Multidirectional A* Search

William Tidwell[1*] , **Niklas Gustafsson**[2] , **Stefano Fenu**[1] , **Thad Starner**[1]
[1]Georgia Institute of Technology
[2]Chalmers University of Technology
{trace.tidwell, sfenu3, thad}@gatech.edu, niklgus@student.chalmers.se

Abstract

We present Multidirectional A*, a generalization of bidirectional heuristic search to cases with n goals. We compare performance in run-time and number of nodes explored for multi-A*, A*, and bi-directional A* for a well-known Romania map and a street map of Atlanta for three to nine goals. Multi-A* offers improved run-time for smaller numbers of goals ($n \leq\sim 5$) while exploring fewer nodes as the number of goals increases ($n >\sim 5$) when compared to the naive approach of finding a path between all pairs of goals with A* or bi-A* and then constructing the solution. Our method maintains partial solutions while searching and discards all paths that cannot be part of the solution. This approach allows search to terminate before all paths between all pairs of goals have been found. However, as n grows, maintaining partial solutions becomes expensive, even though fewer nodes are explored. Thus, the benefit of multi-A* is highest when the cost of calculating the heuristic is expensive, particularly on large graphs with many nodes.

1 Introduction

Search is a ubiquitous task in AI, with applications in fields such as route-finding [Ikeda *et al.*, 1994], game-playing [Korf, 1997], and robotics [Cohen *et al.*, 2014]. Standard algorithms such as uniform cost search (UCS) [Russell and Norvig, 2003] and A* [Hart *et al.*, 1968] assume a start position and a goal. However, what if there are several goals that should be visited and the system can start at any goal?

For example, imagine that a visitor wishes to fly to England and then drive to London, Cambridge, and Oxford. The goal is to optimize the amount of driving by choosing the shortest path between the three cities such that the visitor flies into the first city and departs from the third. The problem is different from the Traveling Salesman Problem [Dantzig *et al.*, 1954] in that the shortest route between any pair of cities is not known and the visitor does not need to return to the initial city. Another example is visiting n local package pick-up locations in a city before driving to a remote airport (i.e., the distance to the destination is far enough away compared to the local distances such that it does not affect the choice of order of pick-up locations). We assume that ride-shares or taxis are ubiquitous and that we can call one to start from any of the n local locations before visiting them all and traveling the long distance to the airport.

One solution to the problem is to find the shortest path between each pair of goals using A* or bi-directional A* [Pohl, 1969] and then choose the $n-1$ paths that connect the goals. Here, instead, we generalize bi-directional A* into a multi-directional A* with $n > 2$ search targets. Our contributions include

- Describing the multi-A* algorithm for heuristic search with multiple goals
- Comparisons of the algorithm's performance against a naive approach using A* or bi-directional A* on a simple and a complex map.

2 Background & Related Work

In this section we give a brief background on A*, bidirectional search, and bidirectional A*.

2.1 A* Search

A* is very similar to UCS, which is itself a special case of Dijkstra's Algorithm [Dijkstra, 1959]. The difference between the two is the order in which nodes are explored. Unlike UCS, which expands uniformly with respect to path cost, A* uses a heuristic to guide the direction of the search. The heuristic function $h(v)$ approximates the cost of reaching the goal from a node v. The heuristic is used to form an augmented cost $f(v) = g(v) + h(v)$, where $g(v)$ is the known cost from start to v, which determines the priority of nodes in the frontier. Thus, A* takes the cost of the whole path into consideration, while UCS only considers the cost of the path up until the current node.

Any heuristic function used by A* must be admissible and consistent. We say that h is admissible as long as it never overestimates the cost of reaching the goal. That is, assuming the cost of reaching the goal from a node v is c, h is admissible iff $0 \leq h(v) \leq c$ [Russell and Norvig, 2003].

For graphs, we also require h to be consistent (monotonic). We say that h is consistent if for every node v and every successor u of v, we have $h(v) \leq c(v,u) + h(u)$ where $c(v,u)$

*Contact Author

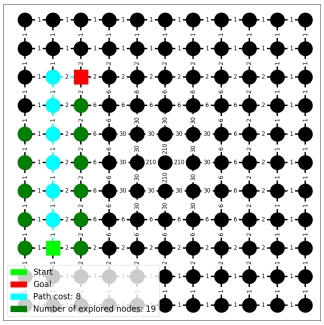


Figure 1: A* using Euclidean distance heuristic. Colored nodes are explored.

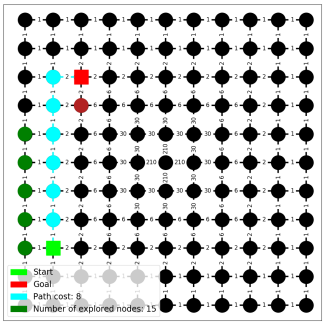


Figure 2: Bidirectional A* using Euclidean distance heuristic. Colored nodes are explored.

is the cost of the path to u via v. In other words, the approximated cost of the remaining path can not increase.

The rest of the algorithm executes almost identically to UCS. In each iteration, the node v with minimal $f(v)$ is popped. When the algorithm terminates a path to the goal with cost μ has been found, and for all nodes v remaining in the frontier we have $f(v) \geq \mu$. For each such node v, denote the cost of reaching the goal from v by c_v. Because h is admissible, we know that for each v, $h(v) \leq c_v$. This means that the cost of reaching the goal via any v is at least $f(v)$. Thus, no node v remaining in the frontier can lead to a lower cost than μ. Hence, A* finds the shortest path to the goal. Figure 1 shows an example of the A* algorithm along with the nodes it explores in finding a path from start to goal.

2.2 Bidirectional Search

While both UCS and A* provide optimal solutions, A* tends to explore fewer nodes. The question is if we can devise an algorithm which performs better in terms of the number of nodes explored. A common approach is to perform two searches simultaneously: one search from the start to the goal, and one in the reverse direction from goal to start. The idea is that if the searches meet in the middle, then maybe they can avoid exploring some of the peripheral nodes.

To understand the motivation behind this, let us consider a simplified version of the problem. Consider performing a bidirectional BFS [Moore, 1959] in a graph where edges have unit cost. Denote the branching factor by b, and assume that the goal is at depth d. The number of nodes explored by the search before the goal is found would be $O(b^d)$. If we were to perform two breadth first searches instead, they would both need to search to a depth $d/2$ in order to meet. When the searches meet a path to the goal would be known and the algorithm would terminate. Thus, the number of explored nodes would be $O(b^{d/2} + b^{d/2}) = O(2 \cdot b^{d/2}) = O(b^{d/2})$, which is significantly less than $O(b^d)$.

2.3 Bidirectional A* Search

Building upon the idea of the bidirectional search is the bidirectional A* Search. Unlike the unidirectional A*, this algorithm maintains two frontiers, two explored sets and two costs and predecessor maps – one for each search direction. The frontiers are initialized with the goal and start node, respectively. In each iteration, the algorithm pops an element from the frontier with the cheapest top element. That is, it considers the top nodes of the two frontiers, and pops the one with lowest cost. Exploring a node is done in a similar way, the only difference being that one needs to check for intersecting frontiers. If the popped node is already explored by the search in the other direction, a path to the goal has been found. Denote the cost of the current candidate path as μ. The algorithm terminates when there does not exist a node v in any of the two frontiers such that $f(v) < \mu$. The optimality argument is similar to that of A* search. Figure 2 shows an example of the bidirectional A* algorithm along with the nodes it explores in finding a path from start to goal. Notice it explored fewer nodes than the unidirectional A*.

2.4 Improvements to Bidirectional A* Search

Since the original Pohl paper on bidirectional heuristic search was published in 1969, much work has been done to improve the performance of the algorithm. A key issue with bidirectional heuristic search was believed to be that search frontiers often pass each other, resulting in more nodes being explored than in the unidirectional case. In [Kaindl and Kainz, 1997], the authors state that the issue is actually the frontiers going through each other, and not around, and that much effort is spent on post-processing to find solutions other than the first one found and determining when to actually terminate. The first issue was more or less solved by BS* [Kwa, 1989] and with the front-to-front evaluations developed by [de Champeaux and Sent, 1977] and [Politowski and Pohl, 1984]. [Kaindl and Kainz, 1997] developed a generic, non-traditional form of bidirectional search in which the direction of the search only changes once, as opposed to alternating back-and-forth between the two searches, along with an approach to dynamically improve the heuristic estimation based on differences between existing calculations and known costs. More recently, [Felner *et al.*, 2010] introduced the single-frontier bidirectional search (SF-BDS) in which each node can decide which way to search, and [Holte *et al.*, 2016] developed a method to guarantee the forward and backward searches meet in the middle.

2.5 Baseline Methods

We compare our method to a "naive" method, of finding a path between all $\binom{n}{2}$ pairs of goals, then building a complete graph with the goals as the nodes and the path costs between

them as edges, and finally solving a modified TSP that does not have to return to where it started. We use a traditional A* and a bidirectional A* for a complete comparison.

3 Method

The goal of the Multidirectional A* method is to develop an algorithm that can find a shortest path containing $n > 2$ goals that does not require finding a path between all $\binom{n}{2}$ pairs of goals and then solving a variant of the TSP on a complete graph to construct the final solution. The key insights include maintaining separate frontiers for each goal; maintaining actual and estimated Leg Costs, or costs between any pair of goals; and maintaining actual and estimated Solution Costs, or costs of solutions that contain all goals. In doing so, we are able to discard solutions that are known to be suboptimal.

3.1 Definitions

Before explaining the algorithm itself in too much detail, we need to first define some terms. Some of these are unique to our method, and some are commonly used but with no widely agreed upon definition.

Leg: Path between a pair of goals e.g. path between goals $a - b$.

Solution: Path containing all goals made up of $n - 1$ Legs.

Optimal Solution: Minimum cost path containing all goals made up of $n - 1$ Legs.

Leg Cost: The cost of the path between two goals e.g. cost of path between goals $a - b$.

Estimated Leg Cost: For two goals a, b, the leg cost is estimated as $\min(Frontier_{a-b}, Frontier_{b-a}, LegCost_{a-b})$, where $Frontier_{a-b}$ is the item in the frontier of goal a pointing toward goal b with the minimum cost.

Solution Cost: The cost of a path that contains all n goals and contains a specific order, equal to the sum of the Leg Costs that comprise it. For example, for goals a, b, c, a solution starting at a and ending at c that travels through b will be a distinct solution with a distinct cost, as will a solution starting at a and ending at b that travels through c.

Estimated Solution Cost: Like the Solution Cost, it is the sum of the Estimated Leg Costs that comprise it.

Frontier: Represented with a Priority Queue, the frontier is made up of nodes that have been processed but have not yet been visited. These are all candidates to be visited on the next iteration.

Visited: A node that has had its neighbors processed, meaning they have been added to the frontier if necessary and have been checked for intersections with other searches, has been explored. Each goal has its own set of visited nodes to prevent a search from visiting a node multiple times.

Explored: Similar to being visited, an explored node is maintained for the entire search. So, while a search from a single goal will not visit a node more than once, multiple searches might visit the same node. The overall count is kept here.

Intersection: When a node has been explored by two searches, those searches are said to have intersected.

Predecessors: The node that precedes a given node in a Leg. When exploring a node, it will be the predecessor for all neighbors added to the frontier.

Heuristic Costs: Costs calculated by the Heuristic function from goals to nodes in the graph.

Search Goals: All goals toward which a given search is actively searching. Once the frontier cost is greater than the Leg Cost, the optimal Leg has been found and must not be searched further.

3.2 Algorithm

The basic algorithm is outlined in Algorithm 1. Prior to starting the search, we must initialize all variables. For each goal, we maintain predecessors, costs, visited nodes, the frontier, and search goals. Then we initialize the Leg Costs and Estimated Leg Costs for each pair of goals as well as the Solution Costs and Estimated Solution Costs for each potential solution to ∞. Finally, we initialize an empty hash table for mapping each pair of goals to its respective Leg once found, and we set $\mu = \infty$, which represents the minimum Solution Cost found so far.

At the beginning of each iteration, we determine the start-state s by finding the frontier with the tuple (f, u, t, w) with the lowest value of f. We then pop the tuple from the frontier of s. The tuple contains information about the node being explored, which is discussed more in the frontier update step below. Every d iterations, an update will be performed on μ and the Estimated Solution Costs. The value of d is currently set to 2^{n-1}, but this value could potentially be optimized further. Since all Solution Costs are initialized to ∞, once $\mu < \infty$, we know we have a Solution. All Solutions with an Estimated Solution Cost $> \mu$ can be discarded. This is due to the admissibility requirement of the heuristic function being used, which means the heuristic never overestimates the cost of a Leg. Because the Estimated Solution Cost includes the heuristic, it is a best-case scenario. If this best-case scenario is already worse than our best solution, there is no need to explore it any further. As we continue to search, better solutions are found, and more of the suboptimal solutions can be discarded.

After the update step, a few checks are performed. First, we check to see if the node has already been visited by the search from this start-state. If so, we discard it and start the next iteration. We then check to see if f is greater than the Leg Cost between the start-state and goal-state. If so, we remove the start-state and goal-state from each other's search goals, respectively, and start the next iteration. If neither check is true, we update the predecessors and visited nodes for the start-state and increment the explored count for the node by one.

Next, we process each neighbor v and check for any new Legs that have been found. If v has not been visited by the search from s, a cost is calculated as $g(s, v)_{new} = g(s, u) + c(u, v)$, where $c(u, v)$ is the cost or weight of the edge joining nodes u and v. If $g(s, v)_{new} < g(s, v)$, then $g(s, v)$ is updated to this new cost. Then, for each search goal t of s, we prepare to add the neighbor to the frontier. We calculate the augmented cost as $f(s, v, t) = g(s, v) + heuristicFn(t, v)$. Though not shown here, we experimented with storing the heurist costs so as to avoid redundancy with positive results. Once f has been obtained, a tuple is added to the frontier of s in the form $(f(s, v, t), v, t, u)$.

Algorithm 1 Multidirectional A*

Input: graph, goals, heuristicFn
Output: optPath, mu, explored

hashMap $predecessors, costs, visiteds$
hashMap $frontiers, searchGoals$
hashMap $legCosts, estLegCosts$
hashMap $solnCosts, estSolnCosts$
while search $\equiv$ True **do**
  $s \leftarrow \min(frontiers[goal]$ for $goal \in goals)$
  $(f, u, t, w) \leftarrow$ pop($frontiers[s]$)
  if update $\equiv$ True **then**
    $\mu \leftarrow \min(solnCosts)$
    $estSolnCosts \leftarrow estSolnCosts \leq \mu$
    if $estSolnCosts \equiv \{\}$ **then**
      $search \leftarrow False$
    end if
  end if
  if $u \in visiteds[s]$ **then**
    continue
  end if
  if $f > legCosts(s, t)$ **then**
    $searchGoals.update(s, t)$
    continue
  end if
  $predecessors[s][u] \leftarrow w$
  $visiteds[s] \leftarrow visiteds[s] + u$
  $explored[u] \leftarrow explored[u] + 1$
  for each $v \in G.neighbors[u]$ **do**
    if $v \in visiteds[s]$ **then**
      continue
    end if
    $g_{new}(s, v) \leftarrow g(s, u) + c(u, v)$
    if $g_{new}(s, v) < costs(s, v)$ **then**
      $g(s, v) \leftarrow g_{new}(s, v)$
    end if
    for each $t \in searchGoals[s]$ **do**
      $f \leftarrow g(s, v) + heuristicFn(G, v, t)$
      $frontiers[s].add((f, v, t, u))$
      if $u \in visiteds[t]$ **then**
        if $g(s, u) + g(u, t) < legCosts(s, t)$ **then**
          $legCosts(s, t) \leftarrow g(s, u) + g(u, t)$
          $leg(s, t) \leftarrow CreateLeg(s, t)$
        end if
      end if
      $estLegCosts.update(s, t)$
    end for
  end for
end while
$\mu \leftarrow \min(solnCosts)$
$order \leftarrow argmin(solnCosts)$
$optPath \leftarrow BuildPath(order)$
return $optPath, mu, explored$

Upon adding the tuple to the frontier, we can now check for an intersection between s and t. If u has already been explored by the search starting from t, then we say the searches have intersected. If $g(s, u) + g(u, t) < LegCost(s, t)$, then $LegCost(s, t) = g(s, u) + g(u, t)$. A new Leg is constructed as $Leg(s, t) = Path(s, u) + Path(u, t)$. After checking for the intersection, the Estimated Leg Costs are updated.

The search continues until all Estimated Leg Costs are greater than μ or all frontiers are empty. At this point, the Optimal Solution is the solution with the lowest Solution Cost. The Optimal Solution, its cost, and the number of explored nodes are all returned.

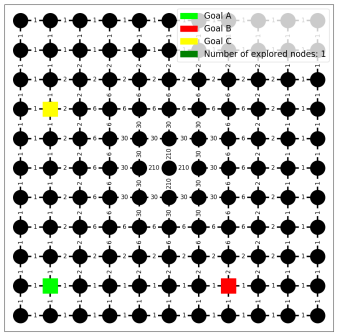
4 Results

We experiment primarily with two graphs. The first is the commonly known Romania map used by Russell and Norvig. It has 20 nodes and 23 edges. For this graph, Euclidean distance is used as the heuristic function. We ran searches for three to nine goals, and for each number of goals, we sampled uniformly at random 100 sets of goals and averaged the results. The second graph is a map of Atlanta, GA consisting of points of latitude and longitude. It has 247,273 nodes and 266,405 edges [OpenStreetMap contributors, 2017]. For this graph, Haversine distance is used as the heuristic function. We also ran searches for three to nine goals, but here we sampled uniformly at random 10 sets of goals and averaged the results, due to the size of the graph.

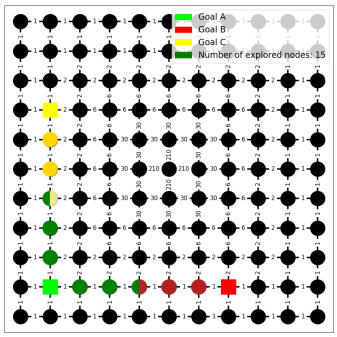
We focus on two main metrics: the number of nodes explored and the total run-time. The number of nodes explored by a search is typically the most salient metric in evaluating the quality of a proposed search algorithm, as it is a useful proxy for how much work the search is performing. However, in practice, we often care more about the actual run-time than any other metric. Especially in our case, where the search has the additional "bookkeeping" of maintaining the Estimated Solution Costs and Solution Costs, we found there were some instances where our run-times suffered even though the algorithm explored many fewer nodes than the baseline methods.

To demonstrate our approach, Figure 3 shows the beginning of a search, when all frontiers have intersected, and the end of the search. We can see that the searches from $A - B$ and $A - C$ intersect between each pair of goals, respectively. A critical observation is that the overall search terminated before any nodes along the Leg from $B - C$ were explored. We know from basic geometry that the hypotenuse of a right triangle is the longest side, so it makes sense that it would be discarded. Figure 4 shows the result of just one of the three bidirectional A* searches needed for the same three goals. In this example, a Leg was found from $B - C$. Because of the construction of the graph, moving diagonally is expensive, involving moving horizontally and vertically as separate steps. Thus, the completed path through $B - C$ passes through A which is reached due to the ease of consistent vertical steps from C and horizontal steps from B. However, the nodes surrounding the completed path must be explored whereas the search was more focused with multi-A*. In addition, Legs must still be found from $A - B$ and $A - C$ with this baseline approach, which leads to redundancy. It is easy to see how inefficient this approach can be compared to multi-A*.

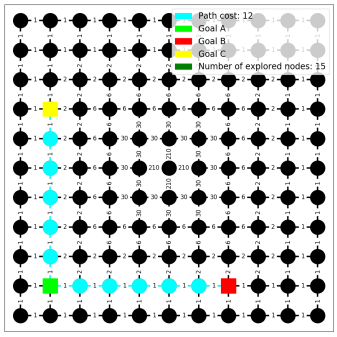
The key advantage of our method is the major reduction in the number of nodes explored as n grows large. Figure 5 demonstrates the average number of nodes explored for 100



(a) Starting state of the multidirectional search



(b) All frontiers in the multidirectional search meet



(c) The multidirectional search is finished

Figure 3: Progression of Multidirectional A*

searches each of three to nine goals on the Romania graph. It is worth noting that for smaller values of n, we actually explore more nodes than some of the baselines, but we still maintain a faster run-time. We believe this result is due to how the solution is constructed by the baseline methods. Recall, after finding all $\binom{n}{2}$ Legs, a variant of the TSP must be solved. In these cases, our "bookkeeping" penalty is quite small, so the average time to explore a node is comparable across methods. However, when including the TSP at the end of the baselines, their total search time increases.

Figure 6 displays the average number of nodes explored for 10 searches each of three to nine goals on the Atlanta graph. The curves are not quite as smooth because we had to run fewer searches due to the size of the graph, but the trend is very much in line with that of the Romania map. At smaller values of n, the naive A* search explores fewer nodes, but our advantage quickly becomes quite large.

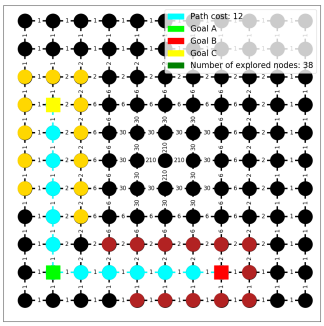


Figure 4: One out of the three bidirectional searches is finished

Previously we mentioned that maintaining the Estimated Solution Costs and Solution Costs penalizes the Multidirectional A* when the heuristic function is simple. Consider Equation 1 below, which is a simple approximation of the run-time:

$$t_r = (t_h + t_i) * n_e + t_b \tag{1}$$

where t_r is run-time, t_h is time to calculate the heuristic, t_i is time to check for intersections, n_e is the number of nodes explored, and t_b is the time for bookkeeping. As the number of goals increases and the heuristic is simple (such as Euclidean distance) t_b for our method increases and eventually comes to dominate t_r. However, when the heuristic becomes even slightly more complex, its contribution to t_r grows, and n_e becomes the dominating component. In such cases, the reduction of the number of nodes explored becomes increasingly valuable. To test this idea, we simulated a more complex heuristic function by adding a delay of 0.05 seconds to the heuristic function each time it was called. Figure 7 shows a comparison of the run-times of the algorithms with and without this delay. With the simple heuristic (no delay), for $n > 7$, the multidirectional A* run-times grow quickly. However, with just a small delay of 0.05 seconds, our method outperforms both baseline methods.

Due to the size of the Atlanta graph and the number of nodes being explored, we were unable to rerun all the searches with a delay. However, we believe we can easily demonstrate the idea. Given that we know the average run-times and nodes explored, we can simply multiply the average nodes explored by some factor and add then add that value to the run-times to simulate a delay. Figure 8 shows a comparison of the run-times of the algorithms with and without a delay. The general shape of the curves matches that of the Romania map.

5 Discussion and Future Work

While the initial results are promising, several improvements are possible. Initializing all Estimated Solution Costs and Solution Costs is costly, even if the Estimated Solution Costs are discarded throughout the search. Perhaps combining these data structures or discarding suboptimal Solution Costs could reduce the complexity and, therefore, run-time. In borrowing from [Felner *et al.*, 2010], we would also like to experiment with a single frontier for all searches. This approach would

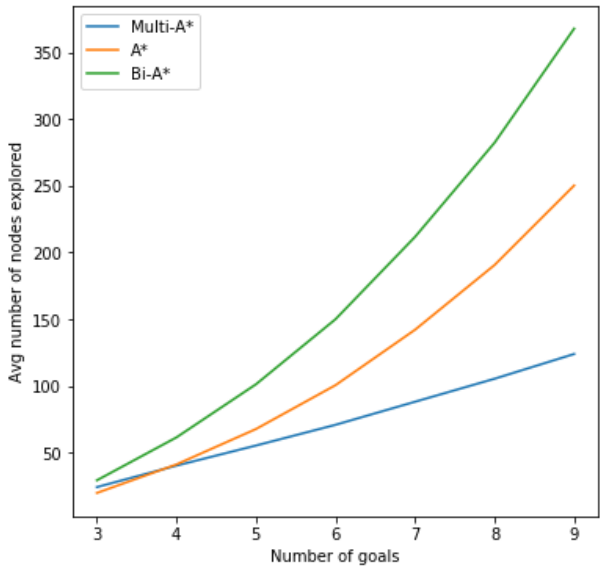


Figure 5: Average Nodes Explored for Romania

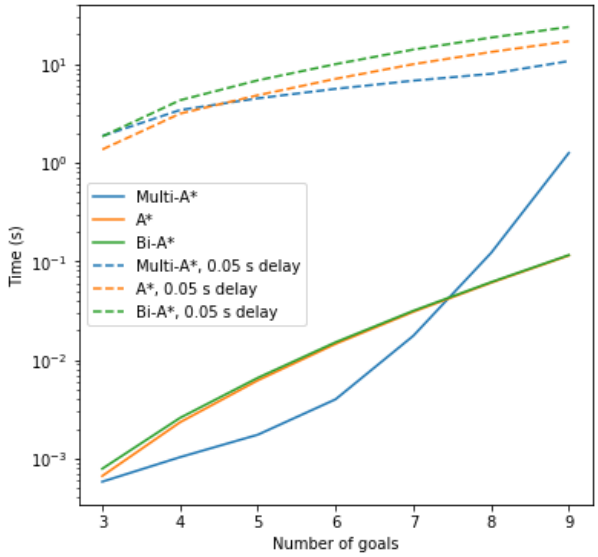


Figure 7: Run-times for Romania

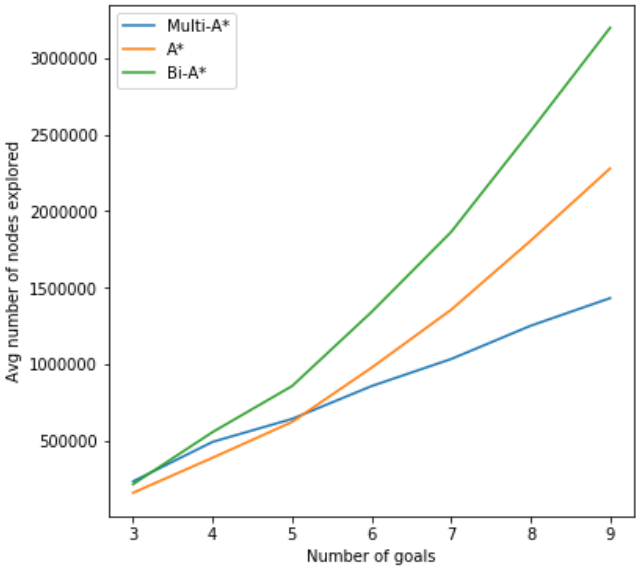


Figure 6: Average Nodes Explored for Atlanta

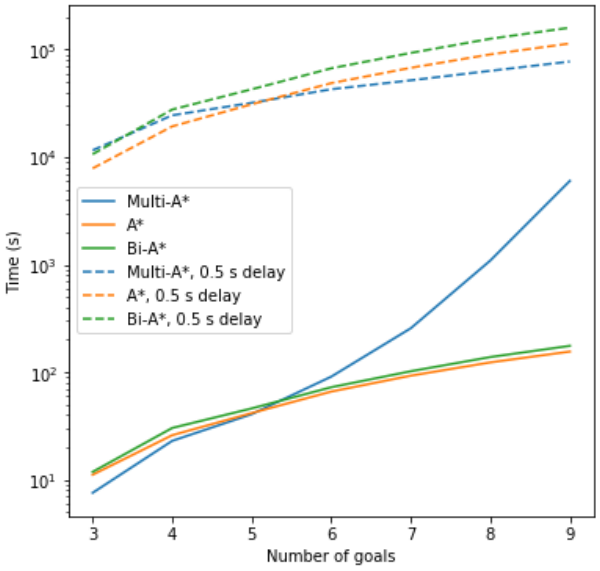


Figure 8: Run-times for Atlanta

allow us to simply pop a tuple off the frontier each iteration without first finding the frontier with the minimum tuple.

Another avenue we explored for $n = 3$ goals but were unable to generalize was the concept of sharing information between the searches. As previously discussed, the baseline methods find all $\binom{n}{2}$ Legs and then construct the Optimal Solution. We proposed performing bidirectional A* and reusing the key elements of the search, like the frontiers, costs, and visited sets, so as to make use of the information gained during previous searches. For example, consider goals a, b, c. We need to find Legs $a-b, a-c, b-c$ to construct the Optimal Solution. After performing the search from $a-b$, we can reuse the frontier of a in the search from $a-c$ and the frontier of c in the search from $b-c$. In doing so, we can reduce the number of nodes explored and, in some cases, reduce the number of searches performed, because a Leg between two goals may be found in the search between two other goals.

6 Conclusion

We have described the Multidirectional A* algorithm which finds the shortest path between n goals. Multi-A* can have significant benefits over a naive implementation with A* and bidirectional A* in number of nodes explored and run-time especially in the case of a high-cost heuristic.

References

[Cohen *et al.*, 2014] Benjamin Cohen, Sachin Chitta, and Maxim Likhachev. Single- and dual-arm motion planning with heuristic search. *The International Journal of Robotics Research*, 33(2):305–320, 2014.

[Dantzig *et al.*, 1954] George B. Dantzig, D. Ray Fulkerson, and Selmer M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[de Champeaux and Sent, 1977] Dennis de Champeaux and Lenie Sent. An improved bi-directional heuristic search algorithm. *Journal of the ACM*, 24(2):177–191, Dec 1977.

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

[Felner *et al.*, 2010] Ariel Felner, Carsten Moldenhauer, Nathan Sturtevant, and Jonathan Schaeffer. Single-frontier bidirectional search. *AAAI Conference on Artificial Intelligence*, 2010.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE*, 4(2):100–107, July 1968.

[Holte *et al.*, 2016] Robert C. Holte, Ariel Felner, Guni Sharon, and Nathan R. Sturtevant. Bidirectional search that is guaranteed to meet in the middle. *AAAI*, 2016.

[Ikeda *et al.*, 1994] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. In *Proceedings of VNIS'94 - 1994 Vehicle Navigation and Information Systems Conference*, pages 291–296, Aug 1994.

[Kaindl and Kainz, 1997] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *JAIR*, 7:283–317, Dec 1997.

[Korf, 1997] Richared E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI Proceedings*, pages 700–705, 1997.

[Kwa, 1989] James B.H. Kwa. Bs: An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence*, 38(1):95 – 109, 1989.

[Moore, 1959] Edward F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, pages 285–292, Cambridge, Massachusetts, 1959. Harvard University Press.

[OpenStreetMap contributors, 2017] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[Pohl, 1969] Ira Pohl. Bi-directional and heuristic search in path problems. https://www.slac.stanford.edu/pubs/slacreports/reports04/slac-r-104.pdf, May 1969.

[Politowski and Pohl, 1984] George Politowski and Ira Pohl. D-node retargeting in bidirectional heuristic search. In *AAAI Proceedings*, pages 274–277, 1984.

[Russell and Norvig, 2003] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, 2nd Edition*. Pearson Education, 2003.