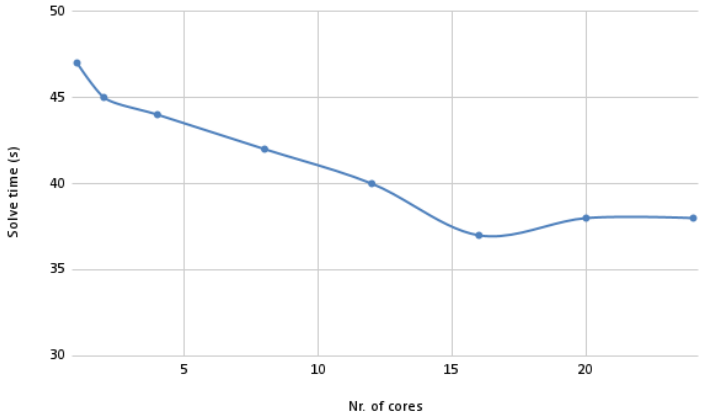

Figure 5.3: Results when scaling the number of cores for `RLFAPSAT-11`

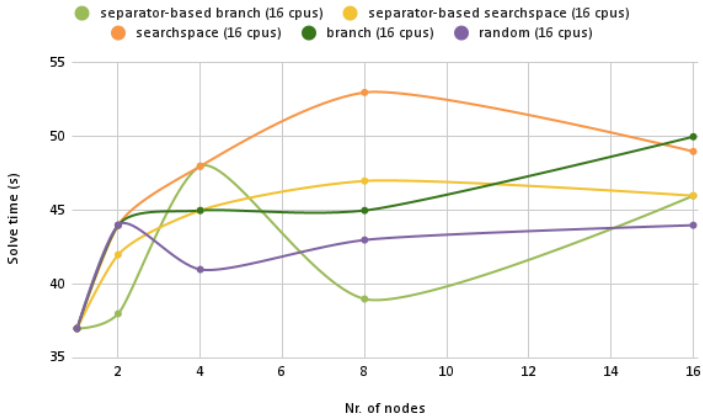

Figure 5.4: Results for different deployment strategies for `RLFAPSAT-11`

RLFAP For the radio-link frequency assignment problem, we can see in Figure 5.3 that on a single machine, the parallelization of the algorithm can improve the solve time up to 16 cores. For this reason, the experiment was run starting at 1 machine with 16 cores, and then gradually adding more machines, up to 16 machines (each with one core).

In Figure 5.4, it can be seen that initially, when increasing the number of nodes to 2

and 4, the overhead increases considerably. However, when continuing to increase the number of nodes, this overhead decreases again for some of the deployment strategies and stabilizes after 8 nodes. This can be explained by the relatively small size of the decomposition. The tree decomposition for this problem instance, which consists of only 27 tree nodes (when using H_4S_5-TD-WT as decomposition method) is quite small. This can also be seen in the total solve time (less than 60 seconds), which is lower than the other problems being tested. When dividing such a small number of bags over a growing number of (up to) 16 machines, each machine contains a smaller number of bags, making it easier to evenly distribute them.

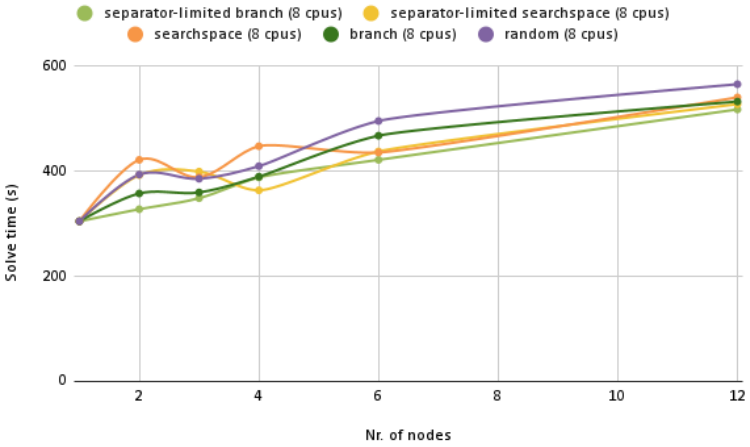

Figure 5.5: Solve times of `RGG-N79-G750-R80-C7` for different deployment strategies when increasing the nr. of nodes with a constant total nr. of cores

RGG-N79-G750-R80-C7 In Figure 5.5, the performance of the deployment strategies for a relatively small but constrained instance of the Random Graph Coloring problem can be seen. From this chart, we can observe that for a larger number of nodes, the random deployment performs the worst. The separator-limited branch deployment performs the best most of the time (except at 4 nodes).

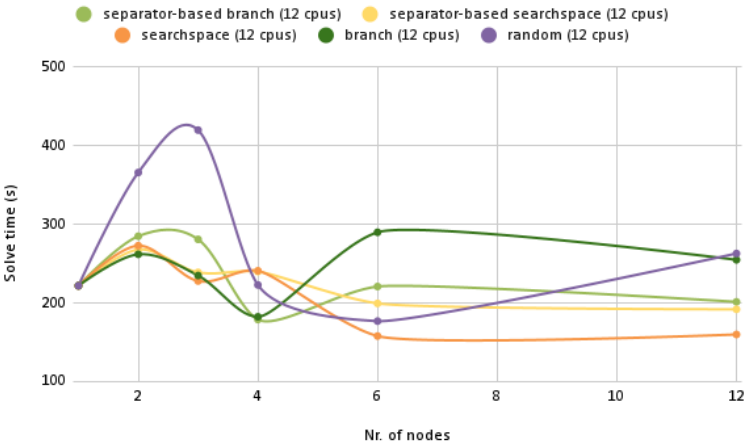

Figure 5.6: Solve times of `RGG-N1000-G2675-R50-C7` for different deployment strategies when increasing the nr. of nodes with a constant total nr. of cores

RGG-N1000-G2675-R50-C7 This instance is considerably larger in terms of variables, but is less constrained, and has a comparable solve time to the previous RGG instance. In Figure 5.6, we see that the solve time, while keeping the total number of CPU cores in the cluster constant, actually leads to a lower solve time for cluster sizes of 4 or larger.

Initially the random deployment performs the worst and both the branch and search space strategies perform considerably better. For more than 4 nodes, the branch deployment starts performing the worst and the search space deployment strategy become the best one.

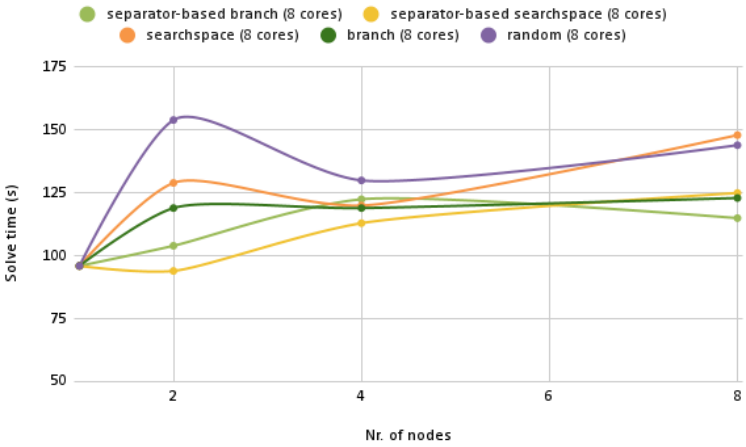

Figure 5.7: Solve times of `RGG-N2000-G8000-R35-C10` for different deployment strategies when increasing the nr. of nodes with a constant total nr. of cores

RGG-N2000-G8000-R35-C10 In Figure 5.7 we can observe again that the random deployment adds the most overhead, which is most visible at 2 nodes. The differences between the other deployment strategies are a lot smaller. The separator-based search space strategy appears to perform the best for this problem, although branch deployment strategy performs very similar when the number of nodes exceeds 4. The separator-based restriction on the deployment of the bags (i.e. bags with large separators are deployed on the same node), seems to have a positive impact on the solve time. So much so in fact, that the search space strategy goes from the second slowest to the fastest strategy up to 6 nodes. The separator-based strategy ensures that bags which share a large separator, and therefore are expected to communicate more often between each other, are deployed on the same machine, such that the communication does not have to span a network. This reduction in network communication overhead causes the separator-based search space deployment to be faster than the regular variant. This effect is however not noticeable for the branch deployment. Most likely, this is due to the structural nature of this deployment. The branch deployment strategy aims to deploy long branches on the same machine. This means that many combinations of child-parent bags are already deployed to the same machine, because they will be part of the same branch, making the effect of the separator-based approach a lot less noticeable.

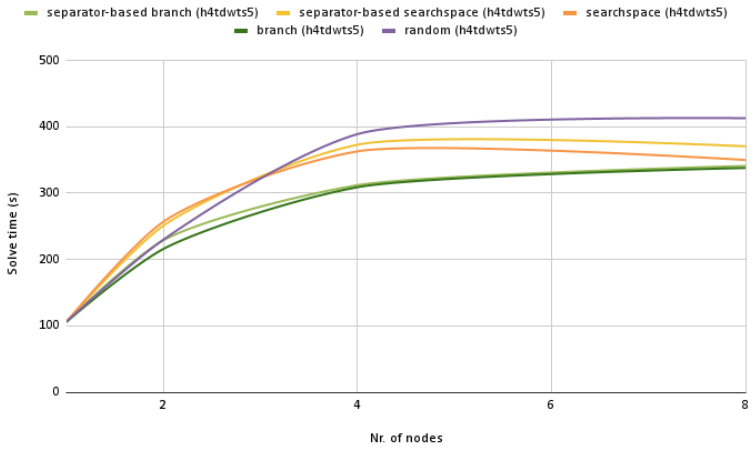

Figure 5.8: Solve times of `TRG-B5-D4-CS5-CC100-C5-S3` for different deployment strategies when increasing the nr. of nodes with a constant total nr. of cores

TRG-B5-D4-CS5-CC100-C5-S3 As can be seen in Figure 5.8, in this instance of the tree-based random graph coloring problem, the random deployment strategy performs the worst once again. The branch deployment strategy performs the best in this instance. This can be explained due to the homogeneous tree structure of the underlying constraint graph. This underlying graph is a tree of clusters, and the amount of clusters and size of the clusters is the same in each of the branches. This leads to the branches having a very similar workload. The branch deployment aims to evenly divide the branches, which for this instance, leads to a deployment where the workload is very evenly divided amongst the machines.

There does not appear to be a significant difference between the separator-based and non-separator based deployments. This is because this instance only has separators lower than the threshold used by this strategy.

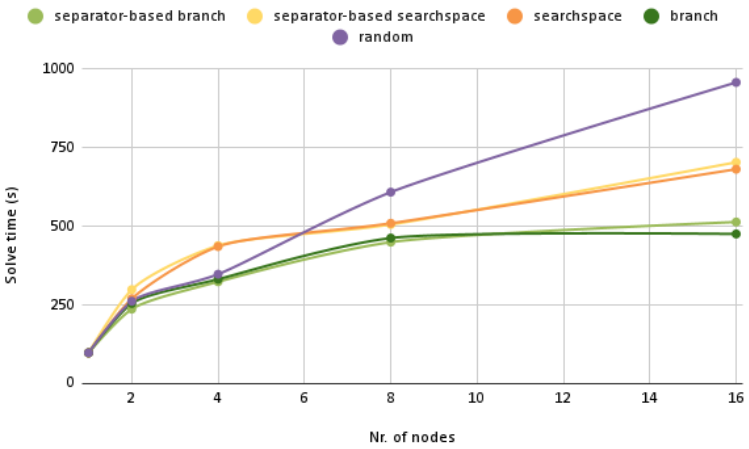

Figure 5.9: Solve times of `TRG-B5-D4-CS4-CC100-C7-S3` for different deployment strategies when increasing the nr. of nodes with a constant total nr. of cores

TRG-B5-D4-CS4-CC100-C7-S3 This is a larger tree-based graph coloring problem consisting of 624 variables. We are solving this problem using the tree-based decomposition method described in Section 5.1.3, because this instance cannot be solved within the 25 minutes time limit using the other decomposition methods.

In Figure 5.9, we can see that initially, out of all deployment strategies, the search space (and the associated separator-based search space) strategy performs the worst, but is overtaken when scaling to more than 6 nodes by the random deployment. The differences between the separator-based and regular deployments is negligible for this instance, because all of the separators in this instance are of size 3 and fall below the threshold the separator-based deployment method uses.

5.5 Evaluating the scalability

For the last experiment, we set out to evaluate the scalability of the developed distributed solver. Additionally, we will also compare the solve time of the implemented BTD algorithm when running locally using a single core to running the Choco solver without our BTD algorithm in order to assess how much performance is gained from leveraging the structural properties of the graph alone without taking parallelism into account.

5.5.1 Setup

Initially, a set of instances is solved on a local version of the Choco solver. This is the same set of instances as the first experiment from Section 5.3. Next, an experiment is done to measure the performance of the distributed BTD algorithm on a single machine with one core. This is done to get a benchmark of the performance gained due to only the leveraging of the structural properties that the BTD algorithm offers, without taking into account parallelism that can be leveraged due to the actor model implementation. Finally, another experiment is done to measure the performance of the distributed algorithm on a single machine with 8 cores, to assess the performance gain of parallelism for each of those instances. H_4S_5-TD-WT is used as the decomposition method, except those instances where this method does not lead to a decomposition that solves the problem within 25 minutes (according to the first experiment), in which case the fastest decomposition method according to the result from Section 5.4.2 is used.

Next to a comparison of the performance against a local solver, we are also interested in evaluating the scalability of the algorithm for increasingly larger cluster sizes in order to evaluate the horizontal scalability. This is done by attempting to solve a relatively large tree-structured instance, `TRG-B5-D4-CS4-CC100-C7-S3`, consisting of 624 variables and 1401 constraints. For this instance, the tree-based decomposition method is used, to take any loss of parallelism as a consequence of a sub-optimal tree decomposition out of the equation.

5.5.2 Results

Instance	Choco	Dist-BTD 1 core	Dist-BTD 8 cores
RG-N21-C10-CC3	1s	84s	105s
RG-N25-C10-CC3	1s	37s	338s
RLFAPSAT-05	1s	3s	2s
RLFAPSAT-11	341s	47s	43 s
RGG-N79-G750-R80-C7	×	226s	295s
RGG-N1000-G2675-R50-C7	×	169s	209s
RGG-N2000-G8000-R35-C10	×	80s	117s
TRG-B2-D3-CS6-CC100-C6-S2	×	6s	3s
TRG-B2-D4-CS7-CC33-C6-S2	×	42s	27s
TRG-B5-D4-CS4-CC100-C4-S3	×	324s	104s

Table 5.3: Results of evaluating local distributed-BTD against local solving.

First, in Table 5.3, a comparison is offered between the solve times of the local Choco solver, distributed BTD on a single core, and distributed BTD on 8 cores. From this, we can conclude that for the unstructured problems (the `RG` instances) and those with a low structure, the local Choco solver manages to solve the instance faster. This is probably due to heuristics and optimizations that the Choco solver can leverage due

to having access to all of the variables and constraints, whereas the BTD algorithm gains very little advantage (if any) from instances with little to no structure.

For the structured instances, we see that the distributed BTD algorithm can solve all of the tested instances, which the local Choco solver cannot solve within the 25 minutes time limit. Furthermore, for those instances which have a high amount of tree-shaped structure (the `TRG` instances), we see considerable speed gains between the 1-core and 8-core distributed BTD algorithm, which indicates that a notable amount of parallelism was achieved. However, for the `RGG` and `RLFAP` instances this speedup is negligible or absent.

Interestingly, for the unstructured instances, the distributed BTD algorithm performs worse on the 8-core configuration than the 1-core configuration. Possibly this is caused by the high amount of backtracking that happens in these unstructured instances, when multiple child nodes are searching for a local solution and one of them does not find a solution. In the 8-core case, these child nodes are searching in parallel and when one of them does not find a solution, the search process in the other children may have already propagated further down (to the grandchildren). Additionally, since actors only process a single message at a time, the message that prevents the other child actors from further propagating the search (i.e. the abort message), cannot be processed until the local search has finished, leading to search work being done unnecessarily. In the 1-core case, only 1 of the child nodes is searching for a local solution at a time, and therefore the other children have not yet propagated the search to their children in case a local solution cannot be found.

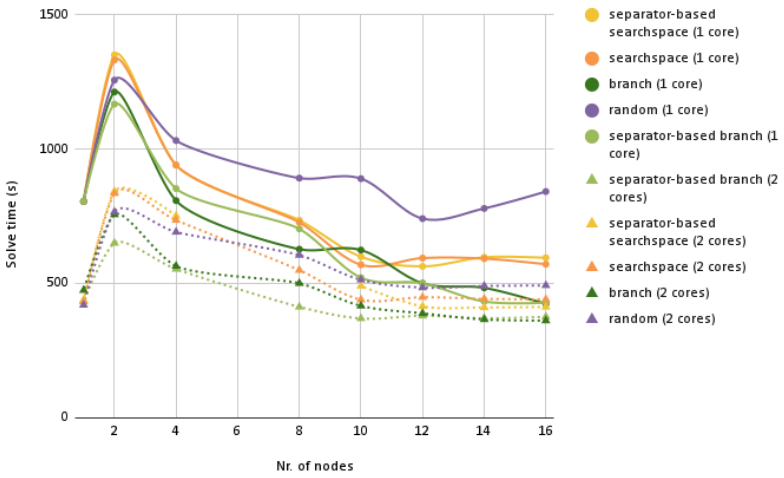

Figure 5.10: Solve times of `TRG-B5-D4-CS4-CC100-C7-S3` when increasing the number of nodes with 1 and 2 CPUs per node

The results of solving the instance of the tree-based random graph coloring problem, `TRG-B5-D4-CS4-CC100-C7-S3`, using the tree-based decomposition method, can be seen in Figure 5.10. It shows, for all of the deployment strategies, the solve time for cluster sizes between 1 and 16 machines, and configurations of 1 and 2 cores per machine. From this, it can be observed that adding more nodes initially leads to a higher solve time, due to the added communication overhead. Adding more nodes, reduces the solve time again, as more computational resources are available in the cluster. This decrease in solve time seems to be roughly linear and stabilizes around a solve time of (for this instance) 500 seconds. The best solve time for the 1 core per machine configuration is the branch deployment, which took 423 seconds. This means a reduction of 47% when compared to the solve time on 1 machine (which was 805 seconds).

The configuration with 2 cores per machine follows a very similar trend as the one core per machine configuration, except the solve times are roughly halved. This is logical, because twice the amount of computational resources are available in the cluster for the same number of machines. The best deployment strategy in this case reduces the total solve time by 14%, considerably less than the 47% of the 1 core per machine configuration. In Figure 5.11, we can see that extra parallelism does not provide more speedup after 12 cores, also not on a single machine. This means there is very little room to make up for the added communication overhead by adding more cores, before this maximum parallelism is reached.

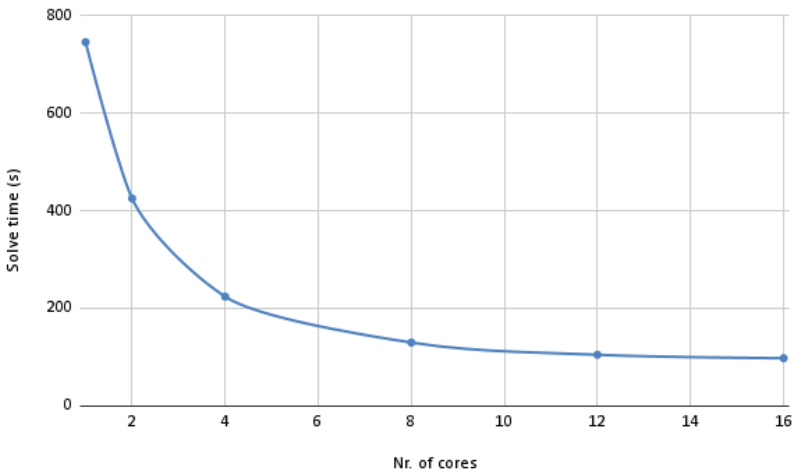

Figure 5.11: Solve time of `TRG-B5-D4-CS4-CC100-C7-S3` on a single machine when increasing the number of cores

Chapter 6

Conclusion

In this thesis, we presented an algorithm for solving constraint satisfaction problems using a distributed cluster of machines. We explored the state of the art algorithms, such as embarrassingly parallel search, multi-agent search and the BTD algorithm. Out of all of these, we decided to use the BTD algorithm as the basis for a distributed implementation. In this thesis, an adapted version of the BTD algorithm was designed and implemented using the actor model. Furthermore, the algorithm was adapted to solve constraint optimization problems as well, for which the BTD algorithm was not originally designed.

The CSP (or COP) is divided into subproblems by means of tree decomposition. We implemented and compared several heuristic algorithms for computing a tree decomposition. We found large differences (up to a 100 times) in the solve time between the different decomposition algorithms when using the produced tree decompositions for solving CSPs/COPs using the distributed BTD algorithm, both in terms of number of instances which could be solved (within 25 minutes) and the time required to solve these instances.

We have done an experimental evaluation of the scalability of the implemented algorithm for several instances of structured CSPs and COPs. From the experiments, it becomes clear that using the algorithm on a cluster of machines adds a considerable amount of communication overhead when compared to running the algorithm on a single machine. A linear reduction in solve time can be observed when continuing to increase the number of machines in the cluster up to some upper-bound. When this upper-bound for the number of machines in the cluster is reached, adding more machines to the cluster no longer decreases the solve time. In the tested instances, the scalability achieved by parallel execution of the branches in the tree decomposition is limited, with an observed upper-bound of at most 16 cores. Taking into account the added communication overhead, a decrease in solve time (when compared with a

single machine) is only observed when scaling to more than 4 machines (in the case of 1 core per machine). This makes the effective cluster size (i.e. the size of the cluster where a speedup can be expected when adding more machines) quite small, namely between cluster sizes of 4 and 16 machines. This range is made even smaller when increasing the number of cores per machine.

The design, reference implementation and the experiments performed using the developed distributed BTD algorithm allow us to answer the first and main research question. We can conclude that a CSP can be solved using a distributed cluster of machines by decomposing the CSP using a tree decomposition and using a distributed version of the BTD algorithm (based on the actor model) for the solving process, although the scalability of this solution is limited by the structural properties of the CSP.

In this thesis we also did exploratory research and an experimental comparison of different deployment strategies. Random, branch and search space deployment strategies were implemented. Additionally, *separator-based* variants of the branch and search space strategies were designed, where pairs of bags with a large separator between them are deployed to the same machine. From the experiments, it becomes clear that the random deployment, in general, has the worst performance. The branch deployment and separator-based branch deployments perform the best overall, especially for the tested tree-shaped random graph coloring instances. Out of all instances, the search space strategy (and the associated separator-based search space strategy) has the most stable performance when changing the number of machines in the cluster. In all but one of the tested instances, the separator-based variants of the branch and search space deployment performed equal or better than their regular counterparts, with a speedup of up to 24%.

Chapter 7

Future Work

In this work, parallelism is achieved during the phase of propagating the search process from the parent to the child actors. Every time a parent actor with more than one child sends an assignment to all of its children, the parallelism increases. However, another possible opportunity exists for adding parallelism: instead of a tree node actor waiting for the responses from all of its children after propagating the search to them, a tree node actor could immediately continue with finding a subsequent local solution and also propagate that one. This has the potential to massively increase the amount of searching being performed in parallel. However, a possible downside is that parts of the search space may be visited which would otherwise have been pruned due to the constraint on the objective function. Research is needed to determine whether the extra parallelism would lead to finding the solution faster despite potentially searching some parts of the search space that would otherwise have been pruned.

In the experiments, several heuristic tree decomposition algorithms were compared by using them in combination with the developed distributed BTD algorithm. We found large differences in the solve times, not just between different decomposition methods, but also for solving different instances using the same method. More research is needed to understand the impact of different properties of the tree decomposition on the solve time, especially when solving in parallel, and how an algorithm can be designed which finds tree decompositions having favorable properties for parallelism.

Several different deployment strategies were implemented and compared experimentally in this work. However, these do not cover all possible strategies and other strategies for deployment of the bags exist that may potentially lead to a shorter solve time. Additionally, all the developed strategies were static in the sense that all bags remained deployed on the same machine during the entire search process. It is hard to predict beforehand which bags in the tree decomposition will generate the most workload, so potentially a dynamic deployment, where bags are moved

to different cluster nodes after the initial deployment, is another possible avenue of research.

In this thesis, the scope of the experiment was limited to relatively small problems, which can be solved in under 25 minutes. Testing the algorithm for larger problems, which have a larger tree decomposition consisting of more bags, and for which it takes longer to find a solution, could potentially yield interesting results. For example, the optimal deployment strategy for a larger problem could be different or it can become more clear which strategy is the best. In particular, it would be interesting to see the performance of the distributed BTD algorithm for the optimization cases of the RLFAP instances. Furthermore, all problems tested in the experiments contain only binary constraints. For problems with higher-arity constraints, a tree decomposition could potentially be less optimal, as some of this information gets lost when creating the primal constraint graph on which the tree decomposition is based, because 3 binary constraints cannot be distinguished from a single ternary constraint on the same variable. Further testing with problems containing higher-arity constraints is needed, to evaluate if the performance is impacted by this.

The experiment was performed on the Peregrine high-performance cluster. This cluster makes uses of a very fast Infiniband network between the different machines that make up the cluster. A consequence of this is that the networking overhead for the communication between nodes is lower than it may be in other types of clusters, such as clusters in a private or public cloud environment. This could have a major impact on which deployment strategy performs the best. The impact of the deployment strategy comes in two different ways: impact on the parallelism/workload (when one machine is very busy and another is idle), and impact on the communication overhead (when placing bags that communicate a lot on different machines). Due to the fast inter-node networking speeds in the experimental setup from this thesis, the impact on the communication overhead is relatively low compared to the impact on workload. Therefore, the results may not be fully representative of every compute cluster, and may not carry over to environments with a slower network. For this reason, it is necessary to do more research and experiments for setups with a slower internal network.

Acknowledgements

First and foremost, I would like to thank my supervisor, Michel Medema, for all of his valuable input, feedback and guidance throughout the project.

Furthermore, I would like to thank Prof. Dr. Alexander Lazovik for his ideas and feedback, not just during this thesis, but throughout my entire studies.

I want to thank the Center for Information Technology of the University of Groningen for providing access to the Peregrine high performance computing cluster, without which the experiments of this thesis would not have been possible.

And lastly, but not least, I would like to thank all of my family, friends and colleagues, for providing me with the support and motivation needed to complete my studies.

List of Figures

List of Tables

Bibliography

[1] J. Pearson and P. G. Jeavons, "A survey of tractable constraint satisfaction problems," tech. rep., Technical Report CSD-TR-97-15, Royal Holloway, University of London, 1997.

[2] P. Jégou and C. Terrioux, "Hybrid backtracking bounded by tree-decomposition of constraint networks," *Artificial Intelligence*, vol. 146, no. 1, pp. 43–75, 2003.

[3] J. Régin, M. Rezgui, and A. Malapert, "Embarrassingly parallel search," in *International conference on principles and practice of constraint programming*, pp. 596–610, Springer, 2013.

[4] G. R. Kip, "Optimal Deployment of Actors in a Distributed Setting." `http://fse.studenttheses.ub.rug.nl/id/eprint/26124`, 2021.

[5] P. Jégou, S. N. Ndiaye, and C. Terrioux, "Computing and exploiting tree-decompositions for solving constraint networks," in *International Conference on Principles and Practice of Constraint Programming*, pp. 777–781, Springer, 2005.

[6] R. Dechter, "Constraint networks," pp. 276–285, 1992.

[7] G. Gottlob, N. Leone, and F. Scarcello, "A comparison of structural CSP decomposition methods," *Artificial Intelligence*, vol. 124, no. 2, pp. 243–282, 2000.

[8] R. Dechter and J. Pearl, "Tree clustering for constraint networks," *Artificial Intelligence*, vol. 38, no. 3, pp. 353–366, 1989.

[9] E. C. Freuder, "A sufficient condition for backtrack-bounded search," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 755–761, 1985.

[10] N. Robertson and P. Seymour, "Graph minors. II. Algorithmic aspects of tree-width," *Journal of Algorithms*, vol. 7, no. 3, pp. 309–322, 1986.

[11] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3, p. 235,

Stanford Research Institute Menlo Park, CA, 1973.

[12] I. Greif, *Semantics of communicating parallel processes.* PhD thesis, Massachusetts Institute of Technology, 1975.

[13] T. Schiex and G. Verfaillie, "Nogood recording for static and dynamic constraint satisfaction problems," *International Journal on Artificial Intelligence Tools*, vol. 3, no. 02, pp. 187–207, 1994.

[14] A. Malapert, J. Régin, and M. Rezgui, "Embarrassingly parallel search in constraint programming," *Journal of Artificial Intelligence Research*, vol. 57, pp. 421–464, 2016.

[15] S. B. Dersarkissian and A. Malapert, "Flexibility and Portability for Embarrassingly Parallel Search," tech. rep., EasyChair, 2020.

[16] S. H. Clearwater, B. A. Huberman, and T. Hogg, "Cooperative solution of constraint satisfaction problems," *Science*, vol. 254, no. 5035, pp. 1181–1183, 1991.

[17] I. P. Gent, C. Jefferson, I. Miguel, N. C. Moore, P. Nightingale, P. Prosser, and C. Unsworth, "A preliminary review of literature on parallel constraint solving," in *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, pp. 499–504, 2011.

[18] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, "The distributed constraint satisfaction problem: Formalization and algorithms," *IEEE Transactions on knowledge and data engineering*, vol. 10, no. 5, pp. 673–685, 1998.

[19] R. Bejar, C. Domshlak, C. Fernández, C. Gomes, B. Krishnamachari, B. Selman, and M. Valls, "Sensor networks and distributed CSP: communication, computation and complexity," *Artificial Intelligence*, vol. 161, no. 1-2, pp. 117–147, 2005.

[20] A. Meisels and O. Lavee, "Using additional information in DisCSP search," in *Proc. 5th workshop on distributed constraints reasoning, DCR*, vol. 4, Citeseer, 2004.

[21] I. P. Gent, I. Miguel, P. Nightingale, C. McCreesh, P. Prosser, N. C. Moore, and C. Unsworth, "A review of literature on parallel constraint solving," *Theory and Practice of Logic Programming*, vol. 18, no. 5-6, pp. 725–758, 2018.

[22] G. Gottlob, N. Leone, and F. Scarcello, "Hypertree decompositions and tractable queries," *Journal of Computer and System Sciences*, vol. 64, no. 3, pp. 579–627, 2002.

[23] K. Liu, S. Löffler, and P. Hofstedt, "Using hypertree decomposition for parallel constraint solving," *INFORMATIK 2017*, 2017.

[24] C. Prudhomme, J.-G. Fages, and X. Lorca, "Choco solver documentation," *TASC, INRIA Rennes, LINA CNRS UMR*, vol. 6241, 2016.

[25] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo, "Adopt: Asynchronous distributed constraint optimization with quality guarantees," *Artificial Intelligence*, vol. 161, no. 1-2, pp. 149–180, 2005.

[26] M. Bannach, S. Berndt, and T. Ehlers, "Jdrasil: A Modular Library for Computing Tree Decompositions," in *Experimental Algorithms 16th International Symposium, SEA 2017, London, England, June 21 23, 2017, Proceedings*, 2017.

[27] H. L. Bodlaender and A. M. Koster, "Treewidth computations i. upper bounds," *Information and Computation*, vol. 208, no. 3, pp. 259–275, 2010.

[28] P. Jégou, S. N. Ndiaye, and C. Terrioux, "Computing and exploiting tree-decompositions for (max-) csp,"

[29] P. Jégou, H. Kanso, and C. Terrioux, "Towards a dynamic decomposition of CSPs with separators of bounded size," in *International Conference on Principles and Practice of Constraint Programming*, pp. 298–315, Springer, 2016.

[30] P. Jégou, H. Kanso, and C. Terrioux, "An algorithmic framework for decomposing constraint networks," in *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 1–8, IEEE, 2015.

[31] R. Samudrala and J. Moult, "A graph-theoretic algorithm for comparative modeling of protein structure," *Journal of molecular biology*, vol. 279, no. 1, pp. 287–302, 1998.

[32] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[33] P. Jégou and C. Terrioux, "Combining restarts, nogoods and bag-connected decompositions for solving CSPs," *Constraints*, vol. 22, no. 2, pp. 191–229, 2017.

[34] V. Jarník, "O jistém problému minimálním," 1930.

[35] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[36] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[37] P. Jégou and C. Terrioux, "Tree-decompositions with connected clusters for solving constraint networks," in *International Conference on Principles and Practice of Constraint Programming*, pp. 407–423, Springer, 2014.

[38] J. Dall and M. Christensen, "Random geometric graphs," *Physical review E*,

vol. 66, no. 1, p. 016121, 2002.

[39] M. Penrose *et al.*, *Random geometric graphs*, vol. 5. Oxford university press, 2003.