An Online Multi-target Optimal Heuristic Search Algorithm

Swetha.N.G[1], Janani.H[2]

[1,2]*UG Student, Department of Information Technology, Meenakshi Sundararajan Engineering College, Chennai, India*

---***---

Abstract - *Optimal Search algorithms are crucial in various types of applications. Not all applications deal with search problems where only a single source vertex and a single target vertex is present. In this paper, we explore the problem of finding the optimal path in a single-source multi-target setting. The problem is defined with the help of a real-world application analogy. After formally defining the problem, a heuristic search algorithm, similar to the single-source single-target counterpart, is explored. The performance of two different heuristic functions are compared with each other and with the lack of a heuristic. Finally, we explore how this algorithm can be modified to accommodate dynamic addition of target vertices while the optimal path is being traversed.*

1. INTRODUCTION

Consider the problem of delivering newspapers to multiple subscribers. In India, newspapers are delivered to the homes of subscribers by a delivery agent who starts at around the time of sunrise. Assuming the agent starts at time 0 from the publishing outlet *s*, their goal is to minimize the maximum waiting time of the n subscribers at locations t_i where i ∈ {1, 2, ..., n}. This can be modeled as a single-source multitarget problem as follows.

A. Problem Formulation

Let *G(V, E)* be an undirected weighted graph where V is the set of vertices and E is the set of edges. $e(v_i, v_j) \in E$ if and only if vertices $v_i, v_j \in V$ are connected by an edge. Let $w(v_i, v_j)$ be the weight of the edge $e(v_i, v_j)$. For this use-case, we consider the weight of the edge to be the amount of time it takes to travel from v_i to v_j. This time is taken to be directly proportional to the Euclidean distance between the vertices. For simplicity, if two vertices v_i and v_j are connected by an edge, let the time taken to travel between them be equal to the Euclidean distance between them. A single source vertex s and multiple target vertices ti where i ∈ {1, 2, ..., n} (n ≥ 1) are given.

The goal is to find the optimal (shortest) path starting from s and ending in a target vertex such that the path traverses every target vertex ti at least once. Since the edge weights correspond to the time taken by the agent to traverse edges, it can be seen that the length of the shortest path gives the maximum waiting time of the subscribers.

B. Organization

The first part of the paper deals with exploring an Optimal Heuristic Search Algorithm to solve the above single-source multi-target problem. Two different Heuristic functions are compared and experimental results are discussed. The second part of the paper deals with handling the addition of new target vertices while the shortest path is being traversed. As described in the original problem, the goal is still to minimize the maximum waiting time of the subscribers at the target vertices. Note that while the waiting time of the original target vertices t_i is computed as the distance in the optimal path from s, the waiting time of the added target vertices t_j is computed as the distance from the vertex s_k at which the agent was present when the new target vertices were added. Therefore, the maximum waiting time of the subscribers is no longer the length of the optimal path.

1.1 Related Works

While optimal search problems have been extensively studied, their solutions cannot be directly translated to solve specific variations of the problem. The single-source multi-target problem described above (where a vertex may be visited more than once) is commonly solved by computing the all-pairs shortest paths between every pair of vertex using popular algorithms like [1]. While the all-pairs shortest path algorithms are polynomial, another algorithm that uses their output to permute different visit orders of the target vertices is required. This algorithm is exponential in the size of the input and makes the solution non-polynomial. A similar variant of the problem is to disallow paths that visit the same vertex twice. This is addressed in [2] and [3]. Note that it is easy to reduce the traveling salesman problem to this variant in polynomial time. Therefore, this variant is an NP-hard problem. In fact, [2] models the input as a TSP and passes it to a TSP solver. To be able to compute an efficient path in polynomial time, [3] settles for a sub-optimal greedy solution. [4] explores various approximation algorithms to identify sub-optimal yet efficient paths in a single-source multitarget setting. The algorithms discussed in [4] are "fully polynomial". That is, they are polynomial not only in time and space but also in the accuracy of the approximate solution.

2. ALGORITHM

The algorithm is similar to the best-first optimal heuristic search for a single-source single-target problem. In order to accommodate for multiple targets, the following modifications are made. • Each vertex may have multiple copies of it present in the successor priority queue. Each copy corresponds to a different set of target vertices yet to be visited from that vertex. • To initialize the algorithm, a copy of the source vertext s is enqueued into the priority queue corresponding to set of all target vertices (since none have been visited so far). • The heuristic function h(vi, {tj}) is a function of the vertex vi and the set of target vertices, {tj}, yet to be visited from vi. • The algorithm terminates when a copy of a target vertex, corresponding to an empty set of target vertices yet to be visited, is dequeued from the priority queue. The pseudocode for the algorithm is as follows.

```
function SEARCH(s, {$t_i$})
 expanded ← ∅
 gScore ← inf map
 fScore ← inf map
 gScore[s[{$t_i$}]] ← 0
 fScore[s[{$t_i$}]] ← h(s, {$t_i$})
 queue ← {s[{$t_i$}]} .
queue is a priority queue ordered by fScore.
while queue 6= ∅ do
 v[{$t_j$}] ← queue.poll()
 if (v ∈ {$t_i$}) ∧ ({$t_j$} = ∅) then
 return reconstructPath(v)
 else
 expanded ← expanded ∪ {v[{$t_j$}]}
 for neighbor n of v do
 T ← {$t_j$} − n
 if n[T] ∈/ expanded then
 gScoreF romV ← gScore[v[{$t_j$}]] + w(v, n) queue.add(n[T])
 if gScoreF romV < gScore[n[T]]
then
 gScore[n[T]] ← gScoreFromV
 fScore[n[T]] ← gScore[n[T]] + h(n, T)
 end if
 end if
 end for
 end if
 end while
 return not found
end function
```

A. Complexity Analysis

Let |T| < |V | be the number of target vertices. Each node may have the following number of copies.

1C|T| +2 C|T| +3 C|T| + ... +|T| C|T| ∈ O(2|T|)

Assuming the time complexity of the heuristic function is in O(|T|), the total time complexity is in O(|V |2 |T| |E|). Therefore, the time complexity is exponential in the input. If |T| ∈ O(log2 |V |), then the time complexity would be O(|V | 2 |E|).

B. Heuristic functions

The following two heuristic functions are used.

• hmin(v, {t_j}) = minj (dist(v, t_j))

• hmax(v, {t_j}) = maxj(dist(v, t_j))

The first heuristic function directs the search towards the nearest target vertex. The second heuristic function directs the search towards the farthest target vertex. Note that in the presence of only one target vertex, both the functions converge to the heuristic function used in the A* algorithm. In fact, in the presence of only one target vertex, the entire solution converges to the A* algorithm.

C. Proof of Optimality

The algorithm terminates when a target vertex is expanded and no other target vertex is yet to be visited from it. Therefore, the path to the target vertex from the source vertex visited every target vertex at least once. Such a path is also the shortest path since it has the least fScore in the priority queue. Also, it is easy to see that both the heuristic functions hmin and hmax produce under-estimates of the actual path distances. The Euclidean distance between the vertex v and any given target vertex t_j is lesser than or equal to the length of the shortest path that covers v and every vertex in {t_j}. Therefore, the algorithm produces the optimal path when either of the heuristic functions are used.

D. Online addition of target vertices

Upon computation of the optimal path for a given source vertex s and a set of target vertices T, the agent starts traversing that path. However, when the agent is midway in a vertex s 0 , a new set of target vertices T 0 may be added. Let Tu ⊆ T be the set of target vertices from the initial set that are still unvisited.

Therefore, T 0 ∪ T_u is the new set of target vertices and s 0 is the new source vertex. However, the vertices in Tu have already waited for a time equal to the length of the path from s to s 0. Let this be g. In order to model this into the algorithm, another map maxWait similar to gScore and fScore is maintained. The default value is 0. For every copy $t_k[\{t_j\}]$ of every vertex $t_k \in T_u$, maxW ait[$t_k[\{t_j\}]$] is set to g. When a copy of a node n[T] is expanded, its maxWait[n[T]] is updated as max(gScore[n[T]] + maxWait[n[T]], maxW ait[v[{t_j}]]). This is used to compute fScore[n[T]] as max(gScore[n[T]] + hScore(n, T), maxWait(n[T])).

In essence, the fScore is computed as the maximum of the length of the path or the maximum waiting time of a target vertex along the path. Also, since the fScore can. Only become higher due to the updated computation model (since the max function is used), the heuristic functions are still under-estimates. Therefore, the algorithm would still produce the optimal path that carries the least maximum waiting time of its target vertices. The updated algorithm is as follows.

```
function ONLINESEARCH (s, {t_i}, maxWait) .
The initial maxWait map contains g for unvisited target vertices from the initial set and 0 for everything else.
expanded ← ∅
gScore ← inf map
fScore ← inf map
gScore[s[{ti}]] ← 0
fScore[s[{ti}]] ← h(s, {ti})
queue ← {s[{ti}]}
    queue is a priority queue ordered by fScore.
    while queue 6= ∅ do
    v[{tj}] ← queue.poll()
    if (v ∈ {ti}) ∧ ({tj} = ∅) then
    return reconstructP ath(v)
else
    expanded ← expanded ∪ {v[{tj}]}
    for neighbor n of v do
    T ← {t_j} − n
    if n[T] ∈/ expanded then
    gScoreFromV ← gScore[v[{t_j}]] + w(v, n) queue.add(n[T])
    if gScoreFromV < gScore[n[T]]
    then
gScore[n[T]] ← gScoreFromV
maxWait[n[T]] ←
max(gScore[n[T]]+maxWait[n[T]], maxW ait[v[{t_j}]])
fScore[n[T]] ←
max(gScore[n[T]] + h(n, T), maxWait[n[T]])
end if
end if
end for
end if
    end while
    return
    not found
end function
```

3. IMPLEMENTATION

A Java swing application was implemented to simulate the execution of the path finding algorithm.

A. Graph Generation

A heuristic randomized algorithm is used to generate a graph whose vertices are placed randomly in the Euclidean space. The number of vertices and the minimum distance between a pair of vertices is input by the user. The application randomly picks a location and makes sure that the closest vertex is farther than the input minimum distance. If it is not farther, a new location is randomly picked. This process is repeated until the desired number ofnodes is reached or if too many retries are attempted. Once the position of the vertices are generated, every pair of vertex within a distance is connected by an edge. This distance is different for different vertices and is a function of the distance between the vertex and the vertex closest to it. Figure 1 is a graph generated using the application with 500 vertices and the minimum distance between a pair of vertices set to 25.

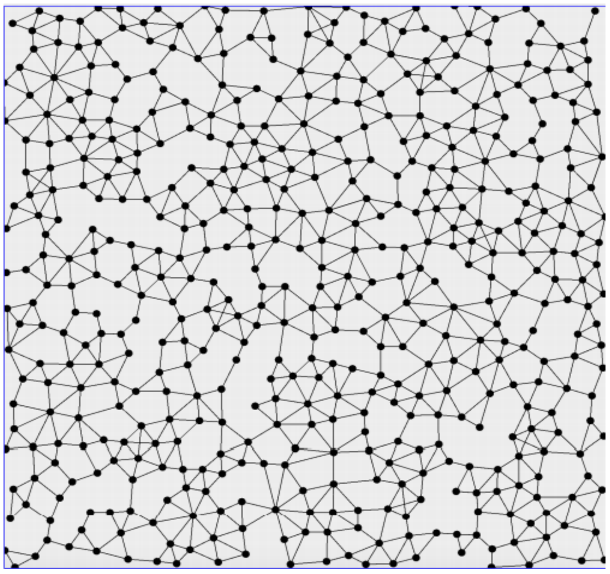


Fig. 1. 500 vertex graph generated using the application.

B. Optimal path generation and visualization

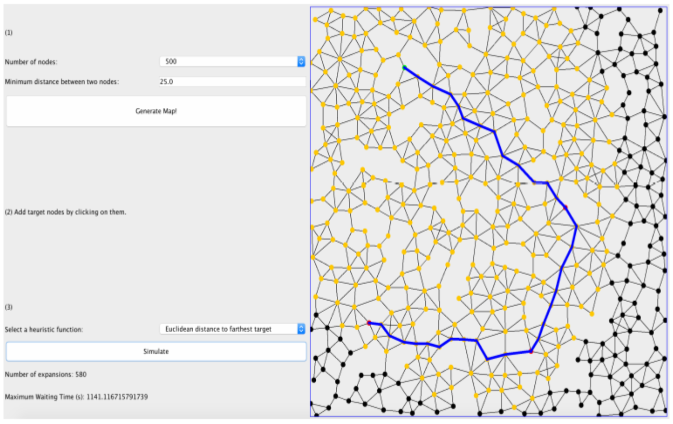
The application allows the user to select a source vertex and one or more target vertices. Upon selecting the source and target vertices, the application runs the algorithm to find the optimal path. This path can be visualized in the UI. Figure 2 is a sample screenshot of the application rendering a shortest path.

C. Online addition of target vertices

Once an optimal path is rendered, the user may select a vertex in the path to mark it as the new source. The unvisited target vertices get a marker on them to show their initial waiting time. The user may add new target vertices. Figure 3 is a sample screenshot of the application rendering a shortest path after new target vertices were added online.

4. EXPERIMENTS AND OBSERVATIONS

The observations are made in graphs generated with 500 vertices and with the minimum distance between a vertex pair set to 25. It was observed that this setting produced



(1)

Number of nodes: 500

Minimum distance between two nodes: 25.0

(2) Add target nodes by clicking on them.

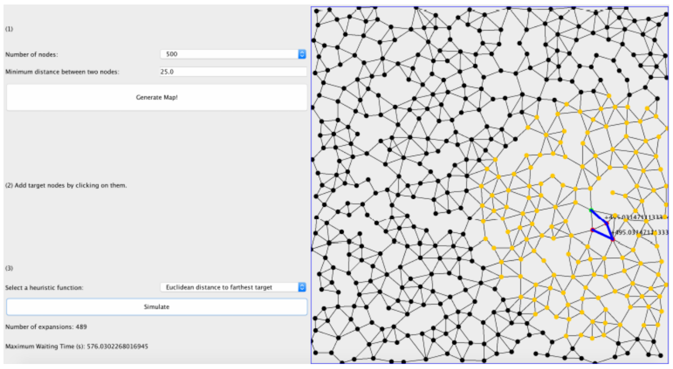
(3)

Select a heuristic function: Euclidean distance to farthest target

Number of expansions: 580

Maximum Waiting Time (s): 1141.116715791739

Fig. 2. Optimal path rendered for visualization.



(1)

Number of nodes: 500

Minimum distance between two nodes: 25.0

(2) Add target nodes by clicking on them.

(3)

Select a heuristic function: Euclidean distance to farthest target

Number of expansions: 489

Maximum Waiting Time (s): 576.0302268016945

Fig. 3. Optimal path after addition of new target vertices online.

Table I shows the observed number of expansions for paths that contain only a single target vertex. As expected, both the min and max heuristic functions produce the same number of expansions. Also, the number of expansions resulting from using the heuristic function is much lesser than the number of expansions resulting from not using any heuristic function.

TABLE I

NUMBER OF EXPANSIONS FOR PATHS WITH A SINGLE TARGET VERTEX. THE COLUMNS INDICATE THE HEURISTIC FUNCTION USED.

	Zero	Min	Max
Run 1	384	75	75
Run 2	467	154	154
Run 3	144	18	18
Run 4	434	100	100
Run 5	400	75	75

Tables II and III show the observed number of expansions for paths that contain three and five target vertices respectively. As expected, the max heuristic function is more efficient since it produces values closer to the actual path length than the min heuristic function. Both the min and the max heuristic functions result in far lesser number of expansions than that from not using any heuristic function.

Another observation is that the number of expansions (irrespective of which heuristic function is used) is lesser than or equal to |V |2 |T|. This is in accordance with the theory.

TABLE II

NUMBER OF EXPANSIONS FOR PATHS WITH THREE TARGET VERTICES. THE COLUMNS INDICATE THE HEURISTIC FUNCTION USED.

	Zero	Min	Max
Run 1	1473	596	280
Run 2	2654	1420	924
Run 3	2114	1059	486
Run 4	1659	647	230
Run 5	1619	647	262

TABLE III

NUMBER OF EXPANSIONS FOR PATHS WITH FIVE TARGET VERTICES. THE COLUMNS INDICATE THE HEURISTIC FUNCTION USED.

	Zero	Min	Max
Run 1	9353	5197	1902
Run 2	6054	3123	1058
Run 3	6608	3416	1176
Run 4	4030	1549	457
Run 5	2042	776	289

5. CONCLUSION

The single-source multi-target problem was formulated and an algorithm was proposed to solve it. The time complexity of the algorithm was shown to be polynomial if the number of target vertices is in O(log2 |V |). It was also shown how the algorithm could be modified to accommodate online addition of new target vertices. Experiments were run to observe the number of expansions. It was observed that the numbers were in accordance with the theory.

6. FUTURE WORK

If the number of target vertices is not in O(log2 |V |), then the algorithm runs in exponential time. It would be useful to explore polynomial time algorithms that produce approximate sub-optimal solutions when several target vertices are present. Another interesting problem to consider is the multisource multi-target problem. That is, how could we coordinate between multiple agents starting from different locations who want to deliver to multiple subscribers at different locations. Each subscriber needs to be visited by at least one agent.

REFERENCES

[1] Demetrescu, C. and Italiano, G.F., 2004. A new approach to dynamic all pairs shortest paths. Journal of the ACM (JACM), 51(6), pp.968- 992.

[2] Spitz, S.N. and Requicha, A.A., 2000. Multiple-goals path planning for coordinate measuring machines. In Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on (Vol. 3, pp. 2322-2327). IEEE.

[3] Lobaton, E., Zhang, J., Patil, S. and Alterovitz, R., 2011, May. Planning curvature-constrained paths to multiple goals using circle sampling. In Robotics and Automation (ICRA), 2011 IEEE International Conference on (pp. 1463-1469). IEEE. [4] Warburton, A., 1987. Approximation of Pareto optima in multipleobjective, shortest-path problems. Operations Research, 35(1), pp.70- 79.