Counting Spanning Trees to Guide Search in Constrained Spanning Tree Problems

Simon Brockbank, Gilles Pesant, and Louis-Martin Rousseau

[1] École polytechnique de Montréal, Montreal, Canada
[2] CIRRELT, Université de Montréal, Montreal, Canada
{simon.brockbank,gilles.pesant,louis-martin.rousseau}@polymtl.ca

Abstract. Counting-based branching heuristics such as maxSD were shown to be effective on a variety of constraint satisfaction problems. These heuristics require that we equip each family of constraints with a dedicated algorithm to compute the local solution density of variable assignments, much as what has been done with filtering algorithms to apply local inference. This paper derives an exact polytime algorithm to compute solution densities for a spanning tree constraint, starting from a known result about the number of spanning trees in a graph. We then empirically compare branching heuristics based on that result with other generic heuristics.

1 Introduction

Constraint programming is a powerful approach that can be used to solve combinatorial problems. However its success depends heavily on heuristics that can guide the search toward promising areas of the search tree. One can design a heuristic dedicated to the particular problem at hand or rely on out-of-the-box generic heuristics that have shown good performance on a variety of problems. The last decade has witnessed renewed interest in the design of robust generic branching heuristics (e.g. [9, 6]). In particular Zanarini and Pesant[15] introduced branching heuristics based on the concept of *solution density*, i.e. the proportion of solutions local to a constraint featuring a given variable-value assignment.

Definition 1 (solution density). *Given a constraint $c(x_1, \ldots, x_n)$, its number of solutions $\#c(x_1, \ldots, x_n)$, respective finite domains D_i $1 \leq i \leq n$, a variable x_i in the scope of c, and a value $d \in D_i$, we will call*

$$\sigma(x_i, d, c) = \frac{\#c(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_n)}{\#c(x_1, \ldots, x_n)}$$

the solution density *of pair (x_i, d) in c. It measures how often a certain assignment is part of a solution to c.*

Specialized algorithms have been designed to compute solution densities for several families of constraints[8]. In this paper we propose an exact polytime algorithm that computes solution densities for a spanning tree constraint.

Definition 2 (Spanning Tree Constraint (adapted from [4])). *Given an undirected graph $G(V, E)$ and set variable $T \subseteq E$, constraint* `spanningTree`(G, T) *restricts T to be a spanning tree of G.*

For the sake of conforming to the previous definition of solution density, especially important if we are to allow the combination of solution density information from different constraints, we instead represent T as an array of boolean variables.

The rest of the paper is organized as follows: Section 2 exposes the related work, Section 3 describes our algorithm to compute solution densities for the spanning tree constraint, Section 4 discusses how our data structures are updated in the course of backtrack search, and Section 5 provides supporting empirical evidence of branching based on solution density.

2 Related work

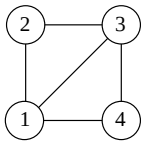
Research in the CP community about imposed tree structures has focused so far on filtering algorithms and not on branching heuristics. Beldiceanu et al.[2] introduced the tree constraint, which addresses the digraph partitioning problem from a constraint programming perspective. In their work a constraint that enforces a set of vertex-disjoint anti-arborescences is proposed. They achieve domain consistency in $\mathcal{O}(nm)$ time, where n is the number of vertices and m is the number of edges in the graph. Their pruning relies on the identification of strong articulation points in the graph and of roots and sinks (to evaluate the minimum and maximum number of trees required to partition the graph).

Dooms and Katriel[3] introduced the MST constraint, requiring the tree variable to represent a minimum spanning tree of the graph on which the constraint is defined. Many variants of the minimum spanning tree problem, such as minimum k-spanning tree and Steiner tree are known to be NP-hard, even though its basic version can be solved in polynomial time. Those problems can be modeled by combining the MST constraint and other constraints. The authors proposed polytime bound consistent filtering algorithms for several restrictions of this constraint. They proceed by classifying edges in three sets: mandatory, possible, and forbidden. Afterwards Dooms and Katriel[4] proposed a weighted spanning tree constraint, in which both the tree and the weight of the edges are variables, and considered several filtering algorithms. In their work a set variable is used, indicating which edges are tree edges.

The filtering proposed by Dooms and Katriel[4] was then simplified and improved by Régin[10], who proposed an incremental filtering algorithm by maintaining a connected component tree which represents disjoint trees merging operations in Kruskal's algorithm, and by computing lowest common ancestors on that tree. Domain consistency was thus achieved in $\mathcal{O}(m + n \log n)$ time. Subsequently, Régin et al.[11] improved the time complexity of that filtering and also considered mandatory edges.

3 Computing Solution Densities

The *Laplacian matrix* $L(G)$ of a graph G is formed by subtracting the adjacency matrix of G from the diagonal matrix whose i^{th} entry is equal to the degree of vertex i in G. Henceforth for notational convenience we will refer to it simply as L. For example Figure 1 shows a graph and its Laplacian matrix.



$$L = \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

Fig. 1. The kite graph and its Laplacian matrix.

The (i, j)-*minor* of a square matrix M, denoted M_{ij}, is the determinant of the sub-matrix obtained by removing from M its i^{th} row and j^{th} column. The Laplacian matrix has the interesting property that its (i, j)-minor, for any row i and column j, is equal to the number of spanning trees of the corresponding graph.

Theorem 1 (Kirchhoff's Matrix-Tree Theorem [13]). *Denote by $\tau(G)$ the number of spanning trees of graph G on n vertices. For any $1 \leq i, j \leq n$,*

$$\tau(G) = L_{ij}.$$

So the number of solutions to a `spanningTree` constraint can be computed as the determinant of a $(n-1) \times (n-1)$ matrix, in $\mathcal{O}(n^3)$ time.

If we remove the first row and column of the Laplacian matrix at Figure 1, the resulting minor is $2 \times (3 \times 2 - (-1) \times (-1)) - (-1) \times (-1 \times 2 - (-1) \times 0) = 8$ and one can easily verify that there are eight possible spanning trees for that graph.

But we are interested in computing the solution density of an edge $(i, j) \in E$. One way to approach this is by counting the number of spanning trees not using that edge, $\tau(G \setminus \{(i, j)\})$, and then dividing that by the total number of spanning trees, yielding the solution density of the corresponding variable being assigned value 0 (i.e. $(i, j) \notin T$):

$$\sigma((i, j), 0, \texttt{spanningTree}(G, T)) = \frac{\tau(G \setminus \{(i, j)\})}{\tau(G)}.$$

Let $L' = L(G \setminus \{(i, j)\})$. How different is L' from L? It will be identical except for entries ℓ_{ii}, ℓ_{jj}, ℓ_{ij}, and ℓ_{ji}. Since we can choose any row and column of L to compute our minor, consider removing row and column i. Then the only difference is $\ell'_{jj} = \ell_{jj} - 1$. The *Sherman-Morrison formula* [12] tells us that if

M' is obtained from matrix M by replacing its j^{th} column, $(M)_j$, by column vector u then

$$\det(M') = (1 + e_j^\top M^{-1}(u - (M)_j))\det(M).$$

In our case $(u - (M)_j) = -e_j$ so the right-hand side of the previous equation simplifies to $(1 - e_j^\top M^{-1} e_j)\det(M) = (1 - m_{jj}^{-1})\det(M)$. So finally we have

$$\sigma((i,j), 0, \texttt{spanningTree}(G,T)) = \frac{L'_{ii}}{L_{ii}} = \frac{(1 - m_{jj}^{-1})L_{ii}}{L_{ii}} = 1 - m_{jj}^{-1},$$

and of course

$$\sigma((i,j), 1, \texttt{spanningTree}(G,T)) = m_{jj}^{-1}.$$

Computing solution densities turns out to be quite simple: for each edge (i,j) incident with vertex i such that $j < i$ (respectively $j > i$), the corresponding value is the j^{th} (respectively $(j-1)^{th}$) entry on the diagonal of the inverse of M, the sub-matrix of Laplacian matrix L obtained by removing its i^{th} row and column. Repeating this from every vertex of a vertex cover of G provides solution densities for every edge. If γ is the size of the vertex cover used then the whole procedure takes $\mathcal{O}(\gamma n^3)$ time.

Example 1. Let M be the sub-matrix of L obtained by removing its first row and column as before. Then

$$M^{-1} = \begin{pmatrix} 5/8 & 2/8 & 1/8 \\ 2/8 & 4/8 & 2/8 \\ 1/8 & 2/8 & 5/8 \end{pmatrix}$$

and the solution density of edges $(1,2)$, $(1,3)$, and $(1,4)$ being used in T is respectively $\frac{5}{8}$, $\frac{4}{8}$, and $\frac{5}{8}$.

4 Integration into Backtrack Search

In this section we describe some of the implementation details and issues. As branching decisions are made and domain filtering is applied, some edges of G will be required in T and others, forbidden. These changes must be reflected in our data structures. Our data structures are reversible so that they are restored upon backtracking.

We use a heuristic greedy algorithm to compute our initial vertex cover — it may be worthwhile spending the time to compute a minimum vertex cover but that cover will need to be revised as vertices are merged following edge contractions. A simple way to update a vertex cover containing vertex j when edge (i,j) is contracted is to replace it with vertex i.

4.1 Updating the Laplacian Matrix

If edge (i, j) is forbidden it is simply removed from the graph. To reflect that in the Laplacian matrix we simply add one to entries ℓ_{ij} and ℓ_{ji}. The degree of each endpoint is also updated by subtracting one to ℓ_{ii} and ℓ_{jj}.

If edge (i, j) is required we contract it in the graph, so that (i, j) is implicitly part of the spanning tree. To update the Laplacian matrix we start by adding to vertex i all the edges (j, k): $\ell_{ik} \leftarrow \ell_{ik} + \ell_{jk}$. This may create multiple edges. The degree of vertex i, ℓ_{ii}, is also updated accordingly. Then, since vertex j is now merged with i, we remove all the edges connected to it, by setting to zero row and column j of the Laplacian matrix. Finally we set ℓ_{jj} to 1 so that minors will be computed correctly when they include row and column j.

Example 2. Recall Figure 1 and suppose edge $(1, 2)$ is now required for the spanning tree: we contract it and merge vertex 2 with 1. The new Laplacian matrix will be (note the double edge $(1, 3)$):

$$L = \begin{pmatrix} 3 & 0 & -2 & -1 \\ 0 & 1 & 0 & 0 \\ -2 & 0 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

4.2 Updating Solution Densities

The solution densities will change and we would like to avoid recomputing them from scratch. Given the inverse of matrix M can we incrementally compute the inverse of a slightly different matrix M'? The *Sherman-Morrison formula* further reveals that if M' is obtained from M by replacing its i^{th} column, $(M)_i$, by column vector u as before then

$$M'^{-1} = M^{-1} - \frac{(M^{-1}(u - (M)_i))(e_i^\top M^{-1})}{1 + e_i^\top M^{-1}(u - (M)_i)}.$$

This can be computed in $\mathcal{O}(n^2)$ time.

In some cases we can lower that time complexity considerably. Consider forbidden edge (i, j). For any edge (i, k) whose solution density was obtained through the inverse of a sub-matrix removing row and column i from L, removing edge (i, j) only changes one entry in that sub-matrix, as we saw before, and the previous formula simplifies to

$$M'^{-1} = M^{-1} - \frac{(M^{-1} \cdot (-e_j)) \cdot (e_j^\top \cdot M^{-1})}{1 - m_{jj}^{-1}} = M^{-1} + \frac{1}{1 - m_{jj}^{-1}} \cdot Q$$

where $Q = (q_{hk})$ is an $(n-1) \times (n-1)$ matrix with $q_{hk} = m_{hj}^{-1} \cdot m_{jk}^{-1}$. Because we only need the k^{th} entry on the diagonal, $m_{kk}^{-1} + (m_{kj}^{-1})^2/(1 - m_{jj}^{-1})$, the update for that edge takes constant time. What preceded equally applies for any edge (j, k) with a sub-matrix removing row and column j from L.

Example 3. Recall that for the graph at Figure 1 the solution density of edge $(1, 4)$ is $\frac{5}{8}$. Suppose edge $(1, 2)$ is now forbidden in the spanning tree. The updated solution density will be $\frac{5}{8} + (m_{42}^{-1})^2/(1 - m_{22}^{-1}) = \frac{5}{8} + (\frac{1}{8})^2/(1 - \frac{5}{8}) = \frac{2}{3}$.

5 Experiments

To demonstrate the effectiveness of using solution density information from a `spanningTree` constraint to guide a branching heuristic on some constrained spanning tree problems, we consider finding degree-constrained spanning trees of a graph. Note that the special case of a maximum degree of 2 corresponds to the Hamiltonian path problem. We created some graphs using a generator designed to produce hard Hamiltonian path instances for backtracking algorithms [14]. We used the IBM ILOG CP v1.6 solver for our implementation and performed our experiments on a AMD Opteron 2.2GHz with 1GB of memory. Our current implementation does not include the incremental algorithm described in Section 4.2 so the times reported are with matrix inversions computed from scratch at every search tree node. We report comparative results between maxSD, impact-based search (IBS), and random variable and value selection (random). Heuristic maxSD considers solution density information from each constraint and branches on the variable-value pair corresponding to the highest solution density observed. For IBS impacts are initialized by probing at the root node. At a search tree node the five best variables according to the approximated impact are identified. For that subset, we compute node impacts and branch on the best variable (highest impact) and value (lowest impact). This is consistent with what is suggested in the IBM ILOG solver documentation. For random we report the average of ten runs.

We used simple filtering rules for our constraint — our objective is not to solve that problem in the best way possible but rather to evaluate a counting-based branching heuristic. The first one forces each vertex to have degree at least one in the tree by lower bounding the sum of the variables corresponding to the edges incident to it. The second one fixes the number of edges that can be part of the spanning tree: as a spanning tree is formed by $n - 1$ edges, the sum of all variables must equal that value. Finally, since a tree is acyclic, we maintain the connected component in which each vertex lies, removing any extraneous intra connected component edges. In addition to the `spanningTree` constraint, we add to our model for each vertex i an upper bound on the sum of the variables corresponding to the edges incident to i.

We first generated random graphs of 15, 20, 25, 30, and 35 vertices (10 instances each). The generator ensures the existence of a Hamiltonian path. Turning first to a degree-2 bound, Table 1 left indicates that using maxSD effectively guides the search to a solution in several orders of magnitude fewer backtracks than the other two branching heuristics. Even though maxSD appears slower on small graphs, as displayed in Table 1 right, as the graphs become larger, this approach becomes faster than IBS and random.

Table 1. Number of backtracks (left) and time in seconds (right) before finding a spanning tree of maximum degree 2. Each line represents an average over 10 instances.

n	maxSD	IBS	random
15	0.2	229.8	49.0
20	1.5	533.0	976.6
25	2.1	1772.3	5919.6
30	71.7	12517.1	91454.4
35	112.2	18405.4	139861.3

n	maxSD	IBS	random
15	0.029	0.001	0.001
20	0.080	0.012	0.020
25	0.187	0.085	0.173
30	0.815	0.897	1.873
35	1.769	4.742	14.646

Table 2. Number of backtracks (left) and time in seconds (right) before finding a spanning tree of maximum degree 3. Each line represents an average over 10 instances.

n	maxSD	IBS	random
15	0.0	225.6	1.3
20	0.0	315.2	53.2
25	0.0	446.7	882.0
30	0.0	495.1	18589.8
35	0.0	566.8	20001.4

n	maxSD	IBS	random
15	0.039	0.002	0.001
20	0.100	0.013	0.001
25	0.222	0.021	0.311
30	0.441	0.039	0.093
35	0.852	0.063	2.333

We then turn to a degree-3 bound (see Table 2). It clearly demonstrates that using solution densities to find spanning trees in random graphs is a very effective approach. A maximum degree of 3 is much less restrictive than a maximum degree of 2 and more spanning trees in that graph will have that property. Therefore the first few spanning trees found satisfy all constraints. For all graphs, the solution density branching heuristic finds a spanning tree without any backtrack, unlike the other approaches. Despite not having to backtrack, maxSD remains slower than IBS on these instances since the latter only requires a few hundred backtracks.

Table 3. Number of backtracks (left) and time in seconds (right) before finding a Hamiltonian path in crossroad graphs. Each line represents an average over 10 instances.

n	maxSD	IBS	random
3	0.2	7721.9	8530.5
4	0.1	262011.7	191195.8
5	0.4	162353.0	-

n	maxSD	IBS	random
3	0.085	0.255	0.062
4	0.280	26.379	3.674
5	0.676	586.679	-

We also generated *crossroad graphs* using the same graph generator. These graphs are made up of small subgraphs only connected to each other via "bridge"

edges. We generated 10 instances each of crossroad graphs containing 3, 4, and 5 subgraphs (with up to 35 vertices in total) and then tried to find a Hamiltonian path (spanning tree of degree 2). Results are shown in Table 3.

Using maxSD on these hard graphs is very effective, always finding a solution in much fewer backtracks than the other approaches. For the instances made up of 5 subgraphs, random could not solve a single instance within 2 hours of computing time. Here maxSD is also orders of magnitude faster than the other two branching heuristics.

6 Conclusion

We presented a new algorithm that computes exact solution densities for the spanning tree constraint in $\mathcal{O}(\gamma n^3)$ time, where γ is the size of a vertex cover for the graph, and updates solution densities in $\mathcal{O}(\gamma n^2)$ time, even in some cases achieving constant time updates per edge. Building the Laplacian matrix of a graph and inverting selected sub-matrices, the proportion of spanning trees including a certain edge of the graph can be calculated. By relying on that information, search can be oriented towards areas of the search space with high solution density with respect to the spanning tree structure and we gave some empirical evidence that this helps solve constrained spanning tree problems.

As future work we would like to try other types of constrained spanning tree problems. There are several application areas that involve finding spanning trees, such as network design, telecommunication, or transportation. Examples of these problems are the degree-constrained problem [7], the hop-constrained problem [5] or the diameter-constrained minimum spanning tree [1]. We also plan to investigate the compatibility of our solution density algorithm with more powerful filtering algorithms and variants of the constraint as proposed in the literature. For example the Matrix-Tree Theorem to count the number of spanning trees has already been generalized to directed graphs.

References

1. Ayman Abdalla and Narsingh Deo. Random-tree diameter and the diameter-constrained mst. *Int. J. Comput. Math.*, 79(6):651–663, 2002.
2. Nicolas Beldiceanu, Pierre Flener, and Xavier Lorca. The tree constraint. In Roman Barták and Michela Milano, editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2005.
3. Grégoire Dooms and Irit Katriel. The *minimum spanning tree* constraint. In Frédéric Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2006.
4. Grégoire Dooms and Irit Katriel. The "not-too-heavy spanning tree" constraint. In Pascal Van Hentenryck and Laurence A. Wolsey, editors, *CPAIOR*, volume 4510 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 2007.
5. Luis Gouveia, Luidi Simonetti, and Eduardo Uchoa. Modeling hop-constrained and diameter-constrained minimum spanning tree problems as steiner tree problems over layered graphs. *Math. Program.*, 128(1-2):123–148, 2011.

6. Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, editors, *CPAIOR*, volume 7298 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2012.
7. Subhash C. Narula and Cesar A. Ho. Degree-constrained minimum spanning tree. *Computers & OR*, 7(4):239–249, 1980.
8. Gilles Pesant, Claude-Guy Quimper, and Alessandro Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *J. Artif. Intell. Res. (JAIR)*, 43:173–210, 2012.
9. Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP-04*, volume LNCS 3258, pages 557–571. Springer, 2004.
10. Jean-Charles Régin. Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In Laurent Perron and Michael A. Trick, editors, *CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2008.
11. Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem Jan van Hoeve. The weighted spanning tree constraint revisited. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *CPAIOR*, volume 6140 of *Lecture Notes in Computer Science*, pages 287–291. Springer, 2010.
12. Jack Sherman and Winifred J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Annals of Mathematical Statistics*, 21:124–127, 1950.
13. W.T. Tutte. *Graph Theory*, volume 21 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, 2001. 333 pages.
14. Basil Vandegriend. Finding hamiltonian cycles : Algorithms, graphs and performance. Master's thesis, University of Alberta, 1998. http://webdocs.cs.ualberta.ca/∼joe/Theses/HCarchive/main.html.
15. Alessandro Zanarini and Gilles Pesant. Solution counting algorithms for constraint-centered search heuristics. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 743–757. Springer, 2007.