A Memetic Algorithm and a Solution Archive 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. We present a memetic algorithm for a combinatorial optimization problem called rooted delay-constrained minimum spanning tree problem arising for example in centralized broadcasting networks where quality of service constraints are of concern. The memetic algorithm is based on a specialized solution representation and a simple and effective decoding mechanism. Solutions are locally improved by a variable neighborhood descent in two neighborhood structures. Furthermore, to tackle the problem of repeated examination of already visited solutions we investigate a simple hash-based method to only detect duplicates or, alternatively, a trie-based complete solution archive to additionally derive new unvisited solutions. Experimental results show that our memetic algorithm outperforms existing heuristic approaches for this problem in most cases. Including the hash-based duplicate detection mostly further improves solution quality whereas the solution archive can only rarely obtain better results due to its operational overhead.

Keywords: network design, memetic algorithm, solution archive, delay constraints.

1 Introduction

When designing a communication network with a central server broadcasting information to all the participants of the network, some applications, such as video conferences, require a limitation of the maximal delay from the server to each client. Beside this delay-constraint minimizing the cost of establishing the network is in most cases an important design criterion. This network design problem can be modeled as an $\mathcal{NP}$-hard 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 an undirected graph $G = (V, E)$ with a set V of nodes, a set E of edges, a cost function $c : E \rightarrow \mathbb{Z}_0^+$, a delay function $d : E \rightarrow \mathbb{Z}^+$, a fixed root node $s \in V$, and a delay bound $B \in \mathbb{Z}^+$. 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 $\sum_{e \in P(s,v)} d_e \leq B$, $\forall v \in V$, where $P(s, v)$ denotes the unique path from root s to node v.

Salama et al. [12] introduced the RDCMST problem, proved its $\mathcal{NP}$-hardness and presented a Prim-based construction heuristic and a local search. Manyem et al. [6] showed that the problem is not in APX. In [9] we presented a Kruskal-based construction heuristic, a GRASP, and a variable neighborhood descent (VND) in two neighborhood structures. We improved these results in [10] by using a general variable neighborhood search (GVNS) and ant colony optimization (ACO). Additionally, preprocessing methods are presented to reduce the size of the input graph in order to speed up the solving process. There are many recent heuristic approaches dedicated to the Steiner variant of the problem where only a subset of the nodes has to be reached within the delay bound, the latest are a GRASP [14], a VNS [14], and a hybrid algorithm in [15] combining scatter search with tabu-search, VND, and path-relinking.

Exact methods based on integer linear programming (ILP) have been explored in [4] describing a node-based formulation using lifted Miller-Tucker-Zemlin inequalities extended by connection cuts. Several ILP approaches have been examined by Gouveia et al. in [1] based on a path formulation solved by standard column generation, Lagrangian relaxation, and a reformulation of the constrained shortest path subproblem on a layered graph. In [11] the whole RDCMST problem is modeled on a layered graph which reduces to solving the classical Steiner tree problem on this graph. Stabilized column generation embedded in a branch-and-prize framework is introduced in [5]. However, all these methods can only solve small instances with about 100 nodes to proven optimality in reasonable time when considering complete graphs.

In this paper we investigate a memetic algorithm (MA) for the RDCMST problem using a specialized solution representation together with an efficient decoding mechanism. Local search in two neighborhoods improves the decoded solutions. Furthermore, we present two methods to tackle duplicate solutions, a hash-based approach and a trie-based complete solution archive.

2 Memetic Algorithm

One of the key aspects in designing genetic algorithms is choosing a meaningful solution representation. This is not an easy task especially when dealing with solutions represented by complex graph structures, e.g. constrained trees. An obvious encoding of general graph structures is a binary array of length $|E|$ indicating which edges are part of the solution. However, only a small subset of all possible edge selections may correspond to feasible solutions, e.g. for the RDCMST problem, and naive standard variation operators are therefore unlikely to produce feasible solutions. There are various encodings intended to uniquely represent spanning trees, e.g. Prüfer codes [7], but here again we are faced with the problem that many trees may violate the delay-constraint.

Algorithm 1. Memetic algorithm with duplicate detection

```
1  $initialize(P)$
2  while $time\ limit\ is\ not\ reached$ do
3      $(p_1, p_2) = select(P)$
4      $o = recombine(p_1, p_2)$
5      $mutate(o)$
6      if $\boldsymbol{is\_duplicate}(o, P)$ then $restart\ loop$ or $transform(o)$
7      $T = decode(o)$
8      $improve(T)$
9      $o = encode(T)$
10     if $\boldsymbol{is\_duplicate}(o, P)$ then $restart\ loop$ or $transform(o)$
11     $replace(o, P)$
```

Similar to [2], the genotype in our genetic algorithm consists of an array of length $|V| - 1$ with one delay value $d_v \in [1, B]$ assigned to each node $v \in V \setminus \{s\}$. d_v here represents the maximal allowed delay of path $P(s, v)$ in the corresponding phenotype. To convert such a delay array to a feasible constrained spanning tree we use the following decoding method:

1. Sort all nodes $v \in V \setminus \{s\}$ by delay values d_v in ascending order.
2. Initialize tree with source s.
3. Add next node v in the given order to the tree by choosing the cheapest possible edge without causing a delay higher than d_v on the path $P(s, v)$; if there is no such edge the shortest delay path to v is added and possibly introduced cycles are dissolved.
4. If the tree spans all nodes we obtain a feasible solution, else go to 3.

The decoding method runs in $\mathcal{O}(|V| \log |V| + |E|)$ time. Encoding a feasible tree to a delay array runs in $\mathcal{O}(|V|)$ time by simply using the actual path delays: $d_v = \sum_{e \in P(s,v)} d_e,\ \forall v \in V \setminus \{s\}$. Important about this representation is that every delay array can be decoded to a feasible solution but there is no bijective mapping between delay array and tree: different delay arrays may decode to the same tree while different trees may be encoded by the same delay array. Even encoding and decoding in a row may not lead to the same tree but the resulting tree costs are guaranteed to be at least as low.

Our MA bases on a steady-state genetic algorithm [13] selecting only two parent individuals to produce one offspring per iteration or time step, see Algorithm 1. The main components and operators are:

– **Population initialization:** a random delay value $d \in [1, B]$ is assigned to each node $v \in V \setminus \{s\}$ of an individual
– **Selection:** parent individuals are selected by binary tournaments
– **Recombination:** an offspring is derived by uniform crossover proportional to the parents' solution quality
– **Mutation** (two different operators):
  • a different random delay is assigned to a node v with probability p_m
  • the delays of two different, randomly chosen nodes are swapped

– **Replacement:** an offspring randomly replaces one of the r worst individuals

Additionally, offsprings are locally improved after mutation by local search methods presented in [9]. Depending on the instance size the individuals are either improved by a local search in a single neighborhood or by a VND switching between the two neighborhood structures:

1. **Edge-Replace:** one edge is removed and the resulting two components are reconnected in the cheapest possible way.
2. **Component-Renew:** one edge is removed, the component separated from the source is completely dissolved, the single nodes are then feasibly readded by a Prim-based algorithm while the remaining nodes are joined to the tree by shortest delay paths.

3 Tackling Duplicates

One of the basic problems of local search and population-based heuristics is the potentially repeated examination of already visited solutions. Duplicates decrease the diversity in a population and time is wasted by analyzing or trying to improve these solutions. In a first rather obvious approach to detect revisits hash values of all individuals are computed and maintained in a hashtable. We only store hash values of individuals in the current population, hashes of replaced solutions are discarded. In Section 4 we will see that this apparently artificial limitation is quite beneficial. However, an efficient transformation of duplicates to guaranteed unvisited solutions is not possible.

In a more sophisticated second approach a complete solution archive is built efficiently storing solutions and making it possible to derive new unvisited solutions as replacements of detected duplicates. Promising experiments with similar solution archives to enhance standard genetic algorithms for binary benchmark problems are presented in [8]. Here we adopt and extend this concept for our MA. As in [8], our archive uses a *trie* data structure, which is mostly known from the domain of (language) dictionaries, where a huge number of words has to be stored in a compact way. In our trie, each node contains an array of B references to nodes at the next level, and at each level a dedicated node's delay in a given solution array decides which pointer to follow. Therefore, a single solution is uniquely represented by $|V| - 1$ trie nodes. An example is given in Fig. 3. In this way, the trie has maximum height $O(|V|)$, and an insertion operation and a check whether or not a solution is already contained can always be done in time $O(|V|)$ independently of the number of stored solutions. Some special adaptions are applied to the basic trie data structure in order to reduce the used space while at the same time not increasing access time too much. More specifically, not all delay values are feasible for a node, so the number of array elements of a trie node can be appropriately reduced. To maintain constant access time to an array element a global mapping between delay values and array indices is stored. Furthermore, fully explored subtrees are pruned and replaced by an appropriate marker. The essential aspect which makes our archive approach different to more common simple solution caching strategies as e.g. described in [3],

v	1	2	3	4	5
d_v	2	3	1	4	2

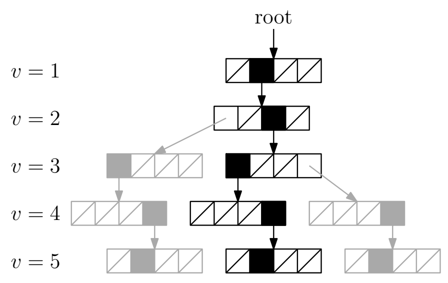

Fig. 1. Given an instance with five nodes and delay bound $B = 4$. The solution archive on the right contains three solutions where the black trie nodes correspond to the solution encoded by the delay array on the left.

is the provision of a function that derives for each duplicate a typically similar but definitely not yet visited delay array. This operation can also be seen as a kind of "intelligent" mutation. In general finding an unvisited delay array in the archive takes $\mathcal{O}(|V|)$ time and the modification is done by assigning a randomly chosen unvisited delay value to a random node. An interesting, although more theoretical side effect of the extension of a metaheuristic by our archive is that the metaheuristic in principle becomes a complete, exact optimization approach with bounded runtime: In each iteration, (at least) one new delay array is evaluated, and by the archive it is also efficiently possible to detect when the whole search space has been covered and the search can be terminated.

An important question is where to integrate the archive in the (meta-)heuristic process and which metaheuristics can benefit from such an extension at all. At some points the solution diversity may be very high lowering the probability of a revisit, e.g. after shaking the solution randomly. Therefore, the archive just grows very large possibly consuming too much space. At other points revisits typically occur more frequently, e.g. after applying local improvement methods, but due to the structure of the metaheuristic it cannot benefit much from consulting the archive. Generally speaking, the solution archive must be used with caution but has the potential to speed up a metaheuristic significantly. We integrated the duplicate check at two different positions in our MA, see Algorithm 1. The first check is performed immediately before decoding the delay array and improving the solution to prevent wasting time on revisits, the second after encoding the solution again to preserve diversity in the population.

4 Computational Results

Our testing environment consists of Intel Xeon E5540 processors with 2.53 GHz and 3 GB RAM per core. The instance sets R500 and R1000 were introduced in [10] and contain 30 complete instances with 500 and 1000 nodes, respectively, and random integer edge costs and delays uniformly distributed in $[1, 99]$. Due to the indeterminism of the MA, 30 runs are performed for each instance and setting. We use a time limit of 300 seconds for each run. All preprocessing methods presented in [10] except the most time-consuming *arbitrary-path* test are applied to the instances reducing the number of edges significantly, see Table 2. We compare our MA to the state-of-the-art heuristics for the RDCMST problem: ACO and GVNS from [10]. In preliminary tests promising parameter values for the MA have been identified: the population size is set to 50, for R500 instances we set the mutation rate to $p_m \in \{0.005, 0.01\}$ and perform full VND for local improvement, for R1000 instances $p_m \in \{0.001, 0.005\}$ and only a single neighborhood is randomly chosen (Edge-Replace with higher probability $p = 0.7$). The replacement parameter $r \in \{10, ..., 40\}$ is dynamically adapted at runtime: initially $r = 10$; if a new best solution is found, r is decreased by 2, and if the search stagnates, r is increased by 2. The higher the parameter r the higher the diversity in the population and the other way round. So if the algorithm should concentrate on intensification of the best solutions, r is automatically lowered while if it gets stuck in a local optimum diversity is increased again.

Experimental results are shown in Table 1. In most cases the MA outperforms existing methods except for two settings where GVNS is still leading. A surprising result is that the use of the trie-based solution archive in general is less beneficial than expected. For the considered problem and MA, the overhead of maintaining the archive is too high even though the operations on it are rather efficient. This can be clearly seen in Table 2 in the average numbers of iterations within the time limit. Here the variant with duplicate detection by hash values yields more kept offsprings after discarding detected duplicates (but one has to consider that only the current population is checked for duplicates). Only for rather low delay bounds the archive is able to yield better results, i.e. both a higher number of new offsprings and a finally higher solution quality. The number of revisits is in general much higher for low delay bounds since the solution space is smaller and the probability of getting stuck in a local optimum after local improvement is higher. Immediately after mutation the duplicate rate is in general rather low provided that the mutation operator is not too limited. Additionally, it can be observed that higher mutation rates are more beneficial in cases of tight bounds, see also [10]. This can be explained again by the fact that it is easier to get stuck in a local optimum requiring a substantial modification of the solution to reach new basins of attraction. In case of loose bounds small changes are enough to escape local optima making a simple swap move in most cases the best choice. Furthermore, the higher the mutation rate, the higher the diversity in the population and the smaller the probability of a revisit. So the solution archive is more effective when having low mutation rates. Generally, most of the time is spent with local improvement and if using full VND the number of achieved iterations further decreases. If only single neighborhoods are examined more iterations are possible and higher mutation rates to cover more areas of the search space are beneficial. Tests without local improvement substantially increased the number of iterations but lead to far worse solution quality.

Table 1. Comparison of GVNS, ACO, and MA with different methods of duplicate detection and mutation operators (swap, two values for p_m); values are average tree costs, B: delay bound, time limit: 300 sec., best results are printed bold

	GVNS	ACO	MA								
			no duplicate detection			hashing			solution archive		
R500			swap	0.005	0.01	swap	0.005	0.01	swap	0.005	0.01
$B = 6$	**8691.6**	8720.4	8716.3	8712.0	8703.8	8707.2	8705.2	8700.3	8710.3	8706.1	8702.0
20	1938.9	1930.8	1930.1	1929.4	1929.2	1929.9	1929.7	**1928.3**	1933.4	1933.8	1933.2
50	893.7	887.5	886.2	**885.9**	886.5	886.1	886.4	886.7	887.2	887.8	888.7
100	599.2	596.8	596.1	596.0	596.0	**595.9**	596.1	596.1	596.3	596.5	596.8
R1000			swap	0.001	0.005	swap	0.001	0.005	swap	0.001	0.005
$B = 6$	9397.2	9367.7	9367.6	9393.6	9366.8	9363.6	9388.1	9366.8	**9353.6**	9369.7	9354.7
20	2346.9	2308.8	**2307.2**	2313.5	2322.8	**2307.2**	2314.2	2322.2	2315.9	2320.9	2333.9
50	1252.5	1238.2	1238.1	1240.9	1248.0	**1237.6**	1241.2	1248.2	1240.7	1242.9	1250.0
100	**1019.4**	1020.9	1020.8	1021.7	1023.1	1021.0	1021.9	1023.5	1021.2	1021.9	1023.3

Table 2. Statistics; $|E_{orig}|$: number of original edges, $|E_{pp}|$: number of edges after preprocessing, I: number of iterations, D: detected duplicates in percent, O: kept offsprings, GB: approx. memory consumption of archive, best results are printed bold

			no det.	hashing			solution archive		
R500	$\overline{\lvert E_{orig}\rvert}$	$\overline{\lvert E_{pp}\rvert}$	$\overline{I} = \overline{O}$	$\overline{I}$	$\overline{D}$ [%]	$\overline{O}$	$\overline{I} = \overline{O}$	$\overline{D}$ [%]	$\overline{GB}$
$B = 6$	7560	3356	**5018**	4774	33	3187	4993	61	0.30
20	25204	15962	**2365**	2260	20	1792	1494	19	0.30
50	63029	26081	**1211**	1185	10	1056	924	7	0.45
100	124750	33282	**804**	797	5	747	678	4	0.65
R1000	$\overline{\lvert E_{orig}\rvert}$	$\overline{\lvert E_{pp}\rvert}$	$\overline{I} = \overline{O}$	$\overline{I}$	$\overline{D}$ [%]	$\overline{O}$	$\overline{I} = \overline{O}$	$\overline{D}$ [%]	$\overline{GB}$
$B = 6$	30254	15264	4251	4309	17	3553	**4820**	27	0.70
20	100874	55603	**1783**	1789	3	1732	1485	2	0.70
50	252217	83986	**1020**	1017	0	1014	958	0	1.00
100	499500	104347	**660**	662	0	659	650	0	1.35

5 Conclusions and Future Work

We presented a memetic algorithm to tackle the rooted delay-constrained minimum spanning tree problem which outperforms existing heuristic approaches in most cases. Additionally, we discussed methods to detect duplicates by either solution hashing or a complete trie-based archive. Hashing works well and is able to improve final solution quality, and in contrast to the solution archive the time overhead is negligible. The trie-based archive can be beneficial for instances with low delay bounds and/or if the number of revisits is very high then providing new unvisited solutions.

In future work we want to adapt our decoding method to improve the quality of decoded solutions. Then maybe less improvement is necessary leading to a higher number of iterations within a given time limit. Additionally, we want to further analyze the integration of solution archives in heuristics, improve the transformation of revisited solutions to more promising ones by considering the solution quality, and decrease the time and space overhead caused by the archive.

References

1. Gouveia, L., Paias, A., Sharma, D.: Modeling and Solving the Rooted Distance-Constrained Minimum Spanning Tree Problem. Computers and Operations Research 35(2), 600–613 (2008)
2. Gruber, M., van Hemert, J., Raidl, G.R.: Neighborhood Searches for the Bounded Diameter Minimum Spanning Tree Problem Embedded in a VNS, EA, and ACO. In: Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1187–1194. ACM, New York (2006)
3. Kratica, J.: Improving Performances of the Genetic Algorithm by Caching. Computers and Artificial Intelligence 18(3), 271–283 (1999)
4. Leggieri, V., Haouari, M., Triki, C.: An Exact Algorithm for the Steiner Tree Problem with Delays. Electronic Notes in Discrete Mathematics, vol. 36, pp. 223–230. Elsevier, Amsterdam (2010)
5. Leitner, M., Ruthmair, M., Raidl, G.R.: Stabilized Branch-and-Price for the Rooted Delay-Constrained Steiner Tree Problem. In: Pahl, J. (ed.) INOC 2011. LNCS, vol. 6701, pp. 124–138. Springer, Heidelberg (2011)
6. Manyem, P., Stallmann, M.: Some approximation results in multicasting. Tech. Rep. TR-96-03, North Carolina State University (1996)
7. Prüfer, H.: Neuer beweis eines satzes über permutationen. Archiv für Mathematik und Physik 27, 142–144 (1918)
8. Raidl, G.R., Hu, B.: Enhancing Genetic Algorithms by a Trie-Based Complete Solution Archive. In: Cowling, P., Merz, P. (eds.) EvoCOP 2010. LNCS, vol. 6022, pp. 239–251. Springer, Heidelberg (2010)
9. Ruthmair, M., Raidl, G.R.: A Kruskal-Based Heuristic for the Rooted Delay-Constrained Minimum Spanning Tree Problem. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2009. LNCS, vol. 5717, pp. 713–720. Springer, Heidelberg (2009)
10. Ruthmair, M., Raidl, G.R.: Variable Neighborhood Search and Ant Colony Optimization for the Rooted Delay-Constrained Minimum Spanning Tree Problem. In: Schaefer, R., Cotta, C., Kołodziej, J., Rudolph, G. (eds.) PPSN XI, Part II. LNCS, vol. 6239, pp. 391–400. Springer, Heidelberg (2010)
11. Ruthmair, M., Raidl, G.R.: A Layered Graph Model and an Adaptive Layers Framework to Solve Delay-Constrained Minimum Tree Problems. In: Günlük, O., Woeginger, G. (eds.) IPCO 2011 Part XV. LNCS, vol. 6655, pp. 376–388. Springer, Heidelberg (2011)
12. 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. IEEE Press, Los Alamitos (1996)
13. Whitley, D.: A genetic algorithm tutorial. Statistics and computing 4(2), 65–85 (1994)
14. Xu, Y., Qu, R.: A GRASP approach for the Delay-constrained Multicast routing problem. In: Proceedings of the 4th Multidisplinary International Scheduling Conference (MISTA4), Dublin, Ireland, pp. 93–104 (2009)
15. Xu, Y., Qu, R.: A hybrid scatter search meta-heuristic for delay-constrained multicast routing problems. Applied Intelligence 1–13 (2010)