Dynamic programming on Nice Tree Decompositions

Experiments on globally solving Weighted Steiner Tree.

LW van der Graaff

February 24, 2015

Abstract

Connectivity problems such as the STEINER TREE are $\mathcal{NP}$-hard problems that are fixed-parameter tractable in the treewidth of the input graph. In this thesis a dynamic programming algorithm on nice tree decompositions is discussed. Based on observations on nice tree decomposition diversity and experiment results, a number of optimization heuristics are proposed to speed up computation. By restructuring the nice tree decomposition, all tested input graphs show a considerable improvement both in amount of processed partial solutions as well as computation time. Furthermore a number of different data structures are analyzed, that allow for various constant time operations for graphs of low treewidth.

Contents

1 Introduction

In this thesis an experimental evaluation of a dynamic programming algorithm on tree decompositions is given, for solving the Steiner Tree problem. The problem definition and algorithm implementation are discussed, with an emphasis on efficient data structures for bounded treewidth. A number of optimization heuristics are proposed which significantly speed up the execution of the algorithm by effectively processing parts of the graph in a different order. The proposed data structures and heuristics are tested on a set of benchmark graph instances to show their relevance in a practical setting.

The algorithm implementation is a continuation of the program code used in a preceding experiment [1] by Fafianie et al. This experiment shows the effectiveness of the reduce-algorithm introduced by Bodlaender et al. [2], applied on the Steiner Tree Problem. The results in this experiments will show that this method remains very effective and will yield even better performance used in conjunction with the proposed improved data structures and heuristics.

1.1 Problem definition

The goal of this experiment is to solve the Steiner Tree problem, which can be defined as follows:

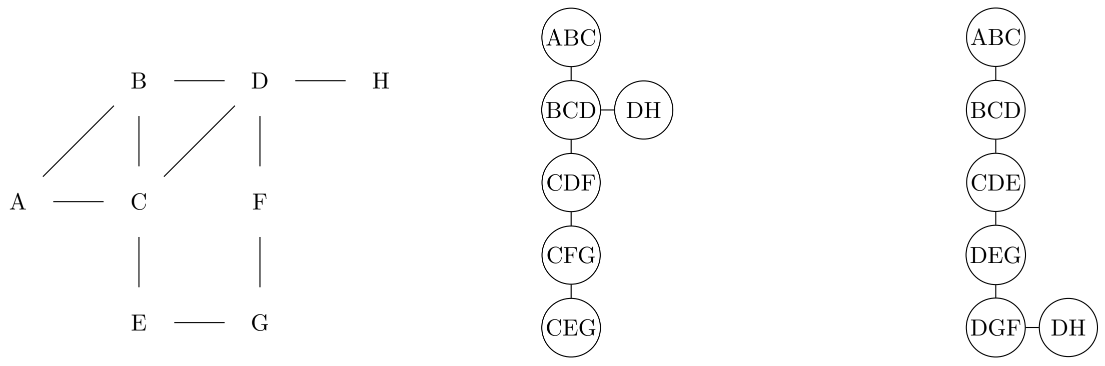
Steiner Tree
Input: A graph $G = (V, E)$, weight function $\omega : E \to \mathbb{N} \setminus \{0\}$, a terminal set $K \subseteq V$ and a tree decomposition TD of G of width `tw`.
Question: The minimum of $\omega(X)$ over all subsets $X \subseteq E$ of G such that $G[X]$ is connected and $K \subseteq V(G[X])$.

1.2 Tree decompositions

Instead of operating on the input graph directly, the algorithm will operate on its tree decomposition(TD). This is a non-unique representation of the graph $G = (V, E)$, consisting of multiple bags $B_x \subseteq V$ in a tree structure, with the following properties:

- $\bigcup_{x \in TD} B_x = V$
- $\forall e = (u, v) \in E, \exists x \in TD \mid u, v \in B_x$
- if $v \in B_x$ and $v \in B_y$, then $v \in B_z$ $\forall z$ on the path from x to y in TD

The width of a TD is the size of its largest bag minus one, and the treewidth of a graph is the minimum width over all possible tree decompositions. In this work the term MBS (MaxBagSize) is used to avoid confusion with the `-1` term that is used in the treewidth definition. The MBS gives a clearer view of the exact size of the problem, especially when discussing algorithm implementation at the data structure level.



(a) Input graph. **(b)** Tree decomposition. **(c)** Tree decomposition.

Figure 1: Input graph with two possible valid tree decompositions, both with a `MBS` of 3.

Figure 1 gives an example of a graph and two possible corresponding tree decompositions. A few observations can be made that can aid in the understanding of the concept of tree decompositions.

- The tree decomposition must obviously satisfy the three rules defined in this section. All vertices are present, all edges can be associated with one or more bags, and the presence of each vertex induces a connected subtree.
- The trivial tree decomposition consists of a bag containing all the vertices. This satisfies all the constraints but is not useful in practice. Tree decompositions with minimal or low MBS allow for efficient algorithm execution.
- Each non leaf bag splits the graph into multiple disconnected components, making it a separator for the graph. All paths between vertices of the surrounding bags must go through one or more of the vertices contained in the bag.
- Each clique is fully contained in a bag.
- Some bags can be connected to the tree at multiple locations, possibly reducing the degree of a bag.

Computing a TD of minimal `MBS` is $\mathcal{NP}$-hard, but can be efficiently approximated using heuristics, resulting in a tree decomposition that has close to optimal `MBS`. In this experiment the TD is generated using the GreedyDegree [3] heuristic, implemented in the libTW [4] library.

1.3 Nice tree decompositions

The TD is converted into a nice tree decomposition(nice TD), to limit the structure to a small set of possible transitions between bags. The nice TD remains a valid TD itself (with the same `MBS`), but only allows for four different transitions between bags, and introduces a root and leaves which both contain no vertices. The definition used in this implementation was introduced by Cygan et al [5], and nice tree decompositions were introduced by Kloks [6].

See Figure 2 for an overview. The four different transitions are:

- An `introduce` bag introduces a new vertex
- A `forget` bag removes a vertex
- A `join` bag joins two bags with equal contents
- An `edge` bag contains the same vertices as its child, but is labeled with the edge it introduces.

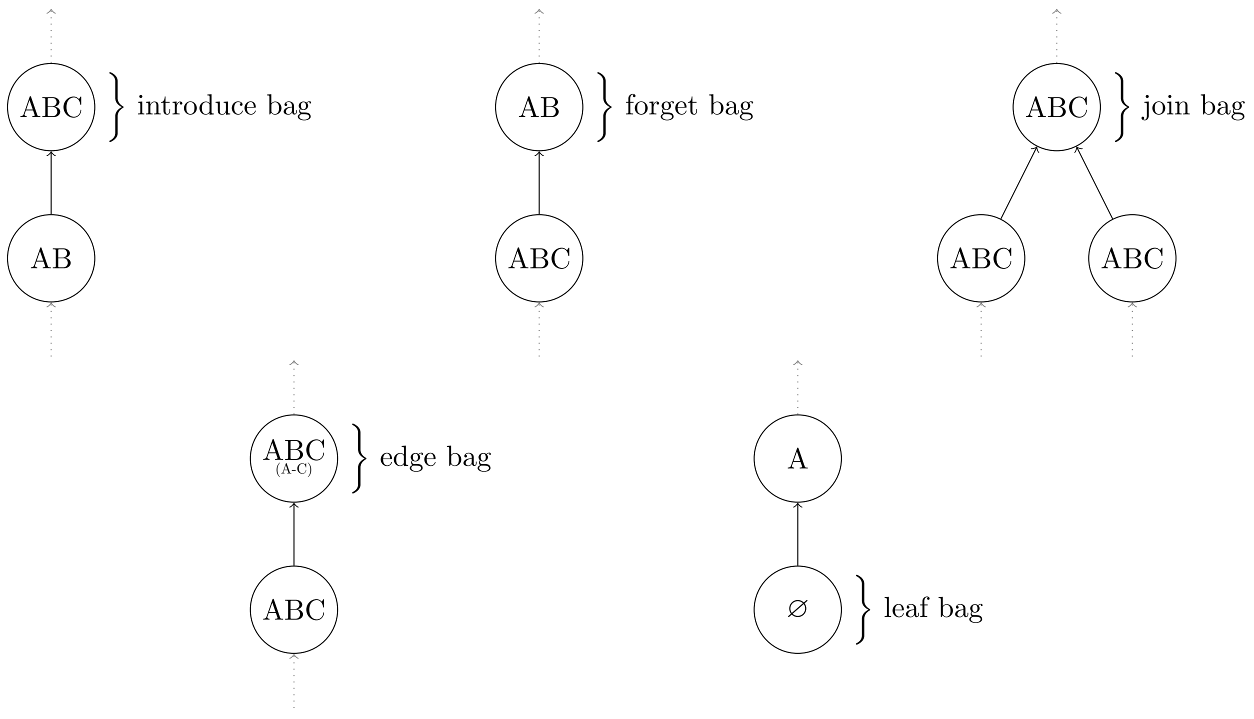


Figure 2: Bags in a nice tree decomposition.

The leaves are empty bags that have `introduce` bags as parents, and the tree is rooted in an empty `forget` bag. Since a vertex can only be present in a connected set of bags (forming a subtree), each vertex can be introduced multiple times, but only forgotten once. Each edge is introduced exactly once, somewhere within the subtree induced by the presence of its two incident vertices.

1.4 Dynamic programming

The dynamic programming algorithm starts in the leaves of the nice TD and traverses the tree up to the root. A table of partial solutions is created after processing a bag, which serves as input for its parent bag. The tables propagate from the leaves up to the root of the tree, resulting in a table with a single optimal solution, if such a feasible solution exists.

At the leaf bag the table just contains a single zero cost solution containing no vertices.

When a new vertex is encountered, it can either be added to a partial solution or discarded, effectively doubling the amount of input partial solutions. Partial solutions are grouped by the set of vertices that are chosen from the current bag. The join bag will then compute the Cartesian product between all the groups with the same used vertices to obtain all the possible joins of partial solutions from both sides. The cost of a joined partial solution is the sum of the costs of the two used partial solutions, since every edge is only introduced once within the nice TD.

A solution is by definition just the set of edges that form some subgraph in the input graph, with its cost being the cost of the subgraph. The partial solutions are however only represented by their accumulated cost and their local connectivity properties.

1.5 Solving the Steiner Tree Problem

The Steiner Tree problem (STP) fits nicely into this paradigm, allow for very straightforward operations to process the different types of nice bags. A partial solution can be uniquely defined as a partition of the chosen vertices into one or more disjoint subsets each representing a (part of a) connected component.

When a new vertex is introduced, it can either be added as a singleton to the partition(used), or be discarded(unused). These two options are applied to all input partial solutions, the output set will be twice the size of the input set. Note that if a vertex is a terminal, it is required to be used, thus no partial solution not using the vertex will be generated, and the output set will be of the same size as the input set.

When a vertex is forgotten, partial solutions that use the vertex but do not have it connected to any other used vertex are discarded. This follows from the definition of a tree decomposition, since after a vertex is forgotten, it cannot be present in any other bag in the nice TD, and thus never possibly be connected to any other vertex to form a connected and thus feasible Steiner tree.

When an edge is added, it can glue two disjoint subsets in a partition into one, if both subsets contain one endpoint of the edge. This increases the total cost of the solution with the cost of the edge.

When two partial solutions are joined, the two partitions are merged with the set union operator. For every vertex the union of the two connected components it was present in forms a new connected component in the output partial solution.

1.6 Reduction of tables

1.6.1 Automatic reduction

During execution, new partial solutions can be generated that are equivalent to existing partial solutions, either with equal or higher cost. Since equivalent partial solutions with equal or higher cost will never lead to better final results, they can safely be discarded. This operation can be regarded as trivial, but clearly represents the notion of eliminating partial solutions of which can be shown that they will not contribute to a better result. A drawback of this approach is that the algorithm only outputs a single optimal solution, even if multiple optimal solutions exist.

1.6.2 Refinement elimination

Given an ordered table of partial solutions, it is easy to spot candidates that will always be outperformed by a solution with a better score. A solution is a refinement if it has at least all the same cuts, showing that it is strictly worse connected. Given a partial solution s, find a partial solution s' solution that is a refinement of the solution s. The solution s'' in Table 1 connects A and B, which s did not, and thus it is not a refinement of s.

	partitions	refinement of s
s	`{{A},{B,C}}`	(yes)
s'	`{{A},{B},{C}}`	yes
s''	`{{A,B},{C}}`	no

Table 1: The refinement operator.

If such a refinement has equal or higher costs it may be discarded. It can be said that s represents s', since any partial solution t that would complete s' to a feasible solution, would also complete s to a feasible solution, and thus s' may be discarded, since it will never contribute to a better result.

If the implementation utilizes a data structure that provides a fast refinement operator, this method will be a viable optimization.

Trivially, if a partial solution is fully connected, all partial solutions of higher cost can be removed, since the unit partition only has itself as a refinement.

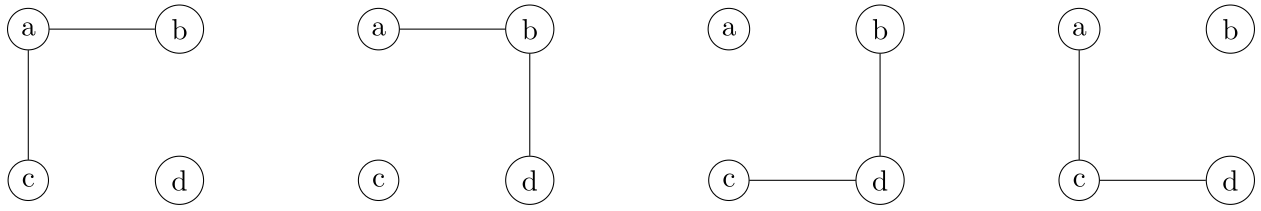
1.6.3 The Reduce method

The reduce-algorithm introduced by Bodlaender et al. [2] extends this concept further by utilizing representative sets. The main idea of this approach is that a table of partial solutions can be reduced by removing all entries of which can be shown that will not be able to contribute to a better final solution.

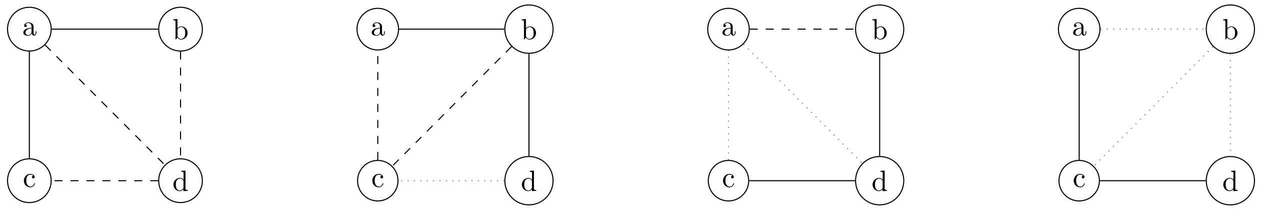
The entries of the resulting reduced table are said to *represent* the entries of the table that were eliminated. Instead of eliminating entries based on comparisons with other entries of equal or lower cost, entries are eliminated by finding a representative set of equal or lower cost entries.

Figure 3 gives an example of a set of three partial solutions that are representative for the complete input set of size four. It is easy to see that for any possible improving edge addition to the fourth partial solution, that same edge addition would also fully connect one of the three first partial solutions, making the fourth solution effectively obsolete.

In other words, for any given solution outside of the representative set, any extension that would complete it, would also complete a partial solution within the representative set. Since the partial solutions are ordered based on cost, the partial solution from the representative set would always produce a feasible solution at least as good as the solution otherwise produced, and therefore all solutions not in representative set can be discarded.



(a) Set of four input partial solutions, ordered from left to right, all of equal cost.



(b) The six possible edge additions(dashed) that would extend a partial solution to make it fully connected, the dotted edges also qualify but would result in a redundant solution.

Figure 3: Example of a representative subset of partial solutions.

A representative set can be found in a naive way by enumerating all possible extensions to a fully connected solution, for each partial solution in the set. If all of the found extensions for a partial solution will also lead to a fully connected solution with any different partial solution of lower cost, the solution can be discarded. The set of partial solutions remaining then form the representative set.

It can be shown that taking a basis in a certain matrix(see Figure 4) will yield the same results, and is substantially easier to compute.

$$\mathcal{M}[p,q] = \begin{cases} 1 & p \sqcap q = \{U\}, \\ 0 & \text{else.} \end{cases}$$

Figure 4: Matrix containing 1's where the meet operation between the two partitions yields the unit partition.

The matrix $\mathcal{M}$ can be written as the product of two cut matrices, $\mathcal{C}\mathcal{C}^T$, where $\mathcal{C} \in \mathbb{Z}_2^{\Pi(U)\times \texttt{cuts}}$. The term $\Pi(U)$ refers to all partitions of the used set of vertices, and `cuts` is the set of all two-partitions of the used vertices, where one vertex is fixed on one side. The matrix $\mathcal{C}$ contains a 1 where the partition is a refinement of the cut, or in other words is consistent with the cut.

The product $\mathcal{C}\mathcal{C}^T$ can then be seen to count the number of partitions that are consistent with both p and q. If $p \sqcap q = \{U\}$, then there is only one such partition (the partition between all vertices and the empty set). Consequently, if $p \sqcap q$ is a partition with at least two disjoint sets, then there is an even number of cuts that are consistent with both p and q, since the amount of consistent cuts is equal to $2^{\#blocks-1}$. Therefore the matrix $\mathcal{M} \equiv \mathcal{C}\mathcal{C}^T$ (in arithmetic modulo 2). Finding the basis of minimum weight in C will suffice to construct the representative set, and that can be achieved with Gaussian elimination, when the entries are ordered by ascending cost.

For the complete proof the reader is directed to the original work by Cygan et al [5].

Sorting the set of partial results on their cost and comparing entries comes at a significant computational cost, so the procedure should only be applied if it can discard enough entries to make up for the increase in computational cost. Therefore the reduce algorithm is only executed if the number of entries in the table is larger than $2^{|U|-1}$, guaranteeing the elimination of at least some entries, since the size of the basis cannot exceed $2^{|U|-1}$(the number of columns).

2 Data structures

2.1 Introduction

For every type of bag in the nice tree decomposition an operation is executed on the input set of partial solutions, resulting in a new set of at that moment still feasible partial solutions. In this section a number of data structures are discussed that allow for either faster execution time for the required operations, or a decrease in memory usage. The proposed implementations will enable the decrease of the overall runtime or memory consumption by a sometimes large constant factor, allowing for a larger instances to still be solvable within a reasonable time limit. The next sections will amongst other optimizations show how a bound on the treewidth can be utilized on the data structure level for a large performance gain.

2.2 Required operations

Table 2 gives an overview of the necessary operations that each implementation must provide for. There are also some auxiliary functions that are needed for some alternative implementations of the dynamic programming algorithm.

`add(c,v)`	Add a vertex as a singleton to a partition
`remove(c,v)`	Remove a vertex from a partition
`connect(c,v1,v2)`	Connect two vertices in a partition
`join(c1,c2)`	Join two partitions
`equals(c1,c2)`	Evaluate if two partitions are equivalent
`hashCode(c)`	Generate a 32-bit hash of a partition
`cutrow(c)`	Generate a cutrow representation of a partition
`areConnected(c,v1,v2)`	Evaluate if two vertices are connected
`isSingleton(c,v)`	Evaluate if a vertex is a singleton in a partition
`isRefinementOf(c1,c2)`	Evaluate if a partition is a refinement of another partition

Table 2: Overview of partition operations, and some optional auxiliary functions.

In the following sections a number of different data structures will be discussed, aided by an overview of the required operations' complexities.

2.3 Hashset

A naive approach is the construction of a set of sets to represent the partition, in the form of either linked-list or a hashset. Each vertex is labeled with a unique integer value, and a partial solution is equal to another if contains the same subsets with the same integer values.

Since the input graphs can be of large size, even for lower treewidth the memory requirement can be relatively large, especially with the added overhead for the set of sets datastructure implementation.

In previous work [1] this representation was used to demonstrate the effectivity of rank based reduce method. The approach discussed in the next sections is in practice an order of magnitude faster, partially because the previous implementation was not necessarily optimized for speed. Therefore it is omitted from further complexity comparisons and experimental evaluations, since it does not reveal anything other than an expected sub-par runtime.

2.4 Partial solution hashing

During execution the current set of partial solutions are stored in a hash table. When an equivalent solution with a lower weight is found, the old entry will be overridden by the new one, effectively removing the non-optimal partial solution. This requires a good hash function for the partial solution objects, that both executes fast and results in unique hashes, to avoid hash collisions.

An injective hash function, that maps every input object to a unique value, will result in an optimally performing hashtable. Theoretically all possible partitions of a set can be enumerated, resulting in a unique value that lies between zero and the bell number of the size of the set. The 15th Bell number is 1382958545, which is the highest bell number that still fits in a 32-bit integer, which is the hash size used in the hashtable implementation. (Applying the reduce function will reduce this number to $2^{\texttt{MBS}-1}$.) In practice a trade-off will need to be made between hash computation time and the performance decrease due to hash collision in the hashtable. Some of the data structures proposed in the next section store the partition in a limited amount of bits. This has the added benefit that the hash for the partition is just the integer value of the used bits, resulting in a constant time hash function that is also injective for sets of limited size.

2.5 Vertex coloring

In any data structure the need arises to uniquely identify each vertex to be able to perform the necessary operations and comparisons. The trivial approach is enumerating all vertices in the graph, resulting in an identifier between zero and the size of the graph ([0,n)).

Given the fact that the algorithm solves the problem progressively on the tree decomposition in a local fashion (bag by bag), we can map each vertex to an identifier(tree-index) in the range of [0,MBS).This concept lends itself form a vertex coloring in the graph, which is trivial to obtain if a tree decomposition is given. Given a nice decomposition, starting at the root assign an unused index when a forget bag is reached, and unassign(free) the index when the vertex is introduced in an introduce bag. This process ensures that in each bag, all vertices are always assigned to unique tree-indices, and each vertex has the same tree-index within its subtree in the nice tree decomposition.

2.6 Cutrow

Fafianie [1] introduced the idea of representing a partition by all of the cuts it contains. All possible cuts are enumerated, resulting in a bit string where each bit represents a unique cut of the set of vertices into two disjoint parts. Whenever the partition is a refinement of a cut, its corresponding bit is set to one. This results in a representation of size $2^{\texttt{MBS}}$ bits, or $2^{\texttt{MBS}-1}$ bits if we fix one vertex to always be on one side of the cut.

One of the benefits of storing the partition as a cutrow lies in the reusability of the representation in the Reduce algorithm. This way the $2^{\texttt{MBS}-1}$ operations previously needed to compute all the cuts are no longer required.

The indices of the bits in the cutrow are easily translated to the partition of vertices the cut represents. The index in two-bit representation defines the set of vertices grouped together on one side of the partition(defined as the *left* side). For example, in a graph with treewidth 3 has a cutrow of size 16. Index 13 is `1101` in two-bit representation, signaling that it represents a cut between vertex v_1, v_3, v_4 and v_2. Note that this is equivalent with index `0010` since this represents the same cut. This also implies that the first and the last index will always be set, since they represent a cut between the empty set and all vertices.

This means that all cuts are effectively represented twice in the cutrow. This can be countered by fixing one vertex to always be selected on the same side of the cut. The problem however is that when this particular vertex is forgotten, the resulting cutrow is obtained by a logical OR of the cutrow and its reverse. Computing the reverse of a bit string is generally not implemented at hardware level and thus a relatively slow operation ($O(\log n)$). Fixing a vertex to one side does result in only half of the memory requirement, and allows for one extra vertex to fit inside a single computer word.

In practice a tree decompositions will usually contain many bags of size smaller than the largest bag. Arguably using the same size cutrow representation is not efficient because of the higher memory usage and slightly more complicated operations. However this might not outweigh the cost of the operations needed for rearranging the cutrow when the size of the bag changes. Note that for smaller bags the tree-indices will most likely not be sequential, thus requiring a transformation operation when size increases or decreases. Filling up the gaps or introducing new gaps cannot be done fast without the usage of CPU instructions for bitwise packing and unpacking such as `PDEP` and `PEXT`, which are as of 2014 only supported in the most recent generation of CPUs.

`10000001`	initial state
`10000011`	introduce v_1
`10110011`	introduce v_3
`10100001`	connect v_1 and v_3
`10010001`	forget v_1

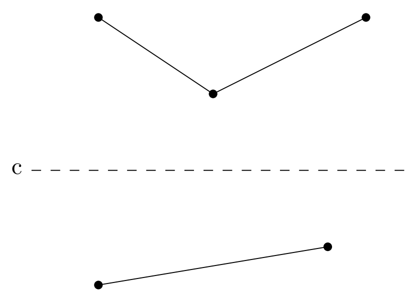
Table 3: Cutrow representation. Gray values represent cuts that are not applicable because the the implied set contains a vertex currently unused.

2.6.1 Introduce and Forget

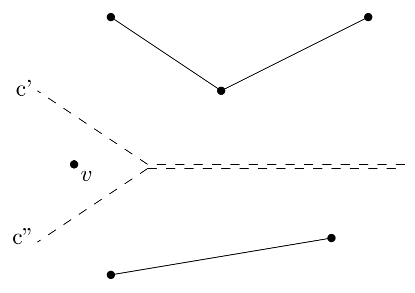
Introducing a new vertex maintains existing cuts(where the new vertex was not on the *left* side), and duplicates them all with the new vertex added (on the *left* side). This operation is easily executed with logical operators by merging shifted copy of the existing cutrow 2^{v_i-1} places to the left.

Consecutively, forgetting a vertex is the same shift back also using the logical OR operator. If a cut exists either with or without the given vertex(or both), it will still exist when the vertex is forgotten. If the cut did not exist with or without the vertex, it will also be nonexistent after the vertex is forgotten.

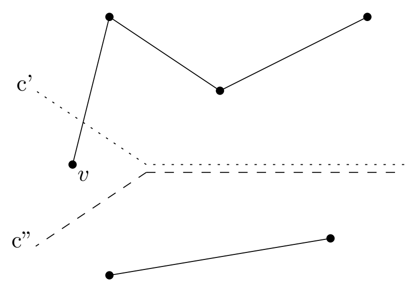
Figure 5 shows the process of adding and removing a vertex, showing the implications for a given cut.



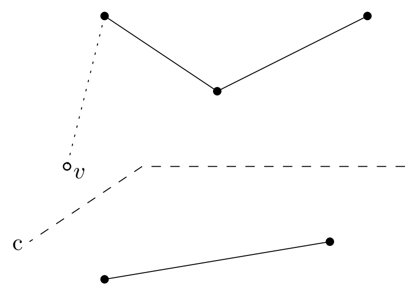
(a) Initial state containt two disconnected components, separated by a cut c.



(b) Introduce vertex v, existing cut c is duplicated into c' and c''.



(c) Cut c' dissolves with the addition of an edge.



(d) Cut c remains after forgetting v because either c' or c'' were still present.

Figure 5: Visualisation of adding and removing a vertex from the perspective of a single cut.

2.6.2 Join and connect

The join operation is trivial for the cutrow representation. When two partial solutions are merged, only the cuts that exist in both input solutions will remain. Simply applying the logical AND operation between the two bit strings will produce the required result.

Connecting two vertices uses precomputed masks to determine which cuts need to be eliminated. As stated earlier, the index of the bit signals which vertices are together on one side(the *left* side) of the cut. For each vertex we can create a mask for the bit string, that contains a one if the given vertex is on one side of the cut, as shown in Table 4. When connecting two vertices, all cuts that had the given vertices on opposing sides can be set to zero. The logical XOR of the two masks shows all relevant cuts, which can all be set to zero by computing the AND between the input bit string and the negation of the XOR between the masks.

`1010101010101010`	mask for v_1
`1100110011001100`	mask for v_2
`1111000011110000`	mask for v_3
`1111111100000000`	mask for v_4

Table 4: Cutrow vertex masks for a bagsize of 4.

2.6.3 Overview

As can be seen in Table 5, the cutrow allows for very fast execution of the required operations, but at a cost of an exponentially growing memory requirement. For larger instances, the relatively slow hash function also becomes a bottleneck resulting in both poor execution time in addition to the high memory usage. Faster less precise hash functions are possible, but will result in more collisions which also slows down the execution.

	MBS ≤ 6	MBS > 6
`add(c,v)`	$O(1)$	$O(\#$`words`$)$
`remove(c,v)`	$O(1)$	$O(\#$`words`$)$
`connect(c,v1,v2)`	$O(1)$	$O(\#$`words`$)$
`join(c1,c2)`	$O(1)$	$O(\#$`words`$)$
`equals(c1,c2)`	$O(1)$	$O(\#$`words`$)$
`hashCode(c)`	$O(1)$	$O(n^2)$
`cutrow(c)`	$O(1)$	$O(\#$`words`$)$
`areConnected(c,v1,v2)`	$O(1)$	$O(1)$
`isSingleton(c,v)`	$O(1)$	$O(\#$`words`$)$
`isRefinementOf(c1,c2)`	$O(1)$	$O(\#$`words`$)$
`memory usage in bits`	2^n	
`MBS for single 64bit word`	6	

Table 5: Operation complexity for the cutrow datastructure.

2.7 Adjacency matrix

Instead of approaching the partition in terms of disjoint subsets, it can also be represented in terms of adjacency between pairs of vertices. When two vertices are both present within the same connected component, they are marked as adjacent in the adjacency matrix, as shown in Table 6. Since this matrix contains redundant and unused fields, we represent it in row form as shown in Table 7.

This representation does contain some redundancy, for example when three vertices are connected, the connection between any two vertices is implied by both connections to the third vertex. However, as holds for every implementation in this chapter, there is always a balance between memory requirement (and thus redundancy), and operator speed.

Adding vertices requires no action, since it does not alter any connectivity.

Removing vertices requires all fields that represent a connectivity between the vertex and another vertex to be set to zero. (If all such fields were already zero, the vertex was a singleton and thus the solution was infeasible.)

Connecting two vertices requires that the neighborhood of both vertices should be merged, and updated for all vertices in that neighborhood. This is also required in the join during step 4 and 5, as explained in Section 2.7.1.

To check if a partial solution is a refinement of another partial solution, the logical OR between the two needs to be equal to the refinement, signaling that it strictly contains more cuts. Generating the cutrow can then be done by enumerating all cuts in the form of the adjacency matrix, and using the refinement operator to check if the partial solution contains the given cut.

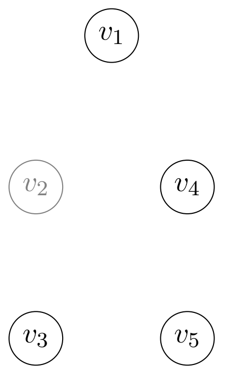
Figure 6 and Table 8 show an example of some consecutive operations with the corresponding adjacency matrix representation.

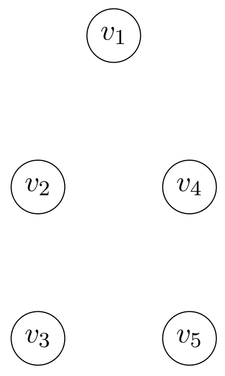
	v_1	v_2	v_3	v_4	v_5
v_1	–	0	1	2	3
v_2	0	–	4	5	6
v_3	1	4	–	7	8
v_4	2	5	7	–	9
v_5	3	6	8	9	–

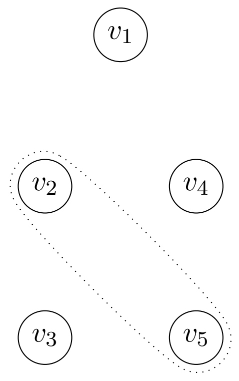
Table 6: Vertex adjacency in matrix form

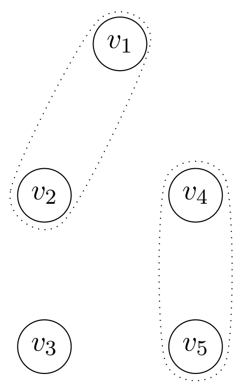
v_4	v_3		v_2			v_1			
v_5	v_5	v_4	v_5	v_4	v_3	v_5	v_4	v_3	v_2
9	8	7	6	5	4	3	2	1	0

Table 7: Vertex adjacency in row form


(a) Initial state.


(b) Introduce v_2.


(c) Connect v_2 and v_5.


(d) Join candidate from other branch.

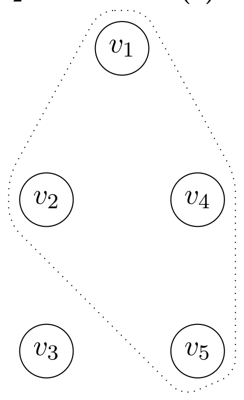

(e) Result after joining partial solution (c) and (d).

Figure 6: Example series of operations on partial solutions.

	v_4	v_3		v_2			v_1			
Figure 6	v_5	v_5	v_4	v_5	v_4	v_3	v_5	v_4	v_3	v_2
a	0	0	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0
c	0	0	0	1	0	0	0	0	0	0
d	1	0	0	0	0	0	0	0	0	1
e	1	0	0	1	1	0	1	1	0	1

Table 8: Adjacency matrix representation of operations shown in Figure 6

2.7.1 Join

The general concept of the join operation is to merge the two input solutions with a logical OR operation, and then to fill in the absent adjacencies that are implied by the context. The join operation consists of the following steps:

1. Merge both partial solutions together with the OR operation, and use the new partial solution

as intermediate result.

2. For both connectivities, mark vertices that are part of a connected component.
3. Select vertices that are part of a connected component within both partial solutions.
4. Extract the current neighborhood of the first marked vertex from the intermediate result.
5. Update the neighborhood of all neighbors of the marked vertex, with the extracted neighborhood.
6. Repeat for the next selected vertex, until all selected vertices are processed.

Each vertex that is not a singleton in both partitions, needs to ensure its neighborhood is fully connected. This can be done by extracting the neighborhood from the intermediate result, and adding it to all its neighbors.

Updating the neighborhood of a vertex can be performed in constant time, since the required portion(only the higher neighbors) of the retrieved neighborhood can be selected using a logical right shift (i.e. removing the lower portion). The neighborhood of a vertex can then be updated using an OR with a logical left shift of the selection. This results in a complexity of $O(n^2)$ for graphs of limited treewidth.

Table 9 shows hows these steps work in practice on the example from Figure 6.

	v_4	v_3		v_2			v_1									
	v_5	v_5	v_4	v_5	v_4	v_3	v_5	v_4	v_3	v_2		v_5	v_4	v_3	v_2	v_1
a	0	0	0	1	0	0	0	0	0	0	$conn(a)$ ②	1	0	0	1	0
b	1	0	0	0	0	0	0	0	0	1	$conn(b)$ ②	1	1	0	1	1
$a \vee b$ ①	1	0	0	1	0	0	0	0	0	1	$conn(a) \wedge conn(b)$ ③	1	0	0	1	0
											$N[v_2]$ ④	1	0	0	1	1
$adj(N[v_2])$ ⑤	0	0	0	1	0	0	$\underline{1}$	0	0	1						
result	1	0	0	1	0	0	1	0	0	1						
											$N[v_5]$ ④	1	1	0	1	1
$adj(N[v_5])$ ⑤	1	0	0	1	$\underline{1}$	0	1	$\underline{1}$	0	1						
final result	1	0	0	1	1	0	1	1	0	1						

Table 9: Join operation steps for the join in Figure 6, $\underline{1}$ marks adjacencies that needed to be added beyond the initial OR of the input solutions.

`add(c,v)`	$O(1)$
`remove(c,v)`	$O(1)$
`connect(c,v1,v2)`	$O(n)$
`join(c1,c2)`	$O(n^2)$
`equals(c1,c2)`	$O(\#\texttt{words})$
`hashCode(c)`	$O(1)$
`cutrow(c)`	$O(2^n)$
`areConnected(c,v1,v2)`	$O(1)$
`isSingleton(c,v)`	$O(1)$
`isRefinementOf(c1,c2)`	$O(1)$
`memory usage in bits`	$\frac{1}{2}n(n-1)$
`MBS for single 64bit word`	11

Table 10: Operation complexity for adjacency matrix.

2.7.2 Sparse adjacency matrix join

In general, a join operation has a complexity of $O(n^2)$ for this data structure, since in the worst case $O(n^2)$ adjacencies have to be set. However, since the neighborhood for a given vertex can be set in constant time if the treewidth is limited, an lower complexity might be achievable.

The adjacency matrix contains a lot of redundant data, since connected components of size n could be represented by $n-1$ bits(i.e. only the bits set for the lowest vertex), but in this form require $\frac{1}{2}n(n-1)$ bits. The consideration has to be made whether the added redundancy brings enough benefits in other operations to warrant the increase in complexity for the join operation. In this section an attempt is made to obtain the best of both worlds by accordingly switching between the two data structures. Instead of merging the two partials solutions directly, and filling in all the missing adjacencies, the partial solutions are transformed to their sparse form and are joined as such.

The sparse form is retrieved by just removing all adjacencies for vertices that do not have the lowest index within their connected component. The result is an adjacency row only containing adjacencies for the representative of the connected component, making it easier to isolate the different connected components from the partial solution. If a vertex is part of a connected component within both partial solutions, both neighborhoods can then be merged into the neighborhood of the lowest representative. Transforming the result back to the original form can then also be executed in $O(n)$ time, maintaining an overall complexity of $O(n)$ for the entire operation.

The remark has to be made that since the treewidth is bounded to fit the data structure within a computer word, adding extra overhead to computations might not be beneficial in practice. Since this discussion is approaching micro-optimizations and measuring the performance of this concept relies heavily on the actual implementation, lowering the complexity of the operations will not necessarily result in an optimal implementation.

On of the benefits of the normal adjacency matrix implementation is the very fast constant time refinement operation. If the refinement operator is used to compute the cutrow of the partial solution, this could become a bottleneck since the cutrow contains an exponential amount of entries. However, since the sparse data structure allows for easy extraction of all connected components(blocks), the cutrow can be computed in just $2^{\#blocks-1}$ time, which is always going to be faster than the otherwise required 2^n time.

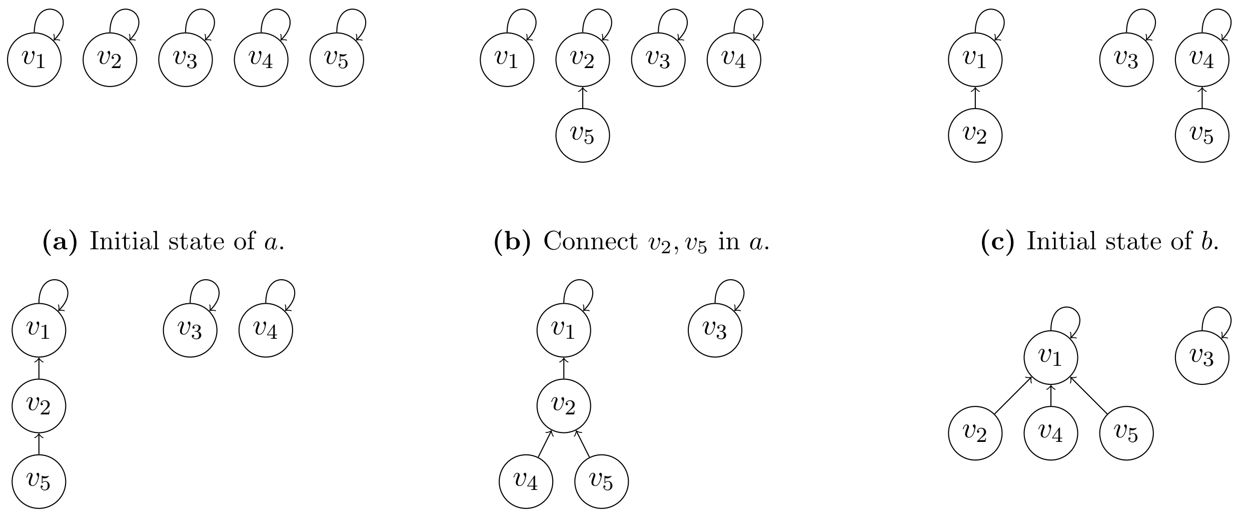
2.8 Union-find

The Union-find data structure stores the partial solution in terms of groups, alike the sparse adjacency matrix. However, instead of storing vertex relations in predetermined positions, each connected component has a vertex representative, and for each vertex only the representative of the component it belongs to needs to be stored. This requires less memory than the other approaches, but adds a slight overhead to some of the operator complexities.

When introducing a vertex, it will not be connected to anything yet thus it will be its own representative. When a vertex is forgotten, it is not allowed to be singleton, so it is only allowed to be its own representative if there are other vertices that have it as a representative. The component it represented should then pick a new representative(the next vertex with the lowest index), and then update all remaining vertices of the same component accordingly. Note that vertex v_1 will always have itself as representative, and thus does not need to be maintained in the data structure.

When connecting two components, the vertices of the component with the highest representative will obtain the representative of the component with the lowest representative.

When joining two partial solutions a and b, for each vertex v both representatives $rep(a.v)$ and $rep(b.v)$ are selected if unequal. Since these three(or possibly two) vertices are connected, they all obtain the lowest vertex as their new representative. $rep(v) = min(rep(a.v), rep(b.v))$, and $rep(max(rep(a.v), rep(b.v)) = min(rep(a.v), rep(b.v))$. This results in a complexity of $O(n)$. Furthermore the resulting structure needs to be compressed, because the partial solutions should be uniquely identifiable to enable fast equivalence checking with other partial solutions. The compression step can also be executed in $O(n)$ time. See Figure 7 for an example of such a join operation.



(a) Initial state of a. **(b)** Connect v_2, v_5 in a. **(c)** Initial state of b.

(d) Merge $b.v_2$ into a. **(e)** Merge $b.v_5$ into a. **(f)** Compress.

Figure 7: Example series of operations on partial solutions using the union-find data structure.

Generating the cutrow cannot be done efficiently using the refinement operator, because it requires linear time. However, each cut can be expressed in terms of a two-partition in the set of connected components. Since this data structure already stores its connectivity properties in terms of connected components, the cutrow can be generated by computing all combinations of connected components.

`add(c,v)`	$O(1)$
`remove(c,v)`	$O(n)$
`connect(c,v1,v2)`	$O(n)$
`join(c1,c2)`	$O(n)$
`equals(c1,c2)`	$O(\#\texttt{words})$
`hashCode(c)`	$O(n)$
`cutrow(c)`	$O(2^{\#\texttt{blocks}-1})$
`areConnected(c,v1,v2)`	$O(1)$
`isSingleton(c,v)`	$O(1)$
`isRefinementOf(c1,c2)`	$O(n)$
`memory usage in bits`	$(n-1) * \lceil \log_2 n \rceil$
`MBS for single 64bit word`	$16(19^*, 20^{**})$

Table 11: Operation complexity for Union find. (*Tightly packed bits,**theoretical maximum.)

2.9 Solution extraction

During execution, locally optimal solutions are maintained(by their local connectivity and their weight) and all non-optimal and infeasible solutions are discarded. On termination the weight of the optimal solution will be returned, if a feasible solution exists. To obtain the set of edges that form such a Steiner tree, the selected edges need to be maintained during the execution of the algorithm.

There are several ways to achieve this, where each solution has its own merits and caveats.

The most straightforward method is to label every edge with a unique number, and for each partial solution maintain a set of edges that were used during its construction. When two partial solutions are joined, the two edge sets can be merged, since every edge will only be introduced once within the nice tree decomposition. This however results in large sets that need to be maintained within every solution, and need to be copied often whenever new partial solutions are generated from existing ones.

A more efficient way is to label every edge with a unique index from 0 to m, and maintain a bit string of length m that represents a set of flags that can be set to true for each edge whenever it is introduced. Joining two solutions can be achieved by simply computing the logical OR between the two bit strings. This results in a fixed amount of m bits needed in the data structure for a partial solution.

If the resulted tree is known to be relatively small, the set of edge numbers might be considered, since this could require less storage space (i.e. if $|E| * \lceil \log_2 m \rceil < m$).

If the amount of edges is very large and the memory requirement for maintaining all used edges becomes a limiting factor, it also possible to opt for not storing the entire set of used edges, but merely a number of snapshots of intermediate results at various positions in the nice tree decomposition. For instance, before execution starts a number of bags can be selected of which the used vertices will be stored in snapshot within the partial solution data structure. Whenever one of these bags is reached, the partial solution stores which vertices it used from that bag, requiring a number of bits equal to the size of the bag. On completion, the algorithm can then be run a second time using the bit string to store the used edges. It will be able to execute much faster in the second run, since a significant amount of partial solutions can be discarded because for set number of vertices it is already known if they will be used in the chosen optimal solution. It is however key to choose promising bags beforehand that will allow for a large reduction of the search space, but also not too many bags because of the resulting increase in memory requirement. For instance bags of large size and bags that join large parts of the graph together will most likely be good candidates.

2.10 Overview

The join operation is often required and will significantly attribute to the execution time, since earlier experiments have shown that generally the majority of the time is spent in a few large join bags. Using a data structure that has a relatively small memory footprint and a fast join operator will therefore be a good choice for most use cases.

For lower `MBS` some benefits arise of fitting the entire data structure in a word(e.g. 32 or 64 bits), allowing for some very fast implementations using bitwise operators, fast hash functions and equivalence checking. For treewidth 5 or lower with 64 bits words the cutrow data structure executes very fast, and for treewidth 10 or lower adjacency matrix data structure performs well.

For larger treewidth the union-find data structure will be the most efficient pick, since it scales very well compared to the other data structures.

Note that it is difficult to give an overall judgment of the benchmark performance of one of these implementations, since despite the discussed theoretical improvements the optimizations introduce, the implementation is done in a language known for its sometimes suboptimal performance. This was however a decision made because of time constraints, and the validness of the underlying concepts will hold despite the less than optimal environment the experimental evaluation is conducted in. Section 5 discusses a series of experiments using the different data structures on a variety of graphs for a quantitative performance comparison.

3 Optimizing nice tree decompositions

Given any input graph, many different valid tree decompositions of the same width are possible. For every tree decomposition, all bags can be selected as root-bag for the nice tree decomposition, and the order of introduce, forget, edge and join bags can be varied. These options lead to a vast amount of possible instances that will all output an optimal solution (not necessarily the same solution), but compute in a different manner. Experiments have shown that execution times for different instances can vary greatly, generating an opportunity for optimization by creating better performing nice TDs.

The following observations can be made on the four different nice TD bag types.

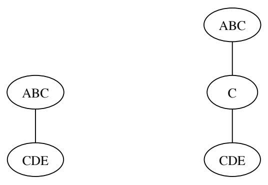
- An `introduce` node will increase the amount of partial solutions, by a factor of 2 (if the vertex is not a terminal).
- A `forget` node will decrease the amount of partial solutions.
- A `join` node will increase the amount of partial solutions.
- An `introduceEdge` node has little influence on the amount of partial solutions.

In the following sections a number of heuristics are defined to minimize the amount of partial solutions that propagate through the nice TD.

3.1 TD simplification

To maintain the ability to optimize the nice TD that needs to be generated, the input TD should not contain unnecessary bags. If a bag contains a subset of vertices of one of its neighbors, it can be safely removed. This results in more freedom for the nice TD generation algorithm, possibly enabling better nice TDs.

Between every two bags in the TD, the nice TD will contain a sequence of zero or more `forget` bags followed by a sequence of zero or more`introduce` bags. This order is optimal since the `forget` nodes will result in a decrease in partial solutions, so the `introduce` nodes can operate on a smaller input set. Because this order is fixed, an extra bag between every two bags in the TD can be added, containing the intersection of the two vertex sets, as shown in figure 8. This will not influence the structure any of the nice TDs that need to be generated, but does results in an increase of bags that can be used as root for the nice TD.

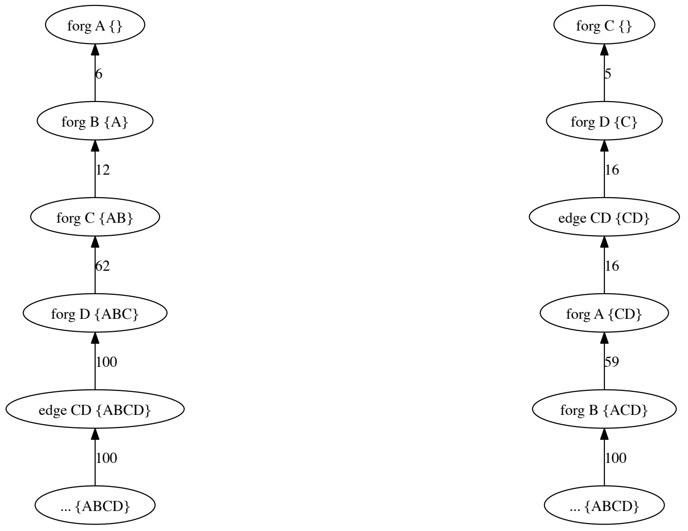


(a) Input TD **(b)** Output TD

Figure 8: The result of adding an intersection bag between connected bags.

3.2 Forget sorting

When a sequence of `forget` nodes is created, the order of the forgets can be optimized based on edges that need to be introduced before the vertex is forgotten. This sequence can be ordered based on the amount of edges that exist between the vertex to forget and its neighbors in the given bag. The `introduceEdge` nodes can the be placed higher within the sequence with the result that the amount of partial solutions will be lower, as shown in figure 9.



(a) unsorted **(b)** sorted

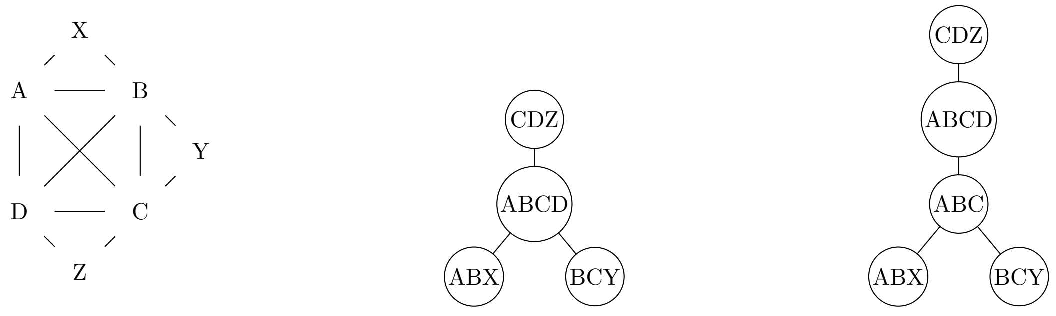
Figure 9: Example of Forget sorting. The edges are labeled with the amount of partial solutions a bag sends to its parent.

3.3 Join minimization

If a bag has a degree higher than two, then every next neighbor will require the addition of a extra join bag in the nice TD (and one more if the bag is the root of the nice TD). It is essential to keep the joins as small as possible since they require a large amount of computation time.

In some cases, an intermediate bag can be added between the bag and two or more of its neighbors, that has a smaller size and thus resulting in some of the joins to be of smaller size.

Specifically, if the union between two of the neighbors intersected with the bag itself returns a smaller bag, it will be beneficial to add this intersection as a bag, so that the two neighbors can meet each other in a smaller join. Figure 10 gives an example of this process.



(a) Input graph. **(b)** Tree decomposition. **(c)** Tree decomposition with minimized join (one viable option).

Figure 10: Example of join minimization.

3.4 Branch relocation

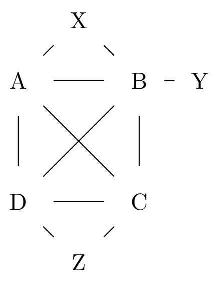
The structure of a given TD can be altered to some extent to reduce the amount of large joins, if all of its defining properties maintain intact. As shown in earlier examples, decreasing the degree of large bags will result in fewer large and computationally expensive joins.

A different approach to achieve this is by relocating branches of the tree such that the degree of a large bag is lowered, at the cost of increasing the degree of a smaller bag.

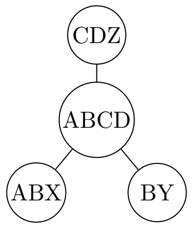
For each bag in the TD, consider all of its neighbors. If the vertices of the intersection between the bag and its neighbor are also present

For each edge in the TD, consider the intersection between its two incident bags. The subtrees induced by the presence of the vertices in the intersection can be split into two different trees, by removing the selected edge. The tree on one side now provides feasible attachment points for the bag on the other side of the selected edge. The goal is to attach one of the incident bags to its opposite subtree, to reconnect the two parts in a more efficient manner. This will result in a still fully connected feasible TD.

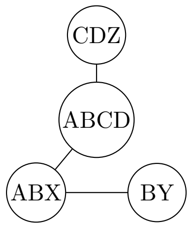
If both subtrees consist only of the incident bag, or if all other attachment points are either of larger size or equal size and lower degree, no improvement is possible.



(a) Input graph.



(b) Tree decomposition.



(c) Tree decomposition with relocated branch.

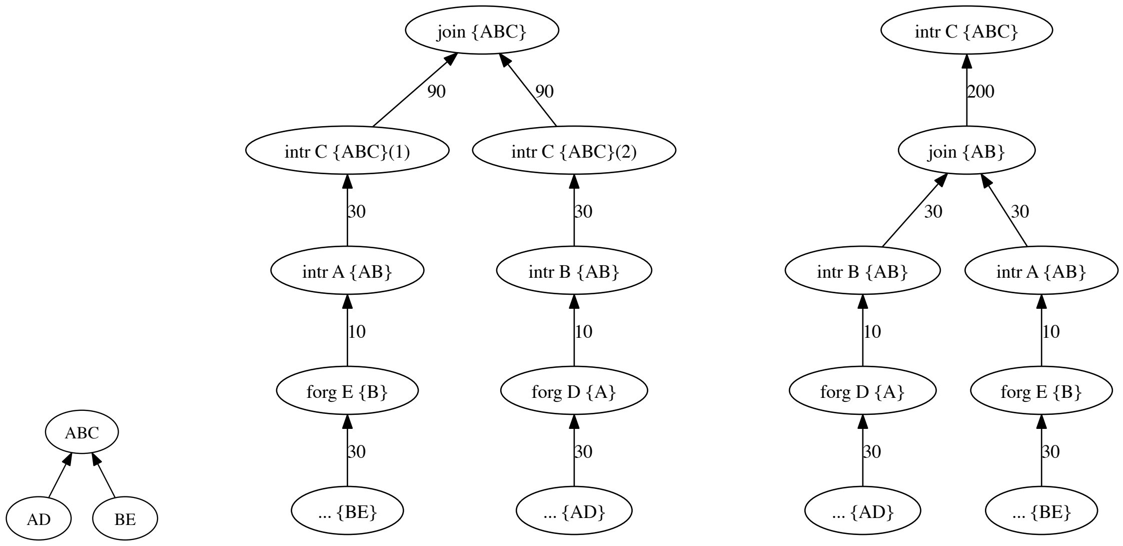
Figure 11: Example of branch relocation.

3.5 Children Intersection Join

If the rooted input TD has a bag with multiple children, the nice TD will insert join bags to accommodate this. In a naive approach, the entire bag will be copied twice, and connected to its two children using introduce and forget bags. Similar to the Join minimization approach earlier discussed, the goal is to minimize the size of the join where possible. Since the root is known, the same technique can be utilized for bags with two children in the rooted tree.

If both children do not require a vertex of the join bag to be present, they will still both need to introduce said vertex before the join occurs. If this introduction is placed above the join node, the join will be smaller (and thus more efficient), and also one less `introduce` node is needed, as shown in Figure 12. The set of vertices in the `join` node is the intersection between the parent, and the union of its children.

Note that even though this process overlaps with the join minimization procedure, it is less powerful since it cannot relocate children to newly formed bags, but benefits from operating on a rooted tree. Therefore this operation should always be executed on an already join minimized rooted tree.



(a) rooted TD **(b)** naive join **(c)** intersected join

Figure 12: Result of sorting forget nodes.

3.6 Edge introduction location

Every edge is introduced exactly once within the nice TD, and can be done anywhere within the subtree induced by its endpoints. The following observations can be made:

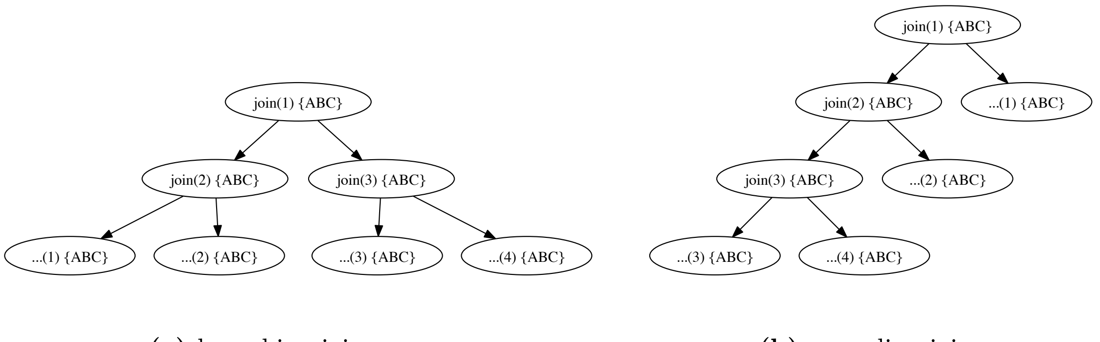
- It will be more efficient to introduce an edge after one ore more `forget` bags, rather than below. Forgetting vertices will decrease the amount of partial solutions, introducing an edge into a smaller set of partial solutions is more efficient.
- It will also be more efficient to introduce an edge before one ore more `introduce` bags, rather than above, since introducing vertices will increase the amount of partial solutions.
- If an edge is introduced directly after a `join` bag, it will be more efficient to introduce the edge above one of its children, since they will have a smaller set of partial solutions than the `join` bag has.
- Adding the edge before the join might provide for better pre-join reduction resulting in a smaller join.
- Adding the edge after the join will allow for a slightly less complex join.

The first two rules can be captured in the simple statement that a bag of smaller size is more likely to contain a smaller set of partial solutions. Adding the edge after the bag with the smallest size, will thus most likely be the most efficient choice.

If this bag is a join bag, one of the children should be picked, for example the child with the smallest subtree.

3.7 Multiple join order

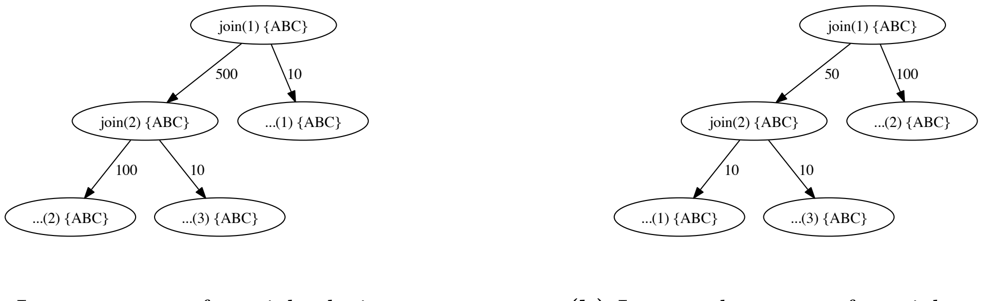
If there exists a bag in the TD that has more than two children, consecutive joins have to be constructed to connect the subtrees together. The structure of these joins can be varied resulting in different nice TDs, with possibly different execution times. An approach could be to cascade the joins, where every `join` bag has another `join` bag and a subtree as child, and the last `join` bag having two subtrees as children. A different approach might be to branch out `join` bags recursively until enough endpoints are created for all the children. Figure 13 shows the different construction techniques.



(a) branching joins **(b)** cascading joins

Figure 13: Two different methods for constructing multiple joins.

It might prove to be beneficial to include subtrees with a large set of partial solutions as high as possible, to avoid large intermediate sets of partial solutions, as shown in figure 14.



(a) Large amount of partial solutions **(b)** Improved amount of partial results

Figure 14: The join is rebalanced, moving bag 2 up, resulting in one small and one large join.

As shown both the structure and the order of joins can be varied and can result computations being executed in a different manner.

Acquiring a good balance can also be achieved by selectively joining branches together at runtime, based on the size of their partial solution tables. This way the opportunity is created to make an informed decision about the problem at hand. Different strategies are possible such as joining smallest first, largest first, or repeatedly combining largest and smallest. The downside of this approach is that the results of all branches need to be stored in memory before all the joins can occur.

3.8 Joincost

Choosing a different root bag for the nice tree decomposition results in a different tree and thus a different order of computations. For instance, if a bag has degree two it will become a join bag, where it would otherwise just result in a series of introduce and forget bags. Choosing a bag of degree higher than 1 as root, will result in a nice tree decomposition with one extra join. Since join bags are generally computationally expensive, it should be preferable to refrain from adding unnecessary extra joins.

Furthermore, if a given subset of vertices inside a join does not allow for a feasible partial solution on one side of the join, the computation of the set of partial solutions on the other side of the join will have been in vain. A different root bag might lead to a different computational order where partial solutions for that subset of vertices are never explored, resulting in an improved execution time.

By predicting the required amount of work in each join bag, the overall workload for a given nice tree decomposition can be estimated. This estimate will be called the joincost, and the root bag will preferably be chosen such that it results in a joincost-minimal nice tree decomposition.

The joincost of a nice tree decomposition is computed by executing the dynamic programming algorithm, but with the exception that all edges, when encountered, are always used. This process can be described as a fast feasibility check for all subsets, since it detects subsets that at some point do not contain any remaining feasible partial solutions and can be discarded.

The reduce function described in Section 1.6.3 states that a subset of n vertices results in a representative set of at most 2^{n-1} partial solutions. Using this bound, the amount of joins that will have to be computed inside a join bag can be counted, and the sum for all join bags results in the joincost of the nice tree decomposition.

Computing the join cost will be $O(2^{\texttt{MBS}})$ faster than executing the algorithm itself, since only a single solution needs to be maintained for each subset in a bag. When computing the join cost for all different root bags, results can be reused since every bag only requires to be computed three times, twice for both directions and once for when it is the root of the tree.

The added benefit of this approach is that for each join and forget bag, a list of infeasible subsets can be maintained. These subsets can then be pushed down the tree to the point of their creation, somewhere in an introduce bag. This will either be a bag that introduces one of the vertices in the infeasible set, or the highest introduce bag that creates a superset of an infeasible set. When the algorithm is executed, every time a bag is processed it will check the list of infeasible subsets and remove the applicable resultsets. This approach will yield a similar result to the method described in Section 4.10.

4 Runtime optimizations

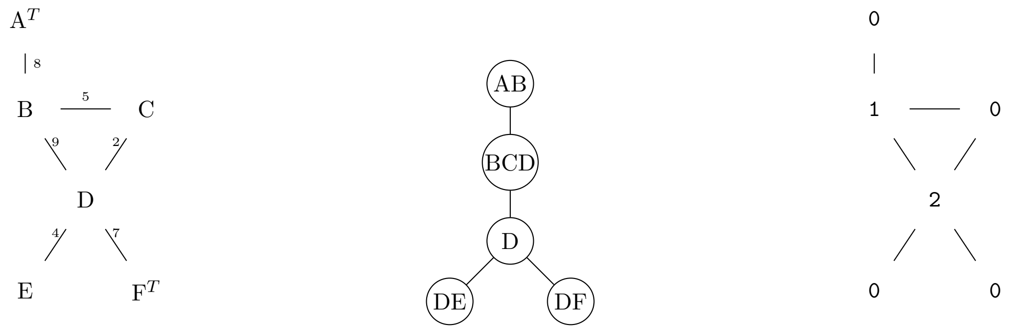
4.1 Vertex coloring

Using the concept of tree-indices as described in Section 2.5, all instructions can be represented efficiently using a small number of bits.

The nice tree decomposition can be transformed to a tightly packed instruction queue, and are sequentially processed while maintaining the state on a stack. The instructions are defined as follows:

- `leaf` pushes an empty set of partial solutions to the stack.
- `intr` introduced a vertex to the top resultset. It has a parameter for the vertex color and a boolean value if it is a terminal.
- `forg` forgets a vertex in the top resultset. It has a parameter for the vertex index.
- `edge` introduces an edge in the top resultset. It has two parameters for the vertex indices and one for the edge weight.
- `join` pops two sets of partial solutions from the stack, merges them and pushes the result back on the stack.

Table 12 gives an example of what the instruction queue looks like for the graph and tree decomposition in Figure 15.



(a) Input graph with two terminals.

(b) Tree decomposition.

(c) Vertices with colors assigned.

Figure 15: Example graph with vertex coloring extracted from the tree decomposition.

00	leaf
01	intr 0 F
02	intr 2 F
03	edge 0 2 4
04	forg 0
05	leaf
06	intr 0 T
07	intr 2 F
08	edge 0 2 7
09	forg 0
10	join
11	intr 0 F
12	intr 1 F
13	edge 0 1 5
14	edge 0 2 2
15	forg 0
16	edge 1 2 9
17	forg 2
18	intr 0 T
19	edge 0 1 8
20	forg 1
21	forg 0

Table 12: Instruction queue.

Each resultset of partial solutions is grouped by the vertices used by the solutions in the set, and identified by a bit string containing ones for all the indices of the used vertices. The presence of one or more vertices in a set can then be checked using the logical AND operator between the identifier and a bit string representing the required vertices.

4.2 Marking last bags

When a forget bag is reached and a partial solution contains the vertex to be forgotten as a singleton, the partial solution will be discarded since it can no longer form a connected tree. It is however possible to detect these cases earlier, by observing the connectivity possibilities for each vertex. Looking at the nice tree decomposition, every vertex has an introduce edge or join bag that has all other edges incident to itself as descendants. After this bag, the vertex can never be connected to any other vertices, thus if it is still singleton it cannot lead to a feasible solution.

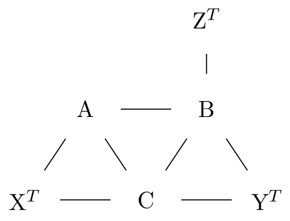
Every join or edge bag is labeled with the vertices it will be the last bag for, and all partial solutions containing any of the labeled vertices as a singleton can be discarded.

In case of an edge introduction, if it is the last bag for one of the incident vertices and it is still singleton, the edge must be introduced. In case of a join bag when two solutions are merged, if both partial solutions contain the labeled vertex as a singleton, the join can be canceled.

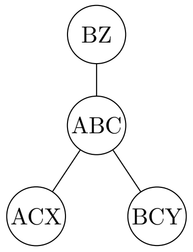
If there is a terminal not in the current bag and not in a descendant bag, there can not be any completed solutions present, since there still remain terminals that need to be connected to the Steiner tree.

Figure 16 shows an example of how this procedure can benefit the execution by enabling earlier removal of infeasible partial solutions. When the edge introduction bag for the edge AB is reached, the vertex A cannot remain singleton. This results in the solutions {A},{A,B},{A,BC} and {A,B,C} being removed before the join in bag ABC occurs.

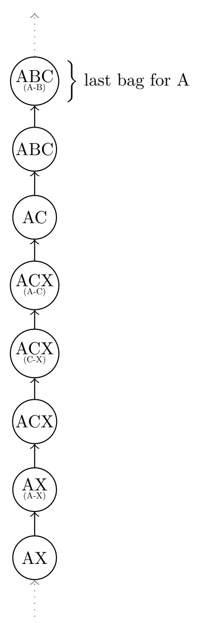
In the partial solution {A}, A is both singleton and the unit partition. Removing A would result in a completed solution, which is not necessarily infeasible. However, since there exist terminals in bags beyond the bag ABC that are not its descendants, a completed solution at this point will always be infeasible. Completed solutions may only emerge at or above the root of the subtree induced by the presence of all terminals, which is either a bag forgetting a terminal or a join bag.



(a) Input graph with three terminals.



(b) Tree decomposition.



(c) Part of the nice tree decomposition, before the ABC join.

Figure 16: Bags in a nice tree decomposition.

4.3 Lattice join

Each bag in the nice tree decomposition outputs a table of partial solutions, which can be subdivided into subtables that use the same set of vertices. For practical reasons, the subtables are stored as map, where the partition is the key, and the cost the value. When the tables are processed by the reduce function, the entries are placed into a list and ordered on their cost.

It is however also possible to store the entries in a lattice, using the refinement operator to create a partial ordering. Figure 17 gives an example of the lattice representation of two input solutions and its join.

A few observations can be made on the structure of this lattice.

- Each row represents the number the number of subsets present in the entry, equivalently signifying the amount of edge additions needed to complete the partition to the unit partition (each edge addition connects two subsets).
- A maximal and minimal element are always[1] present, where the former had accepted all edge introductions, and the latter none(and thus being a set of singletons).
- The cost of elements in the digraph should always be increasing, since if a partial solution is an equal cost or more expensive refinement of another solution, it can be discarded.

The two following two rules can then be constructed.

[1]The discard non-cheaper refinements operation, the mark last bags operation from section 4.2 and the last join limiting operation from section 4.5 do not preserve the lattice property that a minimum element always exists.

- Discard non-cheaper refinements.
- Discard siblings. If the cheapest partial solution join in a series of siblings is equivalent to its coarser parent, all higher cost siblings can be discarded.

The discard refinements rule can be extended to also discard cheaper refinements up to a cost difference the size of the smallest remaining edge in the graph. It is easy to see that the cost of the smallest remaining edge is a lower bound on the cost required to transform the refinement of any given partition to that partition.

Before the join commences, the discard non-cheaper refinements operation is executed on both lattices. The join is then performed by selecting a partial solution from one lattice and merging it with all partial solutions from the other lattice. First it is merged with the maximal element. Then all of its children are processed by increasing cost, and this process is repeated until all elements are visited. The discard siblings rule can be applied by comparing the result of a child with its parent. If it is equivalent, all equal or higher cost siblings need not be visited, since they cannot result in a lower cost or better connected result.

Both rules are applied in Figure 17. In the first phase element B4 can be removed, since it costs more than its coarser parent. During the actual join when A2 gets merged with B5, the result is equivalent to A5 being merged with B5, and thus A3 and A4 do no longer need to be merged with B5. In this example 15 joins are performed to merge two lattices of size 5, whereas that normally would require 25 joins.

4.4 Cycle detection

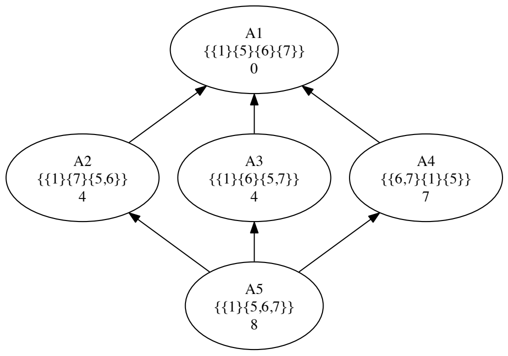
When two partitions are merged, and any two neighbors are adjacent in both partitions, the resulting partial solution will contain a cycle. Since the algorithm should return a tree structure, partial solutions containing a cycle can immediately be discarded. The challenge remains to detect such a cycle as early as possible, to avoid any unnecessary work.

A useful parameter is the amount of subsets a partial solution contains. For any partial solution, $\texttt{numSubsets} - 1$ signifies the number of edges required to complete the partition to the unit partition. Consequently, $\texttt{numVerts} - \texttt{numSubsets}$ is the number of 'connections' currently present, defined as `numConnections`.

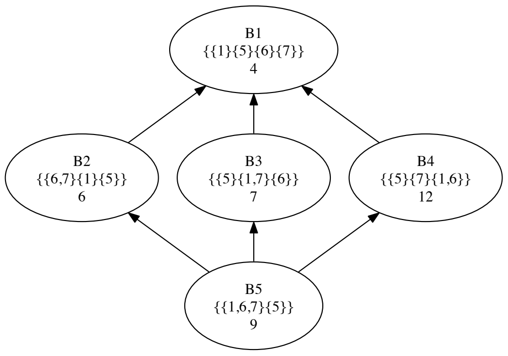
When joining two partitions, if $\texttt{numConnections}_a + \texttt{numConnections}_b$ is larger than $\texttt{numVerts} - 1$, the resulting partial solution must contain a cycle, and can therefore be discarded.

This however only applies to a limited number of pairs in the join. When a data structure is used based on vertex adjacency, a stronger measure can be applied. If the two partial solutions share any vertex adjacencies, a cycle will be formed when the two connected components are joined that both contained that adjacency. By detecting this in advance, the partial solutions do not require merging and can be discarded. In case of the proposed adjacency matrix data structure from Section 2.7, this can easily be computed by checking if the logical AND between the two adjacency matrices equals to zero.

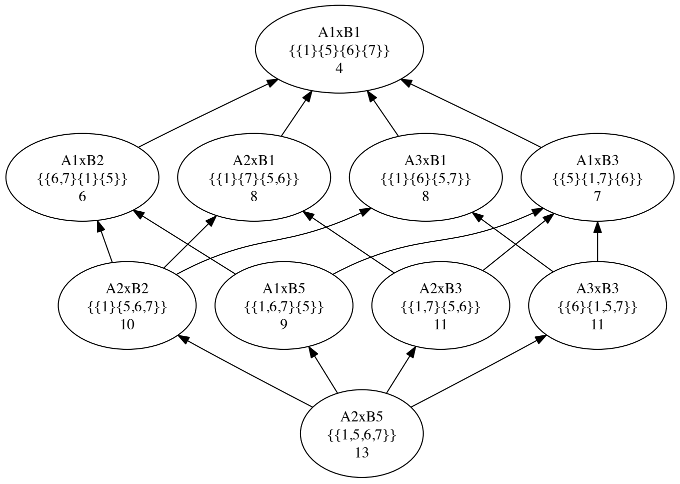
A cycle can also be formed when an even number of connected components are merged on an equal number of connection points. Each connection point between the two partial solutions should result



(a) Input partial solution lattice A.



(b) Input partial solution lattice B.



(c) Join result.

Figure 17: Example join with lattice representation.

in the decrease in connected components by one.

After the join is completed, this can be detected by counting the number of connected components in the resulting partial solution. If this is larger than sum of all input connected components minus the connection points, a cycle must be present somewhere.

4.5 Last join limiting

At the top of the nice tree decomposition, only a limited number of edge introductions will remain, which can be exploited in the final join. As discussed in the previous section, the number of subsets in a partial solution indicates the number of edges required to complete it to a unit partition. By counting the number of remaining edges above the last join, a bound can be created to limit the partial solutions it produces. If $\texttt{numConnections}_a + \texttt{numConnections}_b + \texttt{numRemainingEdges} < \texttt{numVerts} - 1$ for a given combination, not enough edges remain for it to be able to become fully connected so can thus be discarded.

The effectiveness of this process can be maximized by minimizing the number of edge introductions after the last join. When this number is zero, the join will output at most one solution for each unique set of used vertices.

4.6 Join bounding

The lattice representation of the partial solutions allow for optimizations that would otherwise not be possible. However, creating and maintaining this representation is costly, since each element needs to store a sorted list of parents. Traversing the lattice recursively also yields a lot of redundant visits.

Inspired by the lattice representation a number of bounds can be computed that operate on an ordered list, but utilize some of the lattice properties. Processing partial solutions in ascending cost combined with the use of bounds on cost and connectivity, enables the skipping of the remaining partial solutions. Both lists of input solutions are sorted, and every partial solution from one list is joined with all partial solutions from the other side, in ascending order.

During this process, a local and global bound are maintained, consisting of a joined solution and its cost. The global bound is constructed by joining the two maximal partial solutions, and the outcome is injected into the resultset.

The local bound is constructed by joining the solution from the outer loop with the maximal solution from the list in the inner loop. The local bound is also injected into the resultset, and in the case that an equivalent solution of lower cost already existed, the local cost bound will adjusted accordingly.

During execution, if either the local or global cost bound are reached, the inner loop gets terminated. If at any time a join results in a solution equivalent to either the local or global bound, the corresponding cost will be updated to the new value and the inner loop will be terminated.

Figure 18 shows an example of the join bounding procedure in action. In this example, a naive implementation would require 32 joins, or 26 with cycle detection, but now only 21 joins need to be computed.

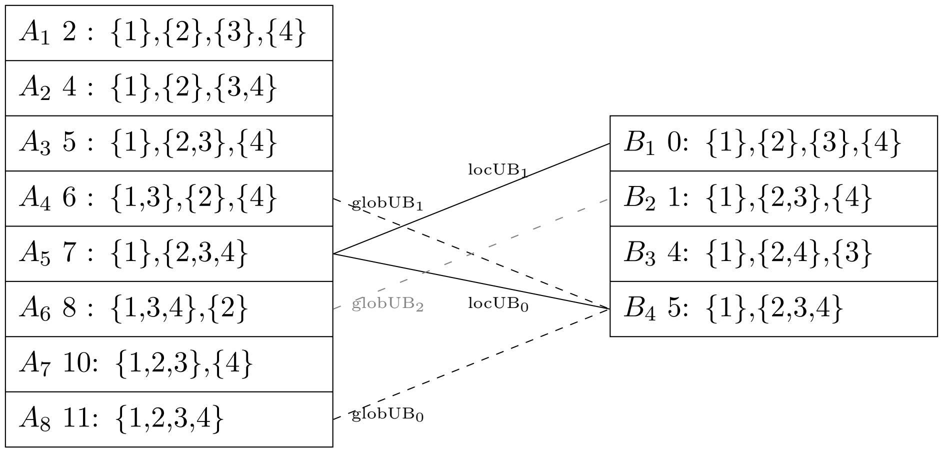


Figure 18: State of an example join showing successive local and global bounds, at point in time when A_5 is processed. The local bound allows that A_5 only needs to be joined with B_4 and B_1. When the global bound is updated at $A_6 \times B_2$ (optimal), no cheaper joins can be made so the procedure will terminate quickly.

A second local bound can also be maintained, by pre-computing the maximum partial solution of the first set with each entry from the second set. This bound can then be used to skip single entries in the inner loop, for which the bound has already been reached at an earlier state.

4.7 Join sorting

Before a join commences, the cost of the result is known as it is just the sum of the two input solutions. This can be used to create a sorted list of outputs, without actually computing the joins. With this order being known, a number of joins can be omitted by maintaining a number of bounds.

The join operation is split into two phases. First all pairs are iterated, and sorted on their combined costs. Each partial solution is joined with the highest cost solution from the other set, and stored in a map. This results in two maps, mapping a partial solution to its maximally connected result.

Then the actual joins are executed in order of increasing cost, and the results are stored in a map, mapping the result to its cost. Before the join is computed, the result map is queried to check if the one of the two maximal solutions is present. If so, it is of equal or lower cost than the current combination, and will be of equal or better connectivity.

Whenever a solution is acquired that is equal to the join of the two maximum partial solutions, the process can halt since all remaining pairs will not be able to yield better results.

The cost upper bound on initiation is the cost of the most expensive partial solution, which is the join between the highest cost partial solution from each set. Whenever a join yields a partial solution equivalent to the

Since the output of results is sorted, the reduce function can be embedded in the process. The cut matrix is continuously constructed and reduced, and when a basis is found the size of the number of columns, the entire process can be halted.

4.8 Lattice based reduce

As discussed in earlier sections, the lattice property of the resultset can be used to construct certain optimizations.

The reduce algorithm discussed in Section 1.6.3 is able to reduce the size of the resultset to 2^{n-1} entries. This bound can be explained in a straightforward matter.

Given a set of used vertices in the graph, a viable solution is just a spanning tree of the given vertices. Every bag in the nice tree decomposition separates a processed part of the graph from the rest of the graph. The partial solutions available in a bag for a given set of used vertices, represent feasible intermediate spanning trees in the processed part of the graph, including the current bag. These intermediate spanning trees can consist of multiple components, all connected to one or more vertices in the set of used vertices. If any component of an intermediate spanning tree connects to a single vertex in the current set of used vertices, it does not matter which vertex this is, since all used vertices will eventually be connected in the spanning tree. A component can also connect two or more vertices, resulting in multiple viable possibilities. The amount of connections the intermediate spanning tree makes between vertices in the bag (between 0 and $n-1$), defines the number of possible configurations.

When a partial solution contains one connection and thus connects two vertices, there are $\frac{1}{2}n(n-1)$ possible combinations of vertices. However, if there are multiple partial solutions containing one connection, it is easy to see that the cheapest $\binom{n-1}{1}$ solutions form a representative set for all other possible partial solutions.

The intuition here is that the number of partial solutions required to represent all possible partial solutions, can be found by combining two representative sets of of a resultset with fewer vertices. When two vertices are used, the lattice contains two rows with each one partial solution, one with zero connections and one with a single connection. When an additional vertex is used, the row containing one connection can be constructed by either connecting the new vertex to any other vertex using all $\binom{1}{0}$ partial solutions, or adding it as a singleton to all $\binom{1}{1}$ partial solutions. The number of partial solutions in each row using k connections can be found by combining $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$, which is $\binom{n}{k}$.

Approached from a different angle, if the graph is a clique of size n and all edges have been added, each row with numConnections k in the representative lattice can be constructed by taking all $\binom{n-1}{k}$ combinations of k edges from a $n-1$ size minimum spanning tree in the clique.

For each number of connections 0 to $maxConn$ there is a row in the lattice, and as is known from Pascal's triangle, this will add up to $2^{maxConn}$ partial solutions. This property can be applied when the reduce algorithm is executed. Whenever the basis contains the appropriate number of partial solutions for the given number of connections, it can discard all other partial solutions with the same amount of connections. This is especially useful for the sorted join discussed in Section 4.7, where the reduction is applied during the computation of a join. If the basis is complete for the given number of connections the joined output will have, the candidate can be discarded without even having to compute the join.

This also means that every optimal resultset is always exactly of size $2^{\mathrm{m}axConn}$. Note that this is a stronger bound than 2^{n-1}, since the maximal partial solution does not necessarily need to be the unit partition and contain $n-1$ connections. In other words, when $\mathrm{m}axConn$ is lower than $n-1$, there are $2^{n-1-\texttt{maxConn}}$ cuts that will be present in all partial solutions. Therefore the basis in the cut matrix will be of smaller since, since all the columns representing these cuts can effectively be discarded.

In earlier work the threshold of 2^{n-1} was introduced to determine if the overhead of executing the reduce algorithm could be justified by its performance gains, only executing the reduce step if the number of partial solutions in the resultset exceeded that bound. Although it showed good results in practice, this bound is slightly misleading. Since $2^{\mathrm{m}axConn}$ forms a lower bound for the size of the resultset, the amount of results can only be lower than 2^{n-1} if at least one cut is present in all partial solutions. Finding a basis in such a matrix will be computationally expensive because the matrix contains a lot more columns than necessary. In stead a cut matrix should be constructed with the appropriate number of columns, significantly reducing the cost of the operation. This does not mean that the overhead is always justified if the amount of partial solutions is only slightly smaller than $2^{\mathrm{m}axConn}$. Choosing a threshold to determine the execution of the reduce algorithm might prove to be useful, for instance if the size of the resultset is more than 5% above the lower bound.

The performance gain of the join minimization step discussed in Section 3.3 can be partly attributed to the existence of the used (incorrect) bound. Whenever a vertex is added as a singleton, the number of partial solutions for a set of used vertices remains the same, whilst the bound grows with a factor of two, removing the opportunity to apply the reduce algorithm.

4.9 Connection based partial solutions

For a given (fully connected) set of used vertices, a representative set of partial solutions will be of size 2^{n-1}. However, in some cases it will be possible to 'represent' the representative set with a smaller number of partial solutions.

Consider a graph that is a clique of size n, and a corresponding tree decomposition consisting of a single bag of size n. Given a set of vertices and a set of edges that connect these vertices, only the $n-1$ edges that form a minimum cost spanning tree are required to construct a representative set. Every row i in the partial solution lattice can be constructed by selecting i edges from the spanning tree, resulting in $\binom{n-1}{i}$ partial solutions in each row, and 2^{n-1} for the whole lattice.

This concept can be applied to the general case, with some extensions. Given a set of used vertices in a bag, all feasible partial solutions that connect none of the selected vertices are stored. For each of these partial solutions the cost of the already selected edges is maintained, and at most $n-1$ edges that form a minimum spanning forest(MSF). Note that these stored edges are not necessarily present inside the current bag, but can also exist elsewhere, as long as their addition connects the given two vertices.

The required operations then become fairly trivial, as only the MSF needs to be recomputed after each step. When an edge is added, the partial solution will contain at most n edges, so at most one edge can be discarded. When two partial solutions are joined, their costs are added, the edge sets are merged and can again be reduced to at most $n-1$ edges. If a vertex is forgotten, a minimum weight incident edge is selected, increasing the cost of the partial solution with its weight. This follows from the cut property of a minimum weight spanning tree, that states that for any cut in the graph, a minimum weight edge in the cut will always be present in an optimal solution. When a vertex is forgotten, all its incident edges must have been added at some point. Because the MSF was maintained during all operations, a minimum weight edge must be present.

All operations can be computed in linear time, and the partial solutions require little memory since only the cost, and $n-1$ edges with their weight need to be maintained.

Since a join is the Cartesian product between two resultsets, the amount of partial solutions will grow quickly.

The straightforward way to reduce the resultset is by eliminating partial solutions that have the same edge set as another partial solution, with costs at least as high. This can be computed in $O(\texttt{numResults}^2)$ time, but will only be able to bound the resultset to a size of $O(n^n)$. Given a representative resultset from the normal approach, its is clear that every solution in this set can also be represented by a connection based partial solution. The resultset bound of 2^{n-1} thus also holds for connection based partial solutions, with an important distinction that it is an upper bound, and that most likely a smaller resultset will suffice.

Each connection based partial solution of $n-1$ edges can generate 2^{n-1} regular partial solutions. After a join, at most 4^{n-1} connection based partial solutions will be present, resulting in 8^{n-1} regular partial solutions, which can then be reduced using the earlier discussed rank based approach in $O(8^{2n})$ time. This is clearly not a very efficient approach of the problem, since computing this reduction will only be feasible if the amount of input partial solutions is significantly lower that the theoretical bound of 2^{n-1}. However, due to time constraints only a straightforward approach using the existing techniques was used, to show the relevance of this alternate representation.

Figure 19 shows a part of a graph during execution, and Table 13 shows a snapshot of the corresponding table of connection based partial results. The results in this table can be used to construct a representative set, shown in Figure 20, after which all unused results can be discarded.

id	cost	edges	ab	ac	ad	ae	bc	bd	be	cd	ce	de	linR	rankR
c_1	0		.	.	.	2	.	6	.	3	.	3		
c_2	1	dx	.	.	.	2	.	6	.	3	.	1		
c_3	1	cy	.	.	.	2	2	.	.	3	.	3		
c_4	2	dx,cy	.	.	.	2	2	.	.	1	.	1		
c_5	2	bz	3	.	.	2	.	.	.	3	.	3		x
c_6	3	bz,cy	.	.	.	2	2	.	.	3	.	3	x	x
c_7	3	bz,dx	3	.	.	2	.	.	.	3	.	1		x
c_8	4	bz,cy,dx	.	.	.	2	2	.	.	1	.	1	x	x

Table 13: Connection based partial solutions. The last two columns mark which entries can be discarded with respectively the linear and the rank based reduce function.

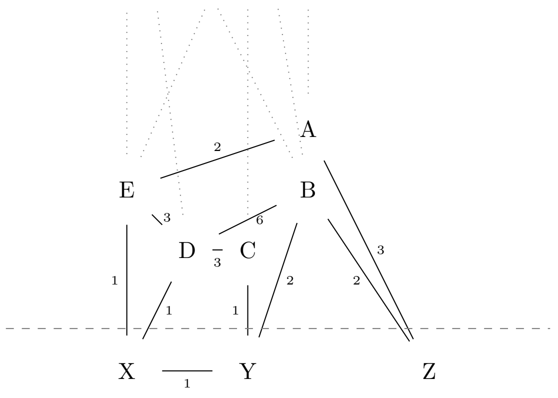


Figure 19: Part of the input graph, where ABCDE exists as a bag in the tree decomposition. X, Y and Z have been forgotten, and some of ABCDE's edges have been added.

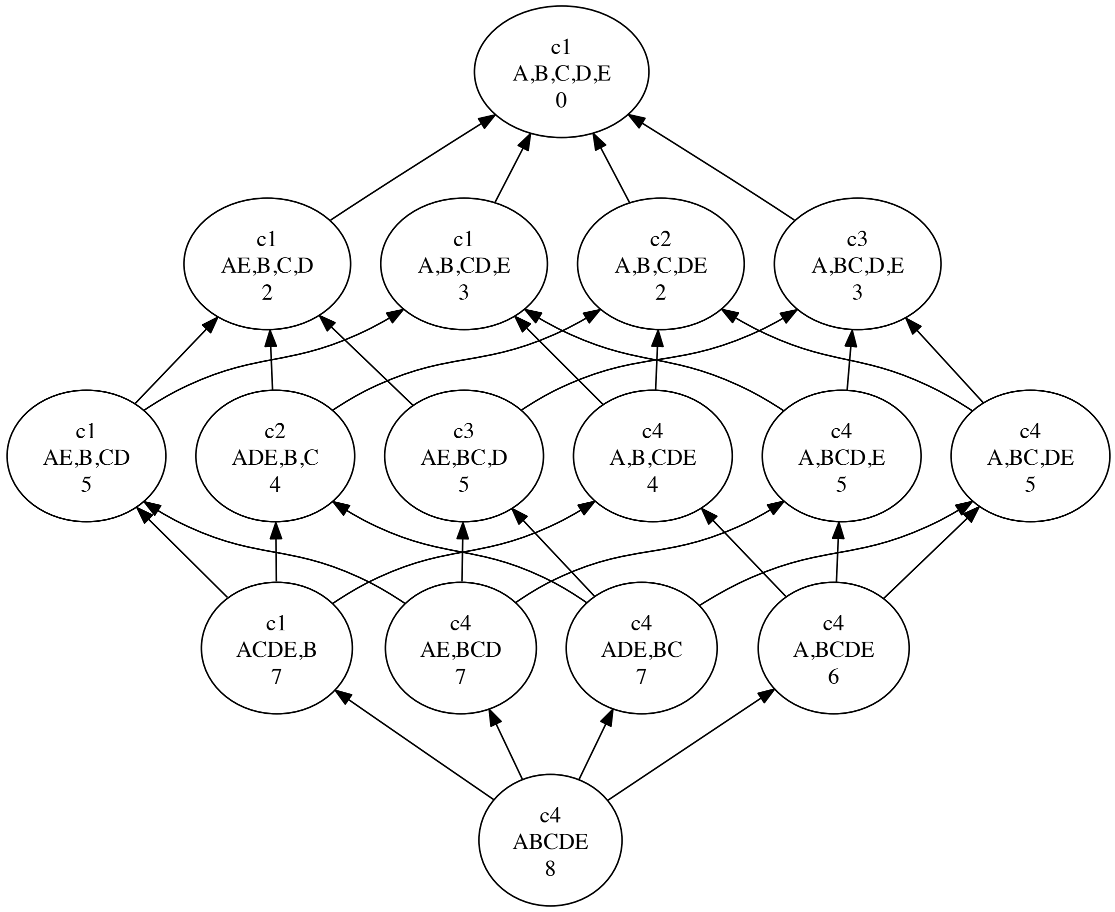


Figure 20: Representative set in lattice form constructed from the set of connection based partial solutions. Each partial solution is identified by its equivalent connection based partial solution($c_1 - c_8$), its connectivity and its cost.

4.10 Vertex constraints

Given the definition of a tree decomposition and a Steiner tree, the observation can be made that there exists a subtree in the tree decomposition induced by the presence of chosen vertices in bags. This follows directly from the fact that a Steiner tree is connected, and for all edges in the graph there must be a bag in the tree decomposition containing both endpoints. From this concept a constraint can be formulated stating that for every bag b for which a bag b' exists for which holds $b' \subset b$, every partial solution in b must contain at least one of the vertices from b'.

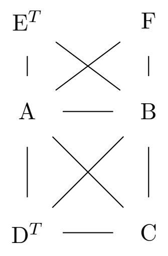
The tree of used bags in the tree decomposition is at least the size of the subtree formed by all the bags in between the bags containing terminals. All bags in this subtree are safe for use as constraining subbags.

Note that if the subbag b' contains a terminal, it will not be useful as a constraining bag. Because bags that are a superset of b' cannot have have partial solution sets containing no vertices form b', since the terminal will by definition always be used.

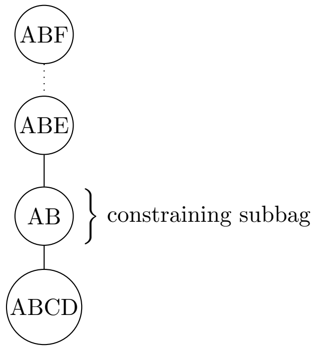
Before computation commences, each introduce bag will get a constraint assigned if available, retrieved from the smallest constraining subbag that is not a descendant. If there exists a set of partial solutions using none of the vertices from the constraint, whilst all of them have already been introduced, the entire set can be discarded.

Figure 21 gives an example of an application of the discussed procedure. After all of A, B, C and D have been introduced, the sets of solutions using only C, D or CD will not exist since the empty solution was discarded after enforcing the constraint when B was introduced.

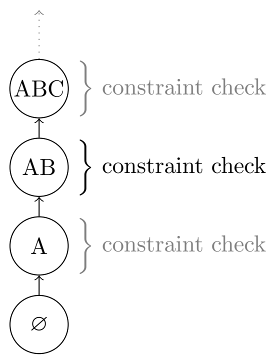
Note that this method is slightly less powerful than the joincost approach described in Section 3.8.



(a) Input graph with two terminals.



(b) Tree decomposition, with subtree of used bags.



(c) Nice tree decomposition with enforced vertex constraints.

Figure 21: Example of vertex constraints. After both A and B are introduced, the constraint checker removes the empty solution, since either A or B must be present.

5 Experiment

5.1 Setup

To gain insight in the performance gains of the proposed optimizations, time measurements and join counts are used compare performance of different implementations on a set of benchmark graphs. Not all graphs are evenly susceptible to the different optimization techniques, so the benchmark set contains different graphs of varying size, treewidth and origin. The input graphs were also used in an earlier experiments [1] by Fafianie et al, and are a combination of graphs from Steinlib [7] and TreewidthLib [8].

The graphs selected from the TreewidthLIB originate from Bayesian network and graph coloring applications. The B and the ES graphs sets from SteinLib are selected, where B contains sparse graphs, and ES consists of rectilinear graphs. The ES graphs are constructed by distributing points on a 2D plane, and preprocessing the set for the geometric rectilinear Steiner tree problem with the GeoSteiner [9] package. Input graph preprocessing such as the techniques described in [10], can reduce the number of vertices and edges substantially. This will most likely lead to less room for the discussed optimizations to be effective, since a lot of trivial constructs will no longer be present in the graphs.

In the following sections all of the earlier discussed optimization techniques are applied one by one, and comparing the algorithm performance to a reference implementation. This approach is chosen to highlight the benefits of each individual optimization. Table 15 provides a more detailed overview of the results of the complete benchmark set, including the properties of all input graphs.

The following implementations will be subject to the performance comparisons

`hash`	Hashset.
`cutrow`	Cutrow.
`adj`	Adjacency matrix.
`uf`	Union find.
`nice`	Straightforward nice tree decomposition generation.
`niceOpt`	Combination of all nice tree decomposition optimizations.
`nored`	Do not use the reduce algorithm.
`reduce`	Use the reduce algorithm.
`ljp`	Last join prediction.
`jb`	Use bounds and sorted input in joins.
`pj`	Compute join on pre-ordered input pairs, and reduce the result table simultaneously.
`vc`	Use vertex constraints.
`iss`	Use infeasible subsets.
`opt`	Combination of `niceOpt`, `ljp`, `jb`, `pj` and `iss`.

Table 14: Overview of different algorithm configurations.

Since each bag can be used as a root for the generation of the nice TD, each computation is repeated selecting each bag once as the root, and measurements are averaged.

The combination of testing multiple configurations, all bags as root and multiple iterations for reliable time measurement results in a large amount of work. The experiment is therefore limited to relatively small graphs, to keep the total amount of work feasible.

The experiments were executed on OSX 1.6, running Java 1.6 on a 2.4Ghz core with 2048MB of available memory.

5.2 Data structures

Figure 22 gives an execution time comparison of the different proposed data structures on a subset of the benchmark graphs. As expected, the non tree-index hashset implementation gets outclassed by an order of magnitude by all other implementations. It is interesting to see that cutrow implementation is generally the best performing data structure. This can be attributed to its constant time operator implementations combined with the minimum memory usage of a single computer word.

Figure 23 gives a similar comparison but for graphs with a higher `MBS`. The cutrow data structure starts performing worse than the other implementations, because of the exponentially growing memory requirement. An `MBS` of 11 is the largest size for which the adjacency implementation will still fit in a single 64-bit computer word, For higher `MBS` the union-find data structure will be a good candidate, or possibly an adjacency data structure using multiple.

5.3 Nice tree decomposition optimizations

Figure 24 gives a join count comparison between executing the algorithm on the unedited tree decomposition that is generated by the heuristic, and a version that is optimized using the proposed rules. A consistent improvement can be seen, varying from small improvements to requiring only half the amount of joins.

Five of the proposed rules are applied, but the multiple-join-order was omitted. Some improvements could be seen, but not significant enough to warrant the increased memory requirement. More importantly, the join optimization and branch relocation rules result in an overall decrease of the degree of join bags. This means that the sequential joins simply do not occur often enough to allow for significant application of the rule.

5.4 Runtime optimizations

Figure 26 shows effectiveness of the rank based approach implemented in the reduce function, compared to a reference implementation. Both implementations use the `uf` data structure and `niceOpt` on the nice tree decomposition. These results are in line with earlier conducted experimental evaluation, where this method was already shown to be very effective.

Figure 27 shows the reduction in the number of joins when either vertex constraints, infeasible subsets or last join prediction is used. All four implementations use the `uf` data structure, `niceOpt` and the `reduce` function. The vertex constraints and infeasible subsets operation are only minimally effective in some graphs, since they require the existence of small separators in the graph to be able to effectively discard sets of partial solutions.

The last join prediction shows a consistent reduction in the required amount of joins. Generally there is always a single join bag that is responsible for a significant amount of work. If this bag is the last join in the nice tree decomposition, it then can be computed in a fraction of the time it normally would require.

Figure 28 gives a performance comparison between a reference implementation, bounded joins and pairwise joins. On all graphs, both methods show a considerable improvement compared to the reference implementation, and in some instances a significant decrease in the size of the largest join. Interestingly enough, the pairwise join method requires the least amount of joins, but its more complex approach generates a significant overhead resulting in an overall slower execution. This can be attributed to generating all pairs beforehand and storing them in an ordered list. For graphs with a larger `MBS` this may still prove to be a viable option, because the generated pairs require a constant amount of memory while the join operation complexity will grow quadratically.

Figure 29 shows the performance variation in terms of computed joins, when different bags are selected as root for the nice tree decomposition. Within the tree decomposition, some branches will perform structurally worse than others, while the largest bag in the center will always be a poor candidate. It is interesting to see that the introduced optimizations not only decrease the ratio between the worst and best case significantly, the worst performing bag is no longer a join bag. The worst performing bag in the unoptimized implementation now performs very well, most likely due to the last join prediction procedure, utilizing the unique property that a last join has.

5.5 Connection based data structure

The connection based data structure does not perform well on most graphs, and often will not terminate within a reasonable amount of time. Oddly enough for a small set of graphs it will perform exceptionally well, especially without the use of the poorly implemented rank based reduce function.

5.6 Complete benchmark

Table 15 gives an overview of the complete benchmark set, and results for two different implementations. Both use `uf` and `reduce`, but the second implementation also uses `opt`, to show the significance of the proposed optimizations.

						`reduce + uf`		`reduce + uf + opt`	
set	name	$\vert V\vert$	$\vert E\vert$	$\vert T\vert$	`MBS`	time(ms)	joins	time(ms)	joins
Steinlib	b01	50	63	9	5	30	536	11	255
	b02	50	63	13	5	47	677	8	467
	b08	75	94	19	7	163	6 708	36	1 805
	b09	75	94	38	7	45	4 297	20	1 441
	b13	100	125	17	8	379	55 821	237	15 413
	b14	100	125	25	8	178	29 048	59	6 276
	b15	100	125	50	9	337	111 605	52	15 404
	i080-001	80	120	6	10	1 139	564 674	811	152 664
	i080-003	80	120	6	10	1 209	1 028 991	236	109 266
	i080-004	80	120	6	11	8 475	5 501 949	1 551	647 420
	b06	50	100	25	11	2 252	1 278 201	686	222 253
	i080-005	80	120	6	12	99 679	32 492 266	12 653	3 071 141
	b05	50	100	13	12	26 003	14 901 666	6 089	1 459 744
Steinlib	es90fst12	207	284	90	6	317	8 171	189	6 854
	es100fst10	229	312	100	6	71	10 580	143	9 203
	es80fst06	172	224	80	7	204	20 990	712	15 199
	es100fst14	198	253	100	7	283	14 526	293	10 252
	es90fst01	181	231	90	8	145	22 423	145	10 742
	es100fst13	254	361	100	8	352	80 105	471	51 418
	es100fst15	231	319	100	9	454	136 880	338	49 634
	es250fst03	543	727	250	9	594	203 645	535	76 197
	es100fst08	210	276	100	10	286	195 280	99	41 702
	es250fst05	596	832	250	10	1 123	741 162	721	425 716
	es250fst07	585	799	250	11	4 993	2 556 609	2 910	1 099 855
	es250fst12	619	872	250	12	19 697	9 318 540	5 237	2 482 310
	es100fst02	339	522	100	13	48 477	24 953 330	11 648	2 545 085
	es250fst01	623	876	250	13	35 000	11 165 018	16 000	1 076 866
	es250fst08	657	947	250	14	300 238	87 295 578	65 000	5 436 438
TreewidthLIB	myciel3	11	20	2	6	103	886	9	417
	BN_28	24	49	4	6	21	616	20	470
	pathfinder	109	211	21	7	207	8 998	49	2 377
	csf	32	94	6	7	25	10 124	37	3 656
	oow-trad	33	72	6	8	146	31 704	273	13 699
	mainuk	48	198	9	8	607	83 420	285	19 935
	ship-ship	50	114	10	9	801	240 277	418	59 455
	barley	48	126	9	9	293	99 724	33	19 061
	miles250	128	387	25	10	897	283 328	299	50 153
	jean	80	254	16	10	1 502	177 876	741	35 393
	huck	74	301	14	11	993	32 290	1 125	3 745
	myciel4	23	71	4	12	9 829	2 807 722	974	281 474
	munin1	189	366	37	12	29 626	30 156 246	3 341	1 380 245
	pigs	441	806	88	13	80 199	52 349 884	7 884	1 799 474
	anna	138	493	27	13	166 444	82 477 869	14 670	3 376 890

Table 15: Overview of all used benchmark graphs, with results for the configurations `reduce+uf` and `reduce+uf+opt`.

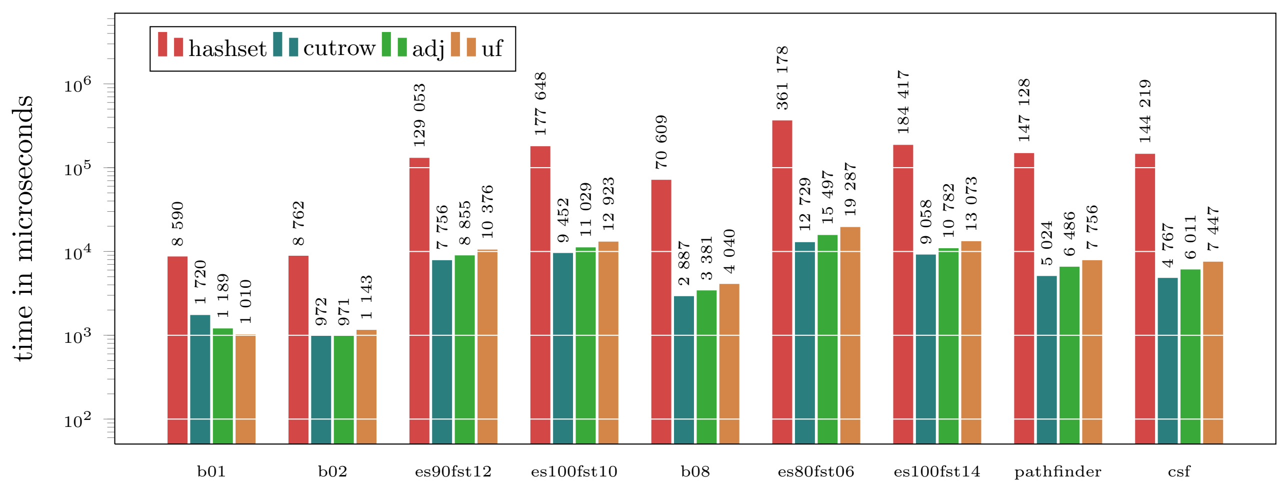


Figure 22: Performance comparison between different datastructures for graphs of `MBS` ≤ 6.

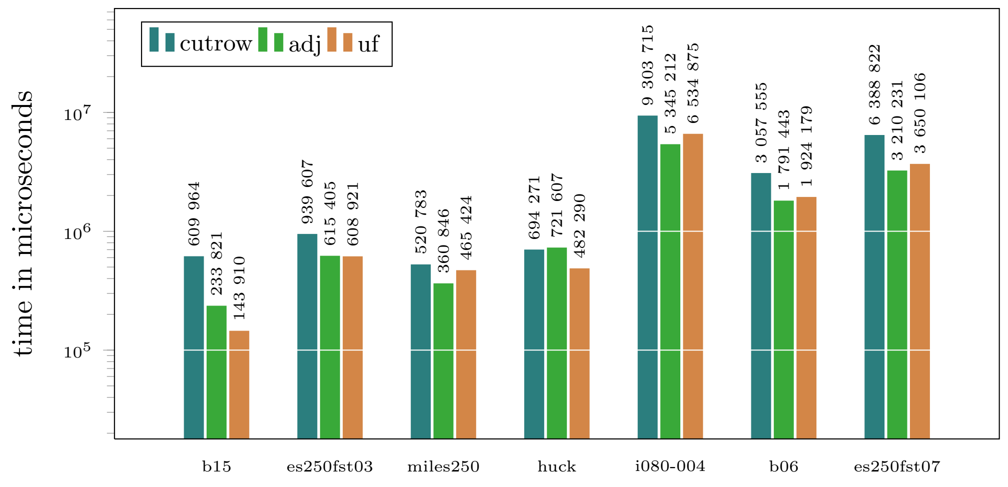


Figure 23: Performance comparison between different data structures for graphs of `MBS` ≤ 11.

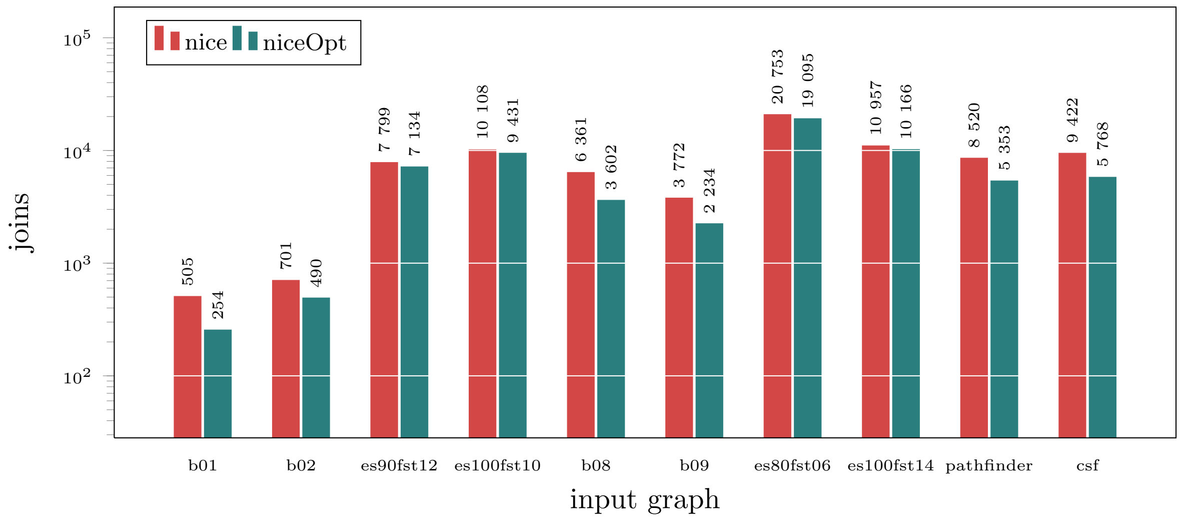


Figure 24: Performance comparison between computing on the standard nice tree decomposition versus its optimized counterpart.

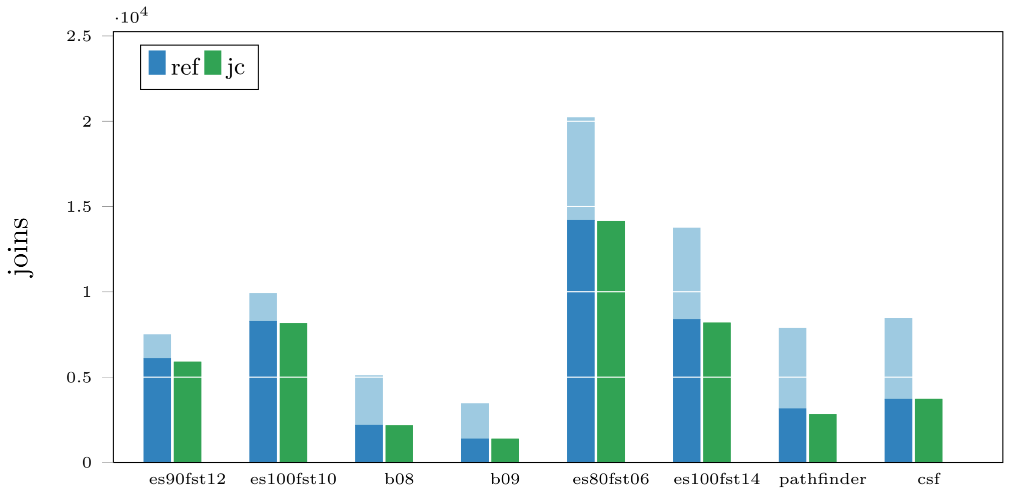
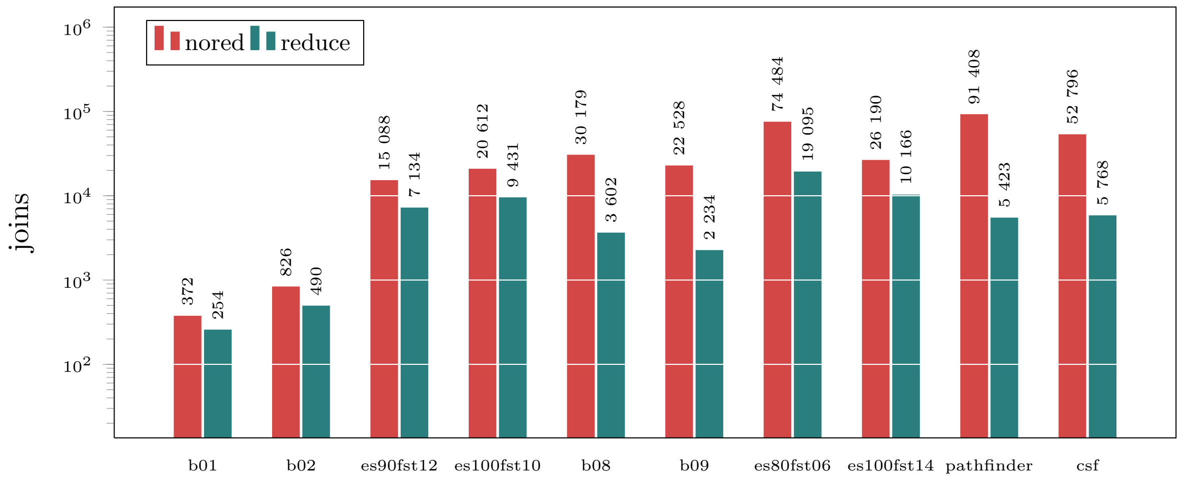
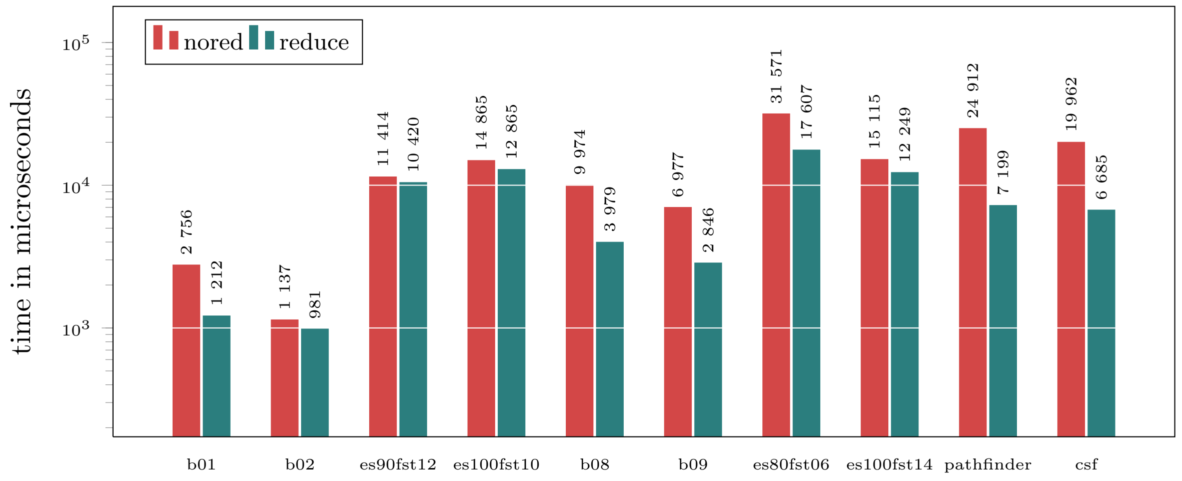


Figure 25: Performance comparison between the reference implementation and an implementation using joincost to choose the optimal root bag. The darker shade represents the best performing root bag, the lighter shade the worst performing.



(a) Join count



(b) Execution time

Figure 26: Performance comparison between the standard implementation without and with the reduce algorithm.

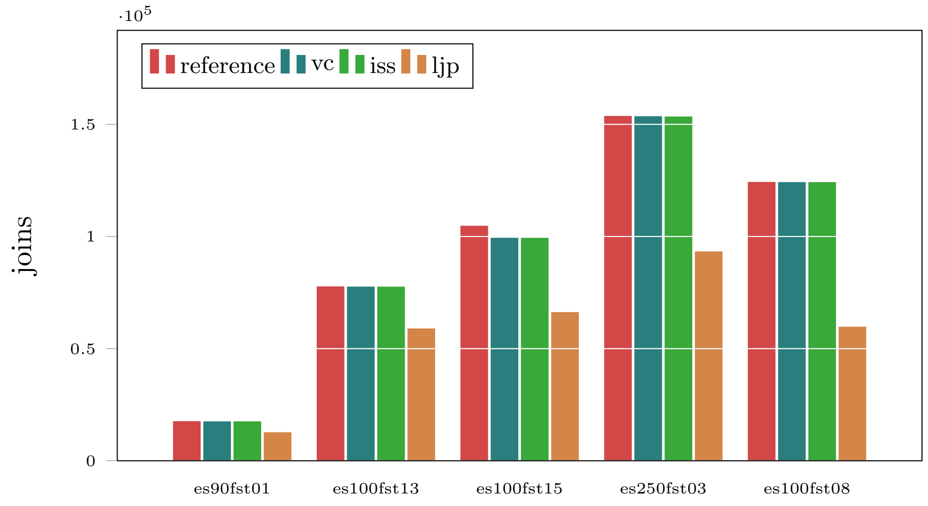


Figure 27: Performance comparison between a reference implementation, vertex constraints, infeasible subset and last join prediction.

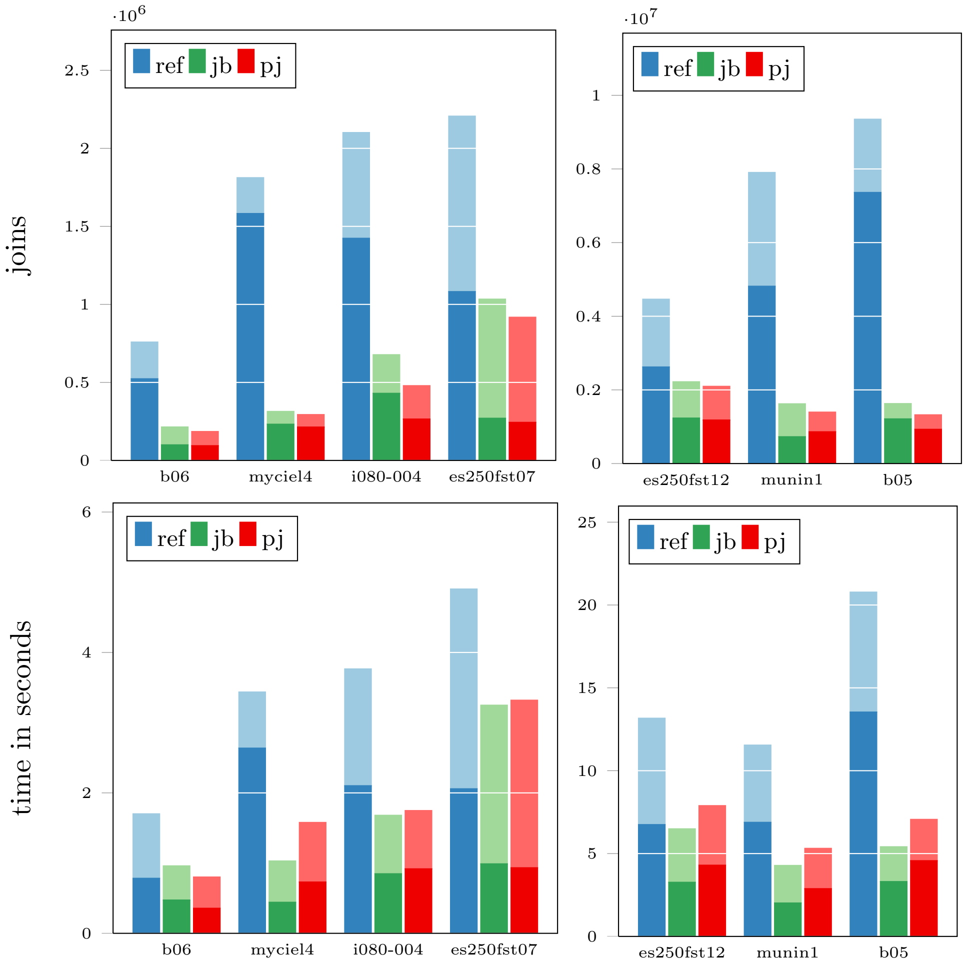
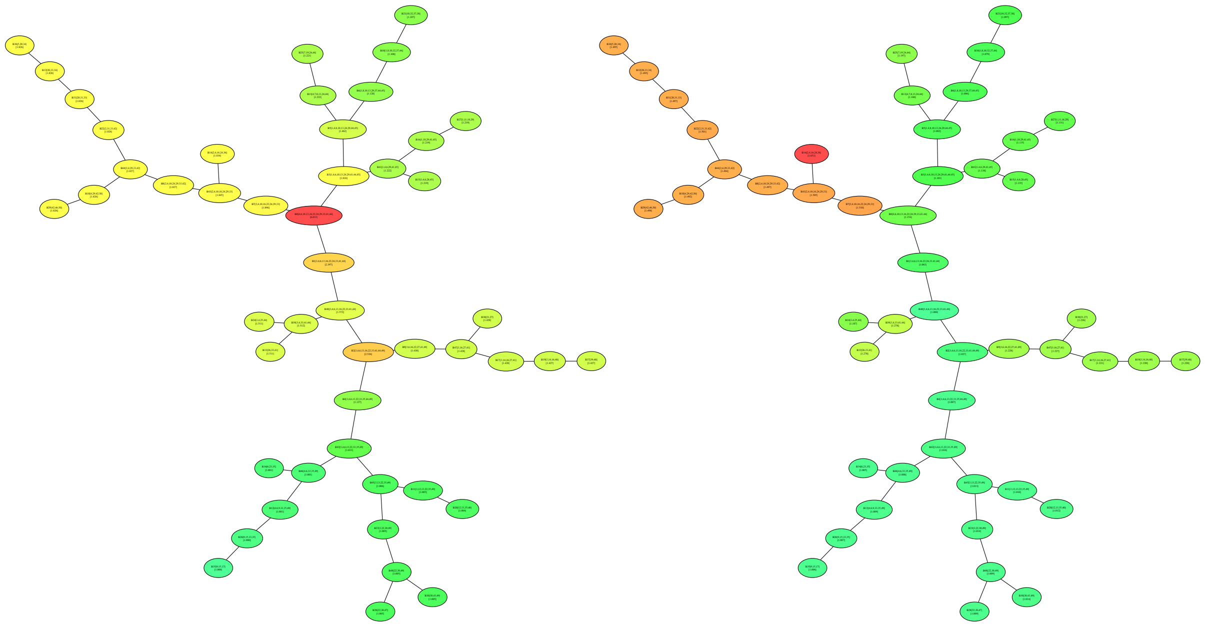


Figure 28: Performance comparison between the reference implementation, bounded joins and pairwise joins. Darker part of each bar represents joincount/time of largest join.



(a) Reference implementation using `reduce + uf`. The red bag computed 6x the optimal amount of joins, the yellow bags approximately 1.8x.

(b) Optimal implementation using `reduce + uf + opt`. The red bag computed 1.6x the optimal amount of joins, the orange bags approximately 1.5x.

Figure 29: Tree decomposition of `b06`, with bags shaded to indicate the amount of joins computed compared to the optimal amount for that implementation.

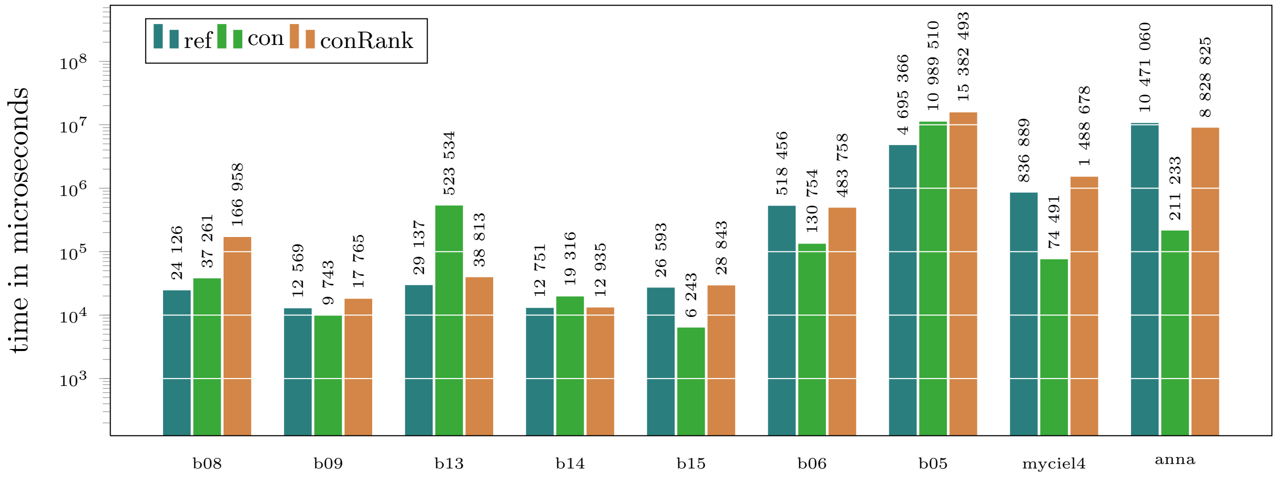


Figure 30: Performance comparison between the standard optimized implementation and the connection based implementation using linear reduce, and using both linear reduce and rank reduce.

6 Conclusion

6.1 Concluding remarks

The experiments have shown that the proposed optimizations can improve performance considerably. Preprocessing the nice tree decomposition shows to be an effective optimization strategy for these types of algorithms. The operations are computationally inexpensive and provide consistent performance improvements, especially with respect to minimizing the joins.

One of the observations that incited this work was the presence of a high variance in algorithm performance between different root bags for the nice tree decomposition. Especially the last bag outputted by the heuristic, which is always has maximum size, showed very poor performance in some input graphs. To some extent these fluctuations have been mitigated by the introduced optimizations, with especially the last join prediction doing very well for when the root bag is of large size. Computing the joincost of the nice tree decompositions shows to be a sound strategy to determine the optimal root bag, albeit with a small added cost.

Using a graph coloring in the form of tree-indices on the data structure level allows for a very concise representation of the partial solutions. This combined with the opportunity to implement the required functions very efficiently using bitwise operators, makes it a superior approach in comparison with a more traditional implementation.

Representing a resultset in terms of a lattice has been a fruitful endeavor for gaining an improved insight of the underlying structure and inner workings of the algorithm. This led to a number of optimizations that were reasonably straightforward but not directly apparent in the absence of this context, and proved to be very effective in practice.

6.2 Further research

6.2.1 Look ahead optimizations

A few of the optimizations introduced in this work can be categorized as look-ahead, in the sense that information that lies beyond the current bag is utilized to discard certain partial solutions. The proposed vertex constraints and last join prediction are good examples of this concept, as they can at times discard large amounts of partial solutions. Overall these methods seem to decrease the variance between the performance of different root bags for the nice tree decomposition. The challenge remains to remove this variance altogether, or to be able to predict the performance of each root bag so the best performing option can be selected. Other techniques such as maintaining the minimum remaining edge cost to predict too expensive refinements, might yield further improvements.

6.2.2 Low level implementations

The single computer word data structures proposed in this work allow for some very fast low level optimizations, if executed correctly. Implementing the algorithm in C++ instead of Java would allow for such optimizations, along with the overall performance benefits that the language exhibits. It would be interesting to see how such an implementation would hold up performance wise, compared to conventional the non-tree-decomposition based methods. It might also be possible to incorporate preprocessing techniques from [10] into the algorithm, allowing for more problem specific optimizations.

6.2.3 Inter resultset reduction

Most of the current runtime optimizations are directed towards reducing the size of the resultset within the scope of a set of used vertices. It might be possible to extend this scope and discard partial solutions that are represented by partial solutions from other sets. The size of the resultset is exponential in the number of used vertices, and the number of used vertices subsets of the bag is also exponential. Discarding whole resultsets alike the vertex constraints rule will generally be very beneficial for the overall performance. For instance, if the neighborhood of a vertex can be represented by the other vertices in the bag, all resultsets containing this vertex can be discarded.

6.2.4 Connection based data structure

The connection based data structure shows good results on some graphs, but often the poor performance of the reduce function results in a very slow execution. There will most likely exist an implementation with a lower complexity, but due to time constraints, only a straightforward approach using the existing techniques was used. Given the fact that this method requires at most 2^{n-1} partial solutions but generally fewer, it might be able to outperform the standard approach.

A good approach would be to start with a small incomplete representative set, and progressively add partial solutions to it. This can be done by generating a partial solution that is not completed by any partial solution in the incomplete representative set, and then find the connection based partial solution that can complete it with minimal cost. The partial solution that completes it can then be added to the representative set, and the process repeated until the representative set is complete. If a function to generate these partial solutions exists using $O(2^{\texttt{bagSize}-1})$ time, the reduce procedure will be of equivalent complexity to the rank based approach, and likely allow for overall improved algorithm performance.

The connection based data structure can also be applied to compute upper bound on the cost of a solution. All partial solutions containing $n-1$ edges are feasible. By removing all partial solutions that have cost greater than $maxCost - k$, the cost of the resulting solution will be at most nk above the optimal cost, where n is the number of bags in the nice tree decomposition.

6.2.5 Minimal joincost tree decompositions

Since the required amount of joins can be accurately computed in a reasonable amount of time, it might be interesting to analyze the construction a joincost-minimal tree decompositions for a given set of terminals. There could be some similarities with the computation of treecost-minimal tree decompositions, since the computation of the joincost also uses a function exponential in the size of the bag.

References

[1] Stefan Fafianie, Hans L. Bodlaender, and Jesper Nederlof. Speeding up dynamic programming with representative sets - an experimental evaluation of algorithms for Steiner tree on tree decompositions. In *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4-6, 2013, Revised Selected Papers*, pages 321–334, 2013.

[2] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part I*, ICALP'13, pages 196–207, Berlin, Heidelberg, 2013. Springer-Verlag.

[3] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259 – 275, 2010.

[4] T. van Dijk, J. van den Heuvel, and W. Slob. Computing treewidth with LibTW. November 2006.

[5] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 150–159, 2011.

[6] T. Kloks. Treewidth. computations and approximations. *Lecture Notes in Computer Science*, 842, 1994.

[7] A. Martin T. Koch and S. Vo. Steinlib, an updated library on Steiner tree problems in graphs. 2000.

[8] H. L. Bodlaender. TreewidthLIB. A benchmark for algorithms for Treewidth and related graph problems. 2004.

[9] P. Winter D. Warme and M. Zachariasen. GeoSteiner, software for computing Steiner trees.

[10] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[11] P. Mutzel Chimani and B. Zey. Improved Steiner tree algorithms for bounded treewidth. *Journal of Discrete Algorithms, 16:6778*, 2012.

[12] J. A. Wald and C. J. Colbourn. Steiner trees, partial 2-trees, and minimum IFI networks. *Networks, 13:159167*, 1983.

[13] Fang Wei-Kleiner. Tree decomposition based Steiner tree computation over large graphs. *CoRR*, abs/1305.5757, 2013.

[14] Hans L. Bodlaender, Paul S. Bonsma, and Daniel Lokshtanov. The fine details of fast dynamic programming over tree decompositions. In *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4-6, 2013, Revised Selected Papers*, pages 41–53, 2013.

[15] Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast hamiltonicity checking via bases of perfect matchings. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 301–310, New York, NY, USA, 2013. ACM.