Linköping University Electronic Press

Booklet of Abstracts

On the Complexity of Finding Spanner Paths

Mikael Nilsson

Part of: Booklet of Abstracts, *The 29th European Workshop on Computational Geometry (EuroCG)*, March 17-20, Braunschweig, Germany, ed Sandor P. Fekete, pp. 77-80, 2013

Available at: Linköping University Electronic Press
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-93332

On the Complexity of Finding Spanner Paths

Mikael Nilsson*

Abstract

We study the complexity of determining if a spanner path exists between two given nodes in a given Euclidean graph. This problem that we call the t-path problem is proven to be NP-complete for non-constant spanner stretches (e.g. $(2n)^{3/2}$). An algorithm to solve the problem is given. It improves on the naive $O(2^n)$ complexity to give $O(2^{0.822n})$.

We also study the same problem for a type of one-dimensional graphs that we call Integer Graphs. A more efficient algorithm can be devised for these graphs resulting in a complexity of $O(2^{c(\log n)^2})$, where c is a constant depending only on the spanner stretch.

1 Introduction

Andersson et al. [1] present an algorithm for building approximate distance oracles for graphs with dense clusters. The algorithm assumes that the Euclidean input graph is partitioned into so-called islands that are all spanners. Nodes connecting islands are called airports. The gain from using the oracle is the reduction in size that can be achieved compared to the naive n^2 lookup table. The size used by the constructed oracle is $O(M^2 + n \log n)$ where n is the total number of nodes and M is the number of airports.

If we want to construct an oracle for a given graph, we first have to put it in the required input format. While doing this we want to minimize the number of airports to keep the oracle's size down. This is a combinatorial optimization problem that has been discussed by Nilsson [5].

If we had knowledge about which nodes could belong to the same island the optimization might become easier. Two nodes can belong to the same island if and only if there is a path linking them that is a spanner path. When applying the spanner concept to paths we get an approximation requirement on the path requiring that any distance between nodes along the path is within the stretch factor times the Euclidean distance between them.

This is a short background of how we get from Andersson et al. [1] to studying spanner paths.

2 Definitions

We start with some definitions that are needed later.

A Euclidean graph is a *t-spanner* if the distance between any pair of nodes via edges in the graph is at most t times the Euclidean distance between them. Hence the factor t, also called the *stretch*, determines the quality by which the graph approximates the real distances.

Given a Euclidean graph, a *t-path* (short for *spanner path* with stretch t) between two nodes is a path between them that respects the spanner requirement. This means that any distance between nodes along the path must be within a factor t of their Euclidean distance. Do not confuse the definition used in this article with that of a t-path used in the context of Restricted Shortest Path problems (see Hassin [2]).

The *t-path problem* consists of deciding if there is a t-path between two given nodes in a Euclidean graph.

Let G be a directed graph containing the special nodes s and e. Furthermore, let C be a list of node pairs. The problem *Path with Forbidden Pairs*, abbreviated *PwFP*, consists of finding out whether there exists a path in G from s to e that at most contains one node from each pair in C. The problem was first formulated and examined by Gabow et al. [3].

It has been proven that this problem is NP-complete and that various versions of it are still NP-complete (see for instance Garey and Johnson [4]). The version we are interested in is the version where the graph is undirected and all forbidden pairs are disjoint; this problem is still NP-complete (see Nilsson [5]).

An *Integer Graph* with n nodes is a one-dimensional Euclidean graph, where all nodes are placed at integer positions on the real line. The leftmost node is placed at 0 and the remaining nodes are placed at $1, 2, 3, \ldots, n-1$. The *t-path problem in the integer graph* consists of finding a t-path from node 0 to node n-1. Figure 1 shows an example of an integer graph.

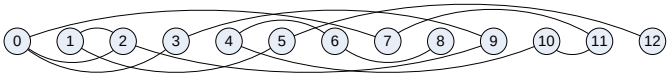

Figure 1: Example of an integer graph.

Our integer graph algorithm uses a data structure we call an *image*. An image is centered at a node. It contains all intervals of nodes that may be visited in future stages given the path used to reach the center of the image. An image centered at node 0 consists of a single interval containing all nodes. When a path is

*Linköping University, Sweden, mikael.a.nilsson@liu.se

built, the interval will be continually subdivided into smaller intervals due to the spanner constraint.

A partial order is imposed on the set of images by the image quality concept. The *image quality* of one image is better than that of another if the intervals of the first image cover all intervals of the second image. If none of the images are better than the other, the images are incomparable. Figure 2 shows an example.

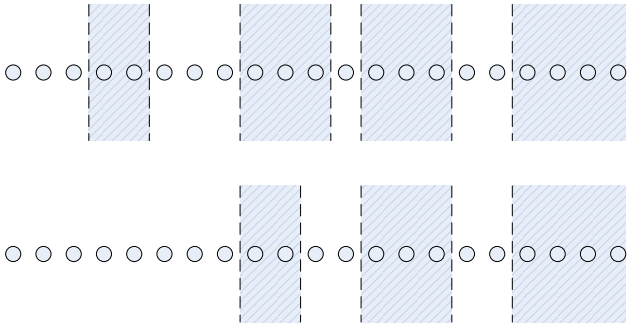

Figure 2: Two comparable images. The dashed intervals can be visited in later stages without breaking the spanner requirement.

3 A simple algorithm

First we present a simple algorithm for the t-path problem. The algorithm is recursive and in each iteration it uses Dijkstra's algorithm to find the shortest path between the start and end node of the t-path.

If this path is not a t-path there are at least two nodes along the path that conflict with the spanner condition. Since they are on a shortest path there can never be a shorter path between these two problem nodes. Hence they can never both be on the t-path. The algorithm proceeds by removing one of them and recursively tries to solve the problem on the resulting smaller graph. If this does not succeed, it puts the node back, removes the other problem node and recurses in the same way.

When analyzed, this algorithm is found to run in time $O(2^n)$ (the recurrence equation is $t_n = O(n^2) + 2t_{n-1}$, where t_n denotes the time to run the algorithm on n nodes and $O(n^2)$ comes from finding the shortest path and testing the spanner requirement). It is possible to improve this to $O(2^{0.822n})$; the calculations cannot fit here though (see Nilsson [5]). The idea is that since the order in which nodes are removed is not taken into consideration, several recursions check the same graph. Especially, as the size of the graphs shrinks the number of times they are checked increases. By building a lookup table of small graphs the recursion can be halted earlier.

4 NP-Completeness of the t-path Problem

We start by observing that the t-path problem is in NP. Given a path from a start node s to an end node e we can check in $O(n^2)$ polynomial time if this path is a t-path. This is done by comparing distances along the path, incrementally calculated in $O(n)$ steps, to the corresponding Euclidean distances for all pairs of nodes on the path (a total of $O(n^2)$ comparisons). The NP-completeness then follows by reducing a known NP-complete problem to the t-path problem. The problem chosen for this reduction is the PwFP problem. We specifically choose the version where the graph is undirected and the forbidden pairs are disjoint. We first give the general reduction scheme and then follow this by an example. It might help to look at the example pictures while following the reduction steps.

Given an instance of the PwFP problem, we start by creating a Euclidean graph. The graph is inscribed in a square that is subdivided into a grid of smaller squares. The number of squares depends on the number of nodes and forbidden pairs. We now add the nodes from the original problem. Each small square will contain one node if the node is not part of any pair and two nodes if they constitute a forbidden pair (remember that pairs are disjoint). The nodes can be added in any order. In a one node square the node is put in the center of its square. In a two node square the nodes are put around the square center with a very small distance between them (to be determined). If we set the smaller square side to 1 this gives us an outer square with sides measuring $\leq \sqrt{n}$. We now add edges that connect the same nodes as in the original problem. Edges are assigned lengths corresponding to the Euclidean distance between the connected nodes.

We now estimate the longest path distance between two nodes in this new graph. A rough upper bound can be calculated assuming that all edges are of maximum length. Because of the size of the outer square we know that any edge is shorter than $\sqrt{2n}$. This means that the longest path is bounded by $\sqrt{2}n^{3/2}$. The shortest Euclidean distance between two nodes in different squares is at least $1/2$. This means that the highest possible stretch in the graph is below $\sqrt{2}n^{3/2}/(1/2) = 2\sqrt{2}n^{3/2} = (2n)^{3/2}$ (here we do not count paths visiting two nodes in the same square as these will be prohibited). We denote this stretch by T. If a stretch of T is allowed there can be no violations of the spanner constraint along paths as long as only one node in each square is visited.

We now estimate the shortest path between two nodes in the same square. Since they are not connected it is a path having at least two edges. Since each edge is at least $1/2$ this path measures at least 1. By now setting the distance between the forbidden pair nodes to be $T^{-1} - \varepsilon$ we can prevent these nodes from both being part of the t-path with stretch T. This is because the distance along a path (≥ 1) divided by the real distance becomes $\geq 1/(T^{-1}-\varepsilon) > T$.

If we now regard this newly created graph as an

instance of the t-path problem with stretch T we will see that we can find a solution to this problem if and only if there is a solution to the original PwFP problem instance. We have then shown that any PwFP problem instance can be reduced to a t-path problem instance which together with the facts that PwFP is NP-complete and t-path is in NP means that the t-path problem is also NP-complete.

Suppose there is a solution to the PwFP instance. This means there is a path which goes from s to e without visiting more than one node from each pair. If we consider the corresponding path in the reduction graph, we see that it is a path that never breaks the spanner condition (since the stretch allowed is large enough to allow any paths that do not contain two nodes from the same square). Hence we have a t-path. If on the other hand we have a t-path between s and e in the reduction graph this must correspond to a path which never visits two nodes in the same square since that would break the spanner condition (the distance between them was set short enough that including them both violates the condition). This means that we have a path which goes from s to e without visiting a forbidden pair, hence there is a solution to the PwFP problem instance. We see that the reduction presented works and conclude that the t-path problem is NP-complete for stretch T.

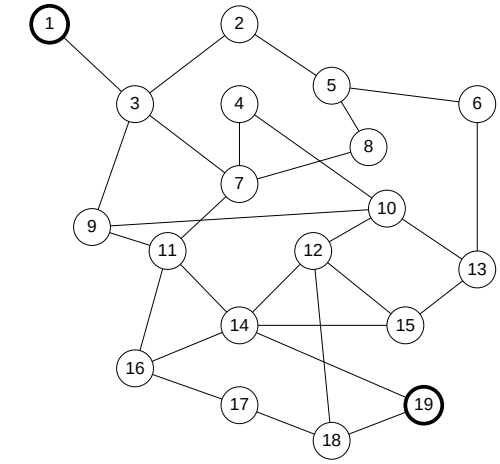

Figure 3: Example of a PwFP problem instance.

Figure 3 and 4 shows an example PwFP instance and an example reduction graph created for it. In the example, a path between node 1 and node 19 is sought. The edges in figure 3 has no edge distances. Distances are present but not shown in figure 4. The edges are drawn so it is possible to see that they still connect the same nodes.

We have shown that for stretches larger than $(2n)^{3/2}$ the t-path problem is NP-complete. By increasing the dimension of the constructed graph (fitting it in a hypercube) it is possible to shrink the required stretch to $2\sqrt{k}n^{1+1/k}$ where the integer k stands for the dimension of the constructed graph.

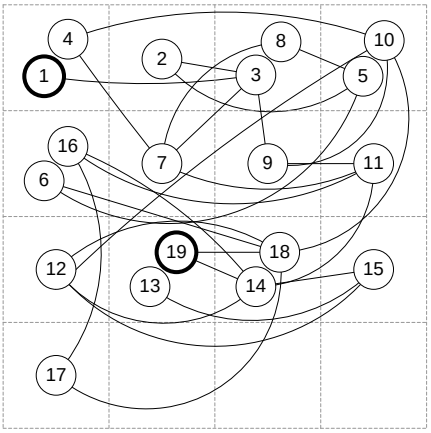

Figure 4: Resulting example reduction graph.

5 The t-path Problem for Integer Graphs

Since nodes in integer graphs are positioned on integer positions the minimum node distance is 1. This together with the fact that the graph is one-dimensional rule out a reduction like the one we just saw. Can the t-path problem be solved efficiently in integer graphs?

It can be solved efficiently for stretch 1 and $n^2/2$. Stretch 1 follows since then the path may never go left. Stretch $n^2/2$ follows since this allows all paths (it can be proven that the maximum length path is $\leq n^2/2$, see Nilsson [5]).

It is tempting to guess that since the problem can be efficiently solved for stretches greater than $n^2/2$ and stretch 1 this also applies to the whole interval $[1, n^2/2]$. Although this remains an open problem we will now examine an algorithm for the integer graph t-path problem which is more efficient than the general algorithm given in section 3.

The algorithm starts from node 0 and works iteratively towards $n-1$. Every time it extends a path to reach a new node an image, capturing the nodes that can be visited in the future, is created.

The algorithm keeps the invariant "in iteration k all images for nodes $\leq k$ created by paths using only nodes $\leq k$ are found".

Iteration k starts with checking the images of nodes $< k$. If any of these, e.g. A, allows for k to be visited a new image is created for k which is created by shrinking the intervals in A to accommodate this new edge. When images for k are created they are checked to see which nodes $< k$ can be reached from k via these new images. If a node $< k$ is found it is visited and gets a new image. This can result in recursive behavior where the path reaches more nodes $< k$.

The algorithm stores for each node only images that are incomparable. If a comparable image is generated, only the best image will be kept. Figure 2 shows an example where the upper image will cause the lower to be discarded.

Comparing two images can be done in time $O(\log n)$ if each image consists of a list of interval endpoints. This follows since the maximum number of intervals in any image is bounded by $\log k$ where k is the node location where the image is created (see Nilsson [5]).

The number of intervals has a direct impact on the complexity of the algorithm so it must be examined. Let the stretch of the problem be $1 + t$. We get an upper bound by assuming the worst in each case in the following discussion. First we check the number of intervals that the image for a node can contain on its left side. The distance these intervals occupy is limited due to the spanner constraint (it is not possible to go back too far). In node k the image cannot contain intervals to the left of point $p = 2k/(2 + t)$. This sets the maximum length of intervals to the left of node k to $kt/(2 + t)$. How many times can this interval be subdivided into smaller intervals? The cause for subdivision is that some node or nodes inside the interval are visited on the path to k.

There exists a pivot point in the original interval. If the path visits nodes to the left of this there will be no new left interval created in the subdivision. However if the path only visits nodes to the right of the pivot point a new interval to the left will be created in the subdivision. An example of this can be seen in figure 5 where a node close to the pivot point is chosen for the path.

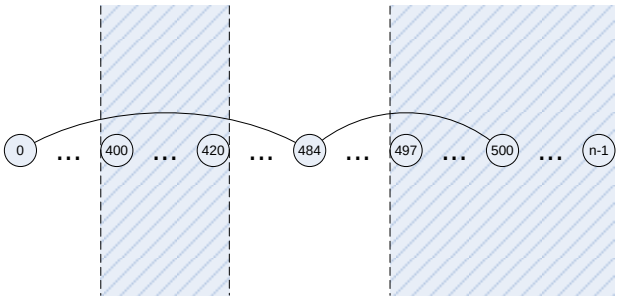

Figure 5: Example of a node close to the pivot node.

In the example figure the stretch is 1.5 allowing the path to reach as far back as 400 once it has visited 500. The pivot point of the interval [400,500] in an image centered at node 500 is 480. If a node to the left of 480 is visited along the path, the interval will have no left subdivision.

Each time a new interval is created the pivot point moves to the right until it finally reaches k. Calculating the number of times the interval can be subdivided gives $\frac{t}{2+t} \log \frac{1}{k}$ subdivisions, where $\frac{t}{2+t}$ is the base of the logarithm. In order to get an upper bound on the number of possible images we assume that each of these intervals can be of any length between 1 and the maximum length. This gives a total number of images centered at k which is bounded by $O(2^{\frac{(\log k)^2}{\log a}})$ where $a = (2 + t)/t$. Further details can be found in Nilsson [5].

We now have an upper bound on the number of images that come from intervals to the left of k. There are also intervals to the right of the node being processed. These intervals are created by the path visiting nodes to the right of k and then in the end going to k. Since the path has already passed by k this right interval will be smaller than the left. However to get an upper bound we let them have the same cardinality. This gives the final number of images in a node during the algorithm's construction of the t-path to be bounded by $O(2^{\frac{2(\log n)^2}{\log a}})$.

Factoring in the time to build new images (using a conservative n^3 for this) and process all nodes from 1 to $n - 1$ the algorithm's runtime is bounded by $O(2^{c(\log n)^2})$ where c is a constant depending only on the spanner stretch. The constant increases approximately linearly and has for example the value 5.32 at stretch 1.5.

The algorithm does not compute the t-path directly since no information of the path which led to a node is kept. However by running the algorithm n times and each time removing a different node it is possible to see which nodes are part of the t-path which can then be found by using Dijkstra's algorithm on the remaining nodes.

Acknowledgments

Thanks go to Christos Levcopoulos for numerous discussions on the t-path problem and related topics.

References

[1] M. Andersson, J. Gudmundsson and C. Levcopoulos. Approximate distance oracles for graphs with dense clusters. *Comput. Geom. Theory Appl.,* 37(3):142-154, 2007.

[2] R. Hassin. Approximation schemes for the restricted shortest path problem. *Math. Oper. Res.,* 17(1):36-42, February 1992.

[3] H.N. Gabow, S.N. Maheshwari and L.J. Osterweil. On Two Problems in the Generation of Program Test Paths. *IEEE Trans. Softw. Eng.,* 2(3):227-231, 1976.

[4] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman & Co., New York, NY, USA, 1979.

[5] M. Nilsson. *Spanneröar och spannervägar.* Institutionen för datavetenskap, Lund, 2009. `http://tinyurl.com/6gbxdnj`.