university of groningen / faculty of science and engineering / bernoulli institute

Backtracking with Tree Decomposition for Solving Constraint Optimization Problems

Bachelor's Thesis

Tran Duy Hieu Nguyen, s3751147, t.d.h.nguyen.1@student.rug.nl
Supervisors: Prof. A. Lazovik & Mr. M. Medema

July 11, 2022

Abstract:

Solving Constraint Satisfaction Problems (CSP) and its sibling the Constraint Optimization Problems (COP) is a prominent research subject in Computing Science in general and in Artificial Intelligence in particular. A notable technique for solving CSPs is using the decomposition of the problem to divide it into independent sub-problems. The algorithm Backtracking with Tree Decomposition (BTD) was developed to solve CSP using the tree decomposition of the constraint network between the variables. In this thesis, we explored the idea of using BTD as a method to solve optimisation problems. We designed and implemented the BTD* algorithm which performs recursive search on the tree decomposition of the constraint network. Following the structure of the tree decomposition, BTD* tries to find the optimal solution to the problem by recursively optimizing the cost of the subtrees until it has determined that no other solution can be more optimized than the one last found, or until it has proven that the problem is unsatisfiable. We then made comparisons between BTD* and the existing COP solver Choco by performing benchmarks using 96 instances selected from the website of the XCSP3 framework. Based on these comparisons, we determined that the performance of BTD* is not comparable to that of the Choco solver, both in terms of time complexity and space complexity. Even though our algorithm and implementation is not entirely efficient, it is still a good starting point for future researches of BTD-like algorithm for solving COP.

Acknowledgement

Before we start with the thesis, I wish to extend my gratitude to all my families and friends whom have supported me through out the project. You all have made the process easier with your kindness.

I would like to thank my supervisors Prf. Dr. Lazovik and Mr. Medema for their instructions and guidance, as well as their patience. Especially to Mr. Medema, with whom the weekly meetings and towards the end of the project, bi-weekly meeting are has helped me progressed so much.

1 Introduction

The formalism Constraint Satisfaction and Optimization Problems (CSP and COP) are powerful frameworks that can represent many academic as well as real life problems. For example, the Eight Queen puzzle is an academic problem that can be modelled using CSP, whereas real life budgeting can be represent by a COP. Finding a solution efficiently for constraint satisfaction problems (CSP) and constraint optimization problems (COP) has always been an important research subject for Artificial Intelligence and Computing Science. This is because problems that can be represented by a CSP or COP tends to have a large search space, thus solving CSP and COP are NP-Complete. One of the known set of techniques for this is decomposition techniques. Taking advantage of the constraints between variables in a CSP, decomposition techniques try to divide a CSP into smaller and more manageable sub-problems. Generally when using this divide and conquer method, after decomposition the original problem has a lower worst case complexity and thus making it more likely that finding a solution would be more efficient. One of the notable method for decomposing a constraint problem is Tree Decomposition. Tree Decomposition separate the problems into smaller sub-problems that are connected via a tree. Algorithms can this navigate the search space following the structure of this tree and exploits certain characteristic of it to lessen the workload.

The Backtracking with Tree Decomposition (BTD) algorithm was developed to solve CSP following this very idea[6]. The algorithm uses standard backtracking search on the tree decomposition of the structure of the constraint network, and at the same time exploiting this structure to prune the search space. Proven to be useful for solving CSP, though out time BTD has been developed further and further as a CSP solving algorithm, with a few work that present BTD or tree decomposition as a method for solving Weighted-CSP. This begs the question of how would an algorithm based on BTD that solves a COP be defined and how would it behave? In this thesis, we explored the idea of altering the BTD algorithm and design a BTD-based COP solving algorithm, named **BTD***. After having defined and implemented the algorithm we compared the results produced by our BTD* algorithm with a well known open-source Constraint Programming solver, the Choco solver.

This thesis is organised as follows. Section 2 contains background information on CSP and COP, the Tree Decomposition method, and the BTD algorithm and framework. In section 3, we address several related works to give a clear view of our position with regards to the current state of the art. Section 4 explains the conceptual definition of BTD* and some design choices that was made during the implementation process of BTD*. In section 5, we show the result of the BTD* implementation, using benchmark tests sourced from available tests on the website www.xcsp3.org/instances/ as well as discuss these results along with making comparison with the results produced by the Choco solver. Section 6 is where we draw our conclusion for the thesis and possible future improvement for BTD_WMC is described in section 7. Finally, the bibliography finalised the paper.

2 Background Information

In this section, we will go over information and definitions that are deemed necessary for the best understanding of important parts of the thesis paper. The definitions are that of: CSP, COP, Tree Decomposition, and the BTD algorithm. We will also mention some information regarding the BTD framework.

2.1 Constraint Satisfaction and Optimization Problems

These definitions regarding CSP are taken almost verbatim from that of S.J.Russell, P. Norvig, et al. in [10].

A **CSP** can be defined as a triplet, consisting of three components, $X, D,$ and C. The definition of these components are as follows:

$X = \{X_1, \ldots, X_n\}$ is the set of variables in the problem,

$D = \{D_1, \ldots, D_n\}$ is the set of domains, each variable possess one domain.

$C = \{C_1, \ldots, C_m\}$, is the set of constraints that specify permissible combinations of values.

A domain D_i in D is a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. This means that only values chosen from the domain D_i can be used to assign to the variable X_i. A constraint $C_i \in C$ consists of a pair $\langle S, r \rangle$. S is a tuple of variables that participate in the constraint, it can also be referred to as the *scope* of the constraint; r is a relation that defines the combinations of values that are *valid* with respect

to the constraint (i.e. *satisfies* the constraint) that can be assigned to variables in S. Should a CSP only have constraints of arity two, it is known as a binary CSP. A **solution** to a **CSP** needs to be a **consistent** and **complete** assignment of values to the variables in the set X. A consistent assignment does not infringe on any of the constraints in C, i.e. no constraints are violated, whereas an assignment in which all of the variables in X are assigned is called a complete assignment. A CSP can have either 0, 1, or multiple solutions; it would have no solution in the case of no complete assignments to X are consistent to C.

A **COP** can be defined as a 4-tuple $\langle X, D, C, O \rangle$, with $X, D,$ and C defined exactly as in a CSP, whereas the objective O is defined as a pair consisting of a *goal* and an objective function $obj()$. The objective function $obj()$ computes the **cost** of a assignment to X by mapping it to a real value. It can also compute the cost of assignments to **individual** variables in X. $obj()$ computes the cost of a complete assignment by summing up the cost of the individual assignments i.e.:

$$obj() = \sum_{i=1}^{n} obj(x_i \leftarrow v_{x_i}) \text{ with } x_i \in X \text{ and } V_{x_i} \in D_x.$$

This implies that $obj()$ can compute the cost of partial assignments as well. The *goal* specifies how the value of $obj()$ should be **optimized**, either maximized or minimized, by a **solution** to the COP. A solution to the COP is a **complete** assignment to X and it is **consistent** to C while at the same time optimize the value of $obj()$. Similar to a CSP, a COP can have 0, 1, or more than 1 solutions. A COP can have multiple solutions because there could be multiple complete and consistent assignments to X which all optimize obj.

The algorithm we defined in this paper was made with the a few **assumptions** regarding the Objective O. These assumptions are:

- Instead of assignments to real values, $obj()$ maps assignments to **integer** values.
- For every variable in X, when assigned a value from its domain, would never have a negative cost, i.e.:

$$\forall x \in X \, (\, \forall v \in D_x \, (\, obj(x \leftarrow v) \geq 0 \,) \,)$$

- The cost of assigning a variable x with value v is computed as:

$$\forall x \in X \, (\, \forall v \in D_x \, (\, obj(x \leftarrow v) = v * c_x \,) \,)$$

  where c_x is the coefficient of x in the Objective.

- The *goal* can be optimizing the cost of a **subset** of X, this subset is to be referred to as the *scope* of the Objective.

2.2 Tree Decompostion

A CSP can be represented by a hypergraph H (or just a graph G for binary CSP) where each variable in the CSP is a vertex in H (or G) and each constraint scope of the CSP is a hyper-edge in H (or an edge in G). We need to first acknowledge the fact that Tree Decomposition is a notion for **graph** and not hypergraph. However, any hypergraph can be represented by a primal graph (also known as clique-graph or 2-section graph) thus Tree Decomposition can be performed on this primal graph instead. A primal graph G of a hypergraph H possesses the same vertices as H, an edge in G connects two vertices if they are connected by a hyperedge in H. After having formally defined tree decomposition, an example of a primal graph of a hyper graph would be shown.

As described in [8], a tree decomposition of a **graph** can be defined as follows: Let $G = (X, C)$ be a graph with X being the set of vertices and C being the set of edges, note that we use the name G, X, and C to relate to a binary CSP. A tree decomposition of G is a pair (E, T) in which $T = (I, F)$ is a tree where:

I is the set of nodes and

F is the set of edges of T,

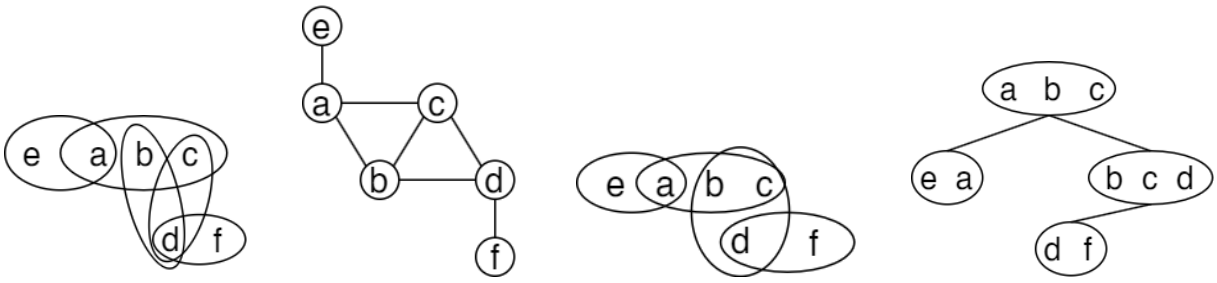
E is $\{E_i : i \in I\}$ a family of subsets (also called clusters) of X

E_i is a node of T and satisfies all three conditions:

- (i) $\cup_{i \in I} E_i = X$,
- (ii) For each edge $x, y \in C$, there exists $i \in I$ so that $x, y \subseteq E_i$, and
- (iii) For all $i, j, k \in I$, if k is in a path from i to j in T, then $E_i \cap E_j \subseteq E_k$

The **width** of a tree decomposition (E, T) is determined to be the size of the largest cluster minus one, i.e. $max_{i \in I}|E_i| - 1$. The *tree-width* w of G is the minimal width over all tree-decompositions of G. An **optimal** tree decomposition is one whose width is equal to w, however, to compute an optimal tree decomposition is an NP-hard problem [8]. Therefore in practice, the tree decomposition computed would have the width $w^+ \geq w$ referred to as an approximation of w. Among the vast amount of works that focus on computing a tree decomposition that minimize w^+ as much as possible, an algorithm that can effectively compute a tree decomposition of a constraint network while at the same time allowing the user to choose certain heuristics to guide the decomposition process was created by P. Jégou et. al. [8]. This algorithm is named Heuristic Tree-Decomposition-Without Triangulation (H-TD-WT). H-TD-WT proves to be effective both in terms of computing time and in terms of the quality of the generated tree decomposition [8].

An example of creating a tree decomposition of a CSP is to first create the primal graph G from the hypergraph H of said CSP. The next step is to create an **acyclic** hypergraph H' from G. Afterwards, simply select an edge of H' to be the root cluster, leaving other edges to be other clusters in the tree, and a tree decomposition is found.



(a) The hypergraph H of (X, C). **(b) The primal graph G of H.** **(c) An acyclic hypergraph H' of G.** **(d) A possible tree decomposition of (X, C)**

Figure 1: An example of steps for creating a tree decomposition of a hypergraph.

Let X be $\{a, b, c, d, e, f\}$ and C be $\{a \leq b \leq c, a \neq e, b \neq d, c \neq d, d \neq f\}$, an illustration of these steps can be found in Figure 1. Figure 1a is the hypergraph H that represent (X, C), 1b is the primal graph G of H, 1c is a possible acyclic hypergraph H' of G, and 1d is the tree decomposition if the edge containing $\{a, b, c\}$ of H' was chosen to be the root cluster.

It is obvious that a parent cluster and a child cluster have shared variables, such sets of shared variables between two clusters are called **separators**. These separators play an important role in the BTD algorithm as well as the BTD* algorithm.

2.3 Backtracking with Tree Decomposition

In 2003, the work of P. Jegou and C. Terrioux on an original framework for solving CSP was published [6]. This is the Backtracking with Tree Decomposition (BTD) algorithm.

The BTD algorithm is described to be a hybrid algorithm, since as the name suggest, BTD solves a CSP based on both backtracking techniques and the notion of tree decomposition. Assuming a tree decomposition of the constraint network has been computed and is available to BTD. It then uses the tree decomposition as a partial variable order to perform enumerative search by always instantiating the variables in the parent cluster first before the ones in the children clusters. BTD assigns value to variables in a cluster $\mathcal{C}_i$ in a certain order that can be decided with the help of a heuristic. When assigning variables in the cluster, BTD of course will try to maintain the consistency of the assignments. Should any variable can not be assigned due to constraint violation, BTD will perform Backtrack and change the current instantiation of the variables either in $\mathcal{C}_i$ or in a ancestor of $\mathcal{C}_i$. Once $\mathcal{C}_i$ has no more variable unassigned, BTD will select a child $\mathcal{C}_j$ of $\mathcal{C}_i$ to continue with the search process by instantiating variables in $\mathcal{C}_j$. Should $\mathcal{C}_i$ have no children to explore, BTD will move to a sibling of $\mathcal{C}_i$ and continue with the search there.

In order to prune the search space, BTD exploit the fact that a cluster $\mathcal{C}_i$ and its child $\mathcal{C}_j$ have a separator between them, which is a set of shared variables. The variables in the separators will be

instantiated before the ones in $\mathcal{C}_j$ and the assignments in $\mathcal{C}_j$ is dependent on the assignments in the separator. Due to this, when BTD has determined certain assignments to the separator will lead to consistent assignments to both $\mathcal{C}_j$ and Descendant($\mathcal{C}_j$) or will definitely lead to inconsistency either in $\mathcal{C}_j$ or Descendant($\mathcal{C}_j$), it will record these assignment in separate sets to later on check if the separator of $\mathcal{C}_i$ and $\mathcal{C}_j$ has been assigned with one of the "(in)consistent-guaranteed" assignments. *Goods* and *nogoods* are, respectively, the name for the consistent-guaranteed and the inconsistency-guaranteed assignments. By recording these *goods* and *nogoods*, BTD can "skip" certain parts of the search space when a *(no)good* is encountered. This is the notion of *structural good* and *structural nogood*[6]. Should the assignments of the separator of $\mathcal{C}_i$ and $\mathcal{C}_j$ is a *good*, BTD will move on to another child of $\mathcal{C}_i$, if there is still an unexplored sibling of $\mathcal{C}_j$. If a *nogood* is encountered, BTD will perform backtracking and change the current assignment, either in $\mathcal{C}_i$ or an ancestor of $\mathcal{C}_i$. In the case of the assignments on separator not being a *good* or *nogood*, BTD will continue the search on to the subtree rooted at $\mathcal{C}_j$ and later on record the newly found *good* or *nogood* accordingly.

Immediately, one can see that BTD uses Chronological Backtracking to change the current assignments should the search process found inconsistency. This method of backtracking is known to be inefficient in practice. However, in [6], BTD is also implemented with extensions that took this into account. Experiments in [6] shows results found when BTD is implemented with extensions either using filtering techniques based on filtering algorithms (Forward Checking - FC, Maintaining Arc Consistency - MAC), non-chronological backtracking (Backjumping - BJ), or both. It was shown that for solving classical random CSPs, FC-BTD and MAC-BTD respectively are comparable to the classical algorithms FC and MAC. When it comes to solving structured random CSPs, FC-BTD-BJ and MAC-BTD-BJ are respectively significantly faster than FC and MAC due to the ability of BTD to exploit *(no)goods*. In term of real word instances, BTD produced better or comparable results when compared to the classical algorithms. [6] also showed that when it come to required space, BTD was proven usable in practice unlike Tree-Clustering[5], which is another decomposition method that was deemed too costly memory wise.

For this project, BTD* was built using the available resources in the source code of the BTD framework that was made available to us. The source code is written in C++ and uses a variant of the XCSP3 Core Parser in C++ that parses the XML format of CSP/WCSP/COP instances named XCSP 3.0 [3] [2]. It has an implementation of the algorithm H-TD-WT with the heuristic H5 [7] for computing a tree decomposition and several heuristics that aides the solver in making "choices" while solving, as well as a filtering system to filter values from domains as the search progresses. More information on the XCSP3 framework and parser developed at the Computer Science Research Institute of Lens (CRIL) at University of Artois can be found on the website: xcsp.org.

3 Related Works

Before discussing our design and implementation of the BTD* algorithm, we need to address the related work that surround this thesis to position our work w.r.t the current state of the art, as well as refer to certain work we will be using for the thesis.

Let us first take a look at a standard technique for solving COPs using backtracking search known as the Branch and Bound algorithm (BB). Unlike BTD* which performs recursive search, BB perform iterative search instead. The way searching works for BB is to build up partial assignments in a depth-first manner while exploiting bounds to prune the search space. Given a COP $\mathcal{C}$ being a reduction of the original COP, and the goal is to minimize the objective function. BB takes in $\mathcal{C}$ and an upper bound value UB. It then first approximates a lower bound and upper bound for the cost of the best solution to $\mathcal{C}$. Afterwards, BB chooses a variable x to assign and tries all possible values for it. With each of these tries, BB will start a "new" search on the further reduced $\mathcal{C}$ with a tightened UB using the smaller between UB and the approximated upper bound, subtracting the cost of assigning x. The bounds returned by these tries are then used to tighten the approximated bounds even more before returning them. In the beginning of each search, should the approximated lower bound be greater than or equal to UB, BB will cancel the search and perform backtracking since the partial solutions at hand will not lead to a better solution, effectively pruning the search space. When BB has finished searching, the returned bounds would have the same value being the minimum cost of $\mathcal{C}$. Though not similar to the way BB works, in section 4 we will see that BTD* would also be exploiting bounds during search to prune the search space should the goal of the objective be to minimize the objective function.

Secondly, the algorithm Russian Doll Search with tree decomposition (RDS-BTD) in [11] that uses BTD to solve the subproblems induced by RDS when solving a Weighted CSP (WCSP). WCSPs can be described as CSPs in which violation of constraints is allowed and has a certain penalty cost. Solving

WCSP generally means minimizing this penalty cost. Even though RDS-BTD is designed to solve WCSP instead of COP, it is still an expansion of BTD to solve problems involving cost and bounding which makes it related to our work. This algorithm was implemented in the toulbar2 solver and competed in the Max-CSP competition 2008 for solving WCSPs hosted by CRIL. It won first place and performed much better than other solvers having solved 89% of the instances while the others solved at most 70% instead.* This result further suggests research on using BTD as a method for solving COPs should be carried out. Still, at the time of research for this project, there seemed yet to be an algorithm or solver that is a variation of BTD or use BTD as a method for solving COP.

The closest work to using BTD or tree decomposition as a method to solve COP is one by M.Kitching and F. Bacchus in [9]. In this work, an iterative search algorithm that is designed to solve a COP is proposed. This algorithm is named OR-Decomp. OR-Decomp searches like BB, i.e by bounding until an optimal cost is found, but it takes in a decomposed problem instead of a reduced problem. The input problem is decomposed into components and OR-Decomp has the freedom to branch on any unassigned variable of any active components. Similar to BTD, this algorithm can also exploit tree decomposition as a partial variable ordering. However, this algorithm can be allowed to not follow the ordering imposed by the tree decomposition should another variable can be deemed more promising. This is so since it has the choice to branch on any unassigned variable of any active components and once all the variables in a cluster are assigned, all of its children became active components. Therefore OR-Decomp can essentially switch between siblings cluster to assign variables. Adding to that, OR-Decomp is an iterative search algorithm, which is very different from the recursive search nature of BTD*. Even though OR-Decomp is designed to solve COP, experiments on OR-Decomp in [9] were performed with WCSP instead and therefore the result does not really translate to the effectiveness of OR-Decomp when solving COPs.

The latest extension of BTD, the algorithm BTD-MAC+RST+Merge [7], should also be mentioned. This algorithm exploits restart and merging clusters to improve BTD, it also uses Generalised Arc Consistency (GAC) to aid the solving process by filtering out value from domains to prune the search space. BTD-MAC+RST+Merge can restart the search to start with a different root cluster setting a new partial variable ordering to follow. Not only that, it can also update dynamically the tree decomposition depending on the need of the solving by merging clusters when needed. This merging method allows BTD-MAC+RST+Merge to change the variable ordering even more since variables in a child cluster after having been merged with its parent would be instantiated before the one in the siblings clusters prior to merging. We can see that BTD has been developed further and more complex for solving CSPs, but yet to be extended to solve COPs. While the BTD* algorithm does not exploit restart or merging, it does use GAC enforced by a propagation-based system exploiting events that was implemented for BTD-MAC+RST+Merge in the source code for BTD we received for this project.

Finally, we must mention the Heuristic-Tree Decomposition-Without Triangulation algorithm for computing a tree decomposition. This algorithm operates without triangulation of the graph to find cliques and then compute clusters. It instead computes set of clusters by traversing the graph using properties related to separators along with their associated connected components [8]. Implemented with different heuristic, H-TD-WT can affect the solving progress differently with the tree decomposition it produces. H-TD-WT is also implemented in the source code for the BTD framework which we received, however it is implemented with the heuristic H5 described in [7]. We find that this heuristic is not suitable for computing a tree decomposition to solve a COP and thus decided to implement a different heuristic.

4 Design and Implementation

This section goes over the design of the BTD* algorithm and reasons for the design choices made regarding BTD* and the heuristics chosen.

4.1 Definition of BTD*

We choose to define BTD* to a be a recursive search algorithm that follows a tree decomposition of the constraint network. This means that BTD* will recursively optimize each subtrees in the tree decomposition in order to optimize the entire tree and thus find an optimal solution if the constraint network is satisfiable. We opted to define BTD* this way to explore the effectiveness of recursive search using tree decomposition for solving COPs.

The way BTD* performs search is as follows. Assuming we have a consistent assignment from the root cluster of the cluster tree to the current cluster $\mathcal{C}_i$ and no variables in $\mathcal{C}_i$ has been assigned. BTD* will

*As shown on the website of the Max-CSP 2008 competition: https://www.cril.univ-artois.fr/CPAI08/

first try to consistently assign all variables in $\mathcal{C}_i$ one by one until $\mathcal{C}_i$ has no unassigned variable left. Then, BTD* proceeds to a child cluster $\mathcal{C}_j$ of $\mathcal{C}_i$ to recursively search for an optimal solution to the subtree rooted at $\mathcal{C}_j$ if the assignments on the separator between $\mathcal{C}_i$ and $\mathcal{C}_j$ is not a *(no)good*. If BTD* succeeded in optimizing the subtree of $\mathcal{C}_j$, meaning the optimal solution to this subtree has been found, BTD* will record a good on the separator between $\mathcal{C}_j$ and $\mathcal{C}_i$; BTD* will so record the solution to $\mathcal{C}_j$ along with the *good* it has just recorded. Should BTD* determined that the subtree rooted at $\mathcal{C}_j$ does not have a solution, i.e. leads to inconsistencty, it will record a *nogood* on the separator between $\mathcal{C}_j$ and $\mathcal{C}_i$ then change the current assignment on $\mathcal{C}_i$ to start searching again. If BTD* has found an optimal solution to all the subtrees of the children of $\mathcal{C}_i$, it will computes the cost of the subtree rooted at $\mathcal{C}_i$ and compare this to the last know best solution of this subtree and then update the last known best solution with the newly found solution to the subtree at $\mathcal{C}_i$ if needed. BTD* will perform a backtrack and change the assignments on $\mathcal{C}_i$ and repeat the whole process. If BTD* found that $\mathcal{C}_i$ has no variables left that can be reassigned, it will return the last known best solution of the subtree rooted at $\mathcal{C}_i$ since this the most optimal solution to the subtree w.r.t the assignments on the separator between $\mathcal{C}_i$ and its parent cluster. Another important detail about how BTD* works is that when it is exploring the children of a cluster $\mathcal{C}_i$, if BTD* encountered a *good* for the child cluster $\mathcal{C}_j$ it will reconstruct the most optimal solution to the subtree rooted at $\mathcal{C}_j$. This is made possible by the fact that when BTD* records a *good*, it also record the assignments to the child cluster induced by the *good*. Therefore, it can reconstruct the optimal solution to the subtree by first reconstructing the optimal the solution for $\mathcal{C}_j$ and then for each cluster in the subtree of $\mathcal{C}_j$.

Algorithm 1: BTD*

```
1: BTD* (A, C_i, V_{C_i})
2: if V_{C_i} = ∅ then
3:     Consistency ← True
4:     newA ← A
5:     F ← Children(C_i)
6:     while F ≠ ∅ and Consistency do
7:         Choose C_j in F
8:         F ← F\{C_j}
9:         if A[C_i ∩ C_j] is a Good of (C_i, C_j) in GoodSet then
10:            newA = newA ∪ Solution_From_Good(C_j, newA[C_j ∩ C_i])
11:        else
12:            if A[C_i ∩ C_j] is a NoGood of (C_i, C_j) in NoGoodSet then
13:                (Consistency, newA) ← (False, A)
14:            else
15:                (Consistency, newA) ← BTD*( newA, C_j, C_j\(C_i ∩ C_j) )
16:                if Consistency then
17:                    Record the good A[C_i ∩ C_j] of C_i ∩ C_j in GoodSet with newA[C_j\(C_i ∩ C_j)]
18:                else
19:                    Record the nogood A[C_i ∩ C_j] of C_i ∩ C_j in NoGoodSet
20:                end if
21:            end if
22:        end if
23:    end while
24:    Return (Consistency, newA)
25: else
26:    Choose x ∈ V_{C_i}
27:    d_x ← D_x
28:    (best_newA, best_cost) ← (∅, ∞)
29:    while d_x ≠ ∅ do
30:        Choose v in d_x
31:        d_x ← d_x\{v}
32:        if ∄c ∈ C such that c is not satisfied by A ∪ {x ← v} then
33:            (Consistency, new_A) ← BTD* (A ∪ {x ← v}, C_i, V_{C_i}\{x})
34:            if Consistency then
35:                cost ← Get_Subtree_Cost(new_A, C_i)
36:                if cost < best_cost then
37:                    (best_cost, best_newA) ← (cost, newA)
38:                end if
39:            end if
40:        end if
41:    end while
42:    if best_cost = ∞ then
43:        Return (False, A)
44:    else
45:        Return (True, best_newA)
46:    end if
47: end if
```

Algorithm 1 describes BTD* when minimizing the objective function $obj()$. The algorithm has input consist of:

- $\mathcal{A}$: a consistent set of assignments from the **root cluster** to $\mathcal{C}_i$,
- $\mathcal{C}_i$: the current cluster, and
- $V_{\mathcal{C}_i}$: the set of unassigned variable of $\mathcal{C}_i$.

The algorithm will return **True** and the most optimal extension of $\mathcal{A}$ called new$\mathcal{A}$ if $\mathcal{A}$ can be consistently

extended to $V_{\mathcal{C}_i}$ and Descendant($\mathcal{C}_i$). Should $\mathcal{A}$ can **not** be consistently extended to $V_{\mathcal{C}_i}$ and Descendant($\mathcal{C}_i$), False and $\mathcal{A}$ will be returned instead.

One can see that the pseudo code in algorithm 1 shows BTD* to be an exhaustive search with no bounding. This is because the way bounding works for BTD* is not easy to describe in pseudo code, therefore we decided mention it separately. Given one of the assumptions made for this thesis is no variable when assigned a value will have a negative cost, we can see that this induces a property of the objective function $obj()$ that it is ascending. having exploited this, when the goal of the objective is to minimize $obj()$, we can perform bounding in BTD* as follows. We created a data structure to keep track of the current best assignment for a cluster and the current best cost of the subtree rooted at said cluster w.r.t to the current state of assignments; we named this data structure Local_Best. If the objective goal is to minimize $obj()$, Local_Best is set to empty assignment and infinity for every cluster and the Local_Best of a cluster will be reset back to this once a variable in it has been reassigned so that the algorithm can keep searching for a better solution, hence the name Local_Best. When a solution has been found for a subtree rooted at a cluster $\mathcal{C}$, Local_Best of $\mathcal{C}$ is set to the cost of this subtree according to this newly found solution, that is if it is indeed better than the current Local_Best of $\mathcal{C}$. However, since Local_Best for every cluster is reset when a variable or more in its parent cluster has been reassigned, it could be the case that this newly found solution for the subtree rooted at $\mathcal{C}$ has a cost smaller than the current Local_Best (∞), but higher than the current Local_Best of the root cluster. Since $obj()$ is ascending, BTD* can conclude that due to this solution to the subtree rooted at $\mathcal{C}$ is already higher than the last best cost of the entire problem, the current assignment on $\mathcal{C}$ is suboptimal, thus Local_Best of $\mathcal{C}$ will not be updated with the cost from this solution and $\mathcal{C}$ will need to have a variable or more reassigned. Should BTD* have tried all the combination for for assigning $\mathcal{C}$ but they all leads to inconsistency or produce suboptimal solution a *nogood* will be recorded on the separator between $\mathcal{C}$ and its parent cluster. Even though in such case the assignments on the separator between $\mathcal{C}$ and its parent does lead to consistency, BTD* would still record a *nogood* because the solutions to the subtree alone is already worse than the last know best solution to the tree so the assignments on the separator will never be able to lead to a better solution, even if all the other variables instantiated before the ones in the separator are reassigned. For that BTD* can record a *nogood* on the separator, without having to worry about missing out on better solution in the future. This action of bounding the cost of subtree to the be less than the last best cost for the entire tree and recording *nogood* on separator that guarantees to break the bound is how BTD* prunes the search space using bounds. Another way of bounding that BTD* uses is for when assign a variable in a cluster. Should the cost of assigning a variable with a certain value is already higher than the Local_Best cost of the subtree rooted at said cluster,i.e there is a current best for the subtree, BTD* will ignore this value and would assign a different one to the variable.

4.2 Design choices

When designing the BTD* solver, we made several decisions regarding our choice for heuristics that will be addressed and reasoned in this section.

Heuristics help guide the solver during the solving process, they provide the solver with the ability to make "choices". For this project, these "choices" are: selecting the next cluster that is to be explored, which variable is to be instantiated next, which value will be chosen for assignment to a variable. Last but not least, the "choice" of how to construct the tree decomposition that the solving process will exploit. These heuristics were already implemented in the BTD framework that we received for this project. However, we decided to implement a different heuristic for creating the next cluster when computing the tree decomposition and a different value heuristic.

Firstly, we address our choice to implement another heuristic for H-TD-WT to compute a tree decomposition. For the implementation of our solver, this algorithm was used along with the heuristic H1 described in [8] to generate the tree decomposition for which the BTD* solver will traverse to find the solution to the problem at hand. Based on [8] and [7], the heuristic H5 that was already implemented in the source code for BTD is the more suitable heuristic out of all five for computing a tree decomposition in order to solve a **CSP**. Because of this, we had to evaluate if the H5 heuristic would be good to use for our implementation of BTD*. We decided that H5 is not suitable for our algorithm since it does not try to minimize the size of a cluster in a tree decomposition. We decided to implement the H1 heuristics in [8] to use with this algorithm instead. Due to its functionality of minimizing locally the size of the next cluster when the tree decomposition is being computed, it was also described to be the one more suited for solving optimization problems [8]. By doing so, the H1 heuristic ensures the tree decomposition computed will be "split" into smaller sized clusters as much as it can. This is beneficial to BTD* since when the

algorithm traverses the cluster tree, it will be able solve each cluster faster. This is because BTD* only use chronological backtracking, if a cluster's size is too big, it would create a bottleneck that would slow down the solving process significantly. Unlike solving a CSP, we will keep looking for more solutions even when one has been found. For that to happen, BTD* must perform backtracking and reassigning variables either in the current cluster or in one of the ancestor clusters. Hence when the size of a cluster is too large, this is likely to take a long time. A cluster with too large of a size will become a bottle neck because the BTD* algorithm optimizes the sub-tree rooted at said cluster by trying all possible combinations of assignments for the variables in the cluster (w.r.t the assignments of its ancestors clusters) and then choose the best one based on the cost of the entire sub-tree resulted by each assignment. For example, a cluster C contains 10 variables without counting the separator with its parents, can be assigned so that 3 variables, no matter how they are assigned to be consistent with previous assignments, will always leads to inconsistency somewhere down the sub-tree. Assuming between these 3 variables the maximum domain size is 9, the solver would need to try in the worst case at least $9^3 = 729$ combinations before it can decide to reassign one of the other variables in the cluster that was assigned before all = 3 or reassign one of the ancestor clusters. After reassignment has been performed, it could be that the solver would run into the same case, thus going through the same bottle neck again. In order to avoid described cases of bottleneck, locally minimizing the size of the next cluster when computing a tree decomposition is the more suitable choice. However, using the H1-TD-WT most likely results in more memory consumption given more Good and NoGood will be recorded since the cluster size will be minimized thus the tree decomposition would have more clusters, i.e. more separators; but it can ensure that such a case of bottle neck as described above will be limited as much as possible and the Goods and NoGoods will be more utilized to speed up the solving progress. This seems to be a reasonable trade off between time complexity and space complexity that we made for this algorithm.

For the heuristic that helps the solver choose the next cluster from the children of the current cluster to continue with the search process, the currently implemented heuristic in the framework seems to be the most suitable. This heuristic is called: "Minimum Size Separator", as the name suggest, it chooses the child cluster with the smallest amount of shared variables with the current cluster to keep furthering the search first. This heuristic can help save some memory for the solver. For example: a cluster $\mathcal{C}$ with 5 children, we label them $\mathcal{C}_1$ to $\mathcal{C}_5$ depending on the size of their separator with $\mathcal{C}$, $\mathcal{C}_1$ being the one with the smallest sized separator and $\mathcal{C}_5$ being the one with the largest. In the case of cluster $\mathcal{C}_1$ and $\mathcal{C}_2$ along with their sub-trees has been solved, meaning a Good was recorded between $\mathcal{C}$ and each of them and other Goods were recorded on any separator in the sub-trees rooted at $\mathcal{C}_1$ and $\mathcal{C}_2$, then BTD* may determine that $\mathcal{C}_3$ does not have a solution. Now the algorithm needs to record a NoGood between $\mathcal{C}_3$ and $\mathcal{C}$ then reassign a variable either in $\mathcal{C}$ or $\mathcal{C}$'s ancestors. If after the reassignment has taken place and $\mathcal{C}$ no longer can have assignments that can use the mentioned Goods with $\mathcal{C}_1$ and $\mathcal{C}_2$ for the rest of the solving process, they would never be used and thus will be wasted. Using this heuristics helps the solver minimize the amount of wasted memory for Goods recording that will never be used again, thus it was decided to keep using this heuristic for BTD*.

Variable ordering is already partially enforced by the tree decomposition. However, we can still choose a heuristic for ordering the variables in a cluster. For this heuristic, we again used the same heuristic implemented in the BTD source code: Dom/Wdeg heuristic. This heuristic guides the solver to choose the unassigned variable in a cluster which has the smallest ratio of its domain size to the sum of its constraints' weights. These weights are named Wdeg for Weighted Degree. Dom/wdeg was first introduced in [1] in 2004. Due to the fact that the main subject of the project is to implement and evaluate BTD*, we decided not to implement a different heuristic for variable ordering in order to save time and focus on implementing the actual algorithm.

The implemented heuristic for choosing a value to assign a variable in the source code of BTD is Lexicographical Ordering. Due to the way this heuristic was implemented in the BTD framework, we were not certain whether it was performing as expected. We decided to implement a simple Minimum Value heuristic to be used by the BTD* solver instead. As the name suggest, it simply select the smallest value currently in the domain of a variable when said variable has been selected for assignment by the solver.

4.3 Implementation

The BTD* algorithm was implemented in C++ as an extension to the the source code we received. Defined recursively, BTD* is very easy to understand and is quite straightforward. However, in terms of practicality, implementing this algorithm recursively would not be ideal since the recursion stack would grow too quickly, which potentially can cause stack overflow. To avoid this, BTD* was implemented

iteratively, but it still perform the search recursively.

5 Experiment and Result Discussion

For the purpose of benchmarking the BTD* algorithm, the used benchmark problems were sourced from the benchmarks available on the website of the XCSP3 framework: xcsp.org. These benchmarks will be hereinafter referred to as *instances*.

In order to ensure that the used instances all follows the assumption that were made for this thesis and all have constraints supported by the BTD source code that we received, we filtered these instances to find the one that are suitable for our experiments. These instances all have objective goal of maximizing or minimizing the sum of a list of variables and no variables in the objective scope can have a negative cost when assigned a variable from its domain. From the instances that were determined to be suitable, we decided to perform the experiment using 96 instances selected from 11 different sets.

The benchmark tests were run in a fresh Ubuntu 20.04 LTS installation on a Lenovo Legion Y540 laptop. The laptop is equipped with an Intel Core i7-9750H having 6 cores, a clock frequency of 2.6GHz, maximum turbo clock frequency of 4.5GHz, and 16 Gigabytes of RAM. The time limit was set to 1800 seconds in CPU time for all instances.

Configurations set for BTD* are:

- H-TD-WT algorithm with the H1 heuristic for computing a tree decomposition.
- Minimum Size Separator for next cluster heuristic.
- Dom/Wdeg for variable ordering in a cluster.
- Minimum Value for value heuristic.
- No memory limit.

For us to better assess BTD*, we used the Choco solver to try and solve the same instances and then compare the results of the two solvers. We downloaded the source code for Choco version 4.10.8, which at the time was the latest release of the Choco solver, directly from the GitHub repository of the Choco team [4].

Configurations set for the Choco solver are:

- Conflict History Search for variable ordering heuristic.
- Minimum Value for value heuristic.
- Default released memory limit of 4000 MiB (roughly 4.2GB)

We note that the maximum amount of memory available for Choco is less than the amount available for BTD*. However, this was not a problem since Choco did not reach the memory limit while trying to solve any of the instances.

All time measurements in this sections are in CPU time unless stated otherwise.

5.1 Overview of BTD*'s result

BTD* managed to solve **49** out of the selected 96 instances, which is approximately **51%** of the total number of instances used. The number of instances solved in a certain time frame are shown in Table 1. We can see that 39 of the solved instances were solved in less than 50 seconds, which is roughly 79.6% of the

Total time **t** (CPU seconds)	#instances solved
$t < 50$	39
$50 \leq t < 100$	2
$100 \leq t < 250$	2
$250 \leq t < 500$	4
$500 \leq t < 1000$	1
$1000 \leq t$	1

Table 1: Time frame of solved instances by BTD*.

solved instances and 40.6% of all selected instances. While this does seems some what impressive, we must also take a look at the instances that BTD* failed to solved. Among the 96 chosen instances, 47 of them were not solved by BTD*. 44 of them are due to the time limit having been exceeded, the other 3 are due to insufficient memory despite having 16GB of RAM at hand. Interestingly, 2 of the instances that timed out did not reach the solving phase, in fact one after 13 hours and one after 2 hours of wall clock time still did not make it pass the initial propagation. We had to stop the process in order to continue with other instances and ruled these 2 instances as having timed out. These instances are: *Pb-factor-S9-P005-Q317* (13 hours) of the subset PseudoBoolean-opt-factor and *PeacableArmies-m1-07_c18* (2 hours) of the set PeacableArmies.

Detailed results produced by BTD* can be found in Appendix B.

5.2 Comparison with Choco

Before going into details of the comparison, there are few things that we would like to address:

- The timer for the Choco solver is implemented differently from the timer in the BTD*. Only the actual solving process of Choco is limited by the time limit set out. The solving process includes the initial propagation and the resolution, but not the building time for relevant data structure. For the BTD*, the entire process is limited to the time limit, i.e. the sum of the building time, initial propagation time, and solving time is limited. Having taken this into account, we decided when comparing the solving time of the 2 solvers the total time will be compared.
- Choco is implemented using Java which is known to be slower than C++ when performing the same computing task.

We will now take a quick look over the results of the two solvers before going into details. A quick

	BTD*	Choco
# solved	49	63
# failed	47	33

Table 2: Number of solved/failed instances of BTD* and Choco.

comparison of the number of solved/failed instances of the two solvers can be found in Table 2. We can see right away that Choco managed to solved more instances than BTD*, it successfully solved 63 instances whereas BTD* only solved 49 instances. This is somewhat expected since Choco did came in third and second position in solving COPs when competed in the XCSP3 2018 and 2019 international competition for solving Constraint Problems.[†]

Total time **t**	#instanced solved	
	BTD*	Choco
$t < 50$	39	48
$50 \leq t < 100$	2	3
$100 \leq t < 250$	2	4
$250 \leq t < 500$	4	5
$500 \leq t < 1000$	1	2
$1000 \leq t$	1	1

Table 3: Number of instances solved in a certain time frame for both solvers.

Table 3 shows how many instanced did BTD* and Choco solved in a certain time frame. Across almost all time frame, Choco solved more than BTD*, especially within less than 50 seconds.

We will now take a closer look into the results of the 2 solvers.

When the results of the two solvers are put together, we found **48** instances were solved by both solvers; 15 of the instances BTD* failed to solve were solved by Choco, whereas only **1** instance solved by BTD* that Choco did not managed to solve.

Let us examine the 48 instances that were solved by both solvers. Table 4 shows the name of the instances, the total time each solver took to solve them, how many nodes assignment did each solver

[†]As shown on the XCSP3 website

Instances name	BTD*			Choco		
	Time	#Nodes	#Backtracks	Time	#Nodes	#Backtracks
Knapsack-30-100-09	29.920	8,366,571	2,647,020	1.342	98,431	196
Knapsack-30-100-15	192.394	51,300,043	19,042,633	1.414	103,074	205,028
Knapsack-30-100-18	76.735	21,726,817	6,695,251	1.070	54,745	108,918
LowAutocorrelation-003	**0.000**	38	**6**	0.191	7	11
LowAutocorrelation-008	**0.045**	3,956	**225**	0.324	802	1,586
LowAutocorrelation-012	4.248	130,723	**3,963**	0.823	2,724	5,409
LowAutocorrelation-015	313.255	1,228,510	**28,742**	20.550	203,969	402,563
Pb-bgr-04	**0.007**	165	**33**	0.226	83	161
Pb-bgr-05	**0.065**	2,966	1,393	0.340	489	959
Pb-bgr-06	0.933	19,174	11,054	0.577	1,279	2,535
Pb-bgr-07	11.913	123,387	69,399	3.254	9,363	18,616
Pb-bgr-08	510.104	3,253,967	1,875,579	42.442	72,791	144,846
Pb-aim-050-2-0-yes1-2	3.227	159,548	72,044	1.385	15,156	30,105
Pb-aim-100-3-4-yes1-2	**3.831**	247,790	**131,807**	19.233	103,076	205,065
Pb-ii08a1.xml	**1.316**	**106,477**	**31,032**	26.728	555,577	1,104,958
Pb-jnh001.xml	45.164	1,949,454	167,017	8.805	18,449	36,605
Pb-jnh207.xml	**0.151**	4,880	**475**	0.762	1,005	2,000
Pb-par08-1-c.xml	**0.007**	823	**50**	0.254	35	69
Pb-par08-5-c.xml	**0.009**	1,053	**101**	0.278	63	125
Pb-garden-4x4	**0.014**	1,910	1,037	0.204	49	97
Pb-garden-7x7	**7.368**	**610,641**	**290,533**	434.609	12,538,190	25,006,132
Pb-gr-04	**0.116**	10,078	2,841	0.233	82	159
Pb-gr-05	**0.336**	33,763	4,793	0.383	616	1,211
Pb-gr-06	30.237	1,879,078	228,365	0.904	2,254	4,460
Pb-gr-07	333.450	12,609,294	1,885,120	3.927	11,240	22,344
Pb-logic-b1	**0.063**	9,471	4,081	0.195	43	75
Pb-logic-bbara-r	120.746	26,489,604	13,084,543	0.490	3,753	7,457
Pb-logic-C17	**0.002**	298	132	0.180	13	21
Pb-msplit-4-30-3	**87.621**	14,269,492	**9,607,678**	474.353	9,992,497	19,941,201
Pb-mps-v2-20-10-bm23	1.254	123,317	74,140	0.648	7,059	14,050
Pb-mps-v2-20-10-p033	**0.007**	1,587	**680**	0.244	741	1,467
Pb-mps-v2-20-10-stein15	**0.042**	6,598	2,808	0.209	337	673
Pb-mps-v2-20-10-stein27	4.694	859,252	367,524	1.259	36,762	73,241
Pb-routing-s3-3-3-1	**0.442**	24,650	**11,023**	3.853	16,321	32,195
Pb-routing-s3-3-3-3	**3.431**	**157,398**	**99,268**	72.443	331,066	656,921
Pb-circ4-3	**0.909**	72,074	**2,102**	3.415	14,201	28,228
Pb-data4-3	**0.960**	72,074	**2,102**	3.518	15,357	30,535
QuadraticAssignment-tai10a	445.990	38,897,885	3,628,799	52.117	937,177	1,871,866
QuadraticAssignment-tai10b	4.724	563,611	135,248	1.256	5,141	10,197
QueenAttacking-03	**0.000**	0	0	0.212	0	0
QueenAttacking-04	**0.046**	2,791	**1,891**	0.503	1,264	2,527
QueenAttacking-05	4.319	151,619	84,959	0.691	2,073	4,124
StillLife-03-03	**0.006**	123	**25**	0.258	17	27
StillLife-03-12	**0.047**	**3,336**	**578**	1.068	12,487	24,823
StillLife-04-08	0.778	62,801	15,056	0.605	4,086	8,118
StillLife-05-09	29.781	824,281	222,044	2.795	41,823	83,259
StillLife-07-07	24.205	477,149	**116,413**	9.048	118,610	236,180
StillLife-08-08	1352.129	3,575,766	**855,978**	229.999	2,259,420	4,501,665

Table 4: 48 instances solved by BTD* and Choco. Total time, number of nodes assignments, and number of backtracks of both solvers. Bold font means for that instance, BTD* performed better than Choco in terms of that metric at face value.

performed, and how many backtrack did each solver performed. It also highlight instances that were solved faster by BTD* which are the one whose total time of BTD* is in bold font.

In total, 26 of the 48 instances were solved faster by BTD*. However, after closer inspection we found that only for 4 of them was BTD* significantly faster than Choco; these instances are highlighted green in Table 4. Overall, we can see that Choco performed better and more consistent than BTD* in terms of

solving time.

In terms of number of nodes assignment made, due to its nature of recursive search, BTD* assigned significantly more nodes than Choco for almost all of the instances. There are only 4 instances where BTD* managed to win over Choco, they are the instances in Table 4 which has the "#Nodes" field of BTD* in bold font. Interestingly, these 4 are also the 4 instances where BTD* was much faster than Choco.

Having looked into the number of backtracks achieved by the two solvers for these 48 instances, we find that for **22** instances BTD* performed less backtracks than Choco. There is an instance for which the solvers tied with each other, it is the *QueenAttacking-03* instance. Both of the solvers determined this instance is unsatisfiable in the initial propagation phase, thus the number of backtracks performed is 0. Instances where BTD* had to perform less backtrack can be found in Table 4, they are the ones with #Backtracks for BTD* in bold font. However, we have to note that due to the way Choco is implemented, its number of backtracks is, most of the time, more than the number of node assigned. This is of course very unusual. We suspect that Choco was making backup of nodes and therefore the counts is higher than anticipated. Given this, information on Backtracks counts does not reflect much about the amount of work done between the two solvers.

Instances name	BTD*			Choco		
	Time	# Nodes	# Backtracks	Time	# Nodes	# Backtracks
Pb-garden-4x4	0.014	1,910	1,037	0.204	49	97
Pb-garden-7x7	7.368	610,641	290,533	434.609	12,538,190	25,006,132
Pb-garden-9x9	395.837	17,047,377	8,145,125	> 1800.000	40,092,684	79,969,674
Pb-garden-15x15	> 1800.000	65,442,709	31,736,821	> 1800.000	40,769,577	80,739,865
Pb-garden-100x100	> 1800.000	33,927,696	16,301,287	> 1800.000	222,525	243,297

Figure 2: Results of the solvers for the subset PseudoBoolean-opt-garden.

Instances name	BTD*			Choco		
	Time	# Nodes	# Backtracks	Time	# Nodes	# Backtracks
Pb-routing-s3-3-3-1	0.442	24,650	11,023	3.853	16,321	32,195
Pb-routing-s3-3-3-3	3.431	157,398	99,268	72.443	331,066	656,921
Pb-routing-s4-4-3-2	> 1800.000	31,310,519	24,580,767	> 1800.000	3,167,222	6,198,627
Pb-routing-s4-4-3-8	> 1800.000	15,366,894	2,562,143	> 1800.000	5,833,311	11,475,298

Figure 3: Results of the solvers for the subset PseudoBoolean-opt-routing.

Interestingly, theres is one instance that BTD* managed to solve, but Choco did not, this is the instance *Pb-garden-9x9* of the subset PseudoBoolean-opt-garden in the set PseudoBoolean. Figure 2 shows the result of the solvers for each instances in this subset. In this subset, BTD* managed to out shine Choco. For every metrics of comparsion, BTD* show significant better quality than that of Choco. This subset and the subset PseudoBoolean-opt-routing, are the 2 subsets where BTD* produced results that can be considered significantly better than Choco. The results of the two solver for the subset PseudoBoolean-opt-routing can be found in Figure 3.

When it comes to other subsets BTD* either performed comparably or worse than Choco. The common trends of these subset are that BTD* performs comparably to Choco, or possibly a bit better, for the easier instances. But as the difficulty grows, by number of variables, number of constraints, contraint arity, etc. the performance of BTD* quickly deteriorates. Some example of cases like these are the subset PseudoBoolean-opt-bgr, QueenAttacking-m1-s1, and StillLife-m1-s1. Results for these instances can be found in, respectively, Figure 4, 5, and 6.

Overall, the experiment show that BTD* is not comparable to the Choco solver. Given the position of Choco in terms of open source Constraint Problems solver, this result is not unexpected. However, on its own, the result for BTD* still seems promising enough to warrant more researches.

Instances name	BTD*			Choco		
	Time	# Nodes	# Backtracks	Time	# Nodes	# Backtracks
Pb-bgr-05	0.065	2,966	1,393	0.340	489	959
Pb-bgr-06	0.933	19,174	11,054	0.577	1,279	2,535
Pb-bgr-07	11.913	123,387	69,399	3.254	9,363	18,616
Pb-bgr-08	510.104	3,253,967	1,875,579	42.442	72,791	144,846
Pb-bgr-09	> 1800.000	7,481,244	4,027,513	772.125	950,017	1,890,934
Pb-bgr-10	> 1800.000	6,897,261	2,976,656	> 1800.000	1,396,193	2,778,689

Figure 4: Results of the solvers for the subset PseudoBoolean-opt-bgr. Showing as the instances gets harder, the performance of BTD* quickly quickly deteriorates.

Instances name	BTD*			Choco		
	Time	# Nodes	# Backtracks	Time	# Nodes	# Backtracks
QueenAttacking-03 *	0.000	0	0	0.212	0	0
QueenAttacking-04 *	0.046	2,791	1,891	0.503	1,264	2,527
QueenAttacking-05	4.319	151,619	84,959	0.691	2,073	4,124
QueenAttacking-06	> 1800.000	28,206,692	14,762,537	2.298	11,991	23,755
QueenAttacking-09	> 1800.000	1,339,544	1,009,890	> 1800.000	8,616,413	17,052,014
QueenAttacking-12	> 1800.000	185,198	148,489	> 1800.000	6,738,392	13,188,477
QueenAttacking-15	> 1800.000	31,730	24,231	> 1800.000	3,076,267	5,949,497

Figure 5: Results of the solvers for the subset QueenAttacking-m1-s1. Showing as the instances gets harder, the performance of BTD* quickly quickly deteriorates.

Instances name	BTD*			Choco		
	Time	# Nodes	# Backtracks	Time	# Nodes	# Backtracks
StillLife-03-03	0.006	123	25	0.258	17	27
StillLife-03-12	0.047	3,336	578	1.068	12,487	24,823
StillLife-04-08	0.778	62,801	15,056	0.605	4,086	8,118
StillLife-05-09	29.781	824,281	222,044	2.795	41,823	83,259
StillLife-06-10	> 1800.000	4,503,572	1,213,085	107.417	1,436,595	2,863,334
StillLife-07-07	24.205	477,149	116,413	9.048	118,610	236,180
StillLife-08-08	1352.129	3,575,766	855,978	229.999	2,259,420	4,501,665
StillLife-09-09	> 1800.000	3,484,693	847,249	> 1800.000	15,854,672	31,584,476
StillLife-10-11	> 1800.000	4,589,140	1,218,205	> 1800.000	18,547,130	36,876,357

Figure 6: Results of the solvers for the subset StillLife-m1-s1. Showing as the instances gets harder, the performance of BTD* quickly quickly deteriorates. Note that instance StillLife-06-10 is harder than StillLife-07-7

5.3 Tree Decomposition

Before the experiment, we believed that having a small tree width would be beneficial to the search time for BTD*. In this section, we show the width, the number of cluster, and the maximum separators size of the tree decomposition that H1-TD-WT produced for each instances to determine if using H1 helped with the solving process as we anticipated.

The results of the 2 solvers for all 96 instances and the information on the tree decomposition computed by H1-TD-WT is shown in Figure 7.

Based on the width, the number of clusters, and the maximum size of separators, we can not conclude anything about whether or not H1-TD-WT benefited the search time. While it does seems that a smaller tree width helps, but there are many instances where our anticipation is contradicted. For example, we look at the set LowAutocorrelation. This set is one of our example for how quickly the performance of BTD* deteriorates as the difficulty increases. Yet, instances of this set have relatively small tree width. Furthermore, for the last 3 instances in this set, the amount of backtracking perform by BTD* is very low compared to the number of nodes assignment. Another example of this is the StillLife set of instances. All instances have relatively good tree width, but the performance of BTD* still dropped as the difficulty

Set	Sub-set	Instances name	BTD*			Choco			H1-TD-WT Information		
			Time	# Nodes	# Backtracks	Time	# Nodes	# Backtracks	w+	# clusters	max-s
Knapsack	Knapsack-m1-s30	Knapsack-30-100-09	29.920	8,366,571	2,647,020	1.342	98,431	196	29	1	0
		Knapsack-30-100-15	192.394	51,300,043	19,042,633	1.414	103,074	205,028	29	1	0
		Knapsack-30-100-18	76.735	21,726,817	6,695,251	1.070	54,745	108,918	29	1	0
	Knapsack-m1-s40	Knapsack-40-150-00	> 1800.000	443,073,025	184,479,977	442.189	11,722,667	23,363,480	39	1	0
LowAutocorrelation	LowAutocorrelation-m1-s1	LowAutocorrelation-003	0.000	38	6	0.191	7	11	3	8	3
		LowAutocorrelation-008	0.045	3,956	225	0.324	802	1,586	8	63	8
		LowAutocorrelation-012	4.248	130,723	3,963	0.823	2,724	5,409	12	143	12
		LowAutocorrelation-015	313.255	1,228,510	28,742	20.550	203,969	402,563	15	224	15
		LowAutocorrelation-020	> 1800.000	6,775,286	51,809	> 1800.000	9,023,187	17,785,783	20	399	20
		LowAutocorrelation-025	> 1800.000	5,583,639	24,600	> 1800.000	7,157,059	13,956,521	25	624	25
		LowAutocorrelation-030	> 1800.000	22,071	52	> 1800.000	4,408,492	8,540,271	30	899	30
PseudoBoolean	PseudoBoolean-opt-bgr	Pb-bgr-04	0.007	165	33	0.226	83	161	29	9	13
		Pb-bgr-05	0.065	2,966	1,393	0.340	489	959	52	23	19
		Pb-bgr-06	0.933	19,174	11,054	0.577	1,279	2,535	86	53	23
		Pb-bgr-07	11.913	123,387	69,399	3.254	9,363	18,616	130	108	24
		Pb-bgr-08	510.104	3,253,967	1,875,579	42.442	72,791	144,846	187	199	25
		Pb-bgr-09	> 1800.000	7,481,244	4,027,513	772.125	950,017	1,890,934	262	339	28
		Pb-bgr-10	> 1800.000	6,897,261	2,976,656	> 1800.000	1,396,193	2,778,689	352	543	28
	PseudoBoolean-opt-dimacs	Pb-aim-050-2-0-yes1-2	3.227	159,548	72,044	1.385	15,156	30,105	29	70	29
		Pb-aim-100-3-4-yes1-2	3.831	247,790	131,807	19.233	103,076	205,065	91	109	91
		Pb-aim-200-1-6-yes1-4	> 1800.000	30,127,381	8,765,647	> 1800.000	14,787,758	29,314,051	86	304	86
		Pb-ii08a1.xml	1.316	106,477	31,032	26.728	555,577	1,104,958	16	90	16
		Pb-ii08a2.xml	> 1800.000	28,431,518	12,037,864	> 1800.000	21,529,590	42,554,475	70	239	66
		Pb-jnh001.xml	45.164	1,949,454	167,017	8.805	18,449	36,605	173	27	173
		Pb-jnh207.xml	0.151	4,880	475	0.762	1,005	2,000	171	29	171
		Pb-par08-1-c.xml	0.007	823	50	0.254	35	69	35	47	34
		Pb-par08-5-c.xml	0.009	1,053	101	0.278	63	125	37	57	36
	PseudoBoolean-opt-factor	Pb-factor-S9-P005-Q317	> 1800.000 **			0.333	332	663	82	17	82
	PseudoBoolean-opt-garden	Pb-garden-4x4	0.014	1,910	1,037	0.204	49	97	8	8	8
		Pb-garden-7x7	7.368	610,641	290,533	434.609	12,538,190	25,006,132	14	35	14
		Pb-garden-9x9	395.837	17,047,377	8,145,125	> 1800.000	40,092,684	79,969,674	18	63	18
		Pb-garden-15x15	> 1800.000	65,442,709	31,736,821	> 1800.000	40,769,577	80,739,865	30	195	30
		Pb-garden-100x100	> 1800.000	33,927,696	16,301,287	> 1800.000	222,525	243,297	200	9800	200
	PseudoBoolean-opt-gr	Pb-gr-04	0.116	10,078	2,841	0.233	82	159	32	8	15
		Pb-gr-05	0.336	33,763	4,793	0.383	616	1,211	53	22	20
		Pb-gr-06	30.237	1,879,078	228,365	0.904	2,254	4,460	89	52	24
		Pb-gr-07	333.450	12,609,294	1,885,120	3.927	11,240	22,344	131	107	24
		Pb-gr-08	> 1800.000	27,536,957	5,824,319	43.443	70,653	140,607	194	198	28
		Pb-gr-09	> 1800.000	24,317,372	4,530,853	673.922	759,810	1,513,280	265	338	28
	PseudoBoolean-opt-logicSynthesis	Pb-logic-b1	0.063	9,471	4,081	0.195	43	75	12	10	10
		Pb-logic-bbara-r	120.746	26,489,604	13,084,543	0.490	3,753	7,457	25	1	0
		Pb-logic-c8	> 1800.000	136,910,582	59,522,367	> 1800.000	15,623,642	30,586,566	52	75	50
		Pb-logic-C17	0.002	298	132	0.180	13	21	6	8	5
		Pb-logic-cm42a	> 1800.000	206,281,030	92,249,814	> 1800.000	24,299,279	48,383,728	1	34	33
		Pb-logic-dk512x-r	> 1800.000	232,359,869	115,788,848	14.02	69,855	138,514	1	1	91
		Pb-logic-fout-r	> 1800.000	132,879,319	64,087,836	> 1800.000	12,122,470	22,718,529	1	131	5
		Pb-logic-m050-100-10-10-r	> 1800.000	111,423,684	52,787,079	> 1800.000	28,891,021	57,521,998	1	28	4
	PseudoBoolean-opt-marketSplit	Pb-msplit-4-30-3 *	87.621	14,269,492	9,607,678	474.353	9,992,497	19,941,201	29	2	0
		Pb-msplit-6-50-2	> 1800.000	156,217,175	93,903,011	> 1800.000	43,436,037	86,765,613	49	2	0
		Pb-msplit-8-70-4	> 1800.000	83,078,015	43,063,384	> 1800.000	8,511,578	16,931,440	69	2	0
		Pb-msplit-opt-4-30-2	> 1800.000	498,812,328	57,956,245	86.766	3,322,371	6,627,026	61	3	29
		Pb-msplit-opt-5-40-1	> 1800.000	473,279,915	38,029,655	> 1800.000	14,099,623	27,771,151	103	2	39
	PseudoBoolean-opt-mps	Pb-mps-v2-20-10-bm23	1.254	123,317	74,140	0.648	7,059	14,050	26	1	0
		Pb-mps-v2-20-10-lseu	> 1800.000	287,921,179	103,683,270	1419.316	24,344,163	48,515,763	56	20	49
		Pb-mps-v2-20-10-neos5	> 1800.000	62,267,738	30,503,908	> 1800.000	1,311,072	2,564,026	162	1	0
		Pb-mps-v2-20-10-p033	0.007	1,587	680	0.244	741	1,467	18	9	10
		Pb-mps-v2-20-10-stein15	0.042	6,598	2,808	0.209	337	673	14	1	0
		Pb-mps-v2-20-10-stein27	4.694	859,252	367,524	1.259	36,762	73,241	26	1	0
		Pb-mps-v2-20-10-stein45	> 1800.000	320,794,996	140,513,795	362.813	5,954,975	11,874,807	44	1	0
	PseudoBoolean-opt-routing	Pb-routing-s3-3-3-1	0.442	24,650	11,023	3.853	16,321	32,195	87	128	87
		Pb-routing-s3-3-3-3	3.431	157,398	99,268	72.443	331,066	656,921	97	142	97
		Pb-routing-s4-4-3-2	> 1800.000	31,310,519	24,580,767	> 1800.000	3,167,222	6,198,627	186	384	186
		Pb-routing-s4-4-3-8	> 1800.000	15,366,894	2,562,143	> 1800.000	5,833,311	11,475,298	123	258	123
	PseudoBoolean-opt-ttp	Pb-circ4-3	0.909	72,074	2,102	3.415	14,201	28,228	72	51	70
		Pb-circ6-3	N/A ****			> 1800.000	2,123,814	4,206,562	303	135	301
		Pb-data4-3	0.960	72,074	2,102	3.518	15,357	30,535	72	51	70
		Pb-data6-3	N/A ****			> 1800.000	1,966,514	3,889,619	303	135	301
	PseudoBoolean-opt-vtxcov	Pb-vtxcov-v2000-e4000-05	> 1800.000	6,558,052	1,453,849	> 1800.000	3,848,239	6,140,998	378	1581	378
QuadraticAssignment	QuadraticAssignment-m1-s1	QuadraticAssignment-esc16e	> 1800.000	227,812,416	29,515,895	40.094	811,037	1,619,191	20	252	2
		QuadraticAssignment-had12	> 1800.000	127,594,947	10,031,999	> 1800.000	21,120,824	42,194,731	21	135	2
		QuadraticAssignment-lipa20b	> 1800.000	68,288,028	3,647,952	> 1800.000	11,580,413	23,053,219	35	385	2
		QuadraticAssignment-tai10a	445.990	38,897,885	3,628,799	52.117	937,177	1,871,866	17	93	2
		QuadraticAssignment-tai10b	4.724	563,611	135,248	1.256	5,141	10,197	17	93	2
		QuadraticAssignment-tai25a	> 1800.000	52,126,212	2,026,604	> 1800.000	7,127,544	14,188,285	47	603	2
QueenAttacking	QueenAttacking-m1-s1	QueenAttacking-03 *	0.000	0	0	0.212	0	0	8	3	4
		QueenAttacking-04 *	0.046	2,791	1,891	0.503	1,264	2,527	15	3	6
		QueenAttacking-05	4.319	151,619	84,959	0.691	2,073	4,124	24	3	9
		QueenAttacking-06	> 1800.000	28,206,692	14,762,537	2.298	11,991	23,755	35	3	11
		QueenAttacking-09	> 1800.000	1,339,544	1,009,890	> 1800.000	8,616,413	17,052,014	80	3	22
		QueenAttacking-12	> 1800.000	185,198	148,489	> 1800.000	6,738,392	13,188,477	143	3	34
		QueenAttacking-15	> 1800.000	31,730	24,231	> 1800.000	3,076,267	5,949,497	224	3	48
StillLife	StillLife-m1-s1	StillLife-03-03	0.006	123	25	0.258	17	27	11	7	6
		StillLife-03-12	0.047	3,336	578	1.068	12,487	24,823	11	25	6
		StillLife-04-08	0.778	62,801	15,056	0.605	4,086	8,118	20	24	16
		StillLife-05-09	29.781	824,281	222,044	2.795	41,823	83,259	22	35	20
		StillLife-06-10	> 1800.000	4,503,572	1,213,085	107.417	1,436,595	2,863,334	24	48	22
		StillLife-07-07	24.205	477,149	116,413	9.048	118,610	236,180	20	40	18
		StillLife-08-08	1352.129	3,575,766	855,978	229.999	2,259,420	4,501,665	22	53	20
		StillLife-09-09	> 1800.000	3,484,693	847,249	> 1800.000	15,854,672	31,584,476	24	68	22
		StillLife-10-11	> 1800.000	4,589,140	1,218,205	> 1800.000	18,547,130	36,876,357	26	93	24
TravellingSalesman	TravellingSalesman-m1-n15	TravellingSalesman-15-30-00	> 1800.000	146,275,908	39,339,346	380.396	7,342,186	14,668,553	15	15	2
		TravellingSalesman-15-30-10	> 1800.000	141,954,083	38,177,029	109.673	2,134,422	4,263,676	15	15	2
Auction	Auction-sum	Auction-sum-matching040_c18	N/A ****			> 1800.000	2,204,877	3,812,781	853	254	825
PeacableArmies		PeacableArmies-m1-07_c18	> 1800.000 ***			194.142	2,051,247	4,077,313	97	1	0
Rlfap	Rlfap-opt	Rlfap-graph-05-opt_c18	> 1800.000	27,415,095	9,030,515	> 1800.000	787,945	1,429,496	60	984	58
SumColoring		SumColoring-dsjc-250-1_c18	> 1800.000	3,004,473	3,004,287	> 1800.000	4,714,835	9,083,187	175	75	175

* Instance is unsatisfiable
** Initial propagation did not finished after 13 hours wall clock
*** Initial propagation did not finished after 2 hours wall clock time
**** Out of memory while solving, processed killed

Figure 7: Results of both solver. Along with the width, number of clusters, and max separator size of the Tree Decomposition computed by H1-TD-WT

level rises. However, if we look at the subset PseudoBoolean-opt-garden, the description applies, but BTD* out performed Choco here instead.

Considering all these information, we can not draw a relation between the width of the tree decomposition and the solving time of BTD*. It is likely that other factors, may be in combination with features of tree decomposition, would be able to predict the behaviour of BTD*.

6 Conclusions

The goal of this thesis was to design, implement, and benchmark as well as compare to another solver, an algorithm for solving Constraint Optimization Problems based on the algorithm Backtracking with Tree Decomposition which was designed to solve Constraint Satisfaction Problem. We have achieved these goals with the BTD* algorithm.

Similar to BTD, BTD* is defined to be a recursive search algorithm. BTD recursively search for a solution to the subtree of the tree decomposition of the constraint network, whereas BTD* recursively search for the optimal solution to the subtree instead. After having implemented BTD*, we ran benchmark tests for BTD* and found that as the difficulty of the problem grow, the time that it takes for BTD* to solve the problem would also be growing quite fast. This is likely due to the recursive search nature of BTD*. The same benchmarks was also run for the Choco solver, a different and well known COP solver, to help us interpret the position of BTD* more. From the comparison of the experimental results, we see that even though BTD* successfully solved over half of the selected instances, it is still not yet to be consider comparable to Choco due to the fact that it is yet to be efficient. However from said results, it does seems promising enough to warrant more research towards COP solving algorithms that ultilise BTD. We can consider our work to be a good starting point for developing, implementing, and benchmarking BTD-like algorithms that solves COP in the future.

7 Future Work

Various future work can be envisioned for BTD*. They can be separated into two trends: (In)directly improving the current state of BTD* or ultilising BTD in a different manner for solving COP.

The first one includes but does not limit to:

- Further optimization of the BTD* source code, for example: introduce more bounding if possible
- Implement more heuristics that BTD* can use and experimenting with their effect on performance, e.g. more value heuristics, variable heuristics, cluster heuristics, and heuristics for H-TD-WT, etc.
- Extending BTD* to consider COP with negative cost for assigning variable.
- More benchmark testing with larger amounts of instances, better hardware, and more diverse settings for both BTD* and the solver to be used for comparison.

A possibility of future work that follows the second trend is to design an algorithm that uses BTD to first create a solution to the constraint network. The next step of such an algorithm could be using that solution and its cost as a bound as it iteratively searches for a better solution and updates the bound if needed. It could also exploit decomposition of the objective cost to create bound for subproblems should the problem be decomposable.

Evidently, there are many possibilities for BTD* to be improved. Given the fact that at the point of research for this project it seemed that BTD* is the first BTD-like algorithm to solve COP, therefore any improvement to BTD* following any of the two trends are warranted.

References

[1] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, page 146–150, NLD, 2004. IOS Press. ISBN 9781586034528.

[2] Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016. URL http://arxiv.org/abs/1611.03398.

[3] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3-core: A format for representing constraint satisfaction/optimization problems. *CoRR*, abs/2009.00514, 2020. URL https://arxiv.org/abs/2009.00514.

[4] Chocoteam. Chocoteam/choco-solver: An open-source java library for constraint programming. URL https://github.com/chocoteam/choco-solver.

[5] Rina Dechter and Judea Pearl. Tree clustering for constraint networks (research note). *Artif. Intell.*, 38(3):353–366, apr 1989. ISSN 0004-3702. doi:10.1016/0004-3702(89)90037-4. URL https://doi-org.proxy-ub.rug.nl/10.1016/0004-3702(89)90037-4.

[6] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artif. Intell.*, 146(1):43–75, 2003. doi:10.1016/S0004-3702(02)00400-9. URL https://doi.org/10.1016/S0004-3702(02)00400-9.

[7] Philippe Jégou, Hanan Kanso, and Cyril Terrioux. Towards a dynamic decomposition of csps with separators of bounded size. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 298–315, Cham, 2016. Springer International Publishing. ISBN 978-3-319-44953-1.

[8] Philippe Jégou, Hanan Kanso, and Cyril Terrioux. An algorithmic framework for decomposing constraint networks. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1–8, 2015. doi:10.1109/ICTAI.2015.15.

[9] Matthew Kitching and Fahiem Bacchus. Exploiting decomposition in constraint optimization problems. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming*, pages 478–492, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-85958-1.

[10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Pearson, 3 edition, 2009.

[11] M. Sanchez, D. Allouche, S. De Givry, and T. Schiex. Russian doll search with tree decomposition. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, page 603–608, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

A Appendix

Set	Sub-set	# chosen instances	Instances name
Knapsack	Knapsack-m1-s30	3	Knapsack-30-100-09
			Knapsack-30-100-15
			Knapsack-30-100-18
	Knapsack-m1-s40	1	Knapsack-40-150-00
LowAutocorrelation	LowAutocorrelation-m1-s1	7	LowAutocorrelation-003
			LowAutocorrelation-008
			LowAutocorrelation-012
			LowAutocorrelation-015
			LowAutocorrelation-020
			LowAutocorrelation-025
			LowAutocorrelation-030
PseudoBoolean	PseudoBoolean-opt-bgr	7	Pb-bgr-04
			Pb-bgr-05
			Pb-bgr-06
			Pb-bgr-07
			Pb-bgr-08
			Pb-bgr-09
			Pb-bgr-10
	PseudoBoolean-opt-dimacs	9	Pb-aim-050-2-0-yes1-2
			Pb-aim-100-3-4-yes1-2
			Pb-aim-200-1-6-yes1-4
			Pb-ii08a1.xml
			Pb-ii08a2.xml
			Pb-jnh001.xml
			Pb-jnh207.xml
			Pb-par08-1-c.xml
			Pb-par08-5-c.xml
	PseudoBoolean-opt-factor	1	Pb-factor-S9-P005-Q317
	PseudoBoolean-opt-garden	5	Pb-garden-4x4
			Pb-garden-7x7
			Pb-garden-9x9
			Pb-garden-15x15
			Pb-garden-100x100
	PseudoBoolean-opt-gr	6	Pb-gr-04
			Pb-gr-05
			Pb-gr-06
			Pb-gr-07
			Pb-gr-08
			Pb-gr-09
	PseudoBoolean-opt-logicSynthesis	8	Pb-logic-b1
			Pb-logic-bbara-r
			Pb-logic-c8
			Pb-logic-C17
			Pb-logic-cm42a
			Pb-logic-dk512x-r
			Pb-logic-fout-r
			Pb-logic-m050-100-10-10-r
	PseudoBoolean-opt-marketSplit	5	Pb-msplit-4-30-3
			Pb-msplit-6-50-2
			Pb-msplit-8-70-4
			Pb-msplit-opt-4-30-2
			Pb-msplit-opt-5-40-1
	PseudoBoolean-opt-mps	7	Pb-mps-v2-20-10-bm23
			Pb-mps-v2-20-10-lseu
			Pb-mps-v2-20-10-neos5
			Pb-mps-v2-20-10-p033
			Pb-mps-v2-20-10-stein15
			Pb-mps-v2-20-10-stein27
			Pb-mps-v2-20-10-stein45
	PseudoBoolean-opt-routing	4	Pb-routing-s3-3-3-1
			Pb-routing-s3-3-3-3
			Pb-routing-s4-4-3-2
			Pb-routing-s4-4-3-8
	PseudoBoolean-opt-ttp	4	Pb-circ4-3
			Pb-circ6-3
			Pb-data4-3
			Pb-data6-3
	PseudoBoolean-opt-vtxcov	1	Pb-vtxcov-v2000-e4000-00
QuadraticAssignment	QuadraticAssignment-m1-s1	6	QuadraticAssignment-esc16e
			QuadraticAssignment-had12
			QuadraticAssignment-lipa20b
			QuadraticAssignment-tai10a
			QuadraticAssignment-tai10b
			QuadraticAssignment-tai25a
QueenAttacking	QueenAttacking-m1-s1	7	QueenAttacking-03
			QueenAttacking-04
			QueenAttacking-05
			QueenAttacking-06
			QueenAttacking-09
			QueenAttacking-12
			QueenAttacking-15
StillLife	StillLife-m1-s1	9	StillLife-03-03
			StillLife-03-12
			StillLife-04-08
			StillLife-05-09
			StillLife-06-10
			StillLife-07-07
			StillLife-08-08
			StillLife-09-09
			StillLife-10-11
TravellingSalesman	TravellingSalesman-m1-n15	2	TravellingSalesman-15-30-00
			TravellingSalesman-15-30-10
Auction	Auction-sum	1	Auction-sum-matching040_c18
PeacableArmies		1	PeacableArmies-m1-07_c18
Rlfap	Rlfap-opt	1	Rlfap-graph-05-opt_c18
SumColoring		1	SumColoring-dsjc-250-1_c18

Figure 8: Instances used for the experiment of the project.

B Appendix B

Instances name	BTD*					
	Time (CPU seconds)	# Nodes	# Backtrack	# Goods recorded	# Good usage	# NoGoods recorded
Knapsack-30-100-09	29.920	8,366,571	2,647,020	0	0	0
Knapsack-30-100-15	192.394	51,300,043	19,042,633	0	0	0
Knapsack-30-100-18	76.735	21,726,817	6,695,251	0	0	0
Knapsack-40-150-00	> 1800.000	443,073,025	184,479,977	0	0	0
LowAutocorrelation-003	0.000	38	6	19	83	0
LowAutocorrelation-008	0.045	3,956	225	2,649	35,782	640
LowAutocorrelation-012	4.248	130,723	3,963	87,023	1,246,075	31,601
LowAutocorrelation-015	313.255	1,228,510	28,742	848,458	12,495,324	295,778
LowAutocorrelation-020	> 1800.000	6,775,286	51,809	5,984,938	142,037,007	595,125
LowAutocorrelation-025	> 1800.000	5,583,639	24,600	5,210,152	168,874,187	258,171
LowAutocorrelation-030	> 1800.000	22,071	52	21,175	953,380	0
Pb-bgr-04	0.007	165	33	32	52	0
Pb-bgr-05	0.065	2,966	1,393	106	146	0
Pb-bgr-06	0.933	19,174	11,054	387	939	0
Pb-bgr-07	11.913	123,387	69,399	1,044	1,924	0
Pb-bgr-08	510.104	3,253,967	1,875,579	29,074	411,418	0
Pb-bgr-09	> 1800.000	7,481,244	4,027,513	124,951	1,647,669	0
Pb-bgr-10	> 1800.000	6,897,261	2,976,656	356,292	5,298,240	0
Pb-aim-050-2-0-yes1-2	3.227	159,548	72,044	373	5,512	72,099
Pb-aim-100-3-4-yes1-2	3.831	247,790	131,807	108	5,677	35,254
Pb-aim-200-1-6-yes1-4	> 1800.000	30,127,381	8,765,647	0	0	17,989,834
Pb-ii08a1	1.316	106,477	31,032	74,503	31,032	0
Pb-ii08a2	> 1800.000	28,431,518	12,037,864	12,542,532	2,932,914,237	0
Pb-jnh001	45.164	1,949,454	167,017	1,489,763	16,091,276	0
Pb-jnh207	0.151	4,880	475	2,343	25,001	0
Pb-par08-1-c	0.007	823	50	46	948	195
Pb-par08-5-c	0.009	1,053	101	56	1,173	249
Pb-factor-S9-P005-Q317	> 1800.000 **					
Pb-garden-4x4	0.014	1,910	1,037	787	6,739	1
Pb-garden-7x7	7.368	610,641	290,533	290,817	12,190,166	1
Pb-garden-9x9	395.837	17,047,377	8,145,125	8,417,282	657,665,824	0
Pb-garden-15x15	> 1800.000	65,442,709	31,736,821	33,706,798	2,693,749,646	0
Pb-garden-100x100	> 1800.000	33,927,696	16,301,287	17,617,543	1,277,536,585	0
Pb-gr-04	0.116	10,078	2,841	443	3,337	0
Pb-gr-05	0.336	33,763	4,793	3,772	25,082	0
Pb-gr-06	30.237	1,879,078	228,365	145,289	2,239,675	0
Pb-gr-07	333.450	12,609,294	1,885,120	678,442	19,314,218	0
Pb-gr-08	> 1800.000	27,536,957	5,824,319	1,294,308	43,797,022	0
Pb-gr-09	> 1800.000	24,317,372	4,530,853	1,478,059	48,553,635	0
Pb-logic-b1	0.063	9,471	4,081	1,354	21,819	37
Pb-logic-bbara-r	120.746	26,489,604	13,084,543	0	0	0
Pb-logic-c8	> 1800.000	136,910,582	59,522,367	37,288,454	2,131,613,454	0
Pb-logic-C17	0.002	298	132	118	394	0
Pb-logic-cm42a	> 1800.000	206,281,030	92,249,814	25,695,895	1,747,101,574	0
Pb-logic-dk512x-r	> 1800.000	232,359,869	115,788,848	0	0	0
Pb-logic-fout-r	> 1800.000	132,879,319	64,087,836	22,690,587	1,386,238,695	0
Pb-logic-m050-100-10-10-r	> 1800.000	111,423,684	52,787,079	53,466,096	1,372,299,937	0
Pb-msplit-4-30-3 *	87.621	14,269,492	9,607,678	0	0	0
Pb-msplit-6-50-2	> 1800.000	156,217,175	93,903,011	0	0	0
Pb-msplit-8-70-4	> 1800.000	83,078,015	43,063,384	0	0	0
Pb-msplit-opt-4-30-2	> 1800.000	498,812,328	57,956,245	930	52,483,536	2,558,015
Pb-msplit-opt-5-40-1	> 1800.000	473,279,915	38,029,655	2	76,059,182	0
Pb-mps-v2-20-10-bm23	1.254	123,317	74,140	0	0	0
Pb-mps-v2-20-10-lseu	> 1800.000	287,921,179	103,683,270	19,708,228	271,897,076	1,065,778
Pb-mps-v2-20-10-neos5	> 1800.000	62,267,738	30,503,908	0	0	0
Pb-mps-v2-20-10-p033	0.007	1,587	680	53	808	35
Pb-mps-v2-20-10-stein15	0.042	6,598	2,808	0	0	0
Pb-mps-v2-20-10-stein27	4.694	859,252	367,524	0	0	0
Pb-mps-v2-20-10-stein45	> 1800.000	320,794,996	140,513,795	0	0	0
Pb-routing-s3-3-3-1	0.442	24,650	11,023	738	36,372	3,700
Pb-routing-s3-3-3-3	3.431	157,398	99,268	2,571	88,957	9,372
Pb-routing-s4-4-3-2	> 1800.000	31,310,519	24,580,767	0	0	2,541,515
Pb-routing-s4-4-3-8	> 1800.000	15,366,894	2,562,143	4	4	11,168,501
Pb-circ4-3	0.909	72,074	2,102	30,298	673,790	70
Pb-circ6-3	N/A ****					
Pb-data4-3	0.960	72,074	2,102	30,360	676,786	8
Pb-data6-3	N/A ****					
Pb-vtxcov-v2000-e4000-05	> 1800.000	6,558,052	1,453,849	5,103,501	1,621,117,290	0
QuadraticAssignment-esc16e	> 1800.000	227,812,416	29,515,895	168	885,476,690	0
QuadraticAssignment-had12	> 1800.000	127,594,947	10,031,999	5,063	1,123,578,844	0
QuadraticAssignment-lipa20b	> 1800.000	68,288,028	3,647,952	4,379	1,181,932,195	0
QuadraticAssignment-tai10a	445.990	38,897,885	3,628,799	3,330	268,527,870	0
QuadraticAssignment-tai10b	4.724	563,611	135,248	2,797	1,056,119	3,388
QuadraticAssignment-tai25a	> 1800.000	52,126,212	2,026,604	5,069	1,118,680,565	0
QueenAttacking-03 *	0.000	0	0	0	0	0
QueenAttacking-04 *	0.046	2,791	1,891	0	0	0
QueenAttacking-05	4.319	151,619	84,959	66	5,549	1,684
QueenAttacking-06	> 1800.000	28,206,692	14,762,537	40	1,720,577	23,894
QueenAttacking-09	> 1800.000	1,339,544	1,009,890	0	0	0
QueenAttacking-12	> 1800.000	185,198	148,489	0	0	0
QueenAttacking-15	> 1800.000	31,730	24,231	0	0	0
StillLife-03-03	0.006	123	25	32	64	0
StillLife-03-12	0.047	3,336	578	644	8,223	11
StillLife-04-08	0.778	62,801	15,056	8,398	173,625	2,777
StillLife-05-09	29.781	824,281	222,044	138,391	3,625,377	86,270
StillLife-06-10	> 1800.000	4,503,572	1,213,085	852,365	26,407,455	607,952
StillLife-07-07	24.205	477,149	116,413	100,846	2,820,860	39,399
StillLife-08-08	1352.129	3,575,766	855,978	781,172	28,539,776	333,487
StillLife-09-09	> 1800.000	3,484,693	847,249	779,740	26,346,248	370,467
StillLife-10-11	> 1800.000	4,589,140	1,218,205	869,525	26,728,706	680,754
TravellingSalesman-15-30-00	> 1800.000	146,275,908	39,339,346	1,122	1,022,821,887	0
TravellingSalesman-15-30-10	> 1800.000	141,954,083	38,177,029	1,119	992,601,645	0
Auction-sum-matching040_c18	N/A ****					
PeacableArmies-m1-07_c18	> 1800.000 ***					
Rlfap-graph-05-opt_c18	> 1800.000	27,415,095	9,030,515	3,168,051	1,581,329,320	0
SumColoring-dsjc-250-1_c18	> 1800.000	3,004,473	3,004,287	119,430	17,872,182	0

* Instance is unsatisfiable
** Initial propagation did not finished after 13 hours wall clock time
*** Initial propagation did not finished after 2 hours wall clock time
**** Out of memory while solving, process killed

Figure 9: Benchmark results of BTD*