A parallel algorithm for the constrained shortest path problem on lattice graphs

Ivan Matic
Published in: Adamatzky, A (Ed.) Shortest path solvers. From software to wetware. Springer, 2018.

Abstract The edges of a graph are assigned weights and passage times which are assumed to be positive integers. We present a parallel algorithm for finding the shortest path whose total weight is smaller than a pre-determined value. In each step the processing elements are not analyzing the entire graph. Instead they are focusing on a subset of vertices called *active vertices*. The set of active vertices at time t is related to the boundary of the ball B_t of radius t in the first passage percolation metric. Although it is believed that the number of active vertices is an order of magnitude smaller than the size of the graph, we prove that this need not be the case with an example of a graph for which the active vertices form a large fractal. We analyze an OpenCL implementation of the algorithm on GPU for cubes in $\mathbb{Z}^d$.

1 Definition of the problem

The graph $G(V,E)$ is undirected and the function $f : E \to \mathbb{Z}_+^2$ is defined on the set of its edges. The first component $f_1(e)$ of the ordered pair $f(e) = (f_1(e), f_2(e))$ for a given edge $e \in E$ represents the time for traveling over the edge e. The second component $f_2(e)$ represents the weight of e.

A path in the graph G is a sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that for each $i \in \{1,2,\ldots,k-1\}$ there is an edge between v_i and v_{i+1}, i.e. $(v_i, v_{i+1}) \in E$. For each path $\pi = (v_1, \ldots, v_k)$ we define $F_1(\pi)$ as the total time it takes to travel over π and $F_2(\pi)$ as the sum of the weights of all edges in π. Formally,

$$F_1(\pi) = \sum_{i=1}^{k-1} f_1(v_i, v_{i+1}) \quad \text{and} \quad F_2(\pi) = \sum_{i=1}^{k-1} f_2(v_i, v_{i+1}).$$

Ivan Matic
Department of Mathematics, Baruch College, CUNY, One Bernard Baruch Way, New York, NY 10010, USA e-mail: Ivan.Matic@baruch.cuny.edu

Let $A, B \subseteq V$ be two fixed disjoint subsets of V and let $M \in \mathbb{R}_+$ be a fixed positive real number. Among all paths that connect sets A and B let us denote by $\hat{\pi}$ the one (or one of) for which $F_1(\pi)$ is minimal under the constraint $F_2(\pi) < M$. We will describe an algorithm whose output will be $F_1(\hat{\pi})$ for a given graph G.

The algorithm belongs to a class of label correcting algorithms [11, 20]. The construction of labels will aim to minimize the memory consumption on SIMD devices such as graphic cards. Consequently, the output will not be sufficient to determine the exact minimizing path. The reconstruction of the minimizing path is possible with subsequent applications of the method, because the output can include the vertex $X \in B$ that is the endpoint of $\hat{\pi}$, the last edge x on the path $\hat{\pi}$, and the value $F_2(\hat{\pi})$. Once X and x are found, the entire process can be repeated for the graph $G'(V', E')$ with

$$V' = V \setminus B, \quad A' = A, \quad B' = \{X\}, \quad \text{and} \quad M' = F_2(\hat{\pi}) - f_2(x).$$

The result will be second to last vertex on the minimizing path $\hat{\pi}$. All other vertices on $\hat{\pi}$ can be found in the same way.

Although the algorithm works for general graphs and integer-valued functions f, its implementation on SIMD hardware requires the vertices to have bounded degree. This requirement is satisfied by subgraphs of $\mathbb{Z}^d$.

Finding the length of the shortest path in graph is equivalent to finding the shortest passage time in first passage percolation. Each of the vertices in A can be thought of as a source of water. The value $f_1(e)$ of each edge e is the time it takes the water to travel over e. Each drop of water has its *quality* and each drop that travels through edge e looses $f_2(e)$ of its quality. Each vertex P of the graph has a label $Label(P)$ that corresponds to the quality of water that is at the vertex P. Initially all vertices in A have label M while all other vertices have label 0. The drops that get their quality reduced to 0 cannot travel any further. The time at which a vertex from B receives its first drop of water is exactly the minimal $F_1(\pi)$ under the constraint $F_2(\pi) < M$.

Some vertices and edges in the graph are considered *active*. Initially, the vertices in A are *active*. All edges adjacent to them are also called *active*. Each cycle in algorithm corresponds to one unit of time. During one cycle the water flows through active edges and decrease their time components by 1. Once an edge gets its time component reduced to 0, the edge becomes *used* and we look at the source S and the destination D of this water flow through the edge e. The destination D becomes *triggered*, and its label will be *corrected*. The label correction is straight-forward if the edge D was inactive. We simply check whether $Label(S) - f_2(e) > Label(D)$, and if this is true then the vertex D gets its label updated to $Label(S) - f_2(e)$ and its status changed to *active*. If the vertex D was active, the situation is more complicated, since the water has already started flowing from the vertex D. The existing water flows correspond to water of quality worse than the new water that has just arrived to D. We resolve this issue by introducing phantom edges to the graph that are parallel to the existing edges. The phantom edges will carry this new high quality water, while old edges will continue carrying their old water flows. A vertex stops

being active if all of its edges become used, but it may get activated again in the future.

2 Related problems in the literature

The assignment of phantom edges to the vertices of the graph and their removal is considered a label correcting approach in solving the problem. Our particular choice of label correction is designed for large graphs in which the vertices have bounded degree. Several existing serial computation algorithms can find the shortest path by maintaining labels for all vertices. The labels are used to store the information on the shortest path from the source to the vertex and additional preprocessing of vertices is used to achieve faster implementations [5, 9]. The ideas of first passage percolation and label correction have naturally appeared in the design of *pulse algorithms* for constrained shortest paths [17]. All of the mentioned algorithms can also be parallelized but this task would require a different approach in designing a memory management that would handle the label sets in programming environments where dynamical data structures need to be avoided.

The method of aggressive edge elimination [22] can be parallelized to solve the Lagrange dual problems. In the case of road and railroad networks a substantial speedup can be achieved by using a preprocessing of the network data and applying a generalized versions of Dijkstra's algorithm [12].

The parallel algorithm that is most similar in nature to the one discussed in this paper is developed for wireless networks [16]. There are two features of wireless networks that are not available to our model. The first feature is that the communication time between the vertices can be assumed to be constant. The other feature is that wireless networks have a processing element available to each vertex. Namely, routers are usually equipped with processors. Our algorithm is build for the situations where the number of processing cores is large but not at the same scale as the number of vertices. On the other hand our algorithm may not be effective for the wireless networks since the underlying graph structure does not imply that the vertices are of bounded degree. The increase of efficiency of wireless networks can be achieved by solving other related optimization problems. One such solution is based on constrained node placement [21].

The execution time of the algorithm is influenced by the sizes of the sets of active vertices, active edges, and phantom edges. The sizes of these sets are order of magnitude smaller than the size of the graph. Although this cannot be proved at the moment, we will provide a justification on how existing conjectures and theorems from the percolation theory provide some estimates on the sizes of these sets. The set of active vertices is related to the limit shape in the model of first passage percolation introduced by Hammersley and Welsh [10]. The first passage percolation corresponds to the case $M = \infty$, i.e. the case when there are no constraints. If we assume that $A = \{0\}$, for each time t we can define the ball of radius t in the first passage percolation metric as:

$$B_t = \{x : \tau(0,x) \leq t\},$$

where $\tau(0,x)$ is the *first passage time*, i.e. the first time at which the vertex x is reached.

The active vertices at time t are located near the boundary of the ball B_t. It is known that for large t the set $\frac{1}{t}B_t$ will be approximately convex. More precisely, it is known [7] that there is a convex set B such that

$$\mathbb{P}\left((1-\varepsilon)B \subseteq \frac{1}{t}B_t \subseteq (1+\varepsilon)B \text{ for large } t\right) = 1.$$

However, the previous theorem does not guarantee that the boundary of B_t has to be of zero volume. In fact the boundary can be non-polygonal as was previously shown [8].

The set of active vertices does not coincide with the boundary of B_t, but it is expected that if ∂B_t is of small volume then the number of active vertices is small in most typical configurations of random graphs. We provide an example for which the set of active vertices is a large fractal, but simulations suggest that this does not happen in average scenario.

The fluctuations of the shape of B_t are expected to be of order $t^{2/3}$ in the case of $\mathbb{Z}^2$ and the first passage time $\tau(0,n)$ is proven to have fluctuations of order at least $\log n$ [23]. The fluctuations are of order at most $n/\log n$ [4, 3] and are conjectured to be of order $t^{2/3}$. They can be larger and of order n for modifications of $\mathbb{Z}^2$ known as thin cylinders [6].

The scaling of $t^{2/3}$ for the variance is conjectured for many additional interface growth models and is related to the Kardar-Parisi-Zhang equation [1, 15, 25].

The constrained first passage percolation problem is a discrete analog to Hamilton-Jacobi equation. The large time behaviors of its solutions are extensively studied and homogenization results are obtained for a class of Hamiltonians [2, 13, 14, 26]. Fluctuations in dimension one are of order t [24] while in higher dimensions they are of lower order although only the logarithmic improvement to the bound has been achieved so far [19].

3 Example

Before providing a more formal description of the algorithm we will illustrate the main ideas on one concrete example of a graph. Consider the graph shown in Figure 1 that has 12 vertices labeled as 1, 2, …, 12. The set A contains the vertices 1, 2, and 3, and the set B contains only the vertex 12. The goal is to find the length of the shortest path from A to B whose total weight is smaller than 19.

The vertices are drawn with circles around them. The circles corresponding to the vertices in A are painted in blue and have the labels 19. The picture contains the time and weight values for each of the edges. The time parameter of each edge is written in the empty oval, while the weight parameter is in the shaded oval. Since

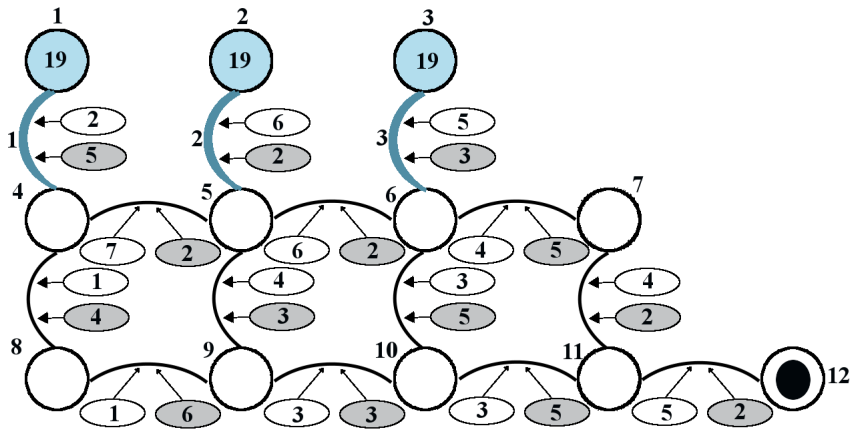

Fig. 1 The initial state of the graph.

the number of edges in this graph is relatively small it is not difficult to identify the the minimizing path $(3, 6, 10, 11, 12)$. The time required to travel over this path is 16 and the total weight is 15.

Initially, the vertices in set A are called *active*. Active vertices are of blue color and edges adjacent to them are painted in blue. These edges are considered *active*. Numbers written near their centers represent the sources of water. For example, the vertex 2 is the source of the flow that goes through the edge $(2,5)$.

Notice that the smallest time component of all active edges is 2. The first cycle of the algorithm begins by decreasing the time component of each active edge by 2. The edge $(1,4)$ becomes *just used* because its time component is decreased to 0. The water now flows from the vertex 1 to the vertex 4 and its quality decreases by 5, since the weight of the edge $(1,4)$ is equal to 5. The vertex 4 becomes active and its label is set to

$$Label(4) = Label(1) - f_2(1,4) = 19 - 5 = 14.$$

The edge $(1,4)$ becomes used, and the vertex 1 turns into inactive since there are no active edges originating from it. Hence, after two seconds the graph turns into the one shown in Figure 2.

The same procedure is repeated until the end of the 5th second and the obtained graph is the top one in Figure 3. In the 6th second the edge $(2,5)$ gets its time parameter decreased to 0 and the vertex 5 gets activated. Its label becomes

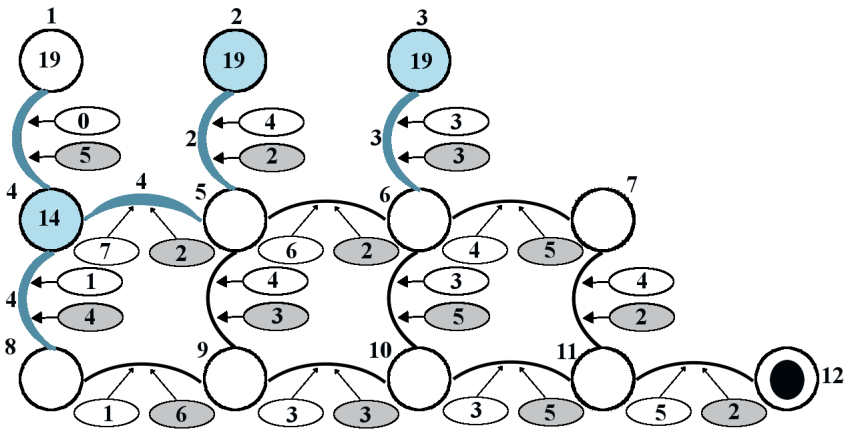

Fig. 2 The configuration after the second 2.

$$Label(5) = Label(2) - f_2(2,5) = 19 - 2 = 17.$$

However, the edges $(4,5)$, $(5,9)$, and $(5,6)$ were already active and the water was flowing through them towards the vertex 5.

The old flow of water through the edge $(4,5)$ will complete in additional 5 seconds. However, when it completes the quality of the water that will reach the vertex 5 will be

$$Label(4) - f_2(4,5) = 14 - 2 = 12 < Label(5)$$

because the label of the vertex 5 is 17. Thus there is no point in keeping track of this water flow. On the other hand, the water flow that starts from 5 and goes towards 4 will have quality

$$Label(5) - f_2(4,5) = 17 - 2 = 15$$

which is higher than the label of the vertex 4. Thus the edge $(4,5)$ will change its source from 4 to 5 and the time parameter has to be restored to the old value 7. At this point the vertex 4 becomes inactive as there is no more flow originating from it.

The same reversal of the direction of the flow happens with the edge $(5,9)$. On the other hand, something different happens to the edge $(5,6)$: it stops being active. The reason is that the old flow of water from 6 to 5 will not be able to increase the label of the vertex 5. Also, the new flow of water from 5 to 6 would not be able to change the label of vertex 6.

A special care has to be taken when a water flow reaches a vertex that is already active. In the case of the graph G such situation happens after the 11th second.

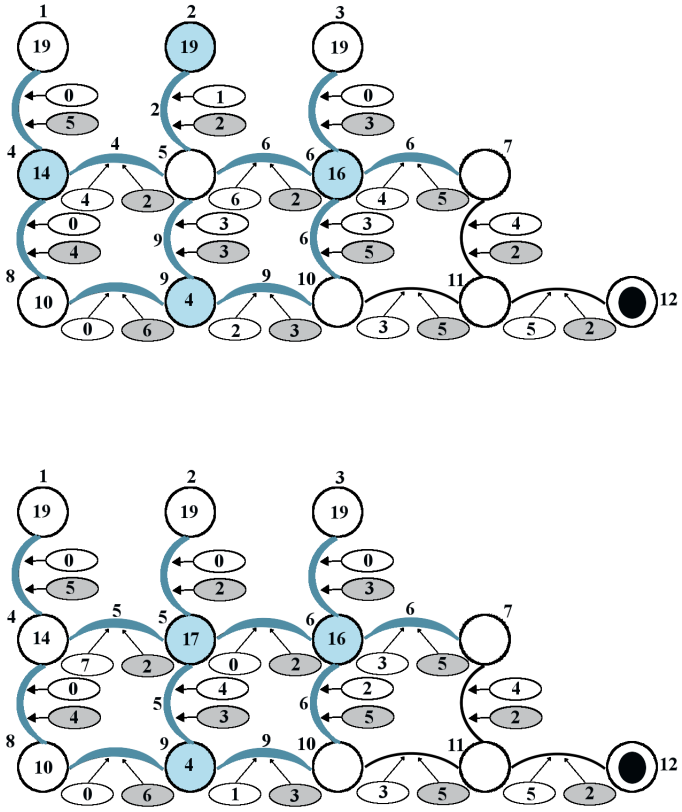

Fig. 3 The configurations after the seconds 5 and 6.

The configuration is shown in the top picture of Figure 4. The edge $(7,11)$ has the smallest time parameter 2. The time will progress immediately to 13 and all active edges get their time parameters decreased by 2. In the 13th second the water from the edge $(7,11)$ reaches the vertex 11. The label of vertex 7 is $Label(7) = 11$, while $Label(11) = 6$. The weight of the flow over the edge between these two vertices is 2, hence this new water is of higher quality than the one present at the vertex 11.

In this situation we consider every active edge originating from 11 and create a phantom edge through which this new water will flow. We will create a new vertex $11'$ with label

$$Label(11') = Label(7) - f_2(7,11) = 9$$

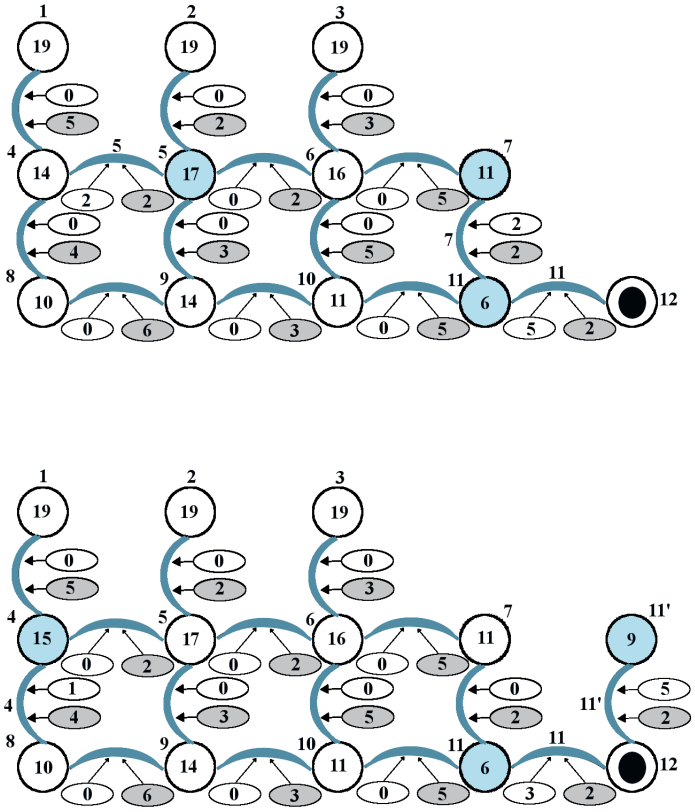

Fig. 4 The configurations after the seconds 11 and 13.

and connect it with each of the neighbors of 11 that can get their label increased with the new flow. The only one such neighbor is 12 and we obtain the graph as shown in the lower part of Figure 4.

It can be now easily verified that after additional 3 seconds, i.e. in the end of the second 16 the vertex 12 becomes active with the label 4. Thus we conclude that it takes water to travel 16 seconds over the shortest path. The total weight of the shortest path is $19-4=15$. The minimizing path is $(3, 6, 10, 11, 12)$.

4 Pseudo-code of the algorithm

We will organize the algorithm by dividing it into smaller components. The first component is the initialization, and the others are performed in the main loop that consists of 9 major steps. The parallelizations will happen only in these individual steps of the main loop. The pseudo-code for the main function is presented in Algorithm 1. Each step will be described in full details and accompanied by a pseudo-code that outlines the main ideas. For the sake of brevity, some data structures in pseudo-code will be modeled with sets. However, the usage of sets is avoided as they cannot support insertion and deletion of elements in parallel. The sets are replaced by indicator sequences for which appropriate operations are easier to parallelize. For the full source code the reader is referred to [18].

Algorithm 1 Main function

Input: Graph $G = (V, E)$; $A, B \subset G$ such that $A \cap B = \emptyset$, $M \in \mathbb{R}$, two functions $f_1, f_2 : E \to \mathbb{R}$. $f_1(e)$ is the time to travel over the edge e and $f_2(e)$ is the weight of the edge e.
Output: The shortest time to travel from A to B over a path whose weight is less than M.

```
1:  function MAIN
2:      Initialization
3:      L =ShortestTravelTimeAndTerminalConditionCheck
4:      while L = 0 do
5:          TriggerVertices
6:          AnalyzeTriggeredVertices
7:          GetInputFromPhantoms
8:          TriggerEdges
9:          TreatTriggeredEdges
10:         L =ShortestTravelTimeAndTerminalConditionCheck
11:         FinalTreatmentOfPhantoms
12:         FinalTreatmentOfVertices
13:         FinalTreatmentOfActiveEdges
14:     return L
```

5 Memory management and initialization

5.1 Labels for vertices and edges

In this section we will describe the memory management of variables necessary for the implementation of the algorithm. Before providing the precise set of variables let us describe the information that has to be carried throughout the execution process. As we have seen before, the vertices will have labels assigned to them. Initially we label each vertex of G with 0 except for vertices in A which are labeled by M.

To each vertex and edge in G we assign a *State*. The vertices have states in the set $\{$*active*, *inactive*$\}$. Initially all vertices in A are *active*, while the other vertices are *inactive*. The states of the edges belong to the set $\{$*active*, *passive*, *used*, *just used*$\}$. Initially the edges adjacent to the vertices in A are set to active while all other are passive.

To each edge we associate a pointer to one of its endpoints and call it *Source*. This variable is used at times when the water is flowing through the edge and it records the source of the current water flow. Initially, to each edge that originates from a vertex in A we set the source to be the pointer to the vertex in A. All other edges have their source initially set to 0.

There is additional variable *Time* that represents the time and is initially set to 0.

5.2 Termination

The algorithm terminates if one of the following two conditions is satisfied:

1° A vertex from B becomes *active*. The variable *Time* contains the time it takes to reach this vertex along the shortest path $\hat{\pi}$, i.e.

$$Time = F_1(\hat{\pi}).$$

The label of the last vertex $\hat{B}$ on the path allows us to determine the value $F_2(\hat{\pi})$. Namely,

$$F_2(\hat{\pi}) = M - \text{Label}(\hat{B}).$$

We will not go into details on how to recover the exact shortest path. Instead we will just outline how this can be done. We need to identify the *used* edge f (or one of the used edges, if there are more than one) that is adjacent to $\hat{B}$. This edge can help us in finding the second to last point of the path $\hat{\pi}$. Let us denote by F the other endpoint of f. It could happen that F is a phantom vertex (i.e. a copy of another vertex), and we first check whether $F \in PhantomVertices$. If this is not the case, then F is the second to last element of the path $\hat{\pi}$. If $F \in PhantomVertices$ then the vertex F is a copy of some other vertex in the graph and the phantom vertex F has the pointer to the original based on which it is created. This original vertex is the second to last point on the path $\hat{\pi}$.

2° There is no *active* edge in the graph. In this case there is no path that satisfies the constraint $F_2 \leq M$.

Algorithm 2 Function that checks whether the algorithm has finished and returns the time for travel over the shortest path

```
1: function SHORTESTPATHLENGTHANDTERMINALCONDITIONCHECK
2:     // Returns 0 if the path is not found yet.
3:     // Returns −1 if there is no path with weight smaller than M.
4:     // Returns the weight of the shortest path if it is found.
5:     // A non-zero return value is the indication that the algorithm is over.
6:     #Performed in parallel
7:     if ∃B_0 ∈ B such that B_0 = active then
8:         result ← Time
9:     else
10:        if there are no active vertices then
11:            result ← −1
12:        else
13:            result ← 0
14:    #barrier
15:    return result
```

5.3 Sequences accessible to all processing elements

It is convenient to store the vertices and edges in sequences accessible to all processing elements. We will assume here that the degree of each vertex is bounded above by d.

5.3.1 Vertices

Each vertex takes 5 integers in the sequence of vertices. The first four are name, label, status, and the location of the first edge in the sequence of edges. The fifth element is be used to store a temporary replacement label. Initially, and between algorithm steps, this label is set to -1.

When a first drop of water reaches an inactive vertex V, we say that the vertex is *triggered*, and that state exists only temporarily during an algorithm cycle. In the end of the algorithm cycle some triggered vertices become active. However it could happen that a triggered vertex does not get a water flow of higher quality than the one already present at the vertex. The particular triggered vertex with this property does not get activated.

5.3.2 Edges

Each edge e takes 8 integers in the sequence of edges. Although the graph is undirected, each edge is stored twice in the memory. The 8 integers are the start point, the end point, remaining time for water to travel over the edge (if the edge is active), the weight of the travel $f_2(e)$, the initial passage time $f_1(e)$, the label of the vertex

that is the source of the current flow through the edge (if there is a flow), status, and the location of the same edge in the opposite direction.

5.3.3 Active vertices

The sequence contains the locations of the vertices that are active. This sequence removes the need of going over all vertices in every algorithm step. The locations are sorted in decreasing order. In the end of the sequence we will add triggered vertices that will be joined to the active vertices in the end of the cycle.

5.3.4 Active edges

The role of the sequence is similar to the one of active vertices. The sequence maintains the location of the active edges. Each edge is represented twice in this sequence. The second appearance is the one in which the endpoints are reversed. The locations are sorted in decreasing order. During the algorithm cycle we will append the sequence with triggered edges. In the end of each cycle the triggered edges will be merged to the main sequence of active edges.

5.3.5 Sequence of phantom edges

The phantom edges appear when an active vertex is triggered with a new drop of water. Since the vertex is active we cannot relabel the vertex. Instead each of the edges going from this active triggered vertex need to be doubled with the new source of water flowing through these new edges that are called phantoms. They will disappear once the water finishes flowing through them.

5.3.6 Sequence of elements in B

Elements in B have to be easily accessible for quick check whether the algorithm has finished. For this reason the sequence should be in global memory.

Listing 3 summarizes the initializing procedures.

6 Graph update

The algorithm updates the graph in a loop until one vertex from B becomes active. Each cycle consists of the following nine steps.

Algorithm 3 Initialization procedure

```
1:  procedure INITIALIZATION
2:      for e ∈ E do
3:          State(e) ← passive
4:          Source(e) ← 0
5:          TimeRemaining(e) ← 0
6:      for v ∈ V \ A do
7:          State(v) ← inactive
8:          Label(v) ← 0
9:      for v ∈ A do
10:         State(v) ← active
11:         Label(v) ← M
12:         for e ∈ Edges(v) do
13:             State(e) ← active
14:             Source(e) ← v
15:             TimeRemaining(e) ← f1(e)
16:     TriggeredVertices ← ∅
17:     PhantomVertices ← ∅
18:     PhantomEdges ← ∅
19:     Time ← 0
```

6.1 Step 1: Initial triggering of vertices

In this step we go over all active edges and decrease their time parameters by m, where m is the smallest remaining time of all active edges. If for any edge the time parameter becomes 0, the edge becomes *just used* and its destination triggered.

To avoid the danger of two processing elements writing in the same location of the sequence of active vertices, we have to make sure that each processing element that runs concurrently has pre-specified location to write. This is accomplished by first specifying the number of threads in the separate variable *nThreads*. Whenever kernels are executed in parallel we are using only *nThreads* processing elements. Each processing element has its id number which is used to determine the memory location to which it is allowed to write. The sequence of triggered vertices has to be cleaned after each parallel execution and at that point we take an additional step to ensure we don't list any of the vertices as triggered twice.

6.2 Step 2: Analyzing triggered vertices

For each triggered vertex Q we look at all of its edges that are just used. We identify the largest possible label that can result from one of just used edges that starts from Q. That label will be stored in the sequence of vertices at the position reserved for temporary replacement label. The vertex is labeled as just triggered. If the vertex Q is not active, this label will replace the current label of the vertex in one of the

Algorithm 4 Procedure TriggerVertices

```
1: procedure TRIGGERVERTICES
2:     TriggeredEdges ← ∅
3:     #Performed in parallel
4:     m ← min{TimeRemaining(e) : e ∈ ActiveEdges}
5:     #barrier
6:     Time ← Time + m
7:     #Performed in parallel
8:     for e ∈ ActiveEdges do
9:         TimeRemaining(e) ← TimeRemaining(e) − m
10:        if TimeRemaining(e) = 0 then
11:            State(e) = just used
12:            S_e ← Source(e)
13:            D_e ← TheTwoEndpoints(e) \ {S_e}
14:            TriggeredVertices ← TriggeredVertices ∪ {D_e}
15:    #barrier
```

later steps. If the vertex Q is active, then this temporary label will be used later to construct an appropriate phantom edge.

We are sure that different processing elements are not accessing the same vertex at the same time, because before this step we achieved the state in which there are no repetitions in the sequence of triggered vertices.

Algorithm 5 Analysis of triggered vertices

```
1: procedure ANALYZETRIGGEREDVERTICES
2:     TempLabel ← ∅
3:     #Performed in parallel
4:     for Q ∈ TriggeredVertices do
5:         TempLabel(Q) ← max{Label(P) − f_2(P,Q) : State(P,Q) = just used}
6:     #barrier
```

6.3 Step 3: Gathering input from phantoms

The need to have this step separated from the previous ones is the current architecture of graphic cards that creates difficulties with dynamic memory locations. It is more efficient to keep phantom edges separate from the regular edges. The task is to look for all phantom edges and decrease their time parameters. If a phantom edge gets its time parameter equal to 0, its destination is studied to see whether it should be added to the sequence of triggered vertices. We calculate the new label that the vertex would receive through this phantom. We check whether this new label is higher than the currently known label and the temporary label from possibly previous triggering of the vertex. The phantoms will not result in the concurrent

writing to memory locations because each possible destination of a phantom could have only one edge that has time component equal to 0.

Algorithm 6 Input from phantoms

```
1: procedure GETINPUTFROMPHANTOMS
2:     #Performed in parallel
3:     Decrease time parameters of fantom edges (as in Listing 4)
4:     Trigger the destinations of phantom edges (as in Listing 4)
5:     #barrier
6:     #Performed in parallel
7:     Analyze newly triggered vertices, in a way similar to Listing 5
8:     #barrier
```

6.4 Step 4: Triggering edges

In this step we will analyze the triggered vertices and see whether each of their neighboring edges needs to change the state. Triggered vertices are analyzed using separate processing elements. A processing element analyzes the vertex Q in the following way.

Each edge j of Q will be considered triggered if it can cause the other endpoint to get better label in future through Q. The edge j is placed in the end of the sequence of active edges.

Algorithm 7 Procedure that triggers the edges

```
1: procedure TRIGGEREDGES
2:     #Performed in parallel
3:     for Q ∈ TriggeredVertices do
4:         for P ∈ Neighbors(Q) do
5:             if TempLabel(Q) − f2(P,Q) > Label(P) then
6:                 State(P,Q) ← active
7:                 TriggeredEdges ← TriggeredEdges ∪ {(P,Q)}
8:     #barrier
```

6.5 Step 5: Treatment of triggered edges

Consider a triggered edge j. We first identify its two endpoints. For the purposes of this step we will identify the endpoint with the larger label, call it the source, and denote by S. The other will be called the destination and denoted by D. In the end of the cycle, this vertex S will become the source of the flow through j.

Notice that at least one of the endpoints is triggered. If only one endpoint is triggered, then we are sure that this triggered endpoint is the one that we designated as the source S.

We then look whether the source S was active or inactive before it was triggered.

6.5.1 Case in which the source S was inactive before triggering

There are several cases based on the prior status of j. If j was passive, then it should become active and no further analysis is necessary. If it was used or just used, then it should become active and the time component should be restored to the original one. Assume now that the edge j was active. Based on the knowledge that S was inactive vertex we can conclude that the source of j was D. However we know that the source of j should be S and hence the time component of j should be restored to the backup value.

Consequently, in the case that S was inactive, regardless of what the status of j was, we are sure its new status must be active and its time component can be restored to the original value. This restoration is not necessary in the case that j was passive, although there is no harm in doing it.

If the edge j was not active before, then the edge j should be added to the list of active edges. If the edge j was active before, then it should be removed from the list of triggered edges because all triggered edges will be merged into active edges. The edge j already appears in the list of active edges and need not be added again.

6.5.2 Case in which the source S was active before triggering

In this case we create phantom edges. Each such triggered edge generates four entries in the phantom sequence. The first one is the source, the second is the destination, the third is the label of the source (or the label stored in the temporary label slot, if higher), and the fourth is the original passage time through the edge j.

6.6 Step 6: Checking terminal conditions

In this step we take a look whether a vertex from B became active or if there are no active edges. These would be the indications of the completion of the algorithm. The function that checks the terminal conditions is presented earlier in Listing 2.

Algorithm 8 Treatment of triggered edges

```
1: procedure TREATTRIGGEREDEDGES
2:     #Performed in parallel
3:     for j ∈ TriggeredEdges do
4:         S ← The endpoint of j with larger label
5:         D ← The endpoint of j with smaller label
6:         OldStateOfS ← State(S)
7:         OldStateOfJ ← State(j)
8:         if OldStateOfS = inactive then
9:             State(j) ← active
10:            Source(j) ← S
11:            TimeRemaining(j) ← f1(j)
12:        if OldStateOfS = active then
13:            Create a phantom vertex S′ and connect it to D
14:            TimeRemaining(S′,D) ← f1(S,D)
15:    #barrier
```

6.7 Step 7: Final treatment of phantoms

In this step we go once again over the sequence of phantoms and remove each one that has its time parameter equal to 0.

Algorithm 9 Final treatment of phantoms

```
1: procedure FINALTREATMENTOFPHANTOMS
2:     #Performed in parallel
3:     for j ∈ PhantomEdges do
4:         if TimeRemaining(j) = 0 then
5:             Remove j and its source from the sequence of phantoms
6:     #barrier
```

6.8 Step 8: Final treatment of vertices

In this step of the program the sequence of active vertices is updated so it contains new active vertices and looses the vertices that may cease to be active.

6.8.1 Preparation of triggered vertices

For each triggered vertex Q we first check whether it was inactive before. If it was inactive then its label becomes equal to the label stored at the temporary storing location in the sequence of vertices. If it was active, its label remains unchanged.

The phantoms were created and their labels are keeping track of the improved water quality that has reached the vertex Q.

We may now clean the temporary storing location in the sequence of vertices so it now contains the symbol for emptiness (some pre-define negative number).

6.8.2 Merging triggered with active vertices

Triggered vertices are now merged to the sequence of active vertices.

6.8.3 Check active vertices for potential loss of activity

For each active vertex Q look at all edges from Q. If there is no active edge whose source is Q, then Q should not be active any longer.

6.8.4 Condensing the sequence of active vertices

After previous few steps some vertices may stop being active in which case they should be removed from the sequence.

Algorithm 10 Final treatment of vertices

```
1:  procedure FINALTREATMENTOFVERTICES
2:      #Performed in parallel
3:      for Q ∈ TriggeredVertices do
4:          if State(Q) = inactive then
5:              State(Q) ← active
6:              Label(Q) ← TempLabel(Q)
7:      #barrier
8:      TempLabel ← ∅
9:      #Performed in parallel
10:     Merge triggered vertices to active vertices
11:     #barrier
12:     #Performed in parallel
13:     for Q ∈ TriggeredVertices do
14:         if there are no active edges starting from Q then
15:             State(Q) ← inactive
16:     #barrier
```

6.9 Step 9: Final treatment of active edges

We first need to merge the triggered edges with active edges. Then all just used edges have to become used and their source has to be re-set so it is not equal to any of the endpoints. Those used edges should be removed from the sequence of active edges.

The remaining final step is to condense the obtained sequence so there are no used edges in the sequence of active edges.

Algorithm 11 Final treatment of active edges

```
1: procedure FINALTREATMENTOFACTIVEEDGES
2:     #Performed in parallel
3:     Merge triggered edges to active edges
4:     #barrier
5:     #Performed in parallel
6:     Transform all just used into used and erase their Source components
7:     #barrier
```

7 Large sets of active vertices

In this section we will prove that it is possible for the set of active vertices in dimension 2 to contain more than $O(n)$ elements. We will construct examples in the case when the time to travel over each vertex is from the set $\{1,2\}$ and when $M = +\infty$.

We will consider the subgraph $V_n = [-n,n] \times [0,n]$ of $\mathbb{Z}^2$. At time 0 the water is located in all vertices of the x axis. For sufficiently large n we will provide an example of configuration ω of passage times for the edges of the graph V_n such that the number of active vertices at time n is of order $n \log n$. This would establish a lower bound on the probability that the number of active vertices at time t is large.

Let us assume that each edge of the graph has the time component assigned from the set $\{1,2\}$ independently from each other. Assume that the probability that 1 is assigned to each edge is equal to p, where $0 < p < 1$.

Theorem 1. *There exists $t_0 \geq 0$, $\mu > 0$, and $\alpha > 0$ such that for each $t > t_0$ there exists n such that the number A_t of active vertices at time t in the graph V_n satisfies*

$$\mathbb{P}(A_t \geq \alpha t \log t) \geq e^{-\mu t^2}.$$

To prepare for the proof of the theorem we first study the evolution of the set of active edges in a special case of a graph. Then we will construct a more complicated graph where the set of active edges will form a fractal of length $t \log t$.

Lemma 1. *If all edges on the y-axis have time parameter equal to 1 and all other edges have their time parameter equal to 2, then at time T the set of active vertices*

is given by

$$A_T = \{(0,T)\} \cup \{(0,T-1)\} \cup \bigcup_{k=1}^{\lfloor \frac{T+1}{4} \rfloor} \{(-k,T-2k),(k,T-2k)\}$$
$$\cup \bigcup_{z \in \mathbb{Z} \setminus \{-\lfloor \frac{T+1}{4} \rfloor, \ldots, \lfloor \frac{T+1}{4} \rfloor\}} \left\{ \left(z, \left\lfloor \frac{T}{2} \right\rfloor \right) \right\}.$$

Proof. After $T-2k$ units of time the water can travel over the path γ_k that consists of vertices $(0,0)$, $(0,1)$, …, $(0,T-2k)$. In additional $2k$ units of time the water travels over the path γ_k' that consists of vertices $(0,T-2k)$, $(1,T-2k)$, …, $(k,T-2k)$. Consider any other path that goes from x axis to the point $(k,T-2k)$ for some fixed

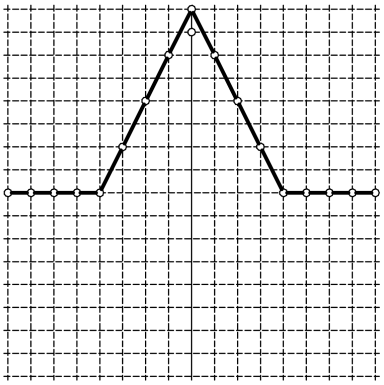

Fig. 5 The active edges at time T.

$k \leq \lfloor \frac{T+1}{4} \rfloor$. If the path takes some steps over edges that belong to y axis then it would have to go over at least k horizontal edges to reach y axis, which would take $2k$ units of time. The path would have to take at least $T-2k$ vertical edges, which would take at least $T-2k$ units of time. Thus the travel would be longer than or equal to T.

However, if the path does not take steps over the edges along y axis then it would have to take at least $T-2k$ steps over edges that have passage time equal to 2. This would take $2(T-2k) = 2T-4k$ units of time. If $T+1$ is not divisible by 4, then $k < \frac{T+1}{4}$ and

$$2T-4k > 2T-T-1 = T-1,$$

which would mean that the travel time is at least T. If $T+1$ is divisible by 4 and $k = \lfloor \frac{T+1}{4} \rfloor$ then the vertical path would reach $(k,T-2k)$ at time $T-1$. However,

the vertex $(k, T-2k)$ would still be active because the water would not reach $(k+1, T-2k)$ which is a neighbor of $(k, T-2k)$.

Let us denote by N_t the number of active vertices at time t whose x coordinate is between $-t$ and t,

$$N_t = \left\{(x,y) \in \{-t, -t+1, \ldots, t-1, t\} \times \mathbb{Z}_0^+ : (x,y) \text{ is active at time } t\right\}.$$

Theorem 2. *There exist real numbers α and $t \geq 0$ and an environment ω for which*

$$N_t(\omega) \geq \alpha t \log t.$$

Proof. Assume that $t = 2^k$ for some $k \in \mathbb{N}$. Let us define the following points with their coordinates $T = (0,t)$, $L = \left(-\frac{t}{2}, 0\right)$, and $O = \left(0, \frac{t}{2}\right)$. We will recursively construct the sequence of pairs $(\omega_1, \mathscr{I}_1)$, $(\omega_2, \mathscr{I}_2)$, …, $(\omega_k, \mathscr{I}_k)$ where ω_j is an assignment of passage times to the edges and $\mathscr{I}_j$ is a subgraph of $\mathbb{Z}^2$. This subgraph will be modified recursively. All edges in $\mathscr{I}_j$ have passage times equal to 2 in the assignment ω_j. Having defined the pair $(\omega_j, \mathscr{I}_j)$ we will improve passage times over some edges in the set $\mathscr{I}_j$ by changing them from 2 to 1. This way we will obtain a new environment ω_{j+1} and we will define a new set $\mathscr{I}_{j+1}$ to be a subset of $\mathscr{I}_j$. The new environment ω_{j+1} will satisfy

$$N_t(\omega_{j+1}) \geq N_t(\omega_j) + \beta t,$$

for some $\beta > 0$.

Let us first construct the pair $(\omega_1, \mathscr{I}_1)$. We will only construct the configuration to the left of the y axis and then reflect it across the y axis to obtain the remaining configuration.

All edges on the y axis have the passage times equal to 1, and all edges on the segment LO have the passage times equal to 1. All other edges have the passage times equal to 2. Define $\mathscr{I}_1 = \triangle LOT$. Then the polygonal line LYT contains the active vertices whose x coordinate is between $-t$ and 0.

The environment ω_2 is constructed in the following way. Let us denote by L_0 and T_0 the midpoints of LO and TO. Let X be the midpoint of LT. We change all vertices on L_0X and T_0X to have the passage time equal to 1. We define $\mathscr{I}_2 = \triangle LL_0X \cup \triangle XT_0T$.

Let L_1 and L_2 be the midpoints of LL_0 and L_0O and let L' and L'' be the intersections of XL_1 and XL_2 with LY. The points T' and T'' are defined in an analogous way: first T_1 and T_2 are defined to be the midpoints of TT_0 and OT_0 and T' and T'' are the intersections of XT_1 and XT_2 with TY.

The polygonal line $LL'XL''YT''XT'T$ is the set of active edges that are inside the triangle LOT. The following lemma will allow us to calculate $N_t(\omega_2) - N_t(\omega_1)$.

Lemma 2. *Let Λ and λ denote the lengths of the polygonal lines $LL'XL''YT''XT'T$ and LYT respectively. If t is the length of OT then*

$$\Lambda = \lambda + \frac{4}{3\sqrt{5}}t.$$

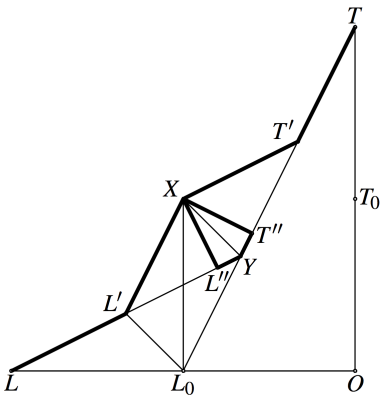

Fig. 6 The set of active edges in configuration ω_2.

Proof. It suffices to prove that $LL'+L'X+XL''+L''Y = LY+\frac{2}{3\sqrt{5}}t$. From the similarities $\triangle LL_0X \sim \triangle LOT$ and $\triangle LL_0L' \sim LOX$ we have that $L_0L' \| OX$. Therefore L' is the midpoint of LY and $LY = LL'+L'Y = LL'+L'X$. It remains to prove that $XL''+L''Y = \frac{2}{3\sqrt{5}}t$. From

$$\angle L_0XL'' = \angle L'XL_0 = \angle L_0LL''$$

we conclude that the quadrilateral $LL_0L''X$ is inscribed in a circle. The segment LX is a diameter of the circle hence $\angle LL''X = \angle LL_0X = 90^\circ$. We also have $\angle L''XY = \angle L_0XY - \angle L_0XL'' = 45^\circ - \angle OLT_0 = 45^\circ - \arctan\frac{1}{2}$.

The point Y is the centroid of $\triangle LOT$ hence $XY = \frac{1}{3}XO = \frac{1}{3\sqrt{2}}t$. Therefore

$$\begin{aligned}
XL''+L''Y &= XY\cos\left(45^\circ-\arctan\frac{1}{2}\right)+XY\sin\left(45^\circ-\arctan\frac{1}{2}\right)\\
&= \frac{\cos\left(45^\circ-\arctan\frac{1}{2}\right)+\sin\left(45^\circ-\arctan\frac{1}{2}\right)}{3\sqrt{2}}t\\
&= \frac{\cos\left(45^\circ-\arctan\frac{1}{2}\right)\cos 45^\circ+\sin\left(45^\circ-\arctan\frac{1}{2}\right)\sin 45^\circ}{3}t\\
&= \frac{\cos\left(45^\circ-\arctan\frac{1}{2}-45^\circ\right)}{3}t = \frac{\cos\left(\arctan\frac{1}{2}\right)}{3}t\\
&= \frac{2}{3\sqrt{5}}t.
\end{aligned}$$

The number of edges on each of the segments of the polygonal lines we obtained is equal to $\frac{u}{\sqrt{5}}$, where u is the length of the segment. Using this fact with the previous lemma applied to both $\triangle LOT$ and its reflection along OT gives us

$$N_t(\omega_2) - N_t(\omega_1) = \frac{4}{3\sqrt{5}}t \cdot \frac{1}{\sqrt{5}} = \frac{4}{15}t.$$

We now continue in the same way and in each of the triangles LL_0X and XT_0T we

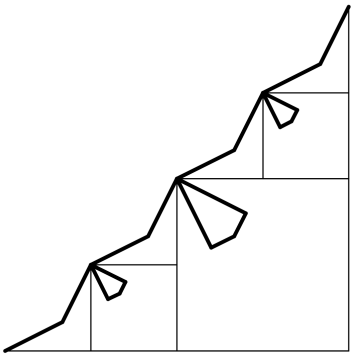

Fig. 7 The set of active edges in configuration ω_3.

perform the same operation to obtain ω_3 and $\mathscr{I}_3$. Since the side length of LL_0X is $\frac{t}{2}$, the increase in the number of elements in the new set of active vertices is $\frac{2}{15} \cdot \frac{t}{2}$. However, this number has to be now multiplied by 4 because there are 4 triangles to which the lemma is applied: $\triangle LL_0X$, $\triangle XT_0T$, and the reflections of these two triangles with respect to OT. Therefore the increase in the number of active vertices is $N_t(\omega_3) - N_t(\omega_2) = 4 \cdot \frac{2}{15} \cdot \frac{t}{2} = \frac{4}{15}t$.

This operation can be repeated k times and we finally get that

$$N_t(\omega_k) = N_t(\omega_1) + (k-1) \cdot \frac{4}{15}t \geq k \cdot \frac{4}{15}t.$$

Thus the theorem holds if we set $\alpha = \frac{4}{15\log 2}$.

Proof (Proof of Theorem 1). Recall that p is the probability that the time 1 is assigned to each edge. Let $\rho = \min\{p, 1-p\}$. The configuration provided in the proof of Theorem 2 has its probability greater than or equal to ρ^{t^2}. Therefore

$$P(A_t \geq \alpha t \log t) \geq \rho^{t^2} = e^{t^2 \ln \rho}.$$

Therefore we may take $\mu = -\ln \rho$.

8 Performance analysis

The algorithm was implemented in C++ and OpenCL. The hardware used has a quad core Intel i5 processor with clock speed of 3.5GHz and AMD Radeon R9 M290X graphic card with 2 gigabytes of memory. The graphic card has 2816 processing elements.

The table provides a comparison of the performance of the algorithm on 4 samples of three dimensional cubes with edges of lengths 50, 75, 100, and 125. The initial configuration for each of the graphs assumes that there is water on the boundary of the cube, while the set B is defined to be the center of the cube. The same program was executed on graphic card and on CPU.

Graph	GPU time (s)	CPU time (s)
$50 \times 50 \times 50$	3	10
$75 \times 75 \times 75$	8	61
$100 \times 100 \times 100$	21	275
$125 \times 125 \times 125$	117	1540

The graph that corresponds to the cube $100 \times 100 \times 100$ has 1000000 vertices and 2970000 edges, while the graph corresponding to the cube $125 \times 125 \times 125$ has 1953125 vertices and 5812500 edges.

9 Conclusion

The algorithm described in this chapter solves the constrained shortest path problem using parallel computing. It is suitable to implement on graphic cards and CPUs that have large number of processing elements. The algorithm is implemented in C++ and OpenCL and the parallelization improves the speed tenfold.

The main idea is to follow the percolation of water through the graph and assign different qualities to drops that travel over different edges. Each step of the algorithm corresponds to a unit of time. It suffices to analyze only those vertices and edges through which the water flows. We call them active vertices and active edges. Therefore, the performance of the algorithm is tied to the sizes of these active sets.

Theorem 1 proves that it is possible to have at time t an active set of size $O(t \log t)$. The proof of the theorem relied on constructing one such set. It is an open problem to find the average size of the active set at time t.

Problem 1. If the weights and travel times of the edges are chosen independently at random, what is the average size of the active set at time t?

At some stages of the execution, the program needs additional memory to store phantom edges in the graph. It would be interesting to know how many phantom edges are allocated during a typical execution. This can be formally phrased as an open problem.

Problem 2. If the weights and travel times of the edges are chosen independently at random, what is the average number of phantoms that need to be created during the execution of the algorithm?

Acknowledgements

The author was supported by PSC-CUNY grants #68387 – 0046, #69723 – 0047 and Eugene M. Lang Foundation.

References

1. G. Amir, I. Corwin, and J. Quastel. Probability distribution of the free energy of the continuum directed random polymer in 1+1 dimensions. *Comm. Pure Appl. Math.*, 64:466–537, 2011.
2. S. Armstrong, H. Tran, and Y. Yu. Stochastic homogenization of a nonconvex Hamilton–Jacobi equation. *Calc. Var. Partial Differential Equations*, 54:1507–1524, 2015. (Submitted) arXiv:1311.2029.
3. M. Benaim and R. Rossignol. Exponential concentration for first passage percolation through modified poincaré inequalities. *Ann. Inst. Henri Poincaré Probab. Stat.*, 44(3):544–573, 2008.
4. I. Benjamini, G. Kalai, and O. Schramm. First passage percolation has sublinear distance variance. *The Annals of Probability*, 31(4):1970–1978, 2003.
5. N. Boland, J. Dethridge, and I. Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations research letters*, 34:58–68, 2006.
6. S. Chatterjee and P. S. Dey. Central limit theorem for first-passage percolation time across thin cylinders. *Probability Theory and Related Fields*, 156(3):613–663, 2013.
7. J. T. Cox and R. Durrett. Some limit theorems for percolation processes with necessary and sufficient conditions. *The Annals of Probability*, 9(4):583–603, 1981.
8. M. Damron and M. Hochman. Examples of nonpolygonal limit shapes in i.i.d. first-passage percolation and infinite coexistence in spatial growth models. *The Annals of Applied Probability*, 23(3):1074–1085, 2013.
9. M. Desrochers, J. Desrosiers, and M. Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, 40(2):342–354, 1992.
10. J. Hammersley and D. Welsh. First-passage percolation, subadditive processes, stochastic networks, and generalized renewal theory. *Bernoulli-Bayes-Laplace Anniversary Volume*, 1965.
11. S. Irnich and G. Desaulniers. Shortest path problems with resource constraints. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, GERAD 25th Anniversary Series, pages 33–65. Springer, 2005.
12. E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. *Lecture notes in computer science*, 3503:126–138, 2005.
13. E. Kosygina, F. Rezakhanlou, and S. R. S. Varadhan. Stochastic homogenization of Hamilton–Jacobi–Bellman equations. *Comm. Pure Appl. Math.*, 59(10):1489–1521, 2006.
14. E. Kosygina, F. Yilmaz, and O. Zeitouni. Nonconvex homogenization of a class of one-dimensional stochastic viscous Hamilton-Jacobi equations. *in preparation*, 2017.
15. J. Krug and H. Spohn. Kinetic roughening of growing surfaces. *Solids far from equilibrium*, pages 412–525, 1991.

16. X.-Y. Li, P.-J. Wan, Y. Wang, and O. Frieder. Constrained shortest paths in wireless networks. *IEEE MilCom*, pages 884–893, 2001.
17. L. Lozano and A. L. Medaglia. On an exact method for the constrained shortest path problem. *Computers and Operations Research*, 40(1):378–384, 2013.
18. I. Matic. Parallel algorithm for constrained shortest path problem in C++/OpenCL. `github.com/maticivan/parallel_constrained_shortest_path`
19. I. Matic and J. Nolen. A sublinear variance bound for solutions of a random Hamilton–Jacobi equation. *J. Stat. Phys.*, 149:342–361, 2012.
20. K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. In *Lecture Notes in Computer Science*, volume 1879, pages 326–337, 2000.
21. S. Misra, N. E. Majd, and H. Huang. Approximation algorithms for constrained relay node placement in energy harvesting wireless sensor networks. *IEEE Transactions on Computers*, 63(12):2933–2947, 2014.
22. R. Muhandiramge and N. Boland. Simultaneous solution of lagrangean dual problems interleaved with preprocessing for the weight constrained shortest path problem. *Networks*, 53:358–381, 2009.
23. C. M. Newman and M. S. T. Piza. Divergence of shape fluctuations in two dimensions. *The Annals of Probability*, 23(3):977–1005, 1995.
24. F. Rezakhanlou. Central limit theorem for stochastic Hamilton–Jacobi equations. *Commun. Math. Phys.*, 211:413–438, 2000.
25. T. Sasamoto and H. Spohn. One-dimensional kardar-parisi-zhang equation: An exact solution and its universality. *Phys. Rev. Lett.*, 104, 2010.
26. P. E. Souganidis. Stochastic homogenization of Hamilton–Jacobi equations and some applications. *Asymptot. Anal.*, 20(1):1–11, 1999.