Computer Science • 13 (3) 2012 http://dx.doi.org/10.7494/csci.2012.13.3.113

Daniel Aioanei

LAZY SHORTEST PATH COMPUTATION IN DYNAMIC GRAPHS

Abstract *We address the problem of single-source shortest path computation in digraphs with non-negative edge weights subjected to frequent edge weight decreases such that only some shortest paths are requested in-between updates. We optimise a recent semidynamic algorithm for weight decreases previously reported to be the fastest one in various conditions, resulting in important time savings that we demonstrate for the problem of incremental path map construction in user-steered image segmentation. Moreover, we extend the idea of lazy shortest path computation to digraphs subjected to both edge weight increases and decreases, comparing favourably to the fastest recent state-of-the-art algorithm.*

Keywords single-source shortest path, dynamic graph, livewire, active snake, interactive image segmentation

1. Introduction

The problem of shortest path computation has found numerous applications, e.g., in Internet routing protocols [14, 15] and user-steered interactive image segmentation [3, 4, 12, 9, 10].

In today's routing protocols, routers exchange link state information so that each router has a complete description of the network topology in its network area [16]. Link state updates are distributed to all other nodes using flooding. In response to updates the full Shortest Path Tree (SPT) to all other routers is recomputed, usually from scratch using a static algorithm such as Dijkstra's [2]. The SPT recomputing time constitutes a limiting factor for the size of the routing area [13].

In the field of image analysis, graph search techniques have been employed in the problem of finding object boundaries in images. Interactive image segmentation techniques take advantage of the better ability of human operators in object recognition and the superior quality of computer algorithms in object delineation. Possible object boundaries are associated with a cost made up of an external energy (e.g., image gradients) [4] and possibly an internal energy component (e.g., curvature) [9]. According to the livewire technique, the image is represented as a graph with non-negative edge costs and the best boundary between the starting point and the user pointer is selected to be a shortest path in the mentioned graph. For the case of a static search area, important response time improvements have been reported in medical image segmentation by stopping Dijkstra's algorithm as soon as the desired optimal path is known, without computing a full SPT. The computation would then resumed as soon as the user pointer is moved to a vertex with optimal distance at least as high as all the previous pointer positions, since all the shortest paths of lower distance have already been computed [3]. To achieve more effective user control the search area can be restricted instead to the union of all local windows traced by the user, with some fixed or variable window size [10]. In this case the search area is a dynamic graph, and it is desirable to stop the computation as soon as the shortest path to the latest position of the user pointer is known. The previous computation results may then be reused when the user moves the pointer and a new shortest path needs to be computed in the larger search area that also includes the local window of the latest pointer position. The approach is equally applicable to centerline extraction of curvilinear objects, e.g. from microscopy images, if the external energy is based on the eigenvalues of a modified Hessian matrix [11]. The problem can be generally cast as shortest path computation in a graph with frequent multiple edge insertions (i.e., edge weight decreases), such that in-between batch updates a single shortest path is of interest rather than a full SPT.

2. Background

2.1. Related work

Existing algorithms for the computation of single-source shortest paths in dynamic graphs focus on the maintenance of a full SPT or more generally shortest path graph. Herein we shortly review the most recent ones that apply to graphs with non-negative edge costs.

FMN is a dynamic algorithm [8] that tries to visit a smaller number of edges than *Dijkstra*, and it has been outperformed [7] by another dynamic algorithm named *DynamicSWSF-FP* [17].

DynamicSWSF-FP has been recently optimised and endowed with the ability to compute an SPT, resulting in the algorithm *MFP* [1]. Experiments have been reported in the same paper showing that *MFP* is outperformed, sometimes by a large margin, by combining semidynamic algorithms as described next.

BallStringInc [15], as corrected in [1] and *BallStringDec* [15] are two semidynamic algorithms that reflect how balls attached to elastic strings, admittedly asymmetric, naturally rearrange themselves when the length of some of the strings is increased or decreased, respectively.

Finally, *DynDijkInc* and *DynDijkDec* [1] are two simple semidynamic algorithms with very good performance. *DynDijkInc* was found to be the second best performing algorithm for the relatively sparse road system graphs of Connecticut, closely following after *BallStringInc*, and the best one for the random graphs with quasi-power-law vertex degrees. *DynDijkDec* was the best performing algorithm for both types of graphs studied in [1].

The combination of *BallStringInc* and *DynDijkDec* was the best performer for road system graphs under mixed edge updates, closely followed by the combination of *DynDijkDec* and *DynDijkInc* (in arbitrary order), which was the best performer instead for random graphs. It should be noted however that the algorithm *DynDijkDec* was anticipated by the algorithm *EnhancedLane* [10] in the field of interactive image segmentation.

Still, it is well accepted that static *Dijkstra* outperforms all non-static algorithms above certain thresholds of changed edges, after which the non-static algorithms degrade rapidly.

2.2. Contributions

For routing networks we are interested in the performance of SPT maintenance under mixed edge updates. Topological stability, i.e., avoiding substitution of a shortest path for another one of equal cost, is also of interest. For image analysis, interactive image segmentation employs a graph subjected to frequent batch edge insertions, i.e., decreases of edge costs from infinity to a non-negative value, intermixed with shortest path computations, and we are interested in the performance of the combination of these two types of operations.

First, we create *LazyDijkDec* by endowing *DynDijkDec* with the ability to stop SPT computation as soon as the desired shortest path is found, in such a way that new edge weight decreases can be immediately incorporated and then new shortest paths can be computed, all while reusing the (unfinished) path tree computation done before new updates are received. *LazyDijkDec* efficiently accommodates the continuous cycle where each iteration consists of a batch of edge insertions followed by exactly one shortest path request, as applicable to the interactive image segmentation domain. In fact *LazyDijkDec* only brings minor changes to *DynDijkDec*, and our contribution is to prove that intermixing early-stopped shortest path computations with batch decreases is indeed a valid approach. The correctness of *DynDijkDec* follows as a special case.

Second, we create *LazyDijkInc* by recasting *DynDijkInc* as a set of edge weight decreases in an altered graph, after which other edge weight decreases can be incorporated via *LazyDijkDec* without the need to compute an intermediate SPT between increases and decreases. However *LazyDijkInc* does require an SPT as the starting point, which can be obtained e.g. by processing the queue of *LazyDijkDec* until empty. In fact *LazyDijkInc* only brings minor changes to *DynDijkInc*, and we prove that such an approach is indeed valid. The correctness of *DynDijkInc* follows as a special case.

Our interactive image segmentation experiments show that *LazyDijkDec* brings important speedups in incremental path map construction as compared to *DynDijkDec* (or equivalently, *EnhancedLane*).

Our experiments on random graphs with quasi-power law distribution (like the Internet [5]) show that under mixed edge weight changes *LazyDijkInc* followed by *LazyDijkDec* and then SPT computation (hereafter called *LazyDynDijk*) is faster than the fastest previously reported approach in similar conditions, namely the chaining of *DynDijkInc* and *DynDijkDec* (hereafter called *DynDijkstra*), in any order. The performance differences are significant however only for larger sets of updates.

Finally, our experiments on a larger real-world sparse road system network, namely the National Highway Planning Network of the USA [6] confirm the performance advantage of *LazyDynDijk* over *DynDijkstra*. Surprisingly for a non-static algorithm, for large sets of updates *LazyDynDijk* mirrors more closely the performance of the faster static Dijkstra algorithm than that of the non-static algorithm *DynDijkstra* that it extends.

3. Preliminary

3.1. Generalities

Let $G = (V, E, w)$ be a simple digraph, where V and E are the sets of vertices and edges, respectively, and $w\colon V \times V \to R^+ \cup \{0, \infty\}$ assigns to each edge in E a finite, non-negative real number, that we call the *weight* of the edge, and to each ordered pair of vertices that is not an edge, the value ∞. The semantics of ∞ are the usual ones of floating-point operations. Given an edge $e = (u, v) \in E$, we call u and v the *tail* $u = t(e)$ and the *head* $v = h(e)$ of the edge, respectively. For a vertex $u \in V$, the set of *outgoing edges* of u is $Out_u = \{e | e \in E \text{ and } t(e) = u\}$. We call *path tree* any subgraph T_s of G with tree structure rooted at a unique, fixed vertex $s \in V$ that we call the *source*. The set of ancestors $anc(v, T)$ where T is a tree rooted at the source vertex s is defined as empty if $v \notin T$, and as the set of all nodes on the unique path from the root s to v in T, including v, otherwise. T can be for example the path tree T_s or an SPT.

The insertion of an edge e is treated as decreasing the value of $w(t(e), h(e))$ from ∞ to a finite non-negative value and adding the edge to set E. The deletion of edge e is treated as increasing the value of $w(t(e), h(e))$ from a finite non-negative value to ∞, and removing e from set E.

A vertex v is *reachable* if there is at least one path from the source vertex s to v in G. The *length* of a path from the source s to a vertex v is defined as the sum of the weights of the edges it contains. The *optimal distance* $d(v, G)$ of a vertex $v \in V$ is defined as the *length* of a shortest path from the source s to v if v is reachable from s, and ∞ otherwise.

3.2. Data structures

The state of the algorithms consists in a priority queue Q and a tree data structure T_s having nodes corresponding to some graph vertices.

The root of the tree T_s is the source vertex s and the tree associates a non-negative finite cost to each node. We define a function $c(v, T_s)$ equal to the finite cost associated to the vertex $v \in G$ when $v \in T_s$, and ∞ if $v \notin T_s$. The function $p(v, T_s)$ denotes the tree parent of v, and it is undefined if $v \notin T_s$. Tree manipulation is represented as assignments to $p(v, T_s)$ and $c(v, T_s)$.

The priority queue Q stores tuples $\langle key, value \rangle$ where *key* is a finite non-negative real number and *value* is a vertex from V. Duplicate keys are allowed, but duplicate values are not (i.e., each vertex can be present at most once in the queue at any time). The operation $Min(Q)$ retrieves the minimum *key* from the queue if not empty, and it is ∞ otherwise. $ExtractMin(Q)$ retrieves and removes a minimum key entry from the queue, breaking ties arbitrarily when multiple such entries exist. $DecreaseKey(value, key, Q)$ decreases the key of a value already present in the queue but associated to a key which compares higher than or equal to the new key, or inserts a new entry if the value was not previously present in the queue.

3.3. Invariants

We call a vertex $v \in V$ *correct* with respect to graph $G = (V, E, w)$ and path tree T_s if $c(v, T_s) = d(v, G)$. If $c(v, T_s) > d(v, G)$ the vertex is called *overestimated*. Otherwise the vertex is called *underestimated*.

Informally, the algorithms presented here maintain a path tree T_s with associated costs which contains a subset of the vertices reachable from the source vertex s in G. The costs associated by T_s to vertices are always greater than (i.e., *overestimated*) or equal to their optimal distance (i.e., *correct*). On any path in T_s from s to a leaf node one encounters, in order, a set of *correct* vertices, optionally followed by some *overestimated* vertices. In other words, the "upper part" of T_s consists of only *correct* vertices, while the "lower part" (if any), consists only of *overestimated* vertices. The proofs (but not the algorithms) also maintain an SPT S_s of G with the property that every *correct* vertex from T_s has the same parent in T_s as in S_s. In other words, the "upper part" of S_s is identical to the "upper part" of T_s. Besides, the algorithms also maintain a priority queue Q which contains a subset of the graph vertices together with the associated cost in T_s. Importantly, Q is maintained such that it contains all the *correct* vertices from T_s that have at least one child vertex in S_s that is *overestimated* by T_s (either not present in T_s, or present in T_s with too high a cost). Note that Q can also contain other vertices besides the mentioned "border" vertices. With such a structure, it turns out that for any vertex v whose cost is less than or equal to the cost of all vertices in Q, it holds that v is *correct*, while the path from s to v in T_s (which is identical to the path from s to v in S_s) indicates a shortest path in G from s to v. The existence of S_s in the initial condition is trivial, and after that at each step its existence is proved by construction, by showing how it can be maintained and modified in parallel to any changes applied to the data structures G, T_s and Q so as to make the invariants hold after each operation.

More formally, the following invariants are crucial to our developments:

Inv 1. For any edge e in the path tree T_s, it holds that $c(h(e), T_s) \geq c(t(e), T_s) + w(e)$.
Inv 2. There exists an SPT S_s of G such that the following hold:

- **Inv 2a.** Every *correct* vertex v reachable in G from s, other than the source s, has the same parent vertex in T_s as in S_s.
- **Inv 2b.** Every *overestimated* vertex has a parent in S_s (i.e., it is not the source s) and its parent in S_s is either *correct* and present in the queue Q, or it is *overestimated*.

Several observations are due about the invariants above. First, Inv 1 guarantees that we never leave *underestimated* vertices in the tree, and it immediately follows that all descendants in T_s of an *overestimated* vertex are also *overestimated*, recursively. Second, since $c(v, T_s)$ is finite iff $v \in T_s$ by definition, it is valid to implicitly assume in Inv 2a that a reachable *correct* vertex is itself present in T_s. Third, it is valid to implicitly assume in Inv 2b that an *overestimated* vertex is present in S_s, i.e., it is reachable in G from s, since it is not possible to *overestimate* an unreachable vertex because its optimal distance is ∞. Fourth, it follows from Inv 1 and Inv 2a that all vertices on the path in T_s from the root to a *correct* vertex are also *correct* and the same path is also present in S_s. Fifth, the *overestimated* vertices form full subtrees in S_s. Indeed, that a correct vertex cannot have an *overestimated* ancestor in S_s follows immediately from our previous observation. Sixth, since every reachable vertex has at least one ancestor in S_s that is *correct*, e.g. the *source*, it follows from Inv 2b that if the queue Q is empty then all vertices are *correct*. Seventh, for each *overestimated* vertex v it holds that $c(v, T_s) > Min(Q)$. Indeed, $c(v, T_s) > d(v, G)$ by virtue of v

being *overestimated* and $d(v, G)$ is greater than or equal to the cost of its first *correct* ancestor in S_s, which is present in Q according to Inv 2b.

Several properties follow trivially from the operation of the algorithms. The root of T_s is always the source vertex s and it always has the optimal distance of 0. Also, for each vertex v, if it is found in Q then it is associated to $c(v, T_s)$ as the key, and the key is finite. Finally, it is implied that the data structure T_s remains cycle-free throughout the operation of the algorithms. This property will be explicitly dealt with whenever structural changes on T_s are performed.

If T_s is a path tree of graph G, and together with the queue Q, they respect the above invariants, we say that the state of the algorithm consisting of T_s and Q is *compatible* with G.

4. Algorithms

We assume that all updates are applied to the graph separately and that the latest updates are visible in the graph when the algorithms request the Out_u set for a vertex $u \in V$. In fact, this is the only way the algorithms interact with the underlying graph. Procedures operating on a set of changed edges do a single pass through the set of changed edges and only access Out_u for any vertex u after the iteration procedure finishes, providing enough leeway for the time and modality in which the underlying graph applies the changes. We also assume, without reducing generality, that the set of vertices $|V|$ is fixed.

In the pseudocode we use the *prime(')* notation to indicate the updated value of w and G. In the proofs we use the *prime(')* notation to indicate the latest value of whichever entity it is applied to.

4.1. Relaxation

The basic building block is that of *relaxation* of an edge, which is a common concept in shortest path algorithms. Let $G = (V, E, w)$ and $G' = (V, E', w')$ be two digraphs such that G' can have an extra edge in comparison with G, or otherwise it has the same edges and for at most one edge it can have a different weight, and that is a lower weight. Formally, there is an edge $(u, v) \in E'$ such that $w(u, v) \geq w'(u, v)$ and for all $(a, b) \in V \times V$ s.t. $a \neq u$ or $b \neq v$, it holds that $w(a, b) = w'(a, b)$. We define the operation of *relaxation* as in Alg. 1.

Proof 1 (Alg. 1 restores Inv 1 and Inv 2 with respect to G') *Here we prove that the postcondition of* Relax *holds at the end. Inv 1 holds trivially. Let S_s by an SPT of G with respect to which Inv 2 holds as required by the precondition. We identify a few possible cases:*

1. *$d'(v, G') = d(v, G)$. Then S_s is a valid SPT in G' and $d'(a, G') = d(a, G), \forall a \in V$. Two cases arise:*
   - *(a) Test in line 2 fails. Then nothing changed in the state of the algorithm and Inv 2 holds with respect to $T'_s = T_s$ and $S'_s = S_s$ as before.*

(b) *Test in line 2 passes. Inv 1 of the precondition implies that* $v \notin anc(u, T_s)$ *so no cycles are introduced in the path tree in line 3.*

   i. v overestimated *in* G' *at the end. Then* $c(v, T_s) > c'(v, T_s') > d'(v, G') = d(v, G)$ *so* v *was also* overestimated *in* G *at the beginning. Then no vertices changed correctness status, and no* correct *vertices in* G' *changed parent in the path tree. So Inv 2 holds with respect to* T_s' *and* $S_s' = S_s$ *as before.*

   ii. v correct *in* G' *at the end. Since* u *satisfies* v*'s optimal distance in* T_s'*,* u *must also be* correct *in* G'*, and thus in* G*. Based on Inv 2a,* $anc(u, T_s) = anc(u, S_s)$*. It follows that* $v \notin anc(u, S_s)$ *so we can define the SPT* S_s' *by starting from* S_s *and making the parent of* v *to be* u*, no matter which parent* v *had in* S_s*. Since the only vertex that may change either correctness status or parent in* T_s' *compared to* T_s *is* v*, Inv 2a holds with respect to* T_s' *and* S_s'*. Because* $v \in Q'$ *thanks to line 5, Inv 2b also holds with respect to* T_s' *and* S_s'*.*

2. $d'(v, G') < d(v, G)$*. It is clear that* v *is reachable in* G' *and* u *must be a parent of* v *in any SPT of* G'*. It should also be noted that for any vertex* $a \in V$ *such that* $d'(a, G') < d(a, G)$*, the inequality* $d'(b, G') < d(b, G)$ *also holds for all vertices* $b \in des(a, S_s)$*. In other words, the vertices that have their optimal distance strictly decreased form full subtrees of* S_s*. Furthermore, all the shortest paths in* G' *from the root to vertices that change optimal distance go through* (u, v)*. Let* S_s' *be an SPT of* G' *that encodes the same optimal paths as in* S_s *from the root to vertices that do not change optimal distance. Such an SPT can be constructed for example by starting from any SPT* S_s^* *of* G'*, and then for each vertex whose optimal path in* S_s *is still an optimal path in* G'*, give it back its path from* S_s*. Procedurally, the parent links of* S_s^* *can be changed to match those of* S_s *if the cost stays optimal, while traversing* S_s *depth-first, visiting the children in* S_s *of a vertex* v *only if* v*'s parent in* S_s^* *matches, or was made to match,* v*'s parent in* S_s*, therefore avoiding cycles of zero-weight edges. We'll show that Inv 2 holds with respect to* T_s' *and* S_s' *thus constructed.*
   All vertices outside the subtree rooted at v *in* S_s' *keep in* T_s' *and* S_s' *the same correctness status and parent as in* T_s *and* S_s*, thus according to the precondition they do not break Inv 2. In the subtree of* S_s' *rooted at* v*, all vertices other than* v *are* overestimated *in* T_s'*. Thus we only have to show that when* v *is referenced in Inv 2, it does not break it.*

   (a) *For Inv 2a we need to show that if* v correct *in* G' *then* $p'(v, T_s') = u$*. Indeed,* u *is the only vertex in* V *that can satisfy the optimal distance of* v *in* G' *since otherwise* $d'(v, G')$ *would be equal to* $d(v, G)$*. It means that the test on line 2 passed and therefore* $p'(v, T_s') = u$*.*

   (b) *For Inv 2b, it is sufficient to show that:*

      i. *If* v *is* correct *in* G' *then* $v \in Q'$*. Indeed, if* v correct *in* G' *it means the test on line 2 passed and line 5 ensures* $v \in Q'$*.*

ii. *If v* overestimated *in G' and u* correct *in G' then $u \in Q'$. We show that this situation is not even possible. Indeed, $c(u,T_s)+w'(u,v) = c'(u,T'_s)+w'(u,v) = d'(u,G')+w'(u,v) = d'(v,G') < d(v,G) \leq c(v,T_s)$, so the test on line 2 must have passed and as a result $c'(v,T'_s) = c(u,T_s)+w'(u,v) = d'(v,G')$, so v is* correct *in G', thus the contradiction.*

Algorithm 1 *Relax*

Require:

path tree T_s rooted at $s \in V$, queue Q *compatible* with $G = (V,E,w)$;

$(u,v) \in V \times V$, $weight = w'(u,v) \leq w(u,v)$

Ensure: T_s, Q *compatible* with $G' = (V,E',w')$

1: $t \leftarrow c(u,T_s) + weight$
2: **if** $t < c(v,T_s)$ **then**
3: $\quad p(v,T_s) \leftarrow u$
4: $\quad c(v,T_s) \leftarrow t$
5: $\quad DecreaseKey(v,t,Q)$
6: **end if**

To start with one can make the invariants hold by making $Q = \{(0,s)\}$ and T_s consist of only one node, namely the *sources* with a cost of 0, while S_s can be taken to be any SPT of G (e.g., produced by the static Dijkstra's algorithm). Otherwise Q can be made empty, T_s can be any SPT of G (as required for example by Alg. 6), and $S_s = T_s$.

It should be noted that the technique we propose in case 2 of the proof of *Relax* for the construction of S'_s is of more general use. It can be applied in conjunction with any SPT maintenance algorithm in the most general case of multiple mixed edge weight changes to ensure that shortest paths that remain optimal after updates (potentially with a different length) do not change unnecessarily. This property is important in routing protocols [15], and it is a way to define the topological stability of SPT maintenance algorithms. A stronger version of stability has been addressed before for the semidynamic case, when the edge weight changes are either all increases or all decreases [15], but it should be noted that the simple chaining of semidynamic algorithms that individually ensure topological stability does not preserve topological stability under mixed edge weight changes.

4.2. Computation of Shortest Paths

Procedure *ShortestPath* computes a shortest path to a vertex by extracting the minimum from the queue and relaxing its outgoing edges until the cost of the vertex of interest is less than or equal to the minimum cost in the queue.

Proof 2 (Alg. 2 finds a shortest path to v, if any, preserving Inv 1 and Inv *We prove the postcondition and that the returned path is indeed optimal. We have* *seen that from Inv 2b it follows that vertices with $c(v, T_s) \leq Min(Q)$ are correct. Assuming no intervening graph updates, because of the strict inequality in procedure* Relax*, a vertex that is removed from Q is not enqueued for the second time. Note that the operation of the algorithm remains unchanged if we assume that in line 2 we only peek at the minimum p, and then later remove p from Q after the outgoing edges have been processed. Clearly, p remains a minimum (though not necessarily the only one) during the edge relaxation loop. Under the peeking assumption, the invariants still hold until after the outgoing edges have been relaxed. We only have to show that removing a just-expanded minimum of the queue does not break the invariants. Since we have seen that p is* correct *prior to the dequeue operation (due to Inv 2b), the only invariant that still requires specific treatment after the dequeue operation is Inv 2b, the others being trivial. But all children of p in S'_s (in fact, in any SPT of G'), are* correct *in T'_s because the relaxation of the edge from p to any such child gives the child its optimal cost. Therefore p does not have any* overestimated *children in S'_s that would require p to be present in Q, so Inv 2b also holds after the dequeue operation.*

Algorithm 2 *ShortestPath*

Require:

path tree T_s rooted at $s \in V$, queue Q *compatible* with $G = (V, E, w)$;

$v \in V$

Ensure: T_s, Q *compatible* with G

1: **while** $c(v, T_s) > Min(Q)$ **do**
2:     $p = ExtractMin(Q)$
3:     **for** $e \in Out_p$ **do**
4:         $Relax(p, h(e), w(e))$
5:     **end for**
6: **end while**
7: **return** path in T_s from s to v if $v \in T_s$; $NULL$ otherwise

---

An SPT can be obtained by looping until the queue becomes empty as shown in procedure *ShortestPathTree.*

Proof 3 (Alg. 3 computes an SPT of G and empties Q) *The correctness of the algorithm follows immediately from the correctness of Alg. 2.*

Finally, *ShortestPathOfMany* can find a shortest path to a vertex with minimum cost out of a set of target vertices and it is useful in the interactive image segmentation domain.

Proof 4 (Alg. 4 finds a shortest path to the closest $v \in D$, if any) *The correctness of the algorithm follows immediately from the correctness of Alg. 2.*

Algorithm 3 *ShortestPathTree*

Require:

path tree T_s rooted at $s \in V$, queue Q *compatible* with $G = (V, E, w)$;

Ensure: T_s is an SPT of G; Q is empty

while $\infty > Min(Q)$ **do**
  $p = ExtractMin(Q)$
  for $e \in Out_p$ **do**
    $Relax(p, h(e), w(e))$
  end for
end while
return T_s.

Algorithm 4 *ShortestPathOfMany*

Require:

path tree T_s rooted at $s \in V$, queue Q *compatible* with $G = (V, E, w)$;

$D \subset V$

Ensure: T_s, Q *compatible* with G

$m \leftarrow min(\{c(v, T_s) | v \in D\})$
while $m > Min(Q)$ **do**
  $p = ExtractMin(Q)$
  for $e \in Out_p$ **do**
    $Relax(p, h(e), w(e))$
    if $h(e) \in D$ **then**
      $m \leftarrow min(m, c(h(e), T_s))$
    end if
  end for
end while
return $NULL$ if $m = \infty$; path in T_s from s to any $v \in D$ s.t. $c(v, T_s) = m$ otherwise

4.3. Semidynamic algorithms

LazyDijkDec processes a set of edge weight decreases simply by relaxing each edge exactly as in Step 1 of *DynDijkDec* of [1], and it is included here for completeness.

Proof 5 (Alg. 5 restores Inv 1 and Inv 2 with respect to G') *The correctness of the algorithm follows immediately from the correctness of Alg. 1.*

LazyDijkInc performs the same operations as Steps 1 and 2 of *DynDijkInc* in [1], starting from a state where T_s is an SPT and Q is empty, and it is included here for completeness. The set of *locally affected* vertices is defined to consist of those vertices $v \in T_s$ such that on the path from root s to v in T_s at least one edge with increasing cost is encountered. The set of *locally unaffected* vertices is defined to consist of those vertices $v \in T_s$ such that on the path from root s to v in T_s there are no edges with increasing cost. Note that since T_s is an SPT, it contains all *reachable* vertices.

Algorithm 5 *LazyDijkDec*

Require:

path tree T_s rooted at $s \in V$, queue Q *compatible* with $G = (V, E, w)$;

set of edges ϵ^- whose weights are decreased

Ensure: T_s, Q *compatible* with G'

for $e_i \in \epsilon^-$ **do**
    $Relax(t(e_i), h(e_i), w'(e_i))$
end for

Algorithm 6 *LazyDijkInc*

Require:

path tree T_s rooted at $s \in V$ is an SPT of $G = (V, E, w)$; queue Q is empty

set of edges ϵ^+ whose weights are increased

Ensure: T_s, Q *compatible* with G'

1: $L_u \leftarrow$ *locally unaffected* vertices with respect to ϵ^+
2: $L_a \leftarrow$ *locally affected* vertices with respect to ϵ^+
3: Remove all vertices of L_a from T_s.
4: **for all** $v \in L_u$ **do**
5:     **for** $e \in Out_v$ **do**
6:         $Relax(v, h(e), w'(e))$
7:     **end for**
8: **end for**

Proof 6 (Alg. 6 restores Inv 1 and Inv 2 with respect to G') *We prove that the invariants hold at the end of* LazyDijkInc*. Let's consider the graph G^* which would be obtained from G after removal of all edges with the tail in L_u and the head in L_a. Then line 3 makes the path tree T_s become an SPT of G^* since* locally affected *vertices are unreachable in G^* and all the other vertices maintain their* correct *status. Then we evolve G^* by adding back all the missing edges that will make G^* identical to G'. Each such edge is* relaxed *to its new value in line 6, which according to the postcondition of* Relax *makes the invariant hold with respect to the graph with the edge added back (with its new weight). Therefore after the last edge is relaxed the state of the algorithm will be* compatible *with G'.*

It should be noted that for each edge with both the tail and the head in L_u operation *Relax* will not perform any changes, because all vertices in L_u stay *correct* with respect to G, G^* and G' all throughout the algorithm, and thus their cost cannot be further decreased. That makes the operation of *LazyDijkInc* identical to Steps 1 and 2 of *DynDijkInc*.

We call *LazyDynDijk* the chaining of *LazyDijkInc* followed by *LazyDijkDec* and then *ShortestPathTree*, and we call *DynDijkstra* the chaining of *DynDijkInc* and *Dyn-*

DijkDec. The order chosen for *DynDijkstra* is arbitrary [1] and it does not influence the results.

4.4. Time complexity

Herein we assume that the priority queue Q executes the operations $Min(Q)$ and $DecreaseKey(value, key, Q)$ in $O(1)$ amortised time, while the operation $ExtractMin(Q)$ runs in $O(log|V|)$ amortised time, where the $|\cdot|$ notation denotes the size of the set it is applied to. One such common data structure is the Fibonacci heap. It follows that Alg. 1 runs in $O(1)$ amortised time, while Alg. 2, Alg. 3 and Alg. 4 have the same time complexity as the static Dijkstra algorithm, namely $O(|E|+|V|\,log|V|)$. Then Alg. 5 runs in $O(|\epsilon^-|)$ and Alg. 6 runs in $O(|\epsilon^+| + |V| + |E|)$, giving *LazyDynDijk* a time complexity of $O(|\epsilon^+|+|\epsilon^-|+|E|+|V|\,log|V|)$, which is the same as static Dijkstra's $O(|\epsilon^-| + |E| + |V|\,log|V|)$ given that $|\epsilon^+| \leq |E|$.

4.5. Example

In Fig. 1 we present an example graph and a set of operations applied to it that show how the algorithms above operate changes in the data structures T_s and Q. After each operation we also show the SPT S_s that results by applying the constructive approach detailed in the proofs above, restoring the invariants Inv 1 and Inv 2 with respect to T_s, Q and S_s after each operation.

Below we point out a few key points about the steps in the example of Fig. 1:

Fig. 1a. Initially we assume that no graph edges are present. As a starting point, we chose the first of two options presented in Section 4.1, namely Q contains only a, while T_s has only the source node a with a cost of 0, and S_s can be taken to be any SPT rooted at a. In this case the only reachable node is a, so S_s has a as the only node.

Fig. 1b. After introducing a few edges but without making any other vertex reachable, T_s, Q and S_s remain unchanged.

Fig. 1c. The introduction of an edge from a to b results in b being a *correct* node, while S_s becomes an SPT that has the same structure of the *correct* nodes a and b as T_s. Visually, within the dashed boxes the blue and red arrows link the same vertices, as required by Inv 2a. It should be noted that the relaxation algorithm did not compute a full path tree of the graph, while the proof already has an SPT containing all 5 vertices to maintain. Note also that in this case the only *overestimated* vertex whose parent in S_s is *correct* is d. Since b, which is the parent in S_s of d, is indeed present in Q, Inv 2b also holds.

Fig. 1d. The introduction of a new edge from b to c with the indicated weight does not change the optimal distance of any vertex in the graph, but it results in c becoming a *correct* vertex. In particular, the optimal cost of vertex c remains equal to its previous value of 3. However, this operation results in modifications of S_s according to Case 1(b)ii of the proof of Alg. 1, which are made to comply with the choice taken by Alg. 1 to assign c as a child of b in T_s. Therefore,

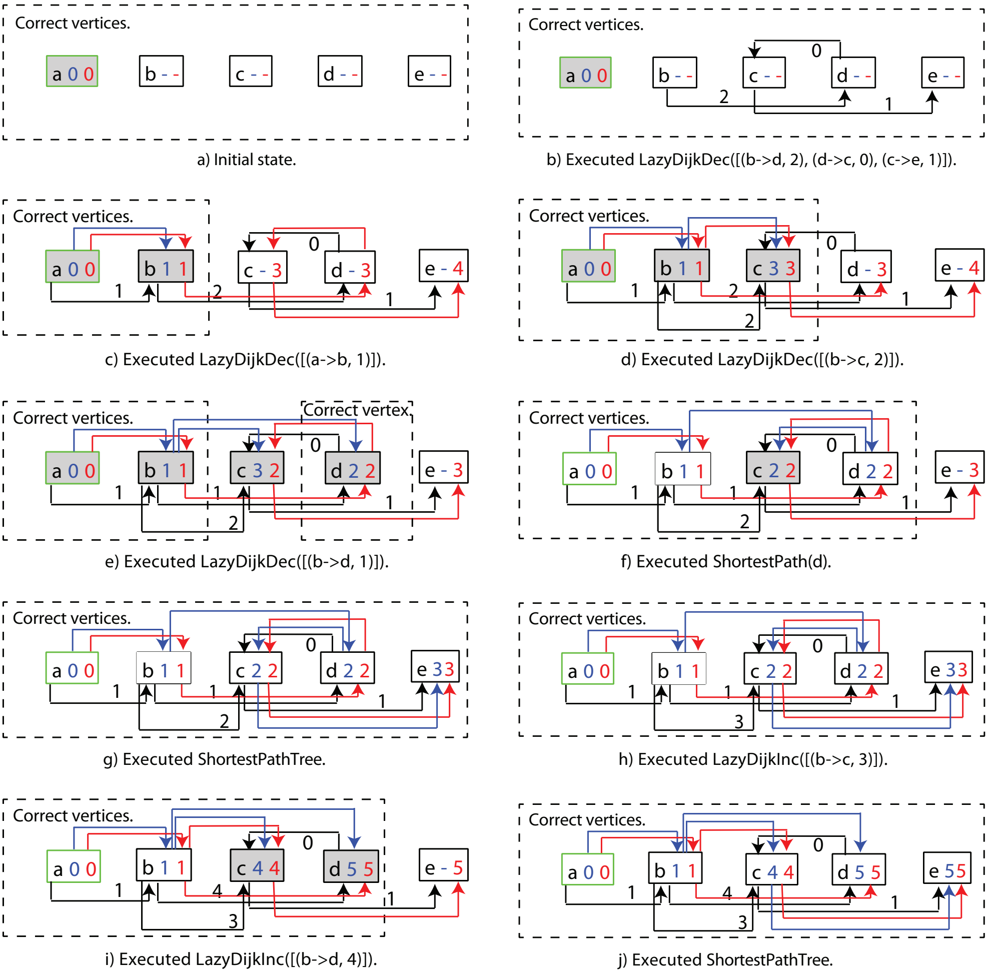

Figure 1. Example of graph with 5 vertices named a, b, c, d and e, respectively, with operations applied as described in the subfigure labels.

The notation $[\ldots]$ denotes a list of items, while the notation $(t \to h, w)$ denotes an edge from tail t to head h with weight w and the dash symbol $-$ denotes positive infinity. In the figure each vertex is represented by a box. The source vertex box has a green border, while all the others have black borders. The boxes of vertices present in Q have a light grey background, while all the other vertex boxes have white background. A vertex box contains three items, which are in order from left to right: an identifier of the vertex v written in black, its cost $c(v, T_s)$ in blue and its optimal distance from the source vertex $d(v, G)$ in red. All graph edges are drawn as arrows labelled with the weight of the edge, in black. The edges of T_s are indicated with blue arrows, while the edges of S_s are indicated with red arrows. All *correct* vertex boxes, i.e., the vertex boxes in which the second and third items are equal, are enclosed for ease of visualisation within one or more dashed boxes. The operations performed on the starting empty graph of subfigure a) are indicated, in order, in the labels of the subfigures b), c), d), e), f), i) and j). Each subfigure depicts the state of G, T_s, Q and S_s after the operation indicated in its label has been performed.

although $a \to b \to d \to c$ is still a shortest path for c, S_s is modified so that it contains for vertex c the shortest path $a \to b \to c$, thereby maintaining Inv 2a. The only *overestimated* vertices are d and e, and their parents in S_s, namely b and c, are both *correct*. Since both b and c are present in Q, Inv 2b holds.

Fig. 1e. The decreased weight of the edge from b to d makes node c become *overestimated*, so it does not matter any longer for Inv 2a. The relaxation of the edge makes node d become *correct*. After the modification of S_s according to the procedure described in Case 2 of Alg. 1, since d becomes *correct*, its parent in T_s is proved according to Case 2a to become b, as can be seen in the figure, thereby ensuring that Inv 2a holds. Similarly, Case 2(b)i proves that d is in Q, as also seen in the figure, thereby ensuring that Inv 2b holds.

Fig. 1f. The computation of a shortest path to d does not even need to extract c from Q, because its optimal cost is equal to that of d, thereby leaving its neighbour e untouched.

Fig. 1g. Alg. 3 ensures that T_s becomes an SPT (equal to S_s) and that Q is empty, thereby making it possible to perform next a (batch) edge weight increase operation.

Fig. 1h. Increasing the weight of an edge that is not present in $T_s = S_s$ does not result in any changes in either T_s, Q or S_s, because it cannot affect any shortest paths. Therefore a (batch) weight increase operation can still be performed.

Fig. 1i. Increasing the weight of the edge from b to d, vertices a and b are *locally unaffected*, while vertices d, c and e are *locally affected*. Therefore d, c and e are first removed from T_s and then the outward edges from a and b are relaxed. The proof of Alg. 6 shows that all invariants hold after the operation is completed, as can also be seen in the figure. Therefore edge weight decreases and shortest path computations can be resumed at this point with no need to compute a full SPT.

Fig. 1j. However, we finish this example with a full SPT computation showing that vertex e, which was *overestimated* at the end of the previous step, becomes *correct*.

5. Experiments

The experiments were performed on an Intel CPU core running at 1.6 GHz. We used Java 7u4 and restricted the memory capacity of the Java Virtual Machine to 1 G. For the queue implementation we used a Fibonacci heap all throughout. For all tests we measured the execution time, and we also counted the following types of elementary operations: queue extractions, key decreases in the queue, graph edge traversals, graph edge weight accesses and tree edge visits where applicable. Cumulative cost computations have been performed in double precision floating point arithmetic.

5.1. Interactive Image Segmentation

5.1.1. Setup

We adopted the version of *EnhancedLane* [10] that makes the computed paths optimal independently of the order in which the user pointer is moved, i.e., by defining the search area as the union of all windows where the user pointer has been detected since the tracing operation of the latest segment was started, as explained in Fig. 2.

The window size was fixed to 90×90 pixels. We adopted the *G-wire* technique [9] in order to include a curvature internal energy component, resulting in a graph with the number of vertices equal to eight times the number of pixels in the image, based on the 8-neighbour system. Each graph vertex corresponds to an image pixel and one of the eight directions from which one can arrive at the image pixel (Fig. 3a). Directed graph edges are created between each pair of graph vertices that correspond to image pixels that are 8-neighbours and the direction of arrival encoded by the head is compatible with the pixel location corresponding to the tail vertex (Fig. 3b).

The curvature component proposed in [9] is proportional to $|v_{k+1} - 2v_k + v_{k-1}|^2$, where v represents the vector of (x, y) coordinates of consecutive pixels on the path $\vec{v}$ and $|\cdot|$ stands for the Euclidean distance. We found this curvature energy to be incompatible with lines that are not perfectly horizontal, vertical or diagonal. Indeed, in such cases in the absence of other strong energy components such a line would be approximated by two segments, one perfectly horizontal or vertical, and the other one diagonal, rather than sticking to the line as closely as possible, and that would require another strong stretching energy component to correct. We used instead a curvature energy for three consecutive pixels based on the angle formed at the middle one: $E_c(0) = 1$, $E_c(\pi/4) = 0.75$, $E_c(\pi/2) = 0.5$, $E_c(3\pi/4) = E_c(\pi) = 0$ where the angle $\angle(v_{k-1}, v_k, v_{k+1})$ is the $[0, \pi]$-normalized angle of the vectors $\vec{v_{k-1}v_k}$ and $\vec{v_{k+1}v_k}$. Our curvature energy opposes sharp bends with angles less than or equal to $\pi/2$ while it is indifferent to the others.

For any three image pixels a, b and c such that a and b are neighbours, and also b and c are neighbours (c could be the same as a), we assigned to the directed graph edge with the tail encoding image pixel b arriving from a and the head encoding the image pixel c arriving from b the cost of $|c - b| * [0.05 + 0.95(2 - |\nabla I(b)| - |\nabla I(c|)/2 + E_c(\angle(a, b, c))]$, where ∇I is the [0, 1]-normalized image gradient computed after Gaussian convolution with a standard deviation of 2.5.

Therefore the resulting empirical total energy function for a path $\vec{v}$ is $E = \sum_k |v_{k+1} - v_k| * [0.05 + 0.95(2 - |\nabla I(v_k)| - |\nabla I(v_{k+1}|)/2 + E_c(\angle(v_{k-1}, v_k, v_{k+1}))]$.

The graph was represented implicitly by keeping track only of the search area, while edge costs were computed on-demand.

We used a dataset of 51 greyscale pictures with 32 bits per pixel and image sizes from 341055 to 1920000 pixels with an average of 748610.6 pixels per image. The images consist of pictures of animals in nature whose contour we delineated. Every time the user moved the pointer, the search area was expanded by union with the new window centered at the user pointer position, and the shortest path was

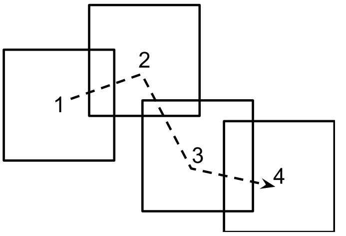

Figure 2. Incremental path map update when the user traces starting at position 1, then moves to positions 2, 3 and 4. Each time the mouse pointer is moved, a new window is added to the search area for the optimal path. Initially the search area consists only in the window centered at 1, but the optimal path is trivial and consists of only the starting point. Then when the mouse pointer is moved to position 2, the search area consists of the union of the windows centered at 1 and 2, therefore an optimal path needs to be found from position 1 to position 2 in the graph corresponding to the union of windows 1 and 2. After the optimal path is found, the mouse pointer is detected at position 3, adding the window centered at 3 to the search area. Finally, the window centered at 4 is added to the search area, and the shortest path between 1 and 4 needs to be found in the search area consisting in the union of all four windows. Because the user generally moves the mouse pointer continuously, shortest path computations need to be as fast as possible to ensure a good user experience.

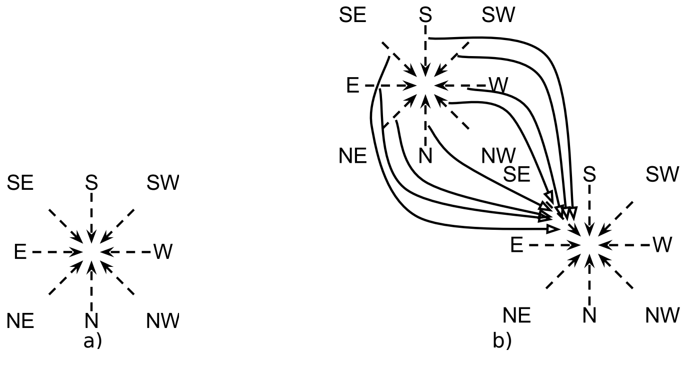

Figure 3. Directed graph based on the 8-neighbour system. (a) There are eight possible ways to arrive at an image pixel, shown as dashed arrows, and each of them will have its own vertex in the graph. (b) Two neighbouring image pixels are considered on the SE-NW axis. The eight graph vertices of each image pixel are shown as in (a), and continuous arcs represent the directed edges having the tail corresponding to the image pixel closer to the NW corner and the head corresponding to the image pixel closer to the SE corner. Note that only one of the eight graph vertices corresponding to the latter image pixel serves as the head of edges from the vertices corresponding to the former image pixel.

Table 1
Interactive Image Segmentation Statistics for 51 images totalling 40276 detected movements of the user pointer.

Unit of measurement	DynDijkDec		LazyDijkDec	
	Mean	Median	Mean	Median
10^6 ops	56.66	54.81	39.90	40.67
ms	14.90	2.97	11.86	2.41

computed from the latest *seed* point (fixed by the user) to the latest pointer position. More exactly, the shortest path to the set of eight graph vertices corresponding to the pointer position was computed using procedure *ShortestPathOfMany* in the case of *LazyDijkDec*, and trivially selecting the vertex with the shortest distance and its shortest path in the case of *DynDijkDec* (or, equivalently, *EnhancedLane*).

To ensure the reproducibility of the results, we recorded all user operations after which we replayed via an automated procedure the same steps taken by the user in a UI-free manner 100 times, measuring for each user pointer movement the response time and taking the average. The response times were measured end-to-end and they also include geometrical computations related to the union of the windows. The number of elementary operations was also recorded. Static *Dijkstra* was too slow to be applied the same procedure and it was omitted.

5.1.2. Results

Summary statistics about the elementary operations performed per image and the response time per user pointer movement are listed in Table 1. One-sided Wilcoxon signed-rank tests show that the speedup of *LazyDijkDec* as compared to *DynDijkDec* (or, equivalently, *EnhancedLane*) is statistically significant both in terms of number of elementary operations ($p = 2.66 \times 10^{-10}$) and response time ($p = 1.63 \times 10^{-22}$).

Dividing the total number of elementary operations of *DynDijkDec* by the total number of elementary operations of *LazyDijkDec* we obtain 1.42. For the total response time we get a ratio of 1.26.

5.2. Random graphs

5.2.1. Setup

To generate the random graph test data we used the approach of [1] with slightly different configurations. The number of vertices (*graphSize*) and the number of edges were kept the same and are given in Table 2. The percentage of changed edges (*pce*) took values in the set $\{0.1, 0.2, 0.5, 1, 2, 5, 7, 9, 20, 50, 75, 100\}$. The percentage of increased edges (*pie*) were picked from the set $\{10, 30, 50, 70, 90\}$. The percentage of changed weight in the decrease case (*pcwDec*) took values in $\{5, 10, 20, 40, 60, 90\}$, while the one in the increase case (*pcwInc*) in $\{100, 200, 1000\}$. Note that since the

Table 2
Artificial Random Graph Statistics.

# vertices	# edges
200	17300
400	61400
800	220400

changed weight in the decrease case is a percentage smaller than 100%, edge weights never become negative. All edge weights have been rounded to the closest integer.

For each combination of *graphSize*, *pce*, *pie*, *pcwDec*, *pcwInc*, we performed $2 \times 3 \times 25 = 150$ SPT computations as described next. Two graphs were first generated with *graphSize* vertices and number of edges as in Table 2. Edge weights were integers uniformly chosen from 1 to 10^6 inclusive. The choice of the range of initial weights together with *pcwInc* makes all cumulative cost computations free of rounding error. For each graph, three groups of edges were randomly selected, each of size corresponding to *pce*. For each group of edges, *pie* indicates how many of the edges would have their cost increased, while the others will have their cost decreased. Then 25 vertices were randomly selected as shortest path sources.

5.2.2. Results

We analyse the effect of the factors *pce*, *pie*, *pcwDec* and *pcwInc* on the performance of the algorithms.

Based on *pce* it can be seen in Fig. 4 that for each graph size there is a *pce* threshold above which static *Dijkstra* becomes faster than the non-static algorithms tested. The *pce* threshold in the tested scenarios varies around 5-9% and it decreases with increasing number of vertices. *LazyDynDijk* always crosses the static algorithm line slightly later, after which it degrades more slowly than *DynDijkstra*.

For the analysis based on *pie*, we used *pce* maximum thresholds of 9, 7 and 5 for 200, 400 and 800 vertices, respectively, so that we fall in the region where both non-static algorithms are faster than static *Dijkstra*. In this range *LazyDynDijk* is only marginally faster than *DynDijkstra*, as seen in Fig. 5. For all three graph sizes we find that the average number of operations and execution time increase first with increasing *pie* reaching a maximum at a *pie* value of 75 and decrease afterwards. This finding is consistent with the observation of Ref. [1] that the execution time of *DynDijkstra* first increases and then decreases as one gradually increases *pie* from 10 to 90.

For the analysis based on *pcwDec* we used the same *pce* maximum thresholds of 9, 7 and 5 for 200, 400 and 800 vertices, respectively. Both *LazyDynDijk* and *DynDijkstra* show an increasing trend in the number of operations and execution time with increasing *pcwDec*. The reason is that larger decreases of the edge weights are

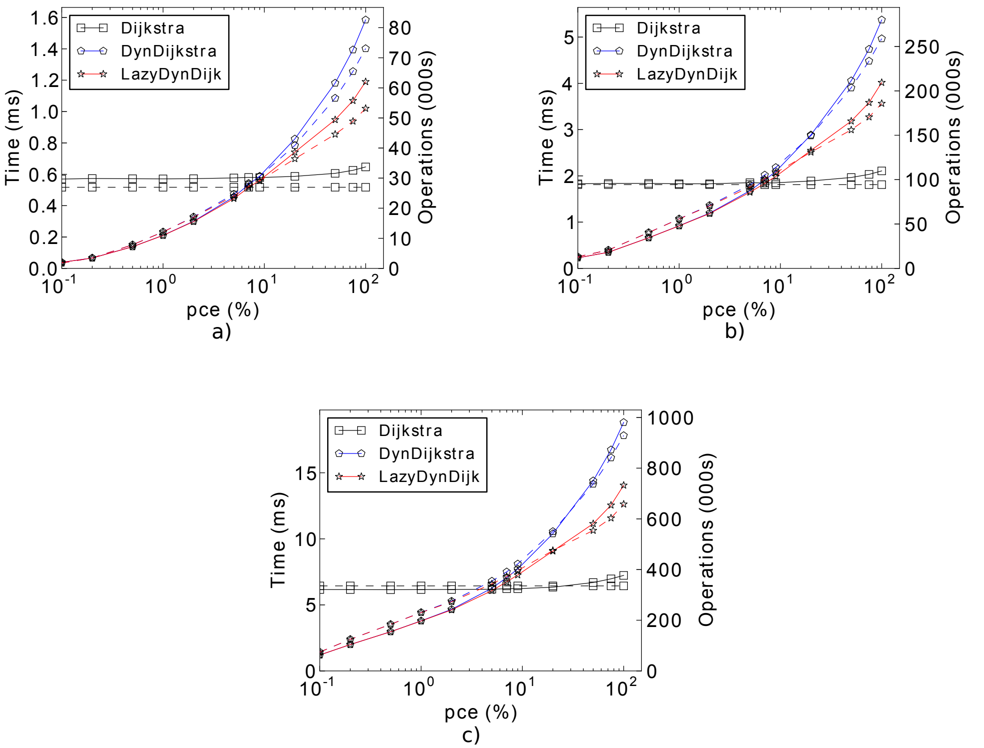

Figure 4. Comparison of mixed edge weight changes by *pce* for random graphs with (a) 200, (b) 400, (c) 800 vertices. Each point represents the average over $5 \times 6 \times 3 \times 150 = 13500$ executions.

more likely to result in structural changes to the SPT. Also in this case, *LazyDynDijk* has a slight edge over *DynDijkstra*, as seen in Fig. 6.

We also analysed the results based on *pcwInc* with the same *pce* maximum thresholds as above. Both the execution time and number of operations are mostly flat lines for all three tested algorithms. The reason is that both non-static algorithms trim the SPT based only on which edges have their weight increased, without taking into account the amount of weight increase. Also in this case, *LazyDynDijk* has a slight edge over *DynDijkstra* (not shown).

5.3. Real road network

5.3.1. Setup

In order to prepare the road system network graph, The National Highway Planning Network (NHPN) [6] for the whole U.S. was preprocessed using OpenJUMP [18] by

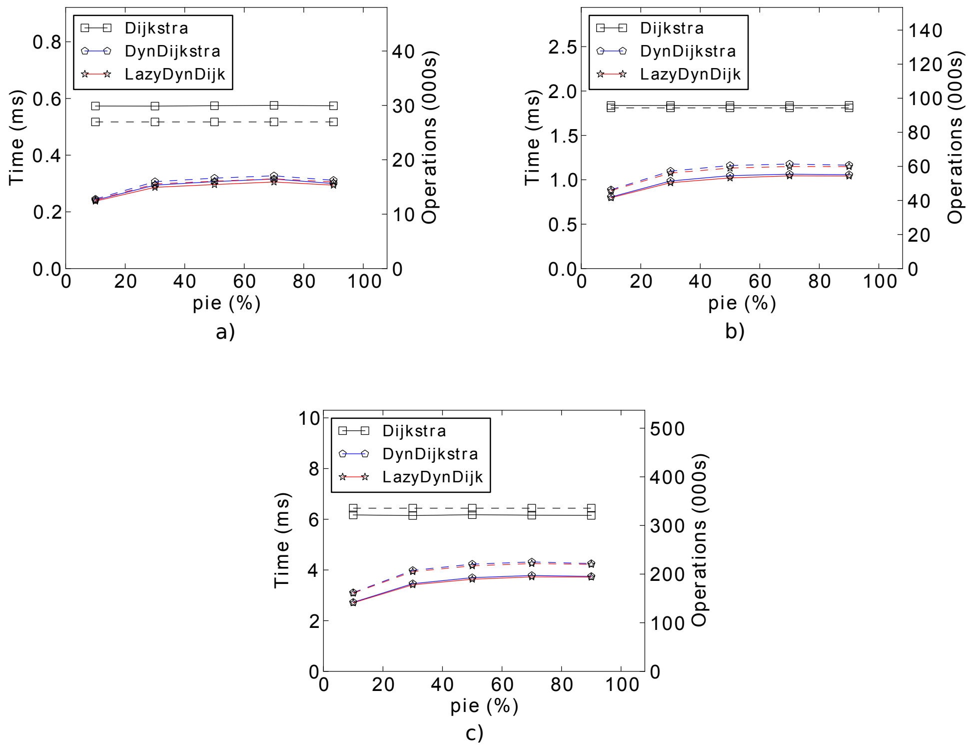

Figure 5. Comparison of mixed edge weight changes by *pie* for random graphs with (a) 200, (b) 400, (c) 800 vertices using *pce* thresholds of 9, 7 and 5, respectively. Each point represents the average over $8 \times 6 \times 3 \times 150 = 21600$, $7 \times 6 \times 3 \times 150 = 18900$ and $6 \times 6 \times 3 \times 150 = 16200$ executions, respectively.

transforming the connected components into "simplified multistrings", extracting the largest connected component and exporting the geometry in CSV format.

The weight of each edge was then computed using the haversine formula and scaled as an integer from 0 to 10^6. After replacing each undirected edge with two opposite directed edges and removing duplicates, we obtained a strongly connected graph with 135820 vertices and 344138 directed edges, therefore averaging only 2.53 edges/vertex. Such low edge-to-vertex ratios are expected since they are a hallmark of real road networks [19].

For the tests we used the same setup procedure as for the random graphs (see Section 5.2.1), with the only difference that for each *pce*, *pie*, *pcwDec* and *pcwInc* configuration, instead of generating 2 random graphs, we always used the same NHPN graph performing $1 \times 3 \times 25 = 75$ SPT computations.

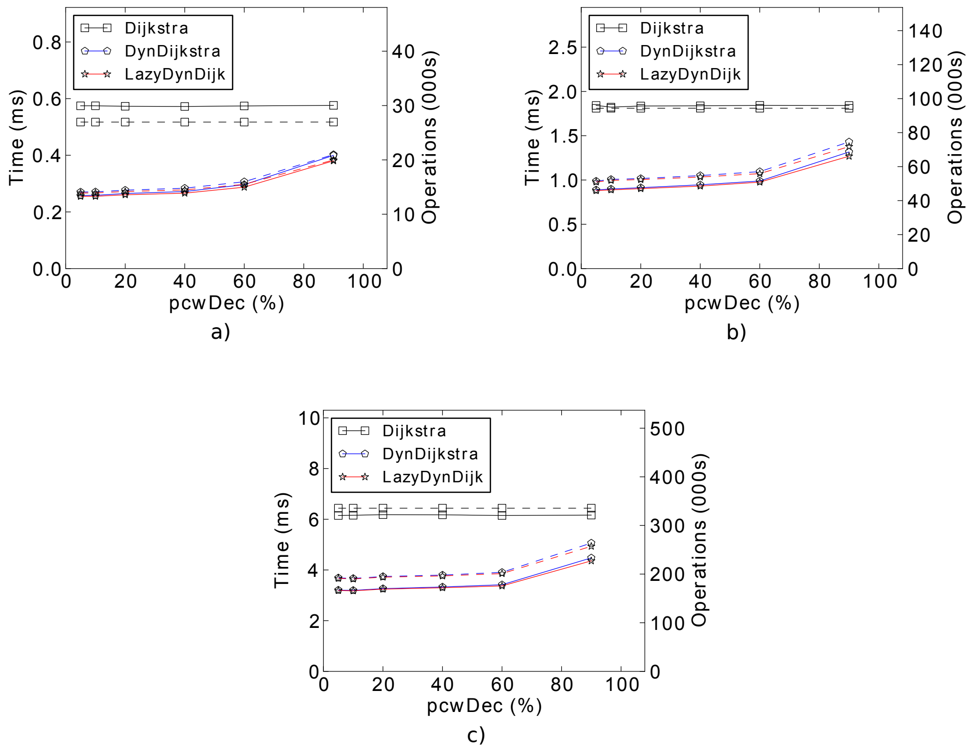

Figure 6. Comparison of mixed edge weight changes by *pcwDec* for random graphs with (a) 200, (b) 400, (c) 800 vertices using *pce* thresholds of 9, 7 and 5, respectively. Each point represents the average over $8 \times 5 \times 3 \times 150 = 18000$, $7 \times 5 \times 3 \times 150 = 15750$ and $6 \times 5 \times 3 \times 150 = 13500$ executions, respectively.

5.3.2. Results

As in Section 5.2.2, we analyse the effect of the factors *pce*, *pie*, *pcwDec* and *pcwInc* on the performance of the algorithms.

Based on *pce*, *LazyDynDijk* outperforms *DynDijkstra* at all *pce* thresholds, while static Dijkstra becomes faster than both tested non-static algorithms after a *pce* threshold somewhere above 0.2. At larger *pce* values, however, the performance of our algorithm *LazyDynDijk* follows more closely that of static Dijkstra, strongly outperforming *DynDijkstra* in terms of both number of operations and execution time, as seen in Fig. 7a.

For the analysis based on *pie* we used a *pce* maximum threshold of 0.2 so that we fall clearly in the region where both non-static algorithms are faster than static *Dijkstra*. In this range *LazyDynDijk* is slightly faster than *DynDijkstra* both in terms of number of operations and execution time. As seen in Fig. 7b, the worst case

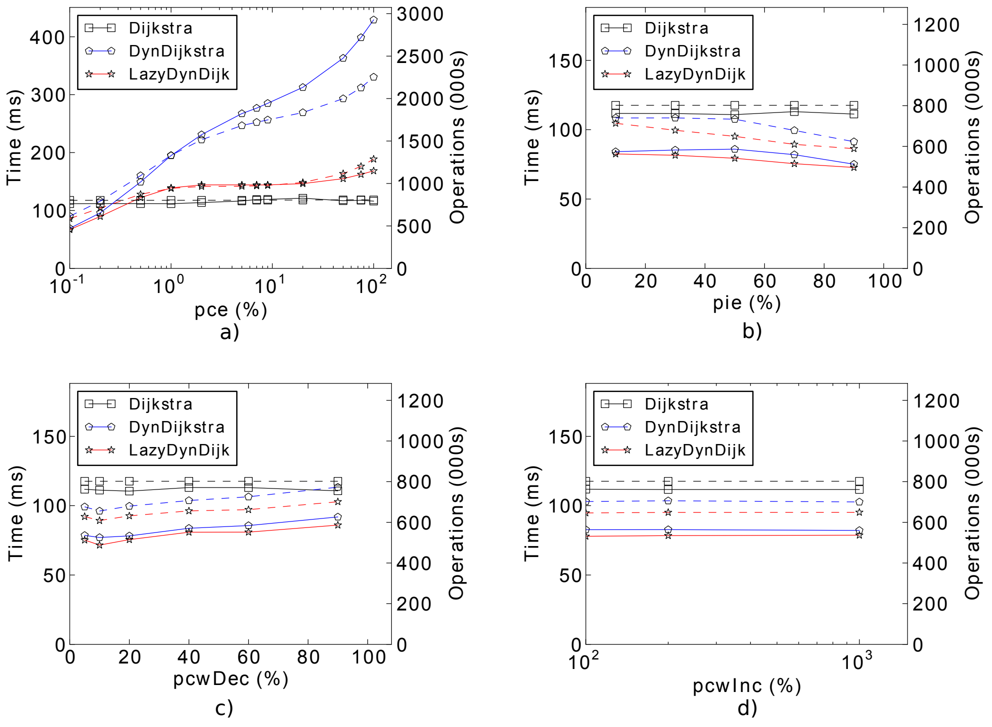

Figure 7. Performance evaluation on the NHPN road system graph. (a) Comparison of mixed edge weight changes by *pce*. Each point represents the average over $5 \times 6 \times 3 \times 75 = 6750$ executions. (b) Comparison of mixed edge weight changes by *pie* using the *pce* threshold of 0.2. Each point represents the average over $2 \times 6 \times 3 \times 75 = 2700$ executions. (c) Comparison of mixed edge weight changes by *pcwDec* using the *pce* threshold of 0.2. Each point represents the average over $2 \times 5 \times 3 \times 75 = 2250$ executions. (d) Comparison of mixed edge weight changes by *pcwInc* using the *pce* threshold of 0.2. Each point represents the average over $2 \times 5 \times 6 \times 75 = 4500$ executions.

for *DynDijkstra* is when the modified edge set is about half increases and about half decreases, as previously found for other road system networks [1]. Conversely, *LazyDynDijk* shows an almost linear performance improvement with increasing *pie*, showing that the performance behaviour of *LazyDynDijk* can be qualitatively different from that of *DynDijkstra* even in the region where both algorithms are faster than static Dijkstra.

For the analysis based on *pcwDec* we used the same *pce* maximum threshold of 0.2. As expected, both *LazyDynDijk* and *DynDijkstra* show a generally increasing trend in the number of operations and execution time with increasing *pcwDec*. Also in this case, *LazyDynDijk* maintains a slight edge over *DynDijkstra*, as seen in Fig. 7c.

We also analysed the results based on *pcwInc* with the same *pce* maximum threshold of 0.2. Like in the random graph case, both the execution time and number of operations are mostly flat lines for all three tested algorithms, with *LazyDynDijk* being only slightly faster than *DynDijkstra*, as seen in Fig. 7d.

6. Conclusion

By exploiting the idea of postponing shortest path computations until actually needed, we improved upon a recent, perhaps fastest-yet algorithm for shortest path computation in dynamic graphs. The improvements we obtained come in two forms. First, when batch edge weight decreases are just as frequent as shortest path requests, the speedup comes from stopping the computation as soon as the required shortest path is found. As a practical application, we report in interactive image segmentation an improvement of about 1.42 in the number of elementary operations and about 1.26 in measured response time. Second, when an SPT needs to be maintained in the face of mixed edge weight changes, we process the multiple increases first by recasting the problem as decreases in an altered graph, which can then be followed by other decreases without having to compute an intermediate SPT. This procedure results in slightly faster SPT computation when few edges change weight, and significantly better degradation behaviour for larger sets of updates.

All throughout the experiments the reported execution time improvement was consistent with the reduction in the number of elementary operations. It follows that the improvements observed are qualitatively independent of implementation details such as the choice of the priority queue data structure or edge weight data type, thus making the technique widely applicable.

References

[1] Chan E. P. F., Yang Y.: Shortest path tree computation in dynamic graphs. *IEEE Trans. Comput.*, 58(4):541–557, 2009.

[2] Dijkstra E. W.: A Note on Two Problems in Connection with Graphs. *Numerical Mathematics*, 1:269–271, 1959.

[3] Falcao A. X., Udupa J. K., Miyazawa F. K.: An ultra-fast user-steered image segmentation paradigm: live wire on the fly. *IEEE transactions on medical imaging*, 19(1):55–62, 2000.

[4] Falcao A. X., Udupa J. K., Samarasekera S., Sharma S., Hirsch B. E., de A. Lotufo R.: User-steered image segmentation paradigms: Live wire and live lane. *Graphical Models and Image Processing*, 60(4):233 – 260, 1998.

[5] Faloutsos M., Faloutsos P., Faloutsos C.: On power-law relationships of the internet topology. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 251–262, New York, NY, USA, 1999. ACM.

[6] Federal Highway Administration (FHWA).: The National Highway Planning Network, 08 2005.

[7] Frigioni D., Ioffreda M., Nanni U., Pasqualone G.: Experimental analysis of dynamic algorithms for the single source shortest path problem. *J. Exp. Algorithmics*, 3:5, 1998.

[8] Frigioni D., Marchetti-Spaccamela A., Nanni U.: Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms*, 34(2):251–281, 2000.

[9] Kang H. W.: G-wire: a livewire segmentation algorithm based on a generalized graph formulation. *Pattern Recogn. Lett.*, 26(13):2042–2051, 2005.

[10] Kang H. W., Shin S. Y.: Enhanced lane: interactive image segmentation by incremental path map construction. *Graph. Models*, 64(5):282–303, 2002.

[11] Meijering E., Jacob M., Sarria J.-C. F., Steiner P., Hirling H., Unser M.: Design and validation of a tool for neurite tracing and analysis in fluorescence microscopy images. *Cytometry. Part A: the journal of the International Society for Analytical Cytology*, 58(2):167–176, 2004.

[12] Mortensen E. N., Barrett W. A.: Intelligent scissors for image composition. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 191–198, New York, NY, USA, 1995. ACM.

[13] Moy J. T.: *OSPF: Anatomy of an Internet Routing Protocol.* Addison-Wesley Professional, New York, 1998.

[14] Narváez P., Siu K.-Y., Tzeng H.-Y.: New dynamic algorithms for shortest path tree computation. *IEEE/ACM Trans. Netw.*, 8(6):734–746, 2000.

[15] Narváez P., Siu K.-Y., Tzeng H.-Y.: New dynamic spt algorithm based on a ball-and-string model. *IEEE/ACM Trans. Netw.*, 9(6):706–718, 2001.

[16] Perlman R.: A comparison between two routing protocols: OSPF and IS-IS. *IEEE Network*, 5(5):18–24, 1991.

[17] Ramalingam G., Reps T.: An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[18] Steiniger S., Bocher E.: An overview on current free and open source desktop GIS developments. *International Journal of Geographical Information Science*, 23(10):1345–1370, 2009.

[19] Zhan F. B., Noon C. E.: Shortest path algorithms: An evaluation using real road networks. *Transportation Science*, 32(1):65–73, January 1998.

Affiliations

Daniel Aioanei

Department of Biochemistry "G. Moruzzi", University of Bologna, Via Irnerio 48, 40126 Bologna, Italy, `aioaneid@gmail.com`

Received: 9.03.2012
Revised: 2.06.2012
Accepted: 9.07.2012