Memory-Constrained Algorithms for Shortest Path Problems

Tetsuo Asano[*] Benjamin Doerr[†]

Abstract

We present an algorithm computing a shortest path between to vertices in a square grid graph with edge weights that uses memory less than linear in the number of vertices (apart from that for storing in the input). For any $\varepsilon > 0$, our algorithm uses a work space of $O(n^{(1/2)+\varepsilon})$ words and runs in $O(n^{O(1/\varepsilon)})$ time.

1 Introduction

It is well known that given a weighted graph of n vertices, the shortest path between any two vertices can be computed in $O(n^2)$ time using work space of $O(n)$ words in addition to arrays keeping graph information, whose total size is $O(m+n)$, where m is the number of edges and $m = O(n^2)$ in general. This is achieved by the original version of Dijstra's algorithms [1].

It is known that the shortest path problem is NL-complete [2]. In other words, it seems hopeless to have an algorithm for the problem using work space of $O(1)$ words of $O(\log n)$ bits.

What happens if we allow a larger, but sublinear work space? Surprisingly, nothing is known for this question as far as the authors know. In this first work on this problem we prove that there is a sublinear-space algorithm for computing the shortest path in a grid graph of size $\sqrt{n} \times \sqrt{n}$. Our algorithm uses a work space of $O(n^{(1/2)+\varepsilon})$ words and runs in $O(n^{O(1/\varepsilon)})$ time for any fixed $\varepsilon > 0$.

2 Computing the Shortest Path Distance

We first present a space-efficient algorithm for computing the length of the shortest path in a grid graph of size $\sqrt{n} \times \sqrt{n}$ where the source and target vertices are located at the lower left corner and upper right corner of the grid, respectively. Once we know an algorithm for computing the shortest path distance, we can report the shortest path by repeatedly applying the algorithm. Throughout the paper we assume that the length of a word is long enough to keep the shortest path distance for any vertex in a given graph.

[*]School of Information Science, JAIST, Japan, t-asano@jaist.ac.jp

[†]Max-Planck-Institut für Informatik, Germany, doerr@mpi-inf.mpg.de

Let $G = (V|E)$ be a square grid graph of size $\sqrt{n}\times\sqrt{n}$. For simplicity, we assume that n is a square number and thus $\sqrt{n}$ is an integer. Two vertices are neighbors if their L_1-distance is one. All edges have positive weights.

First we decompose the grid graph G into $k \times k$ small square grid graphs called "block graphs" $S_1(V_1, E_1), S_2(V_2, E_2), \ldots, S_{k^2}(V_{k^2}, E_{k^2})$ of the same size $(\sqrt{n}/k) \times (\sqrt{n}/k)$. These block graphs are ordered arbitrarily as far as every block graph appears exactly once in the order. The edge sets of these graphs form a partition of E, i.e., we have

$E_i \cap E_j = \emptyset$, for any $i \neq j$, and

$E_1 \cup E_2 \cup \cdots \cup E_{k^2} = E$.

Each vertex set V_i has $O(\sqrt{n}/k)$ boundary vertices which may be common to some other vertex sets, and $O(n/k^2)$ inner vertices which are contained only in the vertex set V_i. Since there are k^2 squares, the total number of boundary vertices is $O(k^2 \times \sqrt{n}/k) = O(k\sqrt{n})$.

Figure 1 shows an example of a grid graph and its decomposition into 6×6 squares together with a shortest path from the lower left corner to the upper right corner. As is seen in the figure, a shortest path may visit a square many times.

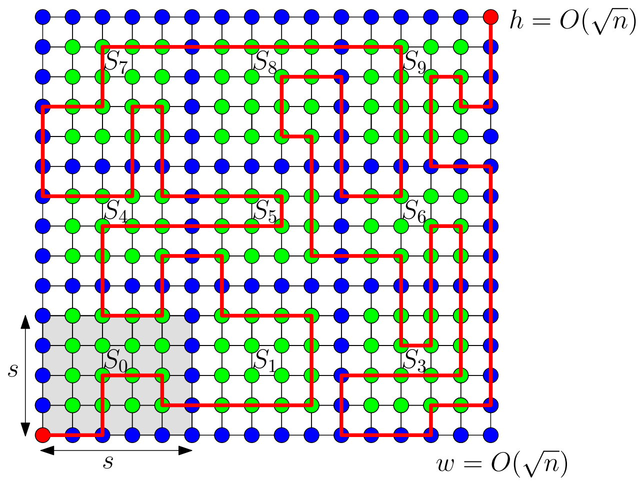

Figure 1: An example of a grid graph of size $k \times k$ and its decomposition into subgraphs called squares. The shortest path from the lower left corner to the upper right corner is also shown.

We are now ready to describe a basic algorithm for computing the shortest path distance between two arbitrarily specified vertices in a given grid graph. We first assume that a source vertex is located at the lower left corner of the grid and a target vertex at the upper right corner. This constraint is removed later.

We execute a simple implementation of Dijkstra's algorithm [1] for small squares again and again. For the implementation we need an array for storing temporary distances from the source vertex. We use two different arrays for this purpose. One is an array C to keep a distance from the source vertex to each boundary vertex in the entire graph G. Its size is $O(k\sqrt{n})$. The array is maintained during the entire algorithm.

The other is an array T to keep a temporary distance from the source vertex (of the whole grid) to each vertex in a square including boundary and inner vertices of the square. It is used for a one-shot implementation of Dijkstra's algorithm for a small square (plus the source vertex of the whole grid). Since each square has the same shape, we can use the array again and again for different squares.

Now, we begin with the lower left square S_1 with the source vertex s in it. After initializing the two arrays C and T with infinity, we set $C[s] = 0$ for the source vertex s. Then, using the array T for all vertices in S_1, we apply a simple implementation of Dijkstra's shortest path finding algorithm to the square S_1. It runs in time quadratic in the number of vertices, that is, in $O(n^2/k^4)$ time. As post-process, we transfer distances of all the boundary vertices of S_1 to the common array C.

Then, we move to the next square S_2. The initialization step is to transfer distances of boundary vertices of S_2 stored in the common array C to the temporary array T. Distance values for all inner vertices of S_2 are initialized to infinity. Then, we implement Dijkstra's algorithm to the square. As is well known, this algorithm computes a shortest path tree which includes every shortest path from a single source vertex to all other vertices. If you needed a source vertex, you could define an imaginary source vertex and imaginary edges from it to all the boundary vertices of the square whose weights are given by the current distance values of those vertices.

After the square S_2, we move to S_3, S_4, and so on until the last square S_m. We call the entire process stated above *"a scan over the grid graph."*

What can we expect after a scan over the graph? Let P be the shortest path. The path P passes through a number of squares. Suppose P passes through $S_1 = S_{\sigma_1}, S_{\sigma_2}, \ldots, S_{\sigma_L}$ in this order. For example, the shortest path in Figure 1 is characterized by a sequence $(S_1, S_2, S_5, S_4, S_5, S_4, S_7, S_4, S_7, S_8, S_9, S_6, S_9, S_8, S_5, S_6, S_3, S_6, S_3, S_6, S_9)$.

How long is the sequence? There are $O(k^2)$ squares and each square is visited by the sequence at most $O(\sqrt{n}/k)$ times (because this is the number of boundary vertices of a square). Since the shortest path must be simple, we can conclude that the length L of the path P is bounded by $O(k^2) \times O(\sqrt{n}/k) = O(k\sqrt{n})$. By induction, we also see that after the i-th scan over the graph, we have computed the shortest path distance up to all boundary vertices of $S_{\sigma i}$. Thus, the shortest path distance must have been computed after $O(k\sqrt{n})$-th scan. Each scan is done in $O((n/k^2)^2 \times k^2) = O(n^2/k^2)$ time using work space of $O(k\sqrt{n} + n/k^2)$ words.

Algorithm 1 is a formal description of this basic procedure.

Theorem 1 *Given a grid graph of size $O(\sqrt{n}) \times O(\sqrt{n})$ with positive edge weights, we can compute the shortest path distance from the lower left corner to the upper right corner of the grid in $O(n^{2+1/2}/k)$ time using work space of $O(k\sqrt{n} + n/k^2)$ words.*

Of course, in an actual implementation of the algorithm we would stop after a whole scan over the grid graph did not result in improving any shortest path distances to boundary vertices.

It is not so hard to extend the algorithm so that it finds the shortest path distance for any two vertices as far as both of them are boundary vertices. We could also extend it to allow inner vertices as source and target vertices.

The work space is minimized to $O(n^{2/3})$ when $k\sqrt{n} = n/k^2$, that is, when $k = n^{1/6}$.

3 Reporting the Shortest Path

Once we have computed the shortest path distance $d(s,t)$ from s to t, we can report the shortest path. When we have computed $d(s,t)$, we have also computed the shortest path distance $d(s, v_i)$ for every boundary vertex v_i not only in the last square S_m, but also in all other squares. Keeping the distances in yet another array D of size $O(n^{2/3})$, we execute Dijkstra's algorithm to the square S_m with the upper right corner vertex t as a new source vertex. Now, for each boundary vertex v_i we have two distances, the global shortest path distance $D(s, v_i)$ from s to v_i and the shortest path distance $d(t, v_i)$ from t to v_i within the square S_m. We choose the boundary vertex v_i of the largest value of $d(v_i, t)$ such that

$D(s, v_i) + d(v_i, t) = d(s,t)$.

Since Dijkstra's algorithm finds all shortest paths from t within the square, we can report the shortest path from t to v_i as the last part of the entire shortest path from s to t.

Our next target is the boundary vertex v_i. To report the shortest path from s to v_i, we repeat the same process again with v_i as a new target vertex instead of t within the square containing v_i which is adjacent to the previous square. Forgetting everything we apply the same algorithm from the scratch again to compute the

Algorithm 1: Basic Algorithm for Computing the Shortest Path Distance between Two Vertices in a Grid Graph.

```
Input: A grid graph $G$ defined by a $\sqrt{n} \times \sqrt{n}$ grid
       with weighted edges, assuming $\sqrt{n}$ is an integer.
Output: The shortest path distance from a source vertex $s$ located at the lower left corner
        to a target vertex $t$ at the upper right corner.
Decompose the grid into $k \times k$ squares $S_1, S_2, \dots, S_{k^2}$ of the same size.;
Let $V_i$ be a set of vertices in a square $S_i$ for each $i = 1, \dots, k^2$.;
// Assume each $V_i$ contains exactly $\sqrt{n}/k \times \sqrt{n}/k = n/k^2$ vertices.;
Let $B_i$ be a set of vertices of $V_i$ that lie on the boundary of $S_i$ (those vertices shared with other squares), called boundary vertices of $S_i$.;
// The number of boundary vertices of each square is $O(\sqrt{n}/k)$.;
Let $B = B_1 \cup \cdots B_{k^2}$ be the set of all boundary vertices.;
// The total number of boundary vertices is $O(k\sqrt{n})$.;
Define an array $C[\,]$ for distances to boundary vertices.;
Define an array $T[\,]$ for distances to vertices in a square.;
for each boundary vertex $v_i$ do
    $C[i] = \infty$.
$C[0] = 0$. // $v_0$ is the source vertex $s$.;
for round = 1 to $k\sqrt{n}$ do
    for each Square $S_i$ do
        for each boundary vertex $v_{i,j} = v_p$ in $S_i$ do
            $T[j] = C[p]$.
        for each inner vertex $v_{i,j}$ in $S_i$ do
            $T[j] = \infty$.
        // Dijkstra's algorithm;
        while there is an unselected vertex in $S_i$ do
            Choose a vertex $v_{i,p}$ such that $T[p] > 0$ and $T[p]$ is smallest.;
            // The source vertex $s$ should be treated exceptionally. for each vertex $v_{i,q}$ in $S_i$ adjacent to $v_{i,p}$ do
                if $T[q] > T[p] + w(v_{i,p}, v_{i,q})$ then
                    $T[q] = T[p] + w(v_{i,p}, v_{i,q})$.;
            $T[p] = -T[p]$. // Mark the vertex
        // Transfer the results into the common array $C$.;
        for each boundary vertex $v_{i,j} = v_p$ in $S_i$ do
            $C[p] = -T[j]$.
return the shortest path distance $C[t]$ of the target vertex $v_t$ at the upper right corner.
```

shortest path distance from s to v_i. In a similar way, we can report the shortest path from v_i to the next intermediate vertex v_j on the shortest path from v_i to s.

Theorem 2 *Given a grid graph of size $O(\sqrt{n}) \times O(\sqrt{n})$ with positive edge weights, we can output the shortest path between any two vertices on the grid in $O(n^3)$ time using work space of $O(k\sqrt{n} + n/k^2)$ words for any value of k with $2 \le k \le n/2$.*

Proof. We only show the time complexity of the algorithm described above. Again, the number of iterations is bounded by $O(k\sqrt{n})$. Since each iteration is done in $O((n^{2+1/2}/k)$ time, the total time complexity is $O(n^3)$. □

3.1 Some Generalizations

We have assumed that source and target vertices are fixed to two corner vertices of the grid. It is rather easy to remove this constraint. Suppose a source vertex s is an inner vertex of a square S_i. Then, we do nothing for the squares $S_1, \dots, S_{i-1}$ in the first scan over the given grid graph. Then, we execute Dijkstra's algorithm to the square S_i after initializing the distance for the inner vertex s as 0. Then, we can correctly compute distances from s to all the boundary vertices of S_i within the square. It is just the same for a target vertex. Thus, just small modifications are enough to adapt the previous algorithm to apply for general cases where source and target vertices are arbitrarily specified.

4 Reducing the work space

We have shown that the shortest path distance between any two vertices in a grid graph of size $\sqrt{n} \times \sqrt{n}$ can be computed in $O(n^{2+1/2}/k)$ time using work space of $O(k\sqrt{n} + n/k^2)$ words after decomposing the grid into $k \times k$ equal small squares. We have also shown that the work space is minimized to $O(n^{2/3})$ when $k = n^{1/6}$. Is it possible to reduce the work space? Our answer is Yes.

A basic idea is to introduce recursion. Given a grid graph of size $\sqrt{n} \times \sqrt{n}$, we decompose it into $k \times k$ squares of equal dimensions. We further decompose each square into $k \times k$ small squares of equal dimensions.

At the top level, we have k^2 squares (called level-1 squares) of dimensions $\sqrt{n}/k \times \sqrt{n}/k$. The number of boundary vertices of each level-1 square is $O(\sqrt{n}/k)$. Thus, the total number of boundary vertices at level 1 is $O(k\sqrt{n})$.

Each level-1 square is decomposed into k^2 small squares (level-2 squares) of dimensions $\sqrt{n}/k^2 \times \sqrt{n}/k^2$. There are $O(\sqrt{n}/k^2)$ boundary vertices in each level-2 square, and thus the total number of boundary vertices at level 2 in a level-1 square is $O(\sqrt{n})$. On the other hand, the number of inner vertice in each level-2 square is $O(n/k^4)$.

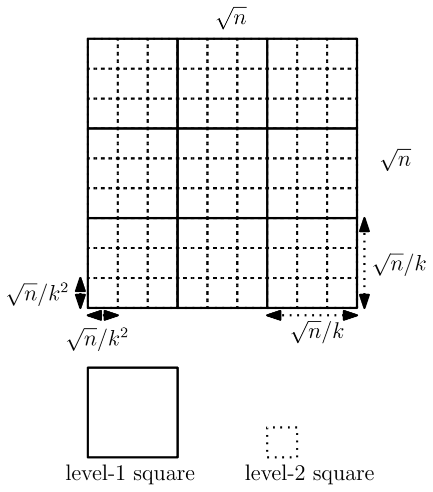

Figure 2: Hierarchical Decomposition of a grid graph. An example of a two-level decomposition.

As before, we apply Dijkstra's algorithm to all level-1 squares in order, but we do not use inner vertices of each square. We recursively apply the previous algorithm to each level-1 square to compute distances for all boundary vertices in the square using inner vertices of level-2 smaller squares.

Now, the work space we use consists of all boundary vertices at level 1, all boundary vertices at level 2 in the currently active level-1 square and all inner vertices in the currently active level-2 small square and thus it amounts to $O(k\sqrt{n}+\sqrt{n}+n/k^4)$. It is minimized to $O(n^{1/10}\sqrt{n})$ when $k\sqrt{n}=n/k^4$, that is, when $k=n^{1/10}$. The time complexity increases. If we denote by T_2 the time for each level-2 square, then we have

$T_2 = O((n/k^4)^2) = O(n^2/k^8)$

since we apply quadratic-time Dijkstra's algorithm for a square of size $\sqrt{n}/k^2 \times \sqrt{n}/k^2$.

The time for each level-1 square, denoted by T_1, is given by

$T_1 = O(k^2 \times T_2 \times \sqrt{n}) = O(n^{5/2}/k^6)$

since we do scan the square just as before. In the similar way, the time for the entire grid, denoted by T_0, is given by

$T_0 = O(k^2 \times T_1 \times k\sqrt{n}) = O(n^3/k^3)$.

Therefore, if we set $k = O(n^{1/10})$, then the work space is given by $O(n^{1/10}\sqrt{n})$ and the time complexity by $O(n^3/n^{1/30})$.

We can extend the recursion into level $\ell > 1$. The smallest square at level ℓ has dimensions $\sqrt{n}/k^\ell \times \sqrt{n}/k^\ell$ and contains $(\sqrt{n}/k^\ell)^2 = n/k^{2\ell}$ inner vertices. When we apply Dijkstra's algorithm to the smallest square, it takes $O(n/k^{2\ell})$ work space for inner vertices and $O(n^2/k^{4\ell})$ time. For the level $i = 0, 1, \ldots, \ell-1$, the number of boundary vertices in the level is given by $O(\sqrt{n}/k^{i+1})$ and the time complexity T_i for the level i is given by

$T_i = O(k^2 \cdot T_{i+1} \cdot (\sqrt{n}/k^{i+1}) \times k^2) = O(\sqrt{n}/k^{i-3}T_{i+1})$.

Thus, we have

$T_0/T_{\ell-1} = (T_0/T_1) \cdot (T_1/T_2) \cdots (T_{\ell-2}/T_{\ell-1}) = n^{(\ell-1)/2}/k^{\ell(\ell+1)/2}$.

We also have

$T_\ell = O((n/k^{2\ell})^2) = O(n^2/k^{4\ell})$,

and

$T_{\ell-1} = O(k^2 \cdot T_\ell \cdot (\sqrt{n}/k^\ell) \cdot k^2) = O(n^{5/2}/k^{5\ell-4})$.

Combining the above results, we have

$$T_0 = O(\frac{n^{\ell/2+2}}{k^{\ell(\ell+1)/2}}).$$

On the other hand, the total work space S is given by

$$S = O(k\sqrt{n} + \sqrt{n} + \frac{\sqrt{n}}{k} + \cdots + \frac{\sqrt{n}}{k^{\ell-2}} + \frac{n}{k^\ell}).$$

This total work space S is minimized to

$$S = O(n^{\frac{1}{2(\ell+1)}}\sqrt{n})$$

when $k = O(n^{\frac{1}{2(\ell+1)}})$.

Substituting it into T_0, we have

$$T_0 = O(n^{2+\frac{\ell^2+\ell}{4(2\ell+1)}})$$

Theorem 3 *Given a grid graph of size $O(\sqrt{n}) \times O(\sqrt{n})$ with positive edge weights and an integer $\ell > 0$, we can output the shortest path between any two vertices on the grid in $O(n^{2+\frac{\ell^2+\ell}{4(2\ell+1)}})$ time using work space of $O(n^{\frac{1}{2(\ell+1)}}\sqrt{n})$ words.*

5 Future Works

One of our future works is to extend the result in this paper to a more general class of graphs. One target class is one of maximal planar graphs. How to use a famous planar separator theorem is important. Another interesting problem is to design a sublinear-space algorithm for computing the shortest path on the Delaunay triangulation of a point set in the plane.

References

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik **1**, 269-271, 1959.

[2] O. Goldreich, "Computational Complexity: A Conceptual Perspective," Cambridge University Press, p. 182, 2008.