A Kruskal-Based Heuristic for the Rooted Delay-Constrained Minimum Spanning Tree Problem

Mario Ruthmair and Günther R. Raidl

Institute of Computer Graphics and Algorithms
Vienna University of Technology, Vienna, Austria
{ruthmair|raidl}@ads.tuwien.ac.at
http://www.ads.tuwien.ac.at

Abstract. The rooted delay-constrained minimum spanning tree problem is an NP-hard combinatorial optimization problem arising for example in the design of centralized broadcasting networks where quality of service constraints are of concern. We present a construction heuristic based on Kruskal's algorithm for finding a minimum cost spanning tree which eliminates some drawbacks of existing heuristic methods. To improve the solution we introduce a greedy randomized adaptive search procedure (GRASP) and a variable neighborhood descent (VND) using two different neighborhood structures. Experimental results indicate that our approach produces solutions of better quality in shorter runtime when having strict delay-bounds compared to an existing centralized construction method based on Prim's algorithm. Especially when testing on Euclidian instances our Kruskal-based heuristic outperforms the Prim-based approach in all scenarios. Moreover our construction heuristic seems to be a better starting point for subsequent improvement methods.

1 Introduction

When designing a network with a single central server broadcasting information to all the participants of the network some applications, e.g. video conferences, require a limitation of the maximal delay from the server to each client. Beside this delay-constraint minimizing the total cost of establishing the network is in most cases an important design criterium. In another example we consider a package shipment organization with a central depot guaranteeing its customers a delivery within a specified time horizon. Naturally the organization wants to minimize the transportation costs but at the same time has to hold its promise of being in time.

These network design problems can be modeled using a combinatorial optimization problem called *rooted delay-constrained minimum spanning tree (RDCMST) problem*. The objective is to find a minimum cost spanning tree of a given graph with the additional constraint that the sum of delays along the paths from a specified root node to any other node must not exceed a given delay-bound.

More formally, we are given a graph $G = (V, E)$ with a set of n nodes V, a set of m edges E, a cost function $c : E \to \mathbb{R}^+$, a delay function $d : E \to \mathbb{R}^+$, a fixed root node $s \in V$ and a delay-bound $B > 0$. An optimal solution to the RDCMST problem is a spanning tree $T = (V, E'),\ E' \subseteq E$, with minimum cost $c(T) = \sum_{e \in E'} c(e)$, satisfying the constraints: $\sum_{e \in P(s,v)} d(e) \leq B,\ \forall v \in V$. $P(s, v)$ denotes the unique path from the specified root node s to a node $v \in V$.

The RDCMST problem is $\mathcal{NP}$-hard because a special case called *hop-constrained minimum spanning tree problem*, where $d(e) = 1,\ \forall e \in E$, is shown to be $\mathcal{NP}$-hard in [1], so all more general variants of this problem are $\mathcal{NP}$-hard too.

2 Previous Work

Exact approaches to the RDCMST problem have been examined by Gouveia et al. in [2], but these methods can only solve small graphs with significantly less than 100 nodes to proven optimality in reasonable time if considering complete instances.

A heuristic approach was presented by Salama et al. in [3], where a construction method based on Prim's algorithm to find a minimum spanning tree [4] is described. This Prim-based heuristic starts from the root node and iteratively connects the node which can be reached in the cheapest way without violating the delay-constraint. If at some point no node can be connected anymore, the delays in the existing tree are reduced by replacing edges. These steps are repeated until a feasible RDCMST is obtained. A second phase improves the solution by local search using the edge-exchange neighborhood structure.

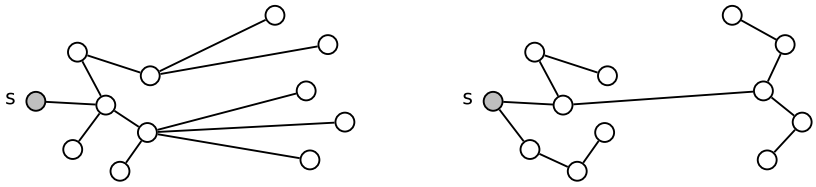
There are many recent publications dedicated to the *rooted delay-constrained minimum Steiner tree problem* which is a generalization of the RDCMST problem. In this variant only a subset of the nodes has to be reached within the given delay-bound, the other nodes can optionally be used as intermediate (Steiner) nodes. Several metaheuristics have been applied to this variant, e.g. a tabu-search in [5], a GRASP in [6] and a path-relinking approach in [7].

3 Kruskal-Based Construction Heuristic

A general problem of the Prim-based heuristic especially on Euclidian instances is the fact that the nodes in the close surrounding of the root node are connected rather cheaply, but at the same time delay is wasted, and so the distant nodes can later only be linked by many, often expensive edges, see Fig. 1. The stricter the delay-bound the more this drawback will affect the costs negatively. This fact led us to a more de-centralized approach by applying the idea of Kruskal's minimum spanning tree algorithm [8] to the RDCMST problem.

3.1 Stage 1: Merging components

In the beginning of stage one of the construction heuristic all edges are sorted by ascending costs and then iteratively added to the solution preventing cycles



(a) Prim-based (b) Kruskal-based

Fig. 1. Prim-based heuristic (a) compared to Kruskal-based heuristic (b).

until a feasible spanning tree is formed. In other words, components initially consisting of single nodes are merged by adding edges to result in one connected tree. The challenge is to maintain the feasibility of the partial solutions, i.e. to satisfy the delay-constraint to the root node throughout the whole merging process. In the Prim-based approach in [3] checking the feasibility of adding an edge to the existing tree naturally runs in constant time whereas our decentralized algorithm needs more effort to achieve this. We have to store and update additional information for each node $v \in V$:

- the path-delay $d_s(v) := \sum_{e \in P(s,v)} d(e)$ from the root node to node v
- the maximum delay $d_{max}(v)$ to any other node in the same component
- the predecessor $pred(v)$ on the path $P(s, v)$, initialized with node v

To initialize $d_s(v)$ Dijkstra's algorithm [9] calculates the path $P(s, v),\ \forall\ v \in V$ with the shortest path-delay. The paths themselves are not added to the solution, we just keep them in mind to always have a possible feasible connection to the root node available. This fallback paths are essential for stage two of the heuristic (see Section 3.2).

At this time we are able to decide if a solution exists or not because if the shortest-delay-path exceeds the specified delay-bound for any node then we cannot build a feasible tree and therefore stop here.

Initially we have a set of components $C = \{C_1, ..., C_k\},\ k = n$. Everytime we add an edge to the solution two components are merged and thereby k is decreased by 1 until set C only contains one component. For each component C_i we specify one node v_{C_i} which is nearest to the root node – it can be seen as the local root node of the subtree C_i. As mentioned above the path $P(s, v_{C_i}),\ v_{C_i} \neq s$, is not part of the tree, we just use it for testing the feasibility of a partial solution.

Now we start iterating over the sorted edge-list. Let $e = (u, v) \in E$ be the next edge on the list and $C_u \ni u,\ C_v \ni v$ be the components incident to e. The decision of adding e to the tree and thereby merging the two components C_u and C_v is based upon fulfilling at least one of the following two conditions:

1. $d_s(u) + d(e) + d_{max}(v) \leq B$
2. $d_s(v) + d(e) + d_{max}(u) \leq B$

So if it is allowed to add edge e to the solution the node information of all nodes in the newely created component C_{uv} has to be updated. First of all we have

to specify the new $v_{C_{uv}}$. There are many possibilities of choosing this node with the only constraint that $d_s(v_{C_{uv}})$ plus the delay of path $P(v_{C_{uv}}, w)$ has to satisfy the delay-bound for all $w \in C_{uv}$. A very simple and fast method turned out to be the most successful one: if only condition 1 is met then $v_{C_{uv}} = v_{C_u}$, when condition 2 holds, we choose v_{C_v}, and if both conditions are satisfied we prefer the v_{C_i} where the corresponding inequality has a larger gap to the delay-bound.

Beginning from this chosen local root node for component C_{uv} we perform a depth-first search to modify $pred(w)$ and $d_s(w)$, $\forall\ w \in C_{uv}$ using $d_s(v_{C_{uv}})$ as the starting delay. The maximal extents $d_{max}(w)$ can be determined in linear time profiting from the tree structure of the component.

The iterations stop if the solution only consists of one component, which means that it is already feasible, or there are more than one components but no more edges left in the list. The latter case is handled in stage two.

To conclude, stage one consists of sorting all edges of the graph in $\mathcal{O}(m \log m)$ time, testing each one for feasibility in constant time and updating the node information in $\mathcal{O}(n)$ time if an edge is added which can happen at most $n-1$ times due to the properties of a tree. So the total runtime is in $\mathcal{O}(m \log m + n^2)$.

3.2 Stage 2: Extension to a feasible solution

At the end of stage one the graph needs not to be connected, so in stage two the remaining subtrees are attached to the component which contains the root node by adding the shortest-delay-path $P(s, v_{C_i})$, $\forall\ C_i \in C$. At least one of the edges of a path $P(s, v_{C_i})$ creates a cycle when adding it to the solution, otherwise all edges of $P(s, v_{C_i})$ would have been included in stage one. So the main task in this stage is to dissolve resulting cycles to form a tree without violating the delay-constraint.

Paths are added by backtracking the shortest-delay-path starting from node v_{C_i} until a node u with minimal delay $d_s(u)$ is reached. We can be sure that path $P(s, u)$ is already the shortest-delay-path and do not have to go further – in the worst case however we end up at the root node. Now we add the missing edges along path $P(u, v_{C_i})$ until we are back at v_{C_i}. Cycles can occur if edge $e = (v, w)$ is added and $pred(w) \neq w \neq v$, indicating that two different paths $P(s, w)$ exist in the tree. Removing edge $(pred(w), w)$ dissolves this cycle and at the same time maintains feasibility because the delay d_s of any node in component C_w can only get smaller or stay equal since $d_s(w)$ now is the smallest possible delay and all other nodes depend on that. In $C_{pred(w)}$ no delays are affected by the removal of edge $(pred(w), w)$ since all nodes are connected to the root node through path $P(s, v_{C_{pred(w)}})$.

Since the dissolving of cycles can be done in constant time and each node is examined at most once, stage two runs in $\mathcal{O}(n)$.

3.3 Modifications

Two modifications in stage one usually lead to better results when applying a subsequent improvement method (see Section 5):

1. A delay-factor $df \geq 1$ is introduced and multiplied with the left side of the inequalities when checking the feasibility of adding an edge. In other words, the delay-bound is lowered by the factor $\frac{1}{df}$.
2. If stage one has added a predefined number of edges $< (n-1)$ it is aborted and stage two uses shortest-delay-paths to attach the left components.

Both modifications provide a solution where the gap between the node-delays $d_s(v)$ and the delay-bound is larger than in the spanning tree of the standard implementation. This higher "residual delay" leads to more possibilites in a following improvement phase and therefore often results in solutions with less total cost.

4 GRASP

To provide many different feasible starting solutions for a subsequent improvement phase we extended stage one of the Kruskal-based construction heuristic with a *greedy randomized adaptive search procedure* (GRASP) [10]. In each iteration of stage one do:

1. store all feasible edges in a candidate list (CL)
2. select a subset of least-cost edges of CL with
$$c(e) \leq \min_{e \in CL} c(e) + \alpha \cdot (\max_{e \in CL} c(e) - \min_{e \in CL} c(e))$$
for a predefined parameter $\alpha \in [0, 1]$ and insert them into a restricted candidate list (RCL)
3. randomly choose an edge from the RCL
4. merge components by adding this edge

5 Variable Neighborhood Descent

We introduce a *variable neighborhood descent* (VND) [11] for improving a constructed solution by performing a local search switching between two neighborhood structures: *Edge-Replace* (ER) and *Component-Renew* (CR). The standard implementation of a VND as it is described in [11] was modified to provide here better results in a shorter runtime: A neighborhood structure is searched by next-improvement until a local optimum is reached; then we switch to the other one continuing until no better solution can be found anymore.

A move in the Edge-Replace neighborhood removes the most expensive edge and connects the resulting two components in the cheapest possible way. A complete neighborhood search is done in $\mathcal{O}(nm)$ time.

A Component-Renew move also deletes the most expensive edge, but completely dissolves the component which is now separated from the root node; it then re-adds the individual nodes by applying a Prim-based algorithm. As before in some cases not all single nodes can be added due to the delay-bound. These remaining nodes are again joined to the root component by shortest-delay-paths, dissolving created cycles. A complete neighborhood search is done in $\mathcal{O}(n^3)$ time.

Table 1. Comparison of Prim- and Kruskal-based heuristics, applied on random instance sets with 500 and 1000 nodes (B: delay-bound, C: only construction, CV: construction and VND, CGV: construction with GRASP and VND, $\bar{c}$: average final objective values, σ: standard deviations, $t[s]$: running times in seconds).

		R500						**R1000**					
		Prim-based			Kruskal-based			Prim-based			Kruskal-based		
B	Test	$\bar{c}$	σ	$t[s]$	$\bar{c}$	σ	$t[s]$	$\bar{c}$	σ	$t[s]$	$\bar{c}$	σ	$t[s]$
6	C	19651	1583	0.1	**10785**	643	0.0	24053	3065	0.5	**14717**	710	0.0
	CV	9624	624	0.8	**9177**	633	0.5	11691	845	4.0	**10123**	544	3.0
	CGV	9340	578	12.2	**9067**	643	9.2	10858	558	64.6	**9942**	505	57.5
8	C	13020	1709	0.0	**8285**	428	0.0	15291	1826	0.0	**11779**	575	0.0
	CV	6795	546	0.8	**6035**	292	0.5	9433	1163	4.2	**6796**	322	3.2
	CGV	6352	368	13.8	**5871**	293	12.8	7719	471	68.8	**6610**	284	60.3
10	C	9555	1666	0.0	**7071**	328	0.0	11275	2051	0.0	**10277**	500	0.0
	CV	5914	686	0.8	**4554**	210	0.8	7299	747	4.3	**5172**	219	3.3
	CGV	4975	274	14.7	**4421**	200	13.5	5715	408	72.7	**5040**	202	70.3
15	C	5793	1037	0.0	**5565**	401	0.0	**6945**	1113	0.1	7996	533	0.0
	CV	3941	432	1.1	**2939**	142	0.8	4726	562	4.7	**3402**	158	3.6
	CGV	3102	238	15.9	**2811**	117	16.0	3459	205	79.8	**3291**	121	86.4
20	C	**4235**	861	0.0	4733	379	0.0	**4972**	892	0.1	6788	437	0.1
	CV	2947	378	1.1	**2215**	117	0.9	3410	415	5.0	**2603**	108	5.1
	CGV	2247	192	15.0	**2124**	87	18.9	2579	112	84.9	**2517**	83	98.7
30	C	**2783**	400	0.0	3757	359	0.0	**3382**	502	0.2	5062	475	0.2
	CV	2011	245	1.2	**1553**	87	1.0	2314	204	7.5	**1888**	67	6.4
	CGV	1501	88	19.2	**1468**	69	21.7	1825	61	111.3	**1812**	56	134.3
40	C	**2070**	318	0.0	3353	353	0.0	**2540**	358	0.5	3979	416	0.5
	CV	1496	194	1.4	**1221**	52	1.1	1894	212	7.4	**1562**	55	7.4
	CGV	1167	56	20.8	**1155**	52	25.4	1491	45	134.1	**1486**	42	189.1

6 Experimental Results

Our testing environment consists of Intel quad-core processors with 2.83 GHz and 8 Gigabytes of RAM. Three kinds of tests are performed to compare the Kruskal-based to the Prim-based heuristic [3]:

1. only the deterministic construction heuristic (in the result tables this test is abbreviated with "C")
2. the deterministic construction followed by the VND, using $df = 1.5$ ("CV")
3. the construction with the GRASP extension followed by the VND, using $\alpha = 0.25$, stopping after ten starts without gain and taking the average values of 30 runs ("CGV")

The instance sets R500 and R1000 each contain 30 complete instances with 500 and 1000 nodes and random integer edge-costs and -delays uniformly distributed in $[1, 99]$. The root node is set to node 0 in all tests. The comparison of only one constructed solution (test "C" in Table 1) indicates that our Kruskal-based heuristic produces usually significantly better solutions than the Prim-inspired algorithm, especially if the delay-constraint is strict. Only in tests with high

Table 2. Comparison of Prim- and Kruskal-based heuristics, applied on Euclidian instance sets with 500 and 1000 nodes (B: delay-bound, C: only construction, CV: construction and VND, $\overline{c}$: average final objective values, σ: standard deviations, $t[s]$: running times in seconds).

		E500						**E1000**					
		Prim-based			Kruskal-based			Prim-based			Kruskal-based		
B	Test	$\overline{c}$	σ	$t[s]$	$\overline{c}$	σ	$t[s]$	$\overline{c}$	σ	$t[s]$	$\overline{c}$	σ	$t[s]$
0.8	C	19.12	0.44	0.1	**18.03**	0.40	0.1	27.56	0.43	0.7	**25.40**	0.32	0.3
	CV	19.00	0.47	1.4	**17.53**	0.40	2.1	27.15	0.65	22.0	**24.81**	0.32	15.6
0.9	C	19.11	0.41	0.1	**18.04**	0.38	0.1	27.48	0.44	0.7	**25.36**	0.32	0.4
	CV	19.02	0.37	1.6	**17.41**	0.36	2.2	26.97	0.76	20.9	**24.65**	0.29	16.3
1.0	C	19.17	0.49	0.1	**17.83**	0.43	0.1	27.38	0.49	0.8	**25.32**	0.29	0.4
	CV	18.97	0.49	1.9	**17.26**	0.34	2.1	26.80	0.93	16.7	**24.51**	0.31	15.4
1.5	C	18.92	0.48	0.2	**17.46**	0.52	0.1	27.30	0.50	1.0	**24.78**	0.32	0.4
	CV	18.75	0.56	2.9	**16.79**	0.36	2.4	26.71	1.07	23.9	**23.85**	0.26	19.4
2.0	C	18.87	0.60	0.2	**17.37**	0.49	0.1	27.29	0.46	1.1	**24.54**	0.37	0.5
	CV	18.69	0.67	3.3	**16.51**	0.33	2.6	26.33	1.29	34.6	**23.49**	0.23	16.6
3.0	C	18.53	0.59	0.2	**17.02**	0.49	0.1	27.04	0.43	1.2	**24.17**	0.29	0.6
	CV	18.09	0.80	4.0	**16.22**	0.30	2.3	25.69	1.43	48.9	**23.14**	0.24	14.0

delay-bounds the Prim-based solution exceeds the Kruskal-based one, but this advantage disappears when also applying the VND. In this test and also when using the GRASP extension ("CV" and "CGV") our heuristic outperforms the Prim-based approach with clear statistical significance. In addition we can observe a higher dependence of the Prim-based heuristic on the specific edge-costs and -delays of the instances noticeable in the higher standard deviation values.

Concerning the runtime the Kruskal-based approach can compete with the Prim-based one and often even beats it, although the administration effort is higher when updating the node information in each step of stage one. We can observe that the runtime is nearly independent of the specified delay-bound B in contrast to the Prim-based heuristic, where tight bounds lead to longer runtimes due to the repeated delay-relaxation process, see Table 1. The general slight increase of the runtime when raising the bound is caused by the fact that in a preprocessing step all edges with $d(e) > B$ are discarded since no feasible solution can include these edges. So tests with lower delay-bounds have to handle less edges.

Additionally we tested our construction heuristic on two sets each consisting of 15 Euclidian instances from the OR-Library originally used for the Euclidian Steiner tree problem [12]. These instances consist of 500 respectively 1000 points randomly distributed in the unit square and the edge-costs correspond to the Euclidian distances between these points. We extended these input data by edge-delays normally distributed around the associated costs and chose a point near the center as root node. The results shown in Table 2 clearly demonstrate the superiority of the Kruskal-based heuristic even if using high delay-bounds. At no time even the VND-improved Prim-based solution reaches the quality of our just constructed spanning tree.

7 Conclusions and Future Work

We introduced a Kruskal-based construction heuristic for the rooted delay-constrained minimum spanning tree problem which produces faster and better results especially for tight delay-bounds and Euclidian edge-costs compared to the Prim-based approach. The runtime is almost independent of the delay-constraint and the cost- and delay-values of the instances. Furthermore the Kruskal-based heuristic seems to be a better starting point for improvement with the presented VND and GRASP.

In the future we want to extend the VND with more neighborhoods maybe based on new solution representations to better diversify the search and therefore find new feasible solutions. Furthermore, we try to apply a modified version of our de-centralized construction heuristic on the rooted delay-constrained minimum Steiner tree problem and compare it to existing approaches.

References

1. Dahl, G., Gouveia, L., Requejo, C.: On formulations and methods for the hop-constrained minimum spanning tree problem. In: Handbook of Optimization in Telecommunications. Springer Science + Business Media (2006) 493–515
2. Gouveia, L., Paias, A., Sharma, D.: Modeling and Solving the Rooted Distance-Constrained Minimum Spanning Tree Problem. Computers and Operations Research **35**(2) (2008) 600–613
3. Salama, H.F., Reeves, D.S., Viniotis, Y.: An Efficient Delay-Constrained Minimum Spanning Tree Heuristic. In: Proceedings of the 5th International Conference on Computer Communications and Networks. (1996)
4. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technical Journal **36** (1957) 1389–1401
5. Skorin-Kapov, N., Kos, M.: The application of Steiner trees to delay constrained multicast routing: a tabu search approach. In: Proceedings of the 7th International Conference on Telecommunications. Volume 2. (2003) 443–448
6. Skorin-Kapov, N., Kos, M.: A GRASP heuristic for the delay-constrained multicast routing problem. Telecommunication Systems **32**(1) (2006) 55–69
7. Ghaboosi, N., Haghighat, A.T.: A Path Relinking Approach for Delay-Constrained Least-Cost Multicast Routing Problem. In: 19th IEEE International Conference on Tools with Artificial Intelligence. (2007) 383–390
8. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematics Society **7**(1) (1956) 48–50
9. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(1) (1959) 269–271
10. Feo, T., Resende, M.: Greedy Randomized Adaptive Search Procedures. Journal of Global Optimization **6**(2) (1995) 109–133
11. Hansen, P., Mladenović, N.: Variable neighborhood search: Principles and applications. European Journal of Operational Research **130**(3) (2001) 449–467
12. Beasley, J.E.: A heuristic for Euclidean and rectilinear Steiner problems. European Journal of Operational Research **58** (1992) 284–292