# Dynamic Routing[1]

Neil Kalinowski[2] and Carola Wenk[2]

## 1 Introduction

The intent of this research is to find a way of applying a version of a shortest path algorithm such as Dijkstra's, or the more general A*, to a graph with parameterized edge weights that are subject to possibly many updates, and apply this to a dynamic travel routing environment. Let a directed graph $G = \{V, E, w\}$ be given with vertex set $V = \{1, 2, \dots, n\}$, edge set $E$, and a *parameterized weight function* $w : E \times T \to \mathbb{R}^+$. In our application $G$ corresponds to a road map with edges corresponding to road segments, vertices corresponding to intersections, and $w(e, t)$ denoting the *travel time* weight on edge $e$ at entrance time $t$. We assume that $T$ is a suitable representation for different times on different days, months, years, etc. The travel time $w(e, t)$ on an edge $e = (u, v)$ at time $t$ is the time it takes a vehicle to travel from $u$ to $v$ when entering $u$ at time $t$. The *dynamic routing task* is to find the shortest path (i.e., the path with the lowest sum of travel time weights) from a start vertex $s$ to a target vertex $t$. In addition, we plan to find a scheme that works best in a system where the edge weights are frequently updated. Hence, the dynamic routing task is dynamic in two ways: (1) The edge weights are parameterized (as opposed to one constant weight per edge), and (2) the weight function $w$ may be subject to updates.

Dynamic routing has a direct application in route planning for automobile travelers. Current in-car navigation systems generally compute shortest paths without taking the current traffic situation into account. Assuming the existence of a system which provides current and frequent updates of travel time weights, a dynamic routing algorithm will have to cope with

[1] Technical report CS-TR-2007-005, Department of Computer Science, University of Texas at San Antonio. This work is based on an independent study conducted in Fall 2006.

[2] University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249-0667, email: {`nkalinow, carola`}`@cs.utsa.edu`

non-constant edge weights as well as with travel time updates. Also note that real-world limitations such as bandwidth and data transfer rates affect the implementation. This paper presents algorithms for two real-world architectures for the system that would handle this task.

## 2 Related Work

Previous work on dynamic routing either considers (1) the edge weights being parametrized by time, or (2) updates to the weight function, but not both.

Extensive research has been done on (mostly all-pairs) shortest paths problems in graphs with type-(2) updates that allow updates to the weight function and insertion or deletion of edges, see [3] for an excellent survey as well as [6, 11, 5, 10, 9] and the references therein. Algorithms have been designed for special graph classes, see for example [5, 6]. It seems that there is no algorithm known for the single source shortest path problem allowing edge-weight updates (including insertion and deletion) whose update time is asymptotically better than recomputing a new solution from scratch. Roditty and Zwick [11] have shown that the incremental dynamic single source shortest path problem, that only allows edge insertions, is as hard as the static all pairs shortest paths problem (the same holds for decremental which only allows deletions).

Type-(1) dynamic shortest paths problems have been considered for example by [1, 10, 9, 8]. Kauffman and Smith [8] have shown that if the edge weights satisfy a so-called FIFO condition, see Section 5 for more details, then a standard single shortest paths algorithm such as Dijkstra's algorithm still computes an optimal shortest path. Orda and Rom [9, 10] have shown that if the FIFO condition is not fulfilled then a shortest path can visit infinitely many edges (although in their framework the edge weights and the travel times per edge are different, it should be possible to construct a similar counter example in our setting). They have also shown that the problem is NP-hard in general, and they give several criteria for feasible special cases.

Another possibly interesting direction is that of Deng and Wong [4] who compute a shortest path based on statistical variance. The paper is written for an electronics application to increase quality. In order to apply this to our setting, the idea would be to search for a path based on statistical analysis of the path. One can set up the algorithm to choose a route that has the lowest historical amount of variance, thus decreasing the likelihood

of the path changing.

# 3 Architecture

The system by which the edge weights are updated drives the algorithm for finding the shortest path. We will explore two of the many options. Both options assume the existence of a central server that holds the road map as well as the most up-to-date travel time weights. In order to update travel times automatically across the whole road map (as opposed to making few manual updates) a recently introduced approach [2] is to collect Global Positioning System (GPS) data from vehicle fleets. In the absence of gathered (current) travel time data for an edge the system relies on the speed limit of the road or on historical data to determine the parameterized edge weight. See Section 6 for more details.

We assume that the vehicle is equipped with a local computer (navigation system) that directs the driver on the shortest path using its own GPS tracking device. It can get an update of the latest published map at any time through the internet while hooked up to a home computer, but while in the vehicle it will receive updates through a lower bandwidth connection such as a cellular telephone. Due to bandwidth issues and the cost of airtime a continuous update of the map or any other information is not feasible.

The two presented dynamic routing architectures differ in where the computation of the shortest path is performed: either local to the vehicle or central to the database that holds the map with the most up-to-date information.

## 3.1 Local Computation

In this option the vehicle's on-board computer calculates the shortest path. Due to the update issues mentioned above, the vehicle's computer will not have the most up to date information. We assume that the computer gets updated every $m$ minutes, where $m$ is a parameter to be chosen. We cover this option in Section 5.1. Bandwidth issues may be addressed by updating the on-board computer only with travel times for edges that are either on or close to the current path.

## 3.2 Central Computation

In this option all shortest path calculations are performed at the central database location with the most up-to-date information. Because of this,

each path calculated is the most accurate and any weight changes to the edges along the path can be dealt with immediately. Unfortunately, the vehicle's computer does not have direct access to this information, so it can only get an updated path every so often.

However, if the central database computer does not actually look for the best path but instead looks for large deviations from the vehicle's current path, this option could provide some advantage. In Section 5.2 we propose a scheme where the central computer is monitoring the current cost of the previously chosen path, based on currently updated travel time edge weights. Then an alternate path (if there is one) would be sent to the vehicle's computer if the path's current cost exceeds the original cost of the path by some threshold.

# 4 Shortest Paths Algorithms: A* and Dijkstra

Algorithm 1 is the basic flow of the A* algorithm [7] to find the shortest path in a weighted graph from a start vertex $s$ to a target vertex $t$. Dijkstra's algorithm is a special case of A* when $h$ (explained below) is always 0. We describe the algorithm for a weighted graph with constant non-negative edge weights and denote with $w(u, v)$ the weight of edge $(u, v)$.

The algorithm keeps a list of vertices called the *open list* that is implemented as a min-heap (or for performance reasons as a Fibonacci min-heap). A vertex $v$ is stored along with the value $f(v)$ which is used as the key value to rank the vertices in the heap. For any vertex $v \in V$ its $f$-value is defined as $f(v) = g(v) + h(v)$, where $g(v)$ is the cost of a shortest path from $s$ to $v$ and $h(v)$ is a heuristic that estimates the cost of a shortest path from $v$ to $t$.

Lines **13–19** are commonly referred to as *relaxing* vertex $v$. Note that $h(u)$ is a heuristic estimate of the cost from $u$ to the target vertex $t$. In Dijkstra's algorithm $h$ is always 0 and the vertices in the open list (heap) are thus ordered by their $g$ values.

Algorithm 1 has been proven to find the optimal path and to be admissible, given that the heuristic $h$ does not give an overestimate [7]. If edge weights correspond to Euclidean distances then a conservative choice for $h(v)$ for $v \in V$ is the shortest (straight-line) distance from the $v$ to $t$ which obviously can never give an overestimate. In our setting $h(v)$ would have to be a lower bound for the travel time from $v$ to $t$ which can be obtained by combining the Euclidean distance with known minimum speed information.

The graph in Figure 1 gives an example of what can happen if $h$ overes-

**Algorithm 1**

```
1  for all v ∈ V
2      Set f(v) = g(v) = ∞
3      Add v to the open list
4      Initialize a predecessors array P to P(v) = −1
5  end
6  Set f(s) = g(s) = 0
7  while the open list is not empty
8      Extract the vertex v with the smallest f-value from the open list.
9      if v == t
10         Trace the shortest path back from s to t using P.
11         Exit.
12     end
13     for every u ∈ V adjacent to v
14         if f(u) > g(v) + w(v, u) + h(u)
15             Set g(u) = g(v) + w(v, u)
16             Set f(u) = g(u) + h(u)
17             Set P(u) = v
18         end
19     end
20 end
21 Output that there is no path from s to t.
```

timates the distance to $t$. The shortest path from $s$ to $t$ is $s - v - u - t$ with a total weight of 7. In the beginning, $g(s) = 0$ and $g(u) = g(v) = g(t) = \infty$. After relaxing $s$, the A* algorithm assigns $f(u) = 11$ and $f(v) = 16$. It would then choose to relax $u$ first, which assigns $g(t) = f(t) = 8$, and afterwards it relaxes $v$, and then it outputs the path $s - u - t$ of weight 8 which is longer than 7. The problem is that the estimate $h(v) = 15$ is an overestimate since the real distance is 6. The estimate for $u$ is also too great. Setting the estimates to something more reasonable, such as the straight line distance to $t$, yields $h(u) = 5$ and $h(v) \leq 6$. The algorithm would then compute $f(u) = 8$ and $f(v) \leq 7$ and would therefore choose to relax $v$ first, leading to the correct answer.

The main drawback of A* is memory use. For large graphs the number of vertices in the *open* list can fill up the available memory [12]. Therefore a

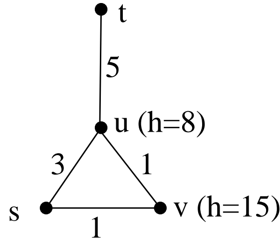


Figure 1: $h$ overestimates the distance from $u$ to $t$.

version of A* such as recursive best first search (RBFS) or memory-bounded A* (MA*) should be used. Both algorithms behave the same as A* except that some vertices with large $f$-values are taken out of active memory.

# 5 Dynamic Routing

Algorithm 1 can be modified to work on graphs with parameterized edge weights as follows: Let $t_0$ be the time at which the route should start at $s$ and initialize $g(s) = t_0$. Instead of accessing the constant weight $w(v, u)$ for the edge $(v, u)$ in lines **14–15**, access the parameterized edge weight $w(v, u, g(v))$. When $v$ is extracted from the open list the algorithm has already computed a shortest path from $s$ to $v$ with total weight $g(v)$. Since edge-weights correspond to travel times, $g(v)$ is the time that a traveler would enter edge $(v, u)$ when having traveled on a shortest path from $s$ to $v$.

It is however not readily apparent that this approach computes the correct shortest path. Kaufman and Smith [8] have shown that this is the case as long as the edge weights fulfill a *non-passing* condition:

$$\begin{array}{l} \text{For all } \ e \in E \ \text{ and } \ t_1, t_2 \in T : \\ \text{If } \ t_1 \leq t_2 \ \text{ then } \ t_1 + w(e, t_1) \leq t_2 + w(e, t_2) \end{array} \tag{1}$$

This *non-passing* condition means that if a car $B$ enters an edge after a car $A$ it will not reach the end of the edge before car $A$. We assume that the parameterized travel-time edge weights in the graph $G$ fulfill this non-passing condition. This means that we assume that the vehicles which collected the travel times did exhibit this non-passing behavior. It would be interesting to investigate the effects in a graph in which a few edges do not fulfill the non-passing condition.

## 5.1 Local Computation

In the case of the vehicle calculating the path locally the car will receive updates every $m$ minutes. The updated information will allow a traveler to avoid traffic jams or unusually long travel times. Receiving updated travel times is of no use if the traveler cannot take advantage of them. Traveling on a path that has few options to be altered, such as a highway, the traveler would often not be able to take advantage of the new traffic information.

To alleviate this, the algorithm can attempt to ensure that the traveler will be at a vertex with a high degree at the time when an update will arrive. This way the traveler will have more options to alter the path in the event of a gross change in expected travel times. So, if the algorithm knows that an update will arrive in $m$ minutes it can make sure that the traveler is at a vertex with a lot of options at that time.

One may think that the best option is to have the algorithm greedily choose the vertex with the highest degree in each step. Unfortunately, this can cause the traveler to end up in a less than desirable location.

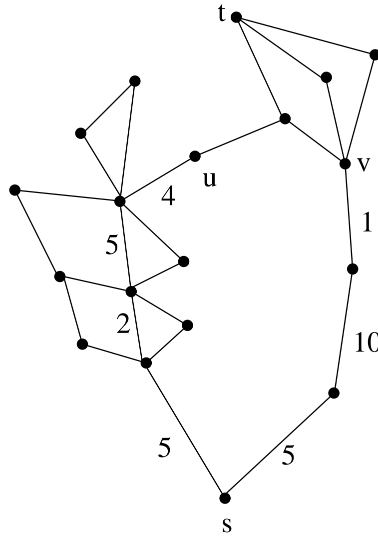


Figure 2: Greedy example with $m = 16$ and $t_0 = 0$.

Figure 2 shows a graph in which this greedy approach fails. Assume $m = 16$. Following the vertices with the highest degree will put the traveler at a vertex $u$, which is of degree 2, when the update comes at time 16, whereas a better algorithm could have placed the traveler at vertex $v$ which

has degree 4 in the same amount of time.

The knowledge that an update will come in $m$ minutes would allow the algorithm to go through some number of low degree vertices to get to a vertex that has a high degree when the information comes in. Without the knowledge that an update will come in $m$ amount of time the algorithm may greedily choose a path with high degree vertices to prepare for an update "any minute" and miss the even higher degree vertices that would allow the algorithm to take advantage of the new information when it arrives.

**Algorithm 2**

To implement this we propose the following algorithm which follows Algorithm 1 with the following modifications:

1. After line **12** in Algorithm 1, if $g(v)$ is greater than the update time then add $v$ to a *ready list* and then break to the beginning of the while loop. The *ready* list is implemented as a min-heap with the $f$-value as the key.

2. Continue relaxing vertices in the *open* list until there are no more vertices in the list.

3. The *ready* list now contains all vertices that can be reached right after the update has come in. Take the top $k$ vertices from the *ready* list and choose the one with the highest degree as the target $t_1$ for this portion of the path. Here $k$ is an arbitrary parameter that could be set to $k = 10$ for example.

4. Backtrack from $t_1$ to the source vertex $s$ to get the partial path for the vehicle.

5. While the vehicle is on the edge directly before $t_1$ the update will arrive. At this point, start the same algorithm over with $t_1$ as the new source. This includes clearing the *ready* list (the *open* list will already be empty by design).

One issue with this algorithm is that it can choose a dead-end. Consider the graph in Figure 3. For an update time of $m = 16$ after the first round of the algorithm vertices $u$ and $v$ would be in the *ready* list and in the top $k$ as in step 3 above. In step 3, the vertex $u$ of degree 5 would clearly be chosen over $v$ which is only of degree 2. However, this vertex would not get the user closer to $t$. In fact, the algorithm would compute an infinite path alternating between $s$ and $u$.

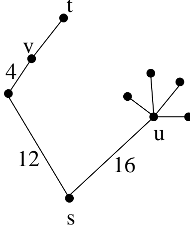


Figure 3: Dead-end example

This problem occurs because the algorithm does not take into account the entire graph when deciding on a sub-path. Furthermore, the straight-line distance heuristic function $h$ does not take into account the fact that the actual path to a goal from a vertex would involve going back the way the vehicle already traveled thereby making the actual path from the vertex much longer.

One way to attack this problem is to rank the top $k$ vertices from the *ready* list not only according to degree but also according to the shortest path from them as well. In other words, compute the actual shortest path using Algorithm 1 from each of the $k$ vertices as the source to the target $t$ and and rank these $k$ paths according to a combination of the degree and the cost of the shortest path. This should allow the algorithm to avoid vertices that may have a low $f$ value that gets them into the *ready* list but have a higher cost to the target $t$.

### 5.2 Central Computation

When a vehicle starts out on a trip it will request a path from the central computer. The central computer will apply Algorithm 1 to find the shortest path based on current information and send it to the vehicle.

As the vehicle travels along the path, the central computer is repeatedly checking the current cost of the path based on new travel time edge weight information. The central computer only needs to calculate the new cost if new information comes in. A data structure could be implemented that allows the computer to know which paths need to be recalculated when a single edge is updated. Each edge would have a pointer to any paths that use it and when it gets updated it could flag the paths for updating. It may, however, ultimately be more efficient to just update the path every 30 seconds, or some other time interval, based on the expected occurrence of

updates to the edge weights.

If the new total travel time of the current path is greater than a predefined threshold, the central computer will recalculate the shortest path using the current vertex in the vehicle's path as the source $s$. If this new path is better than the vehicle's current path by another threshold then it will be sent to the car. Otherwise, the system will continue to monitor the current path's cost.

# 6 Issues

## 6.1 Incomplete Travel Time Data

One issue with the travel time data is that the travel times for the edges will be incomplete, i.e., there will be edges $e$ in the roadmap that will not have a travel time weight $w(e, t)$ for a particular time $t$ that the traveler will be on the path. In those instances one could obtain an estimate of the travel time as follows:

1. Rely on the speed limit for that road or a speed chosen for the type of road.

2. Rely on historical data, possibly an average travel time for that edge at around the same time based on the day of week. Holidays may need to be excluded. For example, the travel time for a road during rush hour on a normal Monday would be much different than the travel time for the same road at the same time on a Monday if it is Christmas.

## 6.2 Partial Paths and the User

Algorithm 2 does not compute one shortest path ahead of time but it repeatedly computes partial paths within update intervals. The user may not appreciate the uncertainty that the remainder of the path has not yet been computed. The algorithm may need to project a total path using its best guess and only change it if necessary.

# 7 Conclusions and Future Research

We presented two schemes for computing shortest paths in a doubly dynamic setting where the edge weights (1) correspond to travel times that are themselves parameterized by time and (2) are subject to possibly many updates.

The central server in the central computaton architecture potentially has a lot of computing power. Therefore the central server could repeatedly search for alternate shortest paths along the vehicle's current path to identify any shorter paths based on the latest travel time information. Since this is of course very expensive it would be interesting to investigate trade-offs between frequency of shortest path computation and the quality of the computed route, as well as impact on the server under multiple shortest path queries from different vehicles.

In general, the central server approach seems to provide more robust solutions than the local approach with updates. However, it may not be feasible to put all the computation load from possibly many shortest path queries onto the server. In order to avoid dead-ends, a different local approach might precompute one shortest path in the beginning and then receive few edge weight updates from the server and either update the path locally or update a precomputed shortest path data structure as in [3, 6, 11, 5, 10, 9].

The update time parameter in Algorithm 1 has currently arbitrarily been set to $m = 15$. A shorter or longer update may be more useful based on the edge weights in the graph or on the bandwidth of the connection to the vehicle's onboard computer. Similarly, the parameter $k$ in Algorithm 2 is arbitrary and it would be interesting to investigate how to choose a good value for it – possibly based on the topology of the graph and on the type of heuristic function $h$.

The constraint of no passing that is needed to apply the simple Dijkstra's (or A*) algorithm (Algorithm 1) to a graph with dynamic edge weights is most likely not going to completely hold for this application. It would be interesting to quantify the error that may occur when using Algorithm 1 on a road network that has a small number of edges that do not satisfy the non-passing condition (1).

# References


[1] R.K. Ahuja, J.B. Orlin, S. Pallottino, and M.G. Scutellà. Dynamic shortest paths minimizing travel times and costs. *Networks*, 41(4):197–205, 2003.

[2] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *Proc. 131st VLDB Conference*, pages 853–864, 2005.

[3] C. Demetrescu and G.F. Italiano. Algorithmic techniques for maintaining shortest routes in dynamic networks. *Electronic Notes in Theoretical Computer Science*, 171:3–15, 2007.

[4] Liang Deng and Martin D.F. Wong. An exact algorithm for the statistical shortest path problem. In *Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 965–970. ACM Press, 2006.

[5] H.N. Djidjev, G.E. Pantziou, and C.D. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, 28:367–389, 2000.

[6] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem (extended abstract). In *SODA: 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 212 – 221, 1996.

[7] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics*, SSC-4(2):100–107, 1968.

[8] David E. Kaufman and Robert L. Smith. Fastest paths in time dependent networks for intelligent vehicle-highway systems application. *IVHS Journal*, 1(1):1–11, 1993.

[9] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge length. *Journal of the ACM*, 37:607–625, 1990.

[10] A. Orda and R. Rom. Minimum weight paths in time-dependent networks. *Networks*, 21:295–320, 1991.

[11] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proc. of 12th. ESA*, pages 580–591, 2004.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.