Pathfinding with Trees

Samuel Bader

August 26, 2016

Abstract

Tree Cache is a pathfinding algorithm that selects one vertex as a root and constructs a tree with cheapest paths to all other vertices. A path is found by traversing up the tree from both the start and goal vertices to the root and concatenating the two parts. This is fast, but as all paths constructed this way pass through the root vertex they can be highly suboptimal.

To improve this algorithm, we consider two simple approaches. The first is to construct multiple trees, and save the distance to each root in each vertex. To find a path, the algorithm first selects the root with the lowest total distance. The second approach is to remove redundant vertices, i.e. vertices that are between the root and the lowest common ancestor (LCA) of the start and goal vertices. The performance and space requirements of the resulting algorithm are then compared to the conceptually similar hub labels and differential heuristics.

Contents

1 Pathfinding

A tuple $\langle V, E \rangle$, where V is a set of *vertices* and E a set of *edges*, is called a *graph*. The edges can be either tuples $\langle v, v' \rangle$ or sets $\{v, v'\}$, where v and v' are vertices from V with $v \neq v'$. If the edges are tuples the graph is called *directed*, if they are sets it is called *undirected*. Each edge has a *cost* $c(v, v') \geq 0$. A graph is called *unweighted* if it has $c(v, v') = 1$ for all edges $v, v' \in E$, otherwise it is called *weighted*. We concentrate on weighted, undirected graphs but the methods we use can be adapted to work on arbitrary graphs.

A sequence of vertices $\pi = \langle v_1, ..., v_n \rangle$ where $\{v_i, v_{i+1}\} \in E$ for all $i \in \{1, ..., n-1\}$ is called a *path*. The value $c(\pi) = \sum_{i=1}^{n-1} c(v_i, v_{i+1})$ is the *cost* of a path. $start(\pi) = v_1$ is called the *start vertex* and $goal(\pi) = v_n$ is called the *goal vertex* of a path. We use the shorthand notation $\pi_{s,g}$ for a path with $s = start(\pi)$ and $g = goal(\pi)$. The problem of *pathfinding* in a graph is to find a path given the start and goal vertices.

A path π is called *optimal* if $c(\pi) \leq c(\pi')$ for all paths π' with $start(\pi') = start(\pi)$ and $goal(\pi') = goal(\pi)$. If π is optimal, $d(start(\pi), goal(\pi)) = c(\pi)$ is called the *distance* between two vertices. $\pi_1 \circ \pi_2 = \langle v_{1,1}, ..., v_{1,n-1}, v_{2,1}, ..., v_{2,n} \rangle$ is the *concatenation* of two paths with $goal(\pi_1) = start(\pi_2)$. $\pi^{-1} = \langle v_n, ..., v_1 \rangle$ is the *inverse* of a path.

A special case is the use of *grid maps*. A grid map is a two-dimensional array of squares which have different types. What types are available depends on the use case, for a game they might include terrain types like grass, woods or water. For our benchmark there are only two types, *passable* and *impassable*.

A grid map can be represented as a graph where each passable square is a vertex. If the squares are only connected to horizontally and vertically adjacent squares the resulting graph is called *4-connected*. If diagonal moves are allowed there are additional edges to diagonally adjacent squares with weight $\sqrt{2}$ and the graph is called *8-connected*.

The Gridbased Path-Planning Competition GPPC [Stu12] uses slightly more restrictive rules, where diagonal connections are only added if both corresponding cardinal connections are also possible. We use maps and benchmarks also used in the GPPC.

Pathfinding algorithms are often comprised of two separate steps, a precomputation step and a search step. The precomputation step is done once and generates data based on the input graph or map. This generated data is then used in the search steps to find a path for given start and goal vertices.

In section 2 we introduce a simple algorithm to find paths on grid maps. In sections 3 and 4 we present two methods to improve the optimality of the found paths. In section 5 we combine these two improvements to further increase the optimality.

In section 6 we describe two algorithms with similar concepts to TreeCache.

In section 7 we evaluate our implementations of the presented algorithms.

In sections 8 and 9 we present our conclusions and describe possible future work on the presented algorithms.

2 Tree Cache

Tree Cache [And12] is an algorithm to find paths on arbitrary graphs. It was the fastest solution in the Gridbased Path Planning Competition 2012, with an average search time 7% as long as the second fastest solution. It had the most expensive paths, on average 20% higher than the optimal solution.

In the precomputation step a single vertex r (e.g. the centre of the map) is chosen as the root and Dijkstra's algorithm [Dij59] is used to construct a tree of cheapest paths. In this tree each vertex is connected to r by a cheapest path. To find a path from a start vertex s to a goal vertex g in the search step, we traverse the tree from s to r to get $\pi_{s,r}$ and from g to r to get $\pi_{g,r}$. The final path $\pi_{s,g}$ can then be obtained with $\pi_{s,r} \circ \pi_{g,r}^{-1}$. The tree is only traversed towards the root and therefore the only information needed for each vertex is its parent.

The search step is very fast, as it only needs an array lookup per edge and invert one part of the path. The quality of a path depends on the location of r, the path found is optimal if r lies on an optimal path from s to g and gets longer the further away r is. Our implementation places r in the center of the map.

3 Lowest Common Ancestor

In a tree with root r, the *level* of a vertex v is the number of edges between v and r, i.e., $level(r) = 0$ and $level(v) = level(parent(v))+1$. The *lowest common ancestor* of two vertices v and v' is the ancestor of both v and v' furthest from the root, i.e., $lca(v, v') = max_m[level(m)]|m \in ancestors(v) \cap ancestors(v')$.

Every path $\pi_{s,g}$ discovered by Tree Cache with root r can be written as $\pi = \pi_{s,lca(s,g)} \circ \pi_{lca(s,g),r} \circ \pi_{lca(s,g),r}^{-1} \circ \pi_{lca(s,g),g}$. The path $\pi' = \pi_{s,lca(s,g)} \circ \pi_{lca(s,g),g}$ is also a path from s to g and is at least as cheap as π.

We compare two ways to implement finding $lca(s, g)$ and constructing π'.

3.1 Naive implementation

While traversing from s to r and constructing $\pi_{s,r}$, we mark each traversed vertex as visited. While traversing from g to r, if the current vertex v is marked as visited, we stop the traversal with $\pi_{g,v}$. We remove each vertex following v from $\pi_{s,r}$ to get $\pi_{s,v}$ and can now easily construct the final path $\pi_{s,g}$ with $\pi_{s,v} \circ \pi_{g,v}^{-1}$.

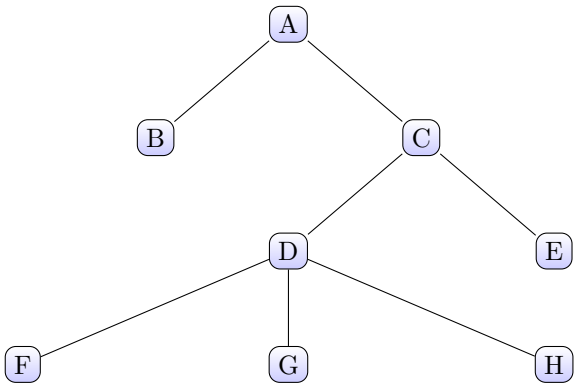


Figure 1: Example Tree

This increases the search time as well as the memory requirements, but the resulting paths are either the same cost (if the root is the lowest common ancestor of the start and goal vertices) or cheaper.

3.2 Implementation using range minimum query

Another algorithm to find lowest common ancestors is based on using a range minimum query on an Euler tour of the tree [BFCP$^+$05].

For an array a, the *range minimum query* finds the smallest value in a given subrange, i.e., $rmq(a, i, j) = argmin_{k|i \leq k \leq j} a[k]$.

An *Euler-tour* of an arbitrary graph is a path starting and ending in the same vertex using each edge exactly once. This is impossible for a non-trivial tree as its leafs are each only connected by a single edge to the rest of the tree. Therefore the tree has to be transformed into a new directed graph by replacing each edge with two directed edges, one for each direction. This transformation can be done implicitly by performing a depth-first traversal where a vertex is added to the tour each time it is passed. For example, the tree in Figure 1 results in the Euler-tour $[A, B, A, C, D, F, D, G, D, H, D, C, E, C, A]$.

If we start from any occurrence of vertex v in this tour and follow it to any occurrence of another vertex v', we pass through the lowest common ancestor of the two vertices, e.g in our example the part of the tour from G to E is $[G, D, H, D, C, E]$, where C is the lowest common ancestor of the two. This is the vertex out of this part which is the closest to the root, i.e. the one with the lowest level. This means that the lowest common ancestor query on the tree becomes a range minimum query on an array containing the levels of each vertex on the Euler-tour.

For this transformation we store three arrays in addition to the tree: One for the Euler-tour, one for the level of each vertex in this tour, and one to quickly find a certain vertex in the tour. We can choose any occurrence of a vertex to

start/end a range minimum query, the correct ancestor is found regardless. To simplify the implementation we always choose the first occurrence of a vertex. For this, the third array contains for each vertex the index of its first appearance in the Euler-tour.

We calculate the range minimum query with an additional two-dimensional array to store the result for all subranges of length $2^i | 1 \leq i \leq log(n)$, which is used to quickly answer the range minimum query for any range. This is done by selecting the value of 2^i closest but smaller to the length of the target subrange and looking at two precalculated subranges of this length, the one that starts at the start of the target subrange and the one that ends at the end of it. As the minima of these two subranges are already known and they cover the whole of the target subrange together, the smaller one of them is the minimum of the target subrange.

All these arrays increase the memory requirements by $O(n * log(n))$ with n being the number of vertices on the map, but the query time is constant, whereas the query time of the naive implementation is dependent on the path length.

4 Multiple Trees

Another way to get cheaper paths is to have multiple roots and compute a tree of cheapest paths for each of them, storing the ancestor vertex as well as the distance to the root. In the search step we calculate the total path length for each tree by adding the distance of the start and goal vertices together, and choose the tree with the cheapest path length. The path is then constructed with the same method as in the standard TreeCache search.

These paths are generally cheaper than in the standard Tree Cache and are guaranteed to be at least as good if the root of the tree in Tree Cache is one of the roots used here. The memory requirement for each tree is twice as big as in the single tree version, as each tree needs to store the distance to the root as well as the parent for each vertex, and increases linearly with the number of roots.

5 Multiple Trees and LCA combined

If we combine these two improvements, i.e. we calculate multiple trees and use lowest common ancestor search on them, they can be further improved. If we use the naive implementation of the lowest common ancestor search we would need to construct the whole path for each tree, calculate their cost and then choose the cheapest one. With the range minimum query implementation the length of a path can be easily calculated from the distances to the root from the start and goal vertices and the lowest common ancestor, which are all stored in the tree. This way, only one path has to be constructed per search.

The memory requirement increases linearly with the number of trees and by $O(n * log(n))$ with the number of vertices on the map.

Name	Space requirement	Time requirement
TreeCache	O(n)	O(p)
SimpleLCA	O(n)	O(p)
MultipleTrees	O(n*t)	O(t+p)
TreeCacheImproved	O(n*log(n)*t)	O(t+p)

Table 1: Comparison of space and time requirements for the different methods (n: number of vertices on the map, t: number of trees, p: number of vertices in the path)

6 Related Work

6.1 Hub labels

Hub labels [ADGW11] calculates a label for each vertex containing a few other vertices (called hubs) and the distances to them. For each pair of vertices, their labels must have at least one hub in common, and the hub must be on a cheapest path between them. This is similar to Tree Cache with multiple trees, with the root vertices as hubs. The main difference here is that in TreeCache each vertex is connected to every root, while the hubs can be different for every vertex. In addition the roots of TreeCache rarely lie on optimal paths.

The labels can be generated in different ways. One of them is to start by adding each vertex to each label and then remove vertices, as long as the conditions are still met (all vertex-pairs have a hub on a cheapest path in common). This is usually done with a greedy algorithm because searching for the optimal vertex to remove requires calculating optimal paths between all vertices over and over.

The vertices in a label are ordered by their distance. This is used in the search step where the labels of the start and goal vertex are traversed from beginning to end until a vertex is found in both of them. Adding the distances of these two entries results in the distance of an optimal path between the start and goal vertices. The path itself is not generated by the Hub label algorithm and has to be constructed with another method.

6.2 Differential heuristics

Differential heuristics [GH05] can be used as a heuristic in algorithms like A* [HNR68]. It is similar to the multiple tree version of tree cache in that it also chooses a number of vertices, here called pivots instead of roots, and calculates cheapest paths from them to all other vertices. In contrast to tree cache, where the parent vertex and the distance to the root vertex are stored, only the distances are stored.

In the precomputation step a set of vertices $P \subset V$ is chosen to act as pivots. This can be done in different ways, e.g. randomly or by adding pivots sequentially, each new pivot as far as possible from all previous ones.

For each vertex v in V the cheapest paths to all vertices p in P are calculated. A lookup table is constructed with the costs of these paths, $d(v, p)$. For any given vertex v and pivot p, $d_p(v, g) = |d(v, p) - d(g, p)|$ is an admissible heuristic, i.e. it never overestimates the actual distance. A straightforward way to get an admissible heuristic using all pivots is just to take the maximum of d_p over all pivots: $dhm(v, g) = max_p d_p(v, g)$.

In the search step A* is used with the following heuristic:

$dh(v, g) = max\{h_{base}(v, g), dhm(v, g)\}$, where $h_{base}(v, g)$ is a basic heuristic, e.g. octile distance for grid maps.

7 Evaluation

7.1 Benchmarks

The different methods were evaluated using maps and scenarios used in the Gridbased Path Planning Competition GPPC [Stu12].

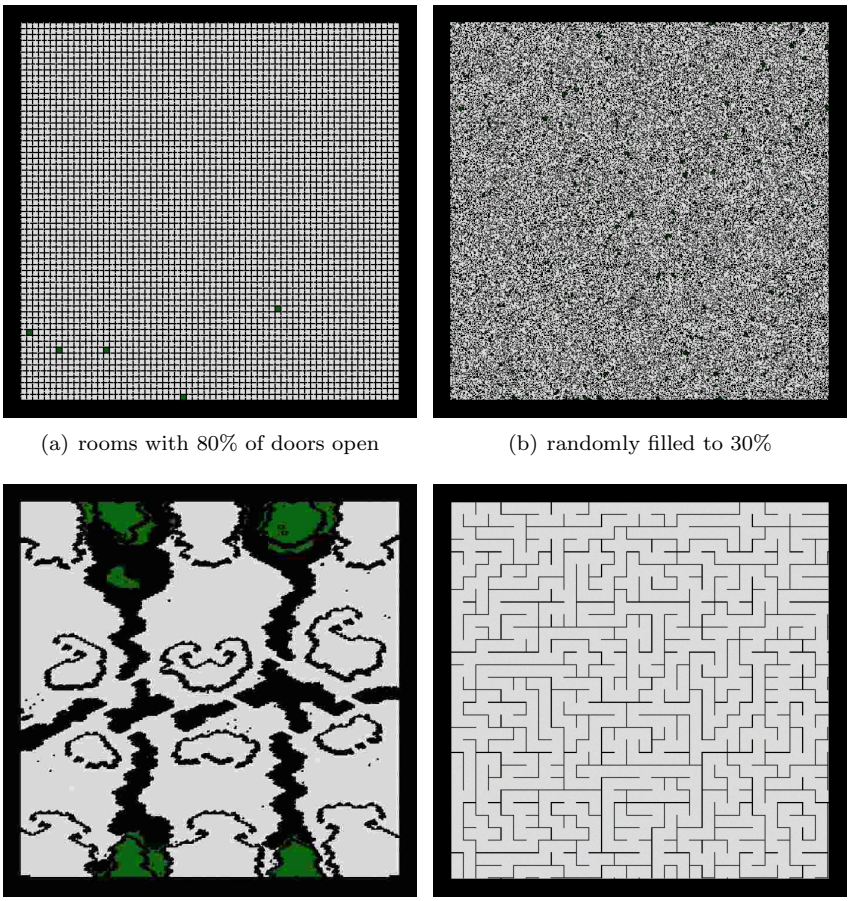
Some maps are taken from the video games Baldurs Gate II, Dragon Age Origins, Starcraft (e.g. Figure 2(c) on the next page), and Warcraft III while others are generic mazes(e.g. Figure 2(d)), rooms (e.g. Figure 2(a)) or filled with random obstacles(e.g. Figure 2(b)). Most maps have a size of 512x512 vertices, while the biggest map (seen in Figure 2(c)) has a size of 768x768 vertices.

Each map has a corresponding scenario with start and goal vertices. For each vertex pair in the scenario the distance of an optimal path is given and used at the end of a benchmark to determine the quality of the paths.

All algorithms were implemented in C++11, compiled with MinGW g++ version 4.9.3-1 and evaluated on a Laptop with a Core i5-2520M CPU @ 2.5GHz and 8 GB of RAM running Windows 10.

7.2 Results

A comparison between standard tree cache and the naive lowest common ancestor approach is shown in Figure 3(a) on page 10. Each dot is one start/goal vertex pair, its x position the cost of the corresponding path using TreeCache and its y position the cost using the lowest common ancestor search. The paths on the diagonal have the same cost in both cases, which means that the root of the tree is the lowest common ancestor of the start and goal vertices. The fact that no paths are above the diagonal shows that the paths do not get worse by using the lowest common ancestor as a cut-off point.



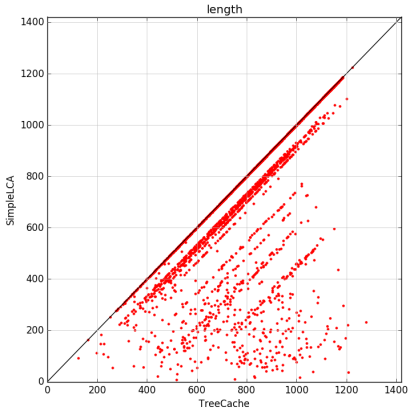
(a) rooms with 80% of doors open

(b) randomly filled to 30%

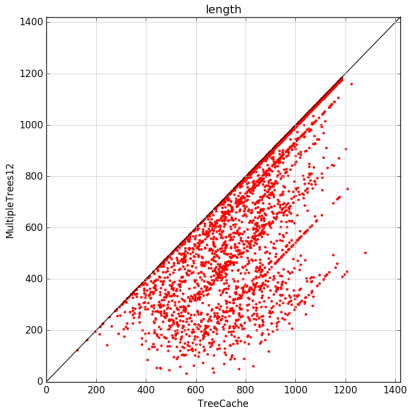
(c) A map from StarCraft called Across the Cape

(d) A maze

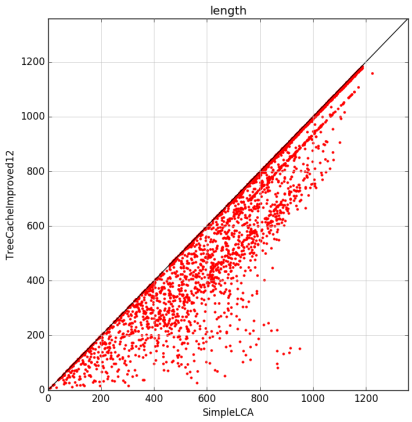
Figure 2: Some maps used in the benchmark



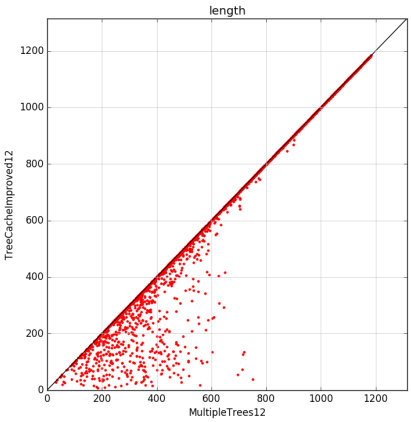
(a) basic TreeCache vs. naive lowest common ancestor search



(b) basic TreeCache vs. 12 trees



(c) basic LCA search vs. 12 trees + LCA search



(d) 12 trees vs. 12 trees + LCA search

Figure 3: Path cost comparisons for the different methods

Figure 3(b) shows a comparison of path cost between standard tree cache and using 12 different trees. The paths on the diagonal use the same root as TreeCache, while all the paths below the diagonal get cheaper by using a different root. The roots where placed in a grid 3 roots wide and 4 roots tall.

Figure 3(c) and Figure 3(d) show the path costs obtained by combining 12 trees with lowest common ancestor search compared to only lowest common ancestor search and 12 trees without lowest common ancestor search, respectively.

Figure 4(a) on the next page shows the same paths as Figure 3(a) but with the computation time instead of the path length and using a log-log scale. Figure 4(b) shows a zoomed in portion of the same plot. The colour of the points indicates how much more expensive the paths are using TreeCache. red points are paths that have the same cost with both methods, while green points are paths where the cost was reduced to a fourth or less by using the lowest common ancestor as a cut-off point. Overall there are more paths that take longer using the lowest common ancestor search. But for paths that are considerably cheaper it is more likely that the additional cost for using the lowest common ancestor search is offset by the gain of having less array lookups.

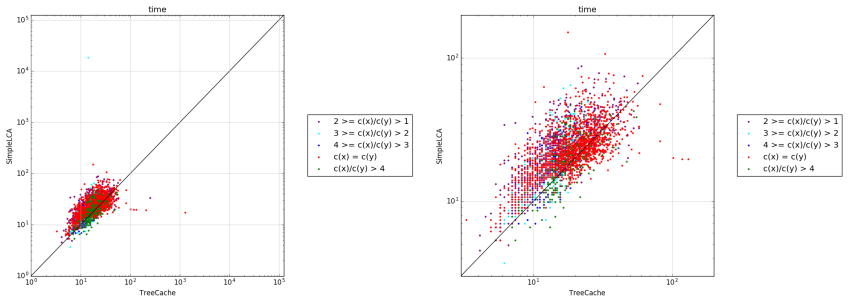
Figure 4(c) shows the computation times for the paths in Figure 3(b) while Figure 4(d) shows it zoomed in. Similar to the lowest common ancestor method, the overhead of checking all trees for the cheapest path is more likely set off if the selected tree results in a much cheaper path than TreeCache.

Figure 4(e) shows the computation times for the paths in Figure 3(d) while Figure 4(f) shows it zoomed in. The combination of multiple trees and using lowest common ancestor search is slower than TreeCache for most paths, but as with previous comparisons the paths that can be improved the most tend to have the shortest computation time.

Figure 5(a) on page 13 shows the geometric mean of path optimality for the variants with multiple trees. The irregularities are due to the way the roots are placed, with e.g. five roots arranged as a 1x5 grid, while four roots arranged as a 2x2 grid resulting in cheaper paths. This shows that with only a few roots, distributed in regular distances across the map, TreeCache can be used to obtain near-optimal paths.

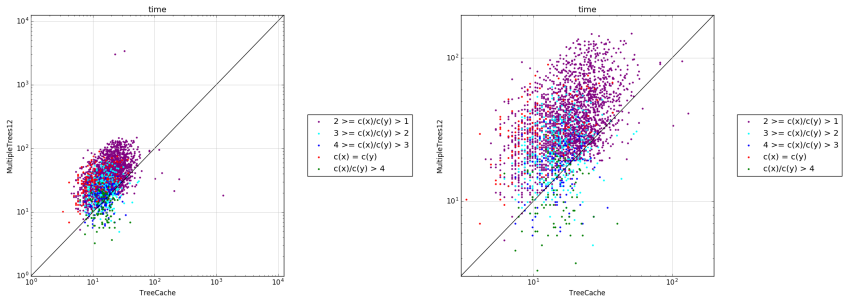
Figure 5(b) shows the average search time for the methods in Figure 5(a). Adding the lowest common ancestor search does not change the search time significantly, but it increases the memory requirements: Using only trees requires 16 bytes per map vertex and tree while trees + LCA search requires 176 bytes per map vertex and tree.

We compare the improved TreeCache to A* with both octile and differential heuristics. The paths obtained with A* are optimal, but the search times are much higher: With octile heuristic, which has no precomputation step, the



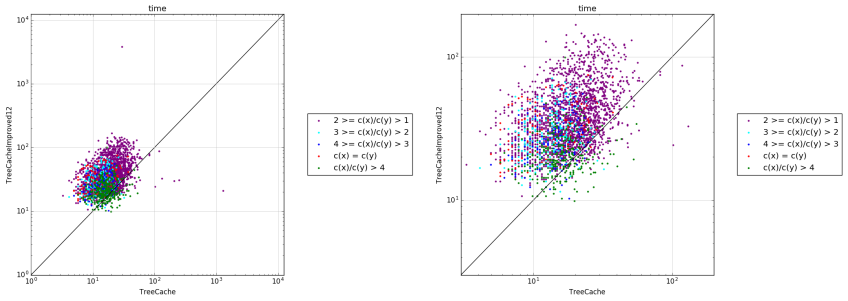
(a) basic TreeCache vs. naive lowest common ancestor search

(b) basic TreeCache vs. naive lowest common ancestor search, zoomed in



(c) basic TreeCache vs. 12 trees

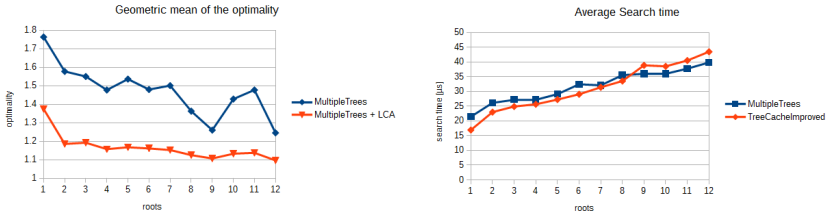
(d) basic TreeCache vs. 12 trees, zoomed in



(e) basic TreeCache vs. 12 trees + LCA search

(f) basic TreeCache vs. 12 trees + LCA search, zoomed in

Figure 4: Computation time comparisons between TreeCache and improvements in μs



(a) Average path cost

(b) Average search time

Figure 5: Results for different numbers of roots

average search time is 145 ms. Differential Heuristics reduces the search time, but with 12 pivots it is still at 46 ms while using 8 bytes of memory per map vertex and pivot.

8 Conclusions

By using multiple trees and lowest common ancestor search TreeCache finds near-optimal paths in a very short time. The downside of the improvements is an increased memory consumption. The improved TreeCache is therefore an ideal algorithm if there is enough memory available and a lot of paths need to be found in a short time.

9 Future Work

Currently our implementation only works on undirected graphs. It can be easily expanded to work on directed graphs by constructing two trees for each root, one for outgoing and one for incoming edges. The search from s to r is then performed in the incoming tree and the search from g to r in the outgoing tree. This would double the required amount of memory but should have no significant effect on search times.

Lowest common ancestor search with the naive implementation would still work with this by simply tracking visited vertices across both trees. The range minimum query implementation is not easily applicable to two trees, which means to use multiple trees and lowest common ancestor search combined it would be necessary to find another fast algorithm for LCA search to achieve similar performance.

An interesting aspect of using multiple trees is the placement of the roots. For our benchmarks we placed the roots in a regular grid. More complex root placement methods which take the geometry of the maps into account might

result in even better paths with fewer roots, which would reduce the memory requirements and make TreeChache applicable in more situations.

References

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th International Conference on Experimental Algorithms*, pages 230–241, 2011.

[And12] Ken Anderson. Tree cache. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS*, 2012.

[BFCP+05] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, November 2005.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[Stu12] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Erklärung zur wissenschaftlichen Redlichkeit
(beinhaltet Erklärung zu Plagiat und Betrug)

~~Bachelorarbeit~~ / Masterarbeit (nicht Zutreffendes bitte streichen)

Titel der Arbeit (Druckschrift):

Pathfinding with Trees

Name, Vorname (Druckschrift): Bada Samuel

Matrikelnummer: 11-054-814

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe.

Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Diese Erklärung wird ergänzt durch eine separat abgeschlossene Vereinbarung bezüglich der Veröffentlichung oder öffentlichen Zugänglichkeit dieser Arbeit.

☒ ja ☐ nein

Ort, Datum: Basel, 26. 8. 16

Unterschrift:

Dieses Blatt ist in die Bachelor-, resp. Masterarbeit einzufügen.