Fast paths on dynamic road networks

G.Nannicini[1,2], Ph. Baptiste[1], D. Krob[1], and L. Liberti[1]

[1] LIX, École Polytechnique, F-91128 Palaiseau, France
{giacomon,baptiste,dk,liberti}@lix.polytechnique.fr
[2] Mediamobile, 10 rue d'Oradour sur Glane, Paris, France
giacomo.nannicini@v-trafic.com

Abstract. A recent industrial challenge for traffic information providers is to be able to compute point-to-point shortest paths very efficiently on road networks involving millions of nodes and arcs, where the arc costs represent travelling times that are updated every few minutes when new traffic information is available. Such stringent constraints defy classic shortest path algorithms. In this paper we review some existing methods that address this scenario and propose a new Polynomial-Time Approximation Scheme heuristic.

Mots-Clefs. Shortest Paths; PTAS; Dynamic Road Networks.

1 Introduction

A current problem faced by traffic information providers is that of offering GPS terminal enabled drivers a source-destination path subject to the following constraints: (a) the path should be fast in terms of travelling time; (b) the travelling times (weights on the edges) vary according to traffic information being available on part of the road network; (c) the graph topology is fixed; (d) traffic information data are updated at regular time intervals; (e) answers to path queries should be computed in real time. Given these constraints, point-to-point SPP (PPSPP) algorithms defined on static graphs are only useful as long as the computation speed of a single point-to-point shortest path is much faster than the edge weight update rate. In practical terms, the solution run of a PPSPP algorithm must be of a few milliseconds in graphs with several million nodes.

In this paper we briefly survey some of the most relevant results for finding PPSPs in dynamic graphs (Sect. 2) and we propose a new Polynomial-Time Approximation Scheme (PTAS) heuristic which performs well in practice (Sect. 3). Our algorithm is based on Dijkstra-type searches performed on clusters of nodes; such clusters are defined on such a way that we are able to give a bound on the solution performance, but small enough to accelerate the search enough to be practically useful within the given time constraints.

2 Literature review

Consider a weighted directed graph $G = (V, A, c)$ (where $c : A \to \mathbb{R}_+$) which represents a road network evaluated by travelling times (so the graph may not be Euclidean). Assuming that c changes every few minutes, that the cardinality of V is very large (several million vertices), and that each shortest path request must be answered in a few seconds, current technology does not allow us to find an exact optimum [5] in the brief time window before the next change of the weight function (if one removes the constraints on c changing every few minutes, some practically efficient algorithms are [18,10]). We assume some lower and upper bounding functions $\lambda, \mu : A \to \mathbb{R}_+$ for c are known. Some interesting results in computing shortest paths satisfying various robustness requirements are given in [16,17,22]. None of these, however, yields algorithms that can answer point-to-point shortest path requests within little time in very large graphs.

2.1 Early history

The first citation we could find concerning the SPP on dynamic graphs with time-dependent edge weight changes is [2] (a good review of this paper can be found in [6], p. 407): Dijkstra's algorithm [5] is extended to the dynamic case through a recursion formula based on the assumption that the network $G = (V, A)$ has the FIFO property: for each pair of time instants t, t' with $t < t'$:

$$\forall\ (u, v) \in A\ \tau_{uv}(t) + t \le \tau_{uv}(t') + t',$$

where $\tau_{uv}(t)$ is the travelling time on the arc (u, v) starting from u at time t. The FIFO property is also called the *non-overtaking property*, because it basically says that if A leaves u at time t and B at time $t' > t$, B cannot arrive at v before A using the arc (u, v).

Although FIFO networks are useful for the study of those means of transportation where overtaking is rare (such as trains), modelling of car transportation yields networks which do not necessarily have the FIFO property. For the SPP on general transportation networks, early studies dealt with a specific type of fastest paths (whose length is called *interzonal travelling times*) defined as shortest paths between centroids of node clusters roughly corresponding to a graph partition minimizing the number of inter-cluster links (typically, these are the links corresponding to cutsets of minimum size, and hence those most likely to be congested) [12]. Two cases were dealt separately: the shortening [15] and the lengthening [11] of an edge, with the latter being more difficult to treat [12]. In the case of increasing edge weight (i.e. longer travelling time associated with an edge), a partial solution was given for the cases where two node clusters i, j were separated by a congested cutset C: the travelling time t_{ij} is then defined as $\min\limits_{\{p,q\} \in C} (t_{ip} + t_{pq} + t_{qj})$ [11].

2.2 Dijkstra's algorithm: uni- and bi-directional

Dijkstra's algorithm (see [5]) solves the single source shortest path problem in static directed graphs with non-negative weights in polynomial time; it also solves the problem in the presence of negative weights, but it may require exponential time in the worst case. Dijkstra's algorithm is a so-called labeling method.

The *labeling method* for the SPP [7] finds shortest paths from the source to all vertices in the graph; the method works as follows: for every vertex v it maintains its distance label $d(v)$, parent node $p(v)$, and status $S(v) = \{\texttt{unreached}, \texttt{explored}, \texttt{settled}\}$. Initially $d(v) = \infty$, $p(v) = NIL$, and $S(v) = \texttt{unreached}$ for every vertex v. The method starts by setting $d(s) = 0$ and $S(s) = \texttt{explored}$; while there are labeled (i.e. explored) vertices, the method picks an `explored` vertex v, relaxes all outgoing arcs of v, and sets $S(v) = \texttt{settled}$. To relax an arc (v, w), one checks if $d(w) > d(v) + c(v, w)$ and, if true, sets $d(w) = d(v) + c(v, w)$, $p(w) = v$, and $S(w) = \texttt{explored}$. If the length function is non-negative, the labeling method terminates with correct shortest path distances and a shortest path tree; its efficiency depends on the rule to choose a vertex to scan next. We say that $d(v)$ is exact if it is equal to the distance from s to v; it is easy to see that if one always selects a vertex v such that, at the selection time, $d(v)$ is exact, then each vertex is scanned at most once. Dijkstra [5] observed that if the cost function c is non-negative and v is an explored vertex with the smallest distance label, then $d(v)$ is exact; so, we refer to the labeling method with the minimum label selection rule as Dijkstra's algorithm. If c is non-negative then Dijkstra's algorithm scans vertices in nondecreasing order of distance from s and scans each vertex at most once; for the point-to-point SPP, we can terminate the labeling method as soon as the target node is `settled`. The algorithm requires $O(m + n \log n)$ time if the queue is implemented as an heap data structure such as binary heaps or Fibonacci heaps [8].

One basic variant of Dijkstra's algorithm for the point-to-point SPP is bidirectional search; instead of building only one shortest path tree rooted at source node s, we also build a shortest path tree rooted at target node t on the reverse graph $\bar{G} : (V, \bar{A})$ where $(u, v) \in \bar{A} \Leftrightarrow (v, u) \in A$. As soon as one node v becomes `settled` in both searches, we are guaranteed that the concatenation of the shortest $s \to v$ path found in the forward search and of the shortest $v \to t$ path found in the backward search is a shortest $s \to t$ path. Since we can think of Dijkstra's algorithm as exploring nodes in circles centered at s with increasing radius until t is reached (see Fig. 1), the bidirectional variant is faster because it explores nodes in two circles centered at both s and t, until the two circles meet (see Fig. 2); the area within the two circles, which represents the number of explored nodes, will then be smaller than in the unidirectional case, up to a factor of two.

We note here that all speed-up techniques based on finding shortest paths in Euclidean graphs [21] cannot be applied either, since the typical arc cost function, the arc travelling time at a certain time of the day, does not yield a Euclidean graph.

2.3 Dynamic Node Routing

Separator-based multi-level methods for the SPP have been used by many authors; we refer to [13] for the basic variant. The main idea behind separator-based methods is to define, given a subset of the vertex set $V' \subset V$, the shortest path overlay graph $G' = (V', A')$ with the property that A' is a minimal set of edges

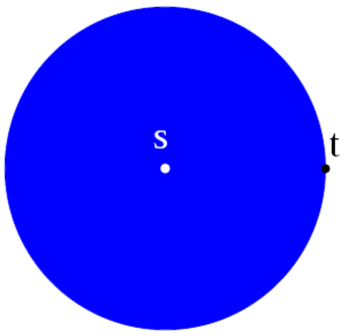

Fig. 1. Schematic representation of Dijkstra's algorithm search space

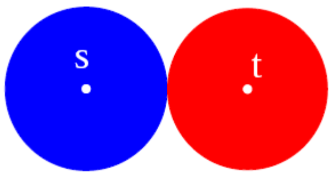

Fig. 2. Schematic representation of bidirectional Dijkstra's algorithm search space.

such that $\forall u, v \in V'$ the shortest path length between u and v in G' is equal to the shortest path length between u and v in G. Usually, the set of separator nodes V' is chosen in such a way that the subgraph induced by $V \setminus V'$ consists of small components of similar size. In a bidirectional query algorithm, the components containing source and target node are wholly searched, but starting from the separator nodes only edges of the overlay graph G' are considered. This approach can be generalized and applied in a hierarchical way, building several levels of overlay graphs with node sets $V = V_0 \supseteq V_1 \supseteq \cdots \supseteq V_L$ so that the following property is mantained: $\forall \ell \leq L - 1$, for all node pairs $s, t \in V_\ell$ the part of the shortest path between s and t that lies outside the level ℓ components to which s and t belong is entirely included in the level $\ell + 1$ overlay graph.

In [20], an arbitrary subset $V' = V'(V)$ of V is considered instead of separator nodes; in practice, the set is chosen in such a way that it contains the most important nodes, i.e. those that appear "more often" on shortest paths. This yields a smaller set V', more uniformly distributed over the whole graph, and thus G' will be smaller, resulting in a smaller space consumption and a faster query algorithm. However, since in this case $V \setminus V'$ is no longer made of small isolated components, the query algorithm is not as simple as in canonical separator-based methods. From a theoretical point of view the same principle holds: we might want to explore nodes from source and target until the queue in

Dijkstra's algorithm only contains nodes that are covered by V' (i.e. there is at least one node $v \in V'$ on the shortest path from the root to any leaf of the current partial shortest path tree), and then switch to the overlay graph G', or to a higher level in the overlay graph hierarchy in the case of a multi-level approach. This, however, does not yield good results in practice, because we cannot tell in advance how many nodes we will have to explore until the whole partial shortest path tree is covered by V'. The main challenge is therefore to compute the set of all covering nodes for the partial shortest path tree T rooted at s as quickly as possible.

Many possible strategies are suggested in [20], including an aggressive variant which stops the search whenever a node in V' is encountered, and which yields a superset of the covering nodes. Another cited technique is "Stall-on-Demand" (see Figure 3), which works as follows: the search from a node $u \in V'$ is stopped as soon as the node is settled. However, if such a node u is reached at some time via another path, then it is *woken up* and a breadth-first search is started from that node in order to *stall* all nodes v for which it can be proven that the tentative path found so far is suboptimal — and this is certainly the case if the shortest path from s to v passes through u. The Dijkstra search is then pruned at stalled nodes. Once the set of all covering nodes for a given level of the overlay graph has been computed, the search can switch to the next level, until the shortest path is found (this is guaranteed to happen at the topmost level). The choice of level node sets $V = V_0, V_1, \ldots, V_L$, where $V_i = V'(V_{i-1})$ for all $i > 0$, is critical for query times. The Highway Hierarchies algorithm [19] is employed in [20].

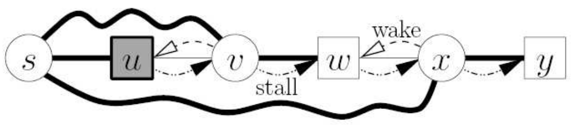

Fig. 3. Example of the Stall-on-Demand technique (see Ex. 21).

21 Example
Consider the following example of the Stall-on-Demand technique applied to the case illustrated in Fig. 3. All edges have cost 1 *except* (s, v) *and* (s, x) *with cost 10; square nodes belong to* V'*, and bold edges belong to the search tree* T *rooted at* s*. We assume that the arc* (v, w) *is relaxed before* (v, u)*; when* u *is woken up by* v *it stalls* v *and* w*, and when* w *is woken up by* x *it stalls* x *and* y*. Only* u *is then returned as covering node.*

The main advantage of this approach is that overlay graphs can be computed in a very short time; besides, if a few arc costs change there is no need to recompute the whole overlay graphs, but only a small part of them — the part which is affected by the change. Certainly, if the changed arc does not belong to the partial shortest path tree of a given node, the construction phase from that node need not be repeated. In particular, during the pre-processing phase we build for each node v a list of all nodes that can be affected if the cost of one of the outgoing arcs from v changes. The construction phase is repeated only when necessary; the authors of [20] claim that the update process takes on average up to a few dozen milliseconds for each arc cost change. After the update step the bidirectional query algorithm will correctly compute shortest paths. The total speed-up, with respect to a "pure" bidirectional Dijkstra's algorithm, is of about three orders of magnitude.

2.4 A^* for dynamic scenarios

Goal directed search, also called A^*, is a search technique which is similar to Dijkstra's algorithm, but which adds a potential function to the priority key of each node in the queue. This function applied on a node v should be an estimate of the distance to reach the target from v; A^* then works exactly as Dijkstra's algorithm, but the use of a potential function has the effect of giving priority to nodes that are (supposedly) closer to target node t. If the potential function π is such that $\pi(v) \leq d(v,t)\,\forall v \in V$, where $d(v,t)$ is the distance from v to t, then A^* always finds shortest paths. A^* is guaranteed to explore no more nodes than Dijkstra's algorithm: if $\pi(v)$ is a good approximation from below of the distance to target, A^* efficiently drives the search towards the destination node, i.e. the search space is not a circle centered at s, bu an ellipse directed towards t (see Fig. 4); if $\pi(v) = 0\,\forall v \in V$, A^* behaves exactly like Dijkstra's algorithm, ie. it explores the same nodes. In [14] it is shown that A^* is equivalent to Dijkstra's algorithm on a graph with reduced costs, i.e. $c_\pi(u,v) = c(u,v) - \pi(u) + \pi(v)$. A^* was first applied in a time-dependent scenario with the FIFO property in [1]; a much more efficient version, presented in [3], makes use of landmarks to compute the potential function.

Fig. 4. Schematic representation of A^* algorithm search space

Landmarks have first been proposed in [9]; they are a preprocessing technique which is based on the triangular inequality. The basic principle is as follows: suppose we have selected a set $L \subset V$ of landmarks, and we have precomputed distances $d(v,\ell), d(\ell,v)\,\forall v \in V, \ell \in L$; the following triangle inequalities hold: $d(u,v) + d(v,\ell) \geq d(u,\ell)$ and $d(\ell,u) + d(u,v) \geq d(\ell,v)$. Therefore

$\pi_t(u) = \max_{\ell \in L}\{d(u,\ell) - d(t,\ell), d(\ell,t) - d(\ell,u)\}$ is a lower bound for the distance $d(u,t)$, and it can be used as a potential function which preserves optimal paths. Bidirectional search can be implemented, using some care in modifying the potential function so that it is consistent for the forward and backward search (see [10]). A^* with the potential function described above is called ALT. It is straightforward to note that, if arc costs can only increase with respect to their original value, the potential function associated with landmarks is still a valid lower bound, and in [3] this idea is applied to a real road network in order to analyse the algorithm's performances.

Two different approaches are considered; the first one is to update the preprocessing information, i.e. update distances to and from landmarks whenever an arc cost changes. Required time for this operation is greatly dependent on the number of updated arcs, their relative importance (urban edge, motorway, etc.) and their position with respect to landmarks, but it is a costly operation if several motorway edges are perturbed. The other possible approach is to compute paths without updating distances to and from landmarks; the algorithm's efficiency decreases with respect to the non-dynamic graph case, depending again on the number and type of perturbed edges. The authors of [3] report that, if 1000 motorway edges out of 42.6 millions edges are perturbed, roughly 95% of queries become slower, but ALT still yields an order of magnitude of speed increase with respect to bidirectional Dijkstra.

3 Polynomial-Time Approximation Scheme

For $s,t \in V$ we denote the set of all paths $(s,\ldots,t)$ from s to t by $P(s,t)$ and the set of all shortest paths from s to t on a graph weighted by function f by $P^*_f(s,t)$; when the weighting function is c, we will omit the subscript, i.e. $P^*(s,t) = P^*_c(s,t)$. Given $U \subseteq V$ such that $s,t \in U$, let $G[U]$ be the subgraph of G induced by U. The set of all paths between s and t in $G[U]$ is denoted by $P[U](s,t)$ and the set of all shortest paths between s and t in $G[U]$ weighted by function f is denoted by $P^*_f[U](s,t)$; as before, we will write $P^*[U](s,t) = P^*_c[U](s,t)$. We naturally extend c to be defined on paths $p = (v_1,\ldots,v_k)$ by $c(p) = \sum_{i=1}^{k-1} c(v_i,v_{i+1})$, and in a similar way for

3.1 Guarantee regions

Let $G_\lambda = (V,A,\lambda)$ and $G_\mu = (V,A,\mu)$ be the graph G weighted by the lower and upper bounding functions λ,μ.

31 Definition
For $K > 1$, $s,t \in V$ and any path $p \in P(s,t)$, we define the guarantee region as:

$$\Gamma_{st}(K,p) = \{v \in V | v \in p \vee \exists\, q \in P(s,t)\ (v \in q \wedge \lambda(q) < \frac{1}{K}\mu(p))\}.$$

Prop. 32 points at a way to compute valid guarantee regions for any given path $p \in P(s,t)$.

32 Proposition
For $K > 1$, $s, t \in V$, $p \in P(s,t)$, $p^* \in P^*(s,t)$ and $r^* \in P^*[\mathit{\Gamma}_{st}(K,p)](s,t)$, we have $c(r^*) \leq Kc(p^*)$.

Proof. Suppose there is a path $q \in P(s,t)$ containing a node $v \notin \mathit{\Gamma}_{st}(K,p)$ such that $c(r^*) > Kc(q)$. For $q^* \in P^*_\mu[\mathit{\Gamma}_{st}(K,p)](s,t)$ We have the chain $\lambda(q) \leq c(q) < \frac{1}{K}c(r^*) \leq \frac{1}{K}c(q^*) \leq \frac{1}{K}\mu(q^*) \leq \frac{1}{K}\mu(p)$, which implies that all nodes of q are in $\mathit{\Gamma}_{st}(K,p)$ by definition: including v, which is a contradiction.

33 Proposition
Let $p^* \in P^*_\mu(s,t)$ be a shortest $s \to t$ path in G_μ, and $p \in P(s,t)$ be another (different) $s \to t$ path. If $p^* \subset \mathit{\Gamma}_{st}(K,p)$ then $\mathit{\Gamma}_{st}(K,p^*) \subseteq \mathit{\Gamma}_{st}(K,p)$.

Proof. By definition, for all $v \in \mathit{\Gamma}_{st}(K,p^*)$ either $v \in p^*$ or there is $q \in P(s,t)$ such that $v \in q$ and $\lambda(q) < \frac{1}{K}\mu(p^*) \leq \frac{1}{K}u(p)$. In the first case $v \in \mathit{\Gamma}_{st}(K,p)$ by hypothesis; in the second case $v \in \mathit{\Gamma}_{st}(K,p)$ by its own definition.

Although the result only holds if $p^* \in P^*_\mu(s,t), p^* \subset \mathit{\Gamma}_{st}(K,p)$, Prop. 33 is useful to characterize the choice of the initial path p (namely, the shortest path in G_μ) that will be used to build a guarantee regions. It is possible to show by counterexample that guarantee regions generated by shortest paths in G_μ are not always minimal. The trouble with the guarantee regions defined above is that, although only a pre-processing step, building all guarantee regions for all node pairs in a very large graph is not a feasible task with current technology. We deal with this problem by covering V with clusters $V_1, \ldots, V_k$.

34 Definition
A covering $V_1, \ldots, V_k$ of V is valid if for all $i \leq k$ there is are two selected (not necessarily distinct) vertices $s_i, t_i \in V_i$ such that for all other vertices $v \in V_i$ there are paths $p \in P(v, s_i), q \in P(t_i, v)$ entirely contained in V_i.

For all $i \leq k$ let $\sigma_i = \max_{v \in V_i, p \in P^*_\mu(v,s_i)} c(p)$ and $\tau_i = \max_{v \in V_i, p \in P^*_\mu(t_i,v)} c(p))$ be the costs of the longest shortest path in G_μ from v to s_i and respectively from t_i to v over all $v \in V_i$.

35 Definition
Given a valid covering $V_1, \ldots, V_k$ of V, for $K > 1$, $i \neq j \leq k$ and any path $p \in P(s_i, t_j)$, we define the guarantee region as:

$$\mathit{\Gamma}_{V_iV_j}(K,p) = \{v \in V | v \in p \cup V_i \cup V_j \vee \exists\, q \in P(s_i,t_j)\ (v \in q \wedge \lambda(q) < \frac{1}{K}(\mu(p) + \sigma_i + \tau_j))\}.$$

We have the following theorem:

36 Theorem
Given a valid covering $V_1, \ldots, V_k$ of V, for $K > 1$, $i \neq j \leq k$, $p \in P(s_i, t_j)$, $p^* \in P^*(u,v)$, and $r^* \in P^*[\mathit{\Gamma}_{V_iV_j}(K,p)](u,v)$ for all $u, v \in V$ we have $c(r^*) \leq Kc(p^*)$.

Proof. Suppose there is a path $q \in P(u,v)$ containing a node $w \notin \Gamma_{V_iV_j}(K,p)$ such that $c(r^*) > Kc(q)$. By definition of G_λ we have $\lambda(q) \leq c(q)$. By definition of q we have $c(q) < \frac{1}{K}c(r^*)$. For $q^* \in P^*_\mu[\Gamma_{V_iV_j}(K,p)](u,v)$ we also have, by definition of G_μ and by optimality of r^*, $c(r^*) \leq c(q^*) \leq \mu(q^*)$. Furthermore, since q^* is shortest, $\mu(q^*) \leq \mu(p) + \sigma_i + \tau_j$, which proves that all the vertices of q, including w, are in $\Gamma_{V_iV_j}(K, p(s_i,t_j))$. By contradiction the result follows.

A result similar to Prop. 33 holds for $\Gamma_{V_iV_j}(K,p^*)$ when $p^* \in P^*_\mu(s_i,t_j)$, and serves as a hint to choose our initial path.

37 Proposition
Given a valid covering $V_1,\ldots,V_k$ of V, for $i \neq j \leq k$ let $p^* \in P^*_\mu(s_i,t_j)$ be a shortest $s_i \to t_j$ path in G_μ, and $p \in P(s_i,t_j)$ be another (different) $s_i \to t_j$ path. If $q^* \subset \Gamma_{V_iV_j}(K,p)$ then $\Gamma_{V_iV_j}(K,p^*) \subseteq \Gamma_{V_iV_j}(K,p)$.

Proof. By definition, for all $v \in \Gamma_{V_iV_j}(K,p^*)$ we have $v \in p^* \vee v \in V_i \vee v \in V_j$ or there is $q \in P(s,t)$ such that $v \in q$ and $l(q) < \frac{1}{K}\mu(q^*) \leq \frac{1}{K}\mu(p)$. In the first case $v \in \Gamma_{V_iV_j}(K,p)$ by hypothesis; in all other cases $v \in \Gamma_{V_iV_j}(K,p)$ by its own definition.

38 Example
We give an example of the fact that $\Gamma_{st}(K,p^*)$ and $\Gamma_{V_iV_j}(K,p^*)$ may fail to have minimal size. We take $s = A, t = E, K = \frac{6}{5}$ in the graph of Fig. 38. Since

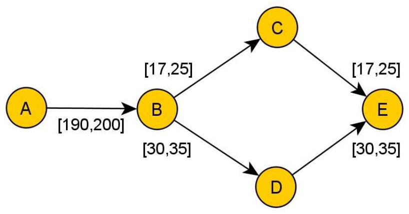

Fig. 5. Nodes A, B, D, E are in set $\Gamma_{AE}(1.2,(ABDE))$.

the shortest path from A to E in G_μ is $p^* = (A,B,D,E)$, those nodes are included in $\Gamma_{AE}(6/5,p^*)$. Furthermore, $\mu(A,B,D,E) = 270$, and that the path $p \in P(A,E), p = (A,B,C,E)$ in G_λ has cost $\lambda(p) = 224 < \frac{5}{6}270 = 225$, hence $\Gamma_{AE}(6/5,p^*) = \{A,B,C,D,E\}$. However, it is easy to see that the set $\Gamma' = \{A,B,D,E\}$ has a smaller size and is enough to guarantee the approximation property.

3.2 Computing the node sets

It is obvious, for the way we have defined $\Gamma_{st}(K,p)$ and $\Gamma_{V_iV_j}(K,p)$ in definition 31 and 35, that the most difficult (and expensive) part during the computation of those sets is computing all paths $p \in P(s,t)|\lambda(p) < H$, where H is a given quantity whose value depends wether we are in the unclusterized or in the clusterized case. Computing H itself is not too difficult, since it requires computing $p^* \in P^*(s,t)$ and, in the clusterized graph, upper bounds to the cost of shortest paths in a small set of nodes. These upper bound can be computed with a slightly modified version of Dijkstra's algorithm: for example, if we want to calculate $\tau_i = \max_{v \in V_i, p \in P^*_\mu(t_i,v)} c(p))$ we just jave to apply Dijkstra on subgraph $G_\mu[V_i]$ with t_i as source node; the cost of the last settled node is our τ_i.

We will now see how we can compute the set of all nodes that belong to a path from a node s to a node t with total cost $< H$, adding some lines to Dijkstra's algorithmm and applying it on the reverse graph (note that, with straightforward changes, the same holds true on the original graph). Let us define the limited-width shortest paths tree $T_{u,L}$ from a given node u of width L as the the shortest paths tree that contains all nodes $v|p^* \in P^*(u,v) \Rightarrow \lambda(p^*) < L$. Given a graph $G = \langle V, A \rangle$ with cost function for edges $\lambda(i,j)$, let us define the reverse graph $\overline{G} = \langle V, \overline{A} \rangle$ where $(i,j) \in \overline{A} \iff (j,i) \in A$ with cost function $\overline{\lambda}(i,j) = \lambda(j,i)$. We will call $l[v]$ Dijkstra's algorithm label of a node $v \in T_{s,L}$ computed on the direct graph, i.e. $l[v] = \lambda(p^*)$ where $p^* \in P^*(s,v)$, and $\overline{l}[v]$ Dijkstra's algorithm label of a node v computed on the reverse graph, i.e. $\overline{l}[v] = \lambda(\overline{p}^*)$ where $\overline{p}^* \in P^*(v,t)$. We assume to set $l[v] = \infty$ if $v \notin T_{s,L}$, and $\overline{l}[v] = \infty \, \forall v \in V$. Algorithm 3.2 computes the desired set.

We can prove its correctness.

39 Proposition
Algorithm 3.2 returns only and all nodes on a path $p \in P(s,t)|\lambda(p) < H$

Proof. First, we note that this algorithm is a modification of Dijkstra's algorithm which simply adds some lines that not inficiate the correctness of Dijkstra's algorithm; there is one additional terminating condition on the main loop: the algorithm stops if it has settled a node u with $\overline{l}[u] > H$. First part: algorithm 3.2 returns all nodes on a path $p \in P(s,t)|\lambda(p) < H$. Suppose there is a node $u \notin S$, $u \in q$ where $q \in P(s,t)$ is such that $\lambda(q) < H$; since there is a path from s to t with cost $< H$, we have that $d(u,t) < H$, and so node u is scanned because the additional terminating condition on the main loop does not apply. Also, we have that $d(s,u) + d(u,t) \leq \lambda(q)$ by optimality. Thus, when the node is scanned, the test on line 11 holds true since $l[u] + \overline{l}[u] = d(s,u) + d(u,t) \leq \lambda(q) < H$, and u is added to S, which is an absurd.

Second part: algorithm 3.2 returns only nodes on a path $p \in P(s,t)|\lambda(p) < H$. Suppose there is a node $u \in S$ such that $\nexists q \in P(s,t)$ such that $\lambda(q) < H, u \in q$. Since $u \in S$, then it has been added on line 12, which means that $l[u] + \overline{l}[u] < H$. In this case, for the way we have defined $\overline{l}[u]$ and $l[u]$, we can concatenate the shortest paths from s to u and from u to t to build a $s \to t$ path with cost $l[u] + \overline{l}[u] < H$. The absurd follows.

Time requirements for Algorithm 3.2 are roughly the same as applying two times Dijkstra's algorithm on the original graph, thus $O(|A| + |V| \log |V|)$ with

Algorithm 1 Find only and all nodes on a path with total cost $< H$ from a node s to a node t

```
1: Build $T_{s,H}$ on graph $G_\lambda$
2: $Q \leftarrow \{t\}$
3: $\bar{l}[t] \leftarrow 0$
4: $S \leftarrow \phi$
5: $E \leftarrow \phi$
6: $stop = \texttt{false}$
7: if $l[t] \neq \infty$ then
8:    while $Q \neq \phi \wedge \neg stop$ do
9:       extract $x \leftarrow \arg\min_{q \in Q}\{\bar{l}[q]\}$
10:      $E \leftarrow x$
11:      if $\bar{l}[x] + l[x] < H$ then
12:         $S \leftarrow S \cup \{x\}$
13:      end if
14:      if $\bar{l}[x] \geq H$ then
15:         $stop = \texttt{true}$
16:      end if
17:      for all arcs $(x, y) \in \overline{A}$ do
18:         if $y \notin E$ then
19:            if $y \notin Q$ then
20:               $\bar{l}[y] \leftarrow \bar{l}[x] + \overline{\lambda}(x, y)$
21:               $Q \leftarrow Q \cup \{y\}$
22:            else if $\bar{l}[x] + \overline{\lambda}(x, y) < \bar{l}[y]$ then
23:               $\bar{l}[y] \leftarrow \bar{l}[x] + \overline{\lambda}(x, y)$
24:            end if
25:         end if
26:      end for
27:   end while
28: end if
29: return $S$
```

Fibonacci's heaps, but each Dijkstra's execution can be stopped whenever we reach a distance of H from source node, so effective running time greatly depends on H, which in turns depends on the choice of K and graph's topology: the higher K, the lower the execution time. Space requirements are linear in $|V|$: in addition to Dijkstra's linear space requirements, we only need to store each node's label in the direct search before applying the reverse search. Note that keeping track of the whole Dijkstra's shortest paths trees is not needed.

3.3 Computational results

We used a subgraph of France's road network, roughly corresponding to Île-de-France (i.e. Paris and surroundings), to validate our approach. This subgraph has roughly 300.000 vertices and 800.000 edges. We ran several bidirectional Dijkstra searches [18] on the full graph and on guarantee regions to assess the usefulness of our heuristic, with source and destination node chosen at random, and for each source-destination pair we repeated the query 5 times with arc costs generated at random with a uniform distribution each time; upper bounds on arc costs are between 5-10 times lower bounds. We recorded solution quality and CPU times in Table 1. For each value of K (first column), we indicate the average number D of nodes explored in bi-directional Dijkstra searches n the full graph, the average number R of nodes explored in bi-directional Dijkstra searches on the guarantee regions, the average percentage increase P of the approximated solution value with respect to the optimum (0% means that the approximated solution is optimal), the average CPU time savings C in percentage of the CPU times taken by the exact algorithm (0% means as slow as the exact algorithm).

K	*D*	*R*	*P*	*C*
3	74559	74532	0%	0%
4	74779	74219	0%	0%
5	74651	65126	0%	8.39%
6	74739	39282	0%	46.85%
7	74647	5609	0.07%	93.86%

Table 1. Computational results on unclustered graph: mean values.

To validate the clustered approach, we generated a valid covering of V, and then, for some random cluster pairs, compared the number of explored and settled nodes between a bidirectional Dijkstra search and a bidirectional Dijkstra search constrained to the guarantee regions, where source and destination node of Dijkstra's search where chosen randomly in their respective cluster, performing 5 queries with arc costs generated at random for each source-destination pair. Results are reported in Table 2 (same column labels as Table 1); cluster size was set to 500 nodes.

K	*D*	*R*	*P*	*C*
6	74493	73262	0%	0%
7	74605	66804	0%	5.83%
8	74129	56761	0%	20.35%
9	74436	34091	0.02%	54.26%
10	74494	13978	1.20%	82.05%

Table 2. Computational results on clustered graph: mean values

4 Conclusion

In this paper we surveyed some of the methods used in computing fast point-to-point shortest paths on dynamic road networks, and proposed a new Polynomial-Time Approximation Scheme heuristic based on limiting a Dijkstra-type search within a set of regions whose definition ensures a desired approximation guarantee.

References

1. I. Chabini and L. Shan. Adaptations of the a^* algorithm for the computation of fastest paths in deterministic discrete-time dynamic networks. *IEEE Transactions on Intelligent Transportation Systems*, 3(1):60–74, 2002.
2. K.L. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14:493–498, 1966.
3. D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In Demetrescu [4].
4. C. Demetrescu, editor. *WEA 2007 — Workshop on Experimental Algorithms*, volume 4525 of *LNCS*, New York, 2007. Springer.
5. E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
6. S.E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.
7. L. R. Ford and D. R. Fulkerson. *Modern Heuristic Techniques for Combinatorial Problems.* Princeton University Press, Princeton, NJ, 1962.
8. M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.
9. A.V. Goldberg and C. Harrelson. Computing the shortest path: a^* meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research, 2003.
10. A.V. Goldberg, H. Kaplan, and R.F. Werneck. Reach for a^*: Efficient point-to-point shortest path algorithms. Technical Report MSR-TR-2005-132, Microsoft Research, 2005.
11. A.K. Halder. The method of competing links. *Transportation Science*, 4:36–, 1970.
12. A.K. Halder. Some new techniques in transportation planning. *Operational Research Quarterly*, 21:267–278, 1970.
13. M. Holzer, F. Schulz, and D. Wagner. Engineering multi-level overlay graphs for shortest-path queries. In *SIAM*, volume 129 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2006.
14. T. Ikeda, M. Tsu, 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 for the IEEE Vehicle Navigation and Information Systems Conference*, pages 291–296, 2004.
15. P.S. Loubal. A network evaluation procedure. *Highway Research Record*, 205:96–109, 1967.
16. R. Montemanni and L. M. Gambardella. An exact algorithm for the robust shortest path problem with interval data. *Computers & Operations Research*, 31:1667–1680, 2004.
17. R. Montemanni, L. M. Gambardella, and A. V. Donati. A branch and bound algorithm for the robust shortest path problem with interval data. *Operations Research Letters*, 32:225–232, 2004.
18. P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In G. Stølting Brodal and S. Leonardi, editors, *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

19. P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA 2006*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.
20. P. Sanders and D. Schultes. Dynamic highway-node routing. In Demetrescu [4], pages 66–79.
21. R. Sedgewick and J. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1(1):31–48, 1986.
22. A. Sengupta and T. K. Pal. Solving the shortest path problem with interval arcs. *Fuzzy Optimization and Decision Making*, 5:71–89, 2006.