

Dynamic Fastest Paths with Multiple Unique Destinations (DynFast-MUD) - A Specialized Traveling Salesman Problem with Intermediate Cities

Jeffrey Miller
Department of Computer Systems Engineering
University of Alaska, Anchorage
jmiller@uaa.alaska.edu

Muhammad Ali
Department of Mechanical Engineering
University of Alaska, Anchorage
ali@uaa.alaska.edu

Abstract – In this paper we present a fastest path algorithm that contains multiple unique destinations, which is a specialization of the Traveling Salesman Problem (TSP) with intermediate cities [1]. This algorithm can be used with Intelligent Transportation System (ITS) applications for determining the fastest route to travel to a set of destinations, such as required by delivery companies. The dynamic behavior of the algorithm is necessary for ITS applications since the amount of time to traverse a roadway in a vehicular network can change constantly. Assuming that all of the potential paths between all nodes in the transportation graph is known, the algorithm will determine the fastest route to a set of nodes in $O(n^3)$, where n is the number of nodes to visit. The algorithm was executed on theoretical and actual transportation graphs in FreeSim (<http://www.freewaysimulator.com>) [2], and an analysis of the running time and a proof of the correctness of the algorithm are included in this paper.

I. INTRODUCTION

Many drivers have often wondered if the route they are current traversing to a destination is really the fastest. Navigation systems and mapping applications will provide shortest routes, and linking with data from departments of transportation, can even attempt to provide fastest routes. The data obtained from the departments of transportation is only at specific locations, though, and may be stale by the time the driver receives it. Intelligent Transportation System (ITS) applications attempt to ameliorate this problem by using Vehicle-to-Infrastructure (V2I) and Vehicle-to-Vehicle (V2V) architectures to gather data from individual vehicles in a continuous manner. With a continuous flow of data, more accurate fastest paths can be determined, and with V2I and V2V networks, the paths can be retrieved in real-time (or at least much faster than is capable of discrete-based gathering).

At the University of Alaska, Anchorage, we have 50 vehicles equipped with tracking devices that report the vehicle's speed, location, and direction to a central server via a V2I architecture every 10 seconds. From this data, we are able to determine the flow of traffic on the roads that are frequented by these vehicles. This data is then used to accurately route vehicles along fastest paths from their current locations to their desired destinations.

To route a vehicle from its current location to its desired destination, we can assume that the transportation network is modeled as a graph with vertices (or nodes) and edges. Each vertex will represent an intersection in the transportation network, and an edge will represent the road connecting two vertices. The weight on an edge is the time to traverse the road, which will constantly change as traffic conditions change. Shortest path algorithms can then be used to compute fastest paths, since the weight on each edge represents the time to traverse that road. In an effort to make fastest path determination more efficient than having to compute the fastest path for vehicles traveling along the same path, all pairs shortest path algorithms can be used. These algorithms are intended to operate on graphs where the weights on the edges are not changing. When an edge weight does change, the entire algorithm must be executed again. These algorithms are explained in detail in [12]. Dynamic fastest path (DynFast) algorithms attempt to reduce the time to re-compute fastest paths when an edge weight changes. Further, taking into account that a transportation network is rather static, though the weights on the edges change frequently, all of the paths between all pairs of nodes can be pre-computed, which can significantly decrease the amount of time to determine the fastest path when required. These algorithms are part of the All Pairs All Paths Pre-Computed class of algorithms.

All of the previous algorithms work for determining the fastest path between one source node and one destination node. Certain applications, such as those required by delivery companies, require a driver to visit a set of intermediate nodes before returning to the start node, which we call the dynamic fastest path problem with multiple unique destinations (DynFast-MUD). In this paper, we will use the term *route* to denote a set of paths that are required to be traversed from a start node such that all of the necessary intermediate nodes will be visited before returning to the start node. The above algorithms can be used to aid in determining the route, but the number of potential routes is factorial with respect to the number of intermediate nodes that must be visited.

In the DynFast-MUD problem, there is a rather static graph representing a transportation network. Although the graph itself is static, the weights on the edges represent the amount of time to traverse the roadway represented by that edge, so the weights on the edges could be constantly changing. In real-time, we want to determine the fastest route from a start node to a set of intermediate nodes (that

FIGURE I. DEFINITIONS USED IN DynFast-MUD ALGORITHM

Variable Type	Description
NODE	vertex in a GRAPH
EDGE	edge connecting two NODEs in a GRAPH with a weight
GRAPH	set of NODEs and set of EDGES
NODE_SET	set of NODEs where adjacent NODEs do not have to be connected by an edge
PATH	set of NODEs where adjacent NODEs are connected by an EDGE
PATH_SET	set of PATHs
ROUTE	set of PATHs where the destination NODE of each PATH is the same as the source NODE of the next PATH, and where the destination NODE of the last PATH is the same as the source NODE of the first PATH (forming a cycle)

can be visited in any order) before returning to the start node. Since the edge weights are changing, the route could change in the middle of the traversal.

The remainder of this paper is arranged as follows. Section II provides an overview of the related work on shortest path and dynamic path algorithms, as well as ITS algorithms. Section III introduces the DynFast-MUD algorithm and provides the pseudo-code. Section IV proves the correctness of the DynFast-MUD algorithm, and Section V analyzes the running time. The conclusion and future work is provided in Section VI.

II. RELATED WORK

Graph algorithms have been studied for decades, with shortest path algorithms being widely known. Given a graph with a set of vertices (or nodes) and edges, where each edge has a weight, single source shortest path algorithms, such as Dijkstra's [7] and Bellman-Ford's [8, 9] algorithms, compute the shortest path between a single node and every other node in the graph. To compute the shortest path between every node and every other node in the graph, there are all pairs shortest path algorithms, such as Johnson's [10] and Floyd-Warshall's [11] algorithms. Since these algorithms are all assuming that the weights on the edges are static, they must be re-executed whenever the weight of an edge changes. Since this is computationally expensive, dynamic fastest path (DynFast) algorithms improve the amortized cost of updating the weight of an edge and determining a shortest/fastest path. More work on the DynFast algorithms are provided in [13, 14].

Specifically related to ITS applications, the graph is relatively static. Because of this, all of the paths between all of the nodes in the graph can be pre-computed, which will allow edge updates to occur or fastest paths to be retrieved in constant time. The All Pairs All Paths Pre-Computed class of algorithms [3] operate in this manner.

The Traveling Salesman Problem (TSP) [12] is similar to the DynFast-MUD problem, but there are some important differences. In the TSP, all of the nodes in the graph must be visited, whereas only a subset of the nodes must be visited in the DynFast-MUD problem. Also, the weights on the edges in the graph are assumed to be static in the TSP, but they could be constantly changing in the DynFast-MUD problem. In light of this, there is a substantial amount of work that has been done on the TSP.

The Dynamic Traveling Salesman Problem (DTSP) is a special case of the TSP where additional nodes can be added into the graph during the traversal. This problem is explained in [4, 5]. Another variation on the TSP is if the nodes must be visited within a certain time window, which has aptly been called the Dynamic TSP with Time Windows [6]. Although these variations are similar to the DynFast-MUD problem, they differ in two key points: the nodes in the graph are static, and the edge weights are changing.

Jaillet [1] presented the Probabilistic TSP, in which a random subset of cities is visited, but his solution used the solution to the traditional TSP. He then removed the nodes from the TSP solution that were not in the random subset to find the solution to the Probabilistic TSP. Although this paper seemed quite relevant to our work, we discovered that the optimal solution to the subset of cities did not have to coincide with the ordering of the cities in the solution to the traditional TSP.

III. DynFast-MUD ALGORITHM

The DynFast-MUD algorithm operates on a graph $G = (V, E)$, where $V = \{v_1, \dots, v_n\}$ and $E = \{e_1, \dots, e_m\}$. Each edge $e_i \in E$ connects exactly two distinct vertices from V and has a weight $w_i > 0$ associated with it. A start node, which is one of the nodes in V , represents the start and end of the overall route. A subset of nodes $V' \subseteq V$ represents the intermediate nodes that must be visited before returning to the start node. A path is defined as the edges required to be traversed when traveling from one node to another. In the DynFast-MUD problem, a path will exist between intermediate nodes, and all of the paths together will comprise a route. A route will form a cycle with the start node being the same as the end node.

The DynFast-MUD algorithm takes the graph G as a parameter, as well as the set of intermediate nodes and the start node. It will return the route of minimum cost from the start node that traverses all of the intermediate nodes before returning to the start node. The variable types used in the DynFast-MUD algorithm are provided in Figure I. The actual pseudo-code of the DynFast-MUD algorithm is in Figure II, and a list of the helper functions used with a description and the running time is in Figure III.

The algorithm first removes the `start_node` from the set of `int_nodes` if it was in the set initially (lines 4-6). It then gets all of the paths between the `start_node` and the intermediate nodes and between all of the intermediate nodes. The paths are populated into the `PATH_SET`

FIGURE II. DynFast-MUD ALGORITHM

```

1 -- returns the ROUTE that starts at start_node and visits all of the nodes in
2 -- int_nodes, with the minimum overall cost
3 ROUTE DynFast-MUD(GRAPH g, NODE_SET int_nodes, NODE start_node) {
4   if (start_node ∈ int_nodes) {
5     int_nodes = int_nodes - {start_node}
6   }
7   -- get all of the paths connecting all of the NODEs in int_nodes
8   PATH_SET original_path_set = {}
9   for each (NODE src ∈ int_nodes U {start_node}) {
10    for each (NODE dest ∈ int_nodes) {
11      if (src != dest) {
12        -- using All_Pairs_All_Paths algorithm from [3]
13        original_path_set = original_path_set U All_Pairs_All_Paths(g, src, dest)
14      }
15    }
16  }
17  -- iterate through each NODE in int_nodes to find the minimum weight route from
18  -- the start_node with the first NODE visited being each NODE in int_nodes
19  ROUTE fin_route[int_nodes.length]
20  path_set = original_path_set
21  for each (NODE u ∈ int_nodes) {
22    ROUTE route = {} U path_set.get_minimum_path(start_node, u)
23    path_set.remove_all_paths_with_src(start_node)
24    path_set.remove_all_paths_with_dest(u)
25    while (path_set != {}) {
26      PATH p = path_set.get_minimum_path()
27      path_set.remove_path(p)
28      -- no simple cycles exist in the route
29      if (!route.contains_cycle_with_path(p)) {
30        route = route U p
31        path_set.remove_all_paths_with_src(p.src)
32        path_set.remove_all_paths_with_dest(p.dest)
33      }
34    }
35    -- order the paths where the dest from one path is the src of the next
36    -- path, starting with start_node
37    fin_route[u] = {}
38    PATH curr_path = route.get_path_from_src(start_node)
39    while (curr_path != NULL) {
40      fin_route[u] = fin_route[u] U curr_path
41      curr_path = route.get_path_from_src(curr_path.dest)
42    }
43    -- complete the route with the path from the last node to the start
44    -- node by calling fastest_path function from [3]
45    fin_route[u] = fin_route[u] U fastest_path(g, fin_route[u].dest, start_node)
46    path_set = original_path_set
47  }
48  -- find route with minimum cost from the fin_route array
49  ROUTE minimum_route = {}
50  for each (t ∈ int_nodes) {
51    if (minimum_route == {} OR minimum_route.weight > fin_route[t].weight) {
52      minimum_route = fin_route[t]
53    }
54  }
55  return minimum_route
56 }

```

FIGURE III. DESCRIPTION OF HELPER FUNCTIONS USED IN DynFast-MUD ALGORITHM

Function	Description	Running Time
<code>PATH_SET All_Pairs_All_Paths(GRAPH g, NODE src, NODE dest)</code>	Returns a PATH_SET that contains all of the PATHs in the GRAPH g between NODE src and NODE dest	O(1) if the PATHs were already pre-computed
<code>PATH PATH_SET.get_minimum_path(NODE src, NODE dest)</code>	Returns the PATH with minimum weight between the NODE src and NODE dest that is in the PATH_SET	O(1) if using the APAP-PC Constant Query algorithm O(Vm) if using the APAP-PC Constant Update algorithm*
<code>void PATH_SET.remove_path(PATH p)</code>	Removes the PATH p from the PATH_SET	O(n) in the worst case if the PATHs are not ordered
<code>void PATH_SET.remove_all_paths_with_src(NODE src)</code>	Removes all of the PATHs from PATH_SET that have NODE src as the source node	O(n) where n is the number of intermediate NODEs that need to be traversed (since there will only be one path from each NODE to every other NODE in the PATH_SET)
<code>void PATH_SET.remove_all_paths_with_dest(NODE dest)</code>	Removes all of the paths from PATH_SET that have NODE dest as the destination node	O(n) where n is the number of intermediate NODEs that need to be traversed (since there will only be one path to each NODE from every other NODE in the PATH_SET)
<code>PATH PATH_SET.get_minimum_path()</code>	Returns (but does not remove) the PATH with minimum weight out of all the PATHs in the PATH_SET	O(1) if using the APAP-PC Constant Query algorithm O(Vm) if using the APAP-PC Constant Update algorithm*
<code>boolean ROUTE.contains_cycle_with_path(PATH p)</code>	Returns true if the PATH p completes a cycle with the other PATHs already in the ROUTE; returns false otherwise	O(n) where n is the number of PATHs currently in ROUTE
<code>PATH ROUTE.get_path_from_src(NODE src)</code>	Returns the PATH from the NODE src that is part of the ROUTE	O(n) where n is the number of PATHs (which is the number of intermediate NODEs) currently in the ROUTE
<code>PATH fastest_path(GRAPH g, NODE src, NODE dest)</code>	Returns the fastest PATH in the GRAPH g from the NODE src to the NODE dest	O(1) if using the APAP-PC Constant Query algorithm O(Vm) if using the APAP-PC Constant Update algorithm*
<code>ROUTE.weight</code>	Gets the weight of the ROUTE, which is the sum of the weights of all of the PATHs that comprise the ROUTE	O(n) where n is the number of PATHs (which is the number of intermediate NODEs) in the ROUTE

* V is the number of NODEs and m is the maximum number of PATHs considered between any pair of NODEs

variable `original_path_set` (lines 8-16). The `fin_route` variable (line 19) will contain n routes, where n is the number of intermediate nodes. Each route in the `fin_route` variable will correspond to the fastest route from the `start_node` with the first node visited being each of the intermediate nodes. The `path_set` variable (line 20) contains

the same paths that are in the `original_path_set` variable, but allows us to remove paths and still retain the original paths.

Since we will be determining the fastest route from the `start_node` through each of the intermediate nodes, the `for` loop iterates through each of the intermediate nodes (line 21). The minimum weight path from the `start_node` to an

intermediate node is added to the `route` variable (line 22). Since we have already taken a path from the `start_node`, we do not need to consider other paths from the `start_node`, so we remove all paths that originate from that node (line 23). Similarly, since we have already taken a path to the intermediate node `u`, we do not need to consider other paths that terminate at that node, so we remove all paths that have a destination of that node (line 24).

The `while` loop (line 25) then performs a greedy algorithm by getting the minimum weight path from all of the paths that have not been considered yet (line 26). The path does not have to correspond to a node that is already in the route – it just has to have the minimum value of all the remaining paths that were added in line 13. One of the limitations of this algorithm is that all of the edges must have distinct weights so that the path with minimum weight is unique. The path is then removed from the `path_set` (line 27). To ensure that a sub-cycle does not exist within the route, the path will only be added to the route if the path does not create a cycle in the existing route (lines 28-29). If the path does get successfully added to the route, all paths that have a source or destination that is the same as the path that was added will be removed from the `path_set` (lines 31-32). The `while` loop will continue until there are no more paths in the `path_set`, which means that a fastest route has been added to the `route` variable, through which the first intermediate node visited is the NODE `u` from line 21.

Lines 37-42 order the paths in the route so that the destination from one path is the source of the next path in the route. The route is also placed into an array variable called `fin_route` that stores the route based on the first intermediate node visited. The `fin_route` for the intermediate node is then appended with the fastest path from the last intermediate node to the `start_node` to complete the cycle (line 45). The `path_set` is then set back to the `original_path_set` (line 46).

After the `for` loop terminates from line 21, the variable `fin_route` will have the fastest routes from the start node through each of the intermediate nodes as the first visited node. Lines 49-53 determine which of those routes has the minimum weight, and line 55 returns that route from the DynFast-MUD function.

IV. DynFast-MUD PROOF OF CORRECTNESS

The DynFast-MUD algorithm was run on numerous graph structures that corresponded to real transportation networks, as well as theoretical graphs, in the traffic simulator FreeSim [2] to verify correctness. To prove the correctness of the DynFast-MUD algorithm, we use two loop invariants. The first loop invariant operates on the `for` loop on lines 21-47 of Figure II.

Loop Invariant

The `fin_route` variable contains the route of minimum weight from the `start_node` through all intermediate nodes, with `u` being the first intermediate node visited.

Initialization

Before the loop begins, the variable `fin_route` does not contain any routes since no intermediate nodes have yet been considered.

Maintenance

Assume that `fin_route` contains the fastest routes for the first j intermediate nodes. After the `for` loop on lines 21-47, `fin_route` will contain the fastest route for the $j+1$ intermediate node. The proof of the fastest path populating the `fin_route` variable is shown in the next loop invariant.

Termination

When the loop terminates, we have iterated through all of the intermediate nodes, so the `fin_route` variable contains one fastest route for each of the intermediate nodes. That route has the corresponding intermediate node as the first intermediate node visited.

The second loop invariant operates on the `while` loop on lines 25-34 of Figure II.

Loop Invariant

The `route` variable contains a path of minimum weight between two intermediate nodes, where the destination node has not yet been visited in the route.

Initialization

Before the loop begins, the variable `route` contains the path of minimum weight between the `start_node` and the intermediate node `u`.

Maintenance

Assume that after the k^{th} iteration, the `route` variable contains k paths of minimum weight between k pairs of intermediate nodes, where the destination of a path is the destination of at most one path.

In the $(k+1)^{\text{th}}$ iteration, a path of minimum weight is added to the `route` variable. The path that is added is for a destination that has not yet been the destination of a path in `route` (which we know from lines 31-32 since all paths that have a destination that has already been added to route is removed from the `path_set` variable). The path also has minimum weight among all of the paths with destinations that have not yet been visited (which we know from line 26, which retrieved the path with minimum weight remaining in the `path_set` variable).

Therefore, after the $(k+1)^{\text{th}}$ iteration, the `route` variable will contain $(k+1)$ paths of minimum weight between intermediate nodes, and there are $(k+1)$ distinct destinations in the paths that comprise the `route` variable.

Termination

When the loop terminates, we have iterated over all of the k' intermediate nodes, producing k' paths with k' distinct destinations. The k' distinct destinations are from the k' nodes in the `int_nodes` set. Since all of the paths included in the `route` variable had minimum weight, the overall cost of the route is minimum for visiting all k' intermediate nodes.

The last step in the proof comes directly from the loop on lines 50-54, where the minimum route from the route

corresponding to each intermediate node is found. The loop populates the `minimum_route` variable with the route that has minimum cost, and that route is returned on line 55.

V. DynFast-MUD RUNNING TIME

To determine the running time of the DynFast-MUD algorithm, we need to analyze the different loops in the algorithm. We will assume that the number of nodes in the entire graph is V , the number of edges in the entire graph is E , and the number of intermediate nodes to visit is n .

The loop declared on line 9 will iterate $n+1$ times, and the loop on line 10 will execute n times, which will make line 11 execute $n * (n+1)$ times. Line 13 has a running time of $O(n^2)$, and each execution of the `All_Pairs_All_Paths` function will take constant time. The overall running time for lines 9-16 will then be $O(n^2)$.

The `for` loop declared on line 21 will iterate n times. Assuming we are using the APAP-PC Constant Query function, the call to `get_minimum_path` on line 22 will take $O(1)$. Each of the following two lines will take time $O(n)$.

The `while` loop on line 25 will iterate *at most* $n-1$ times because there will be one path added for each of the intermediate nodes. The function calls between lines 26-33 will all execute in either constant time or linear time. The overall running time of the loop on lines 25-34 is $O(n^2)$.

The `while` loop on line 39 will execute n times, which is once for each of the intermediate nodes that is being put in order in the `route` variable. The call to `get_path_from_src` will take linear time, giving the overall running time of the loop on lines 39-42 as $O(n^2)$.

Considering the nested loops inside of the outer `for` loop declared on line 21, the overall running time of the loop between lines 21-47 is $O(n^3)$.

There is one additional `for` loop on line 50 that is used to determine the minimum route from all the routes that were determined in the previous loop. That loop will execute n times, and all of the code in the loop will execute in constant time, giving a running time of $O(n)$.

Adding the running times together produces an overall running time of $O(n^3)$ for determining the fastest route with n intermediate node. It is important to note that the pre-processing step does take $O(V^2E!)$ to execute, though in a transportation network where the edges are rather static, this will only have to be executed when a new road is added or removed. More information on the pre-processing step can be found in [3].

VI. CONCLUSION

In this paper, we have presented an algorithm for determining the fastest route in a network from a start node through a set of intermediate nodes, ultimately returning to the start node. The algorithm, called the dynamic fastest path algorithm with multiple unique destinations (DynFast-MUD) was proven correct through two proofs by loop invariant. The overall running time of the algorithm was

shown to be $O(n^3)$, where n is the number of intermediate nodes that must be visited. The DynFast-MUD algorithm makes use of the All-Pairs-All-Paths Pre-Computed class of algorithms [3]. The pre-processing step to those algorithms allows queries for fastest paths to occur in constant time (if using the APAP-PC Constant Query algorithm). The DynFast-MUD algorithm was verified and proven correct for undirected graphs with unique edge weights, though very few modifications are necessary for the algorithm to work on directed graphs with non-distinct weights.

The DynFast-MUD algorithm is similar to a specialized case of the Traveling Salesman Problem (TSP) in which not all of the nodes must be traversed. The traditional TSP has a running time of $O(V!)$, whereas the running time of the DynFast-MUD algorithm for determining fastest routes is polynomial with respect to the number of intermediate nodes (though factorial with pre-processing).

In the future, we are planning to determine the DynFast algorithm for multiple destinations where we may need to visit the same node more than once. We also will determine the modifications necessary to the algorithm with non-unique edge weights and directed edges, which are necessary for use with ITS applications. Since we already have tracking devices installed in 50 vehicles in the Anchorage area, we will verify our algorithm on a live data set in the FreeSim traffic simulator as well.

VII. REFERENCES

- [1] Jaillet, Patrick. "A Priori Solution of a Traveling Salesman Problem in which a Random Subset of the Customers are Visited." *Operations Research*, Volume 36, Number 6, November 1988.
- [2] Miller, Jeffrey, Ellis Horowitz. "FreeSim – A Free Real-Time Traffic Simulator." *IEEE 10th International Intelligent Transportation Systems Conference*, September 2007.
- [3] Miller, Jeffrey, Ellis Horowitz. "Algorithms for Real-Time Gathering and Analysis of Continuous-Flow Traffic Data." *IEEE 9th International Intelligent Transportation Systems Conference*, September 2006.
- [4] Ghiani, Gianpaolo, Antonella Quaranta, Chéfi Triki. "New Policies for the Dynamic Traveling Salesman Problem." *Optimization Methods and Software*, Volume 22, Issue 6, December 2007.
- [5] Lu, Xiangwen, Amelia C. Regan, Sandra Irani. "The Dynamic Traveling Salesman Problem: An Examination of Alternative Heuristics." *Transportation Science*, 2001.
- [6] Larsen, Allan, Oli B.G. Madsen, Marius M. Solomon. "The A-Priori Dynamic Traveling Salesman Problem with Time Windows." *Transportation Science*, Volume 38, Issue 4, November 2004.
- [7] Dijkstra, E.W. "A note on two problems in connexion with graphs." *Numerische Mathematik*, 1959.
- [8] Bellman, Richard. "On a Routing Problem", *Quarterly of Applied Mathematics*, Volume 16, Issue 1, 1958.
- [9] Ford Jr., Lester R., D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
- [10] Johnson, Donald. "Efficient Algorithms for Shortest Paths in Sparse Networks." *Journal of the ACM*, 1977.
- [11] Floyd, Robert. "Algorithm 97 (SHORTEST PATH)." *Communications of the ACM*, 1977.
- [12] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms – 2nd Edition*. The MIT Press, 2001.
- [13] Demetrescu, Camil and Giuseppe Italiano. "A New Approach to Dynamic All Pairs Shortest Paths." *ACM Symposium on Theory of Computing*, June 2003.
- [14] Misra, S, B.J. Oommen. "New Algorithms for Maintaining All-Pairs Shortest Paths." *IEEE 10th Symposium on Computers and Communications*, June 2005.