Arc Flags

Tobias Walter

October 27, 2010

Abstract

In this report we will consider the point-to-point shortest path problem. The basic idea will be that for a given graph, information can be preprocessed. We will discuss an approach called arc flags which was introduced by Lauther in [Lau04] and improved by Köhler et al. in [KMS09]

Contents

1 Introduction

The point-to-point shortest path problem in a network with nonnegative arc weights is widely used as a basic operation in many applications. Most often it is used in route planning for cars or timetable information systems for trains. The main approach, which is used nowadays, relies on Dijkstra's algorithm as described in [Dij59]. Using fibonacci heaps as described by Fredman and Tarjan in [FT87] the Dijkstra algorithm has a complexity of $\mathcal{O}(m+n\log(n))$. In many applications the graphs are still too huge for Dijkstra's algorithm. Since those graphs are often static, we may however preprocess data in order to reduce the search space of Dijkstra's algorithm. In this report we present the results which were obtained in [KMS09].

2 Preliminaries

2.1 Notation

A directed simple *graph* G is a pair (V, A). We call the finite set V *nodes* and $A \subseteq V \times V$ *arcs*. We set $n := |V|$ and $m := |A|$. Our primary interest lies in weighted graphs. A *weighted* graph is a graph (V, A) with a function $l : A \to \mathbb{R}$. l is called the *arc weight*. A *path* p is a sequence $p = (v_1, \ldots, v_k)$ of nodes, such that $(v_i, v_{i+1}) \in A$ for $1 \leq i < k$. p is called a *cycle* if $v_1 = v_k$. We define the length of a path p as $l(p) := \sum_{i=1}^{k-1} l(v_i, v_{i+1})$. The *reversed graph* $G_{rev} = (V, A_{rev}, l_{rev})$ is defined as $A_{rev} := \{(v, u) \mid (u, v) \in A\}$ and $l_{rev}(v, u) := l(u, v)$. In this paper we adapt Dijkstra's algorithm to solve the point-to-point shortest path problem. Given a source $s \in V$ and a target $t \in V$, we calculate the shortest the s-t-path. A shortest s-t-path $(s, \ldots, t)$ is a path from s to t that minimizes the path length. If the graph has no negative cycle, i. e. no cycle p with $l(p) < 0$, Johnson's algorithm shows that we can transform the arc weights to nonnegative arc weights, such that all shortest paths are preserved. Therefore we may assume that all arc weights are nonnegative.

2.2 Arc Flags

We want to solve the point-to-point shortest path problem for a fixed graph. Hence we have the advantage that we can preprocess shortest paths. Storing every shortest s-t-path however consumes $\mathcal{O}\left(n^2\right)$ space. In practice this is too much space for large graphs. We introduce arc flags with the intend to approximate this approach. Therefore the graph G is divided into p different regions. For every node v we denote the corresponding region with r_v. More formal r is a function from V into the regions $\{1, \ldots, p\}$. For every arc a we store a flag vector $f_a : \{1, \ldots, p\} \to \{\text{true}, \text{false}\}$. It is $f_a(i) = \text{true}$ if and only if a is used on a shortest path into the region i or the target of a is already in the region i. The modification to Dijkstra's algorithm is now easy. In order to find a shortest s-t-path we will only consider those arcs $a \in A$ with $f_a(r_t) = \text{true}$. As an example we can look at the graph in Figure 1. The nodes are mapped to the regions according to the different colors. Assume we want to find a shortest

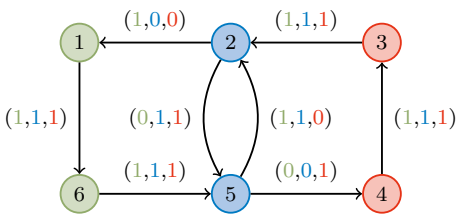

Figure 1: Example of arc flags for a small graph

way from node 3 to node 1. The only choice for our algorithm is the arc $(3, 2)$. Dijkstra's algorithm would now have two choices, either the arc $(2, 5)$ or the arc $(2, 1)$. But the modified Dijkstra algorithm with arc flags doesn't follow the arc $(2, 5)$ since we have $f_{(2,5)}(r_1) = 0$. The following lemma shows that our modification is still correct.

Lemma 2.1. *Dijkstra's algorithm with arc flags finds a shortest path from s to t, $s, t \in V$ if one exists.*

Proof. Consider a shortest path $s = v_0, \ldots, v_n = t$ found by Dijkstra's algorithm. By definition we have $f_{(v_i,v_{i+1})}(r_{v_{i+1}})$ is true. Therefore the modified Dijkstra algorithm with arc flags also finds this way. If a path is found, it is still a shortest path since the order of the processed arcs remains unchanged. □

3 Preprocessing

3.1 All pairs shortest path

Let $d(u, v)$ denote the distance from $u \in V$ to $v \in V$ in G. Assume we have all the distances given. We can compute the flag vectors of an arc a for an region R by checking if a lies on a shortest path into R. Hence we consider the shortest paths from u to any node $r \in R$. The arc $a = (u, v) \in A$ lies on a shortest path if and only if $d(u, r) = d(u, v) + d(v, r)$[1]. If so, or $v \in R$, we set $f_a(R) =$ true. We can use the shortest path trees from u and v calculated by Dijkstra's algorithm for this. This means we have to do $2m$ Dijkstra runs in order to calculate the flag vectors. In practice with large graphs (approximately 1 million nodes and 2.5 million arcs) this would take weeks.

3.2 Basic preprocessing without all pairs shortest path

We don't have to calculate every shortest path into a region R. It is sufficient to calculate only those shortest paths which end at the border of a region R. In order to allow easier terminology this is briefly defined.

Definition 3.1. *$(u, v) \in A$ is a boundary arc if $r_u \neq r_v$. v is then called a boundary node.*

Using the above described insight, we obtain the following lemma.

Lemma 3.2. *If the flag vectors f_a are computed with the set of shortest paths to boundary nodes only, then Dijkstra's algorithm with arc flags is still correct.*

Proof. Every shortest path from s to t, where $r_s \neq r_t$ has to enter the region r_t with some boundary arc (u, v). The subpath from s to v is also a shortest path. Hence the flag vectors are still the same. □

Instead of calculating the shortest paths to a boundary node v, we will now calculate a shortest path tree for v in G_{rev}. This results in those arcs used for shortest paths to the boundary node v. Hence we can set the flag at those arcs to true. We can calculate the shortest path tree for v in $\mathcal{O}(m + n \log(n))$. Assuming we have a sparse graph and k boundary nodes this results in a time complexity of $\mathcal{O}(kn \log(n))$. The running time is hence directly dependent on the number of boundary nodes, which is dependent on the given partition of the graph. Therefore having a small number of boundary nodes is an aim when we choose the parition. This will be further discussed in Section 4.

[1] This is true because we assumed nonnegative arc weights.

3.3 Centralized Shortest Path Search

Up to now we have only used single Dijkstra runs in order to get the flag vectors. Shortest path trees are quite similiar if we start from the same neightborhood. Therefore we run Dijkstra at the same time for every boundary node of a given region and try to use the information gained on Dijkstra runs from the other boundary nodes. Let $B = \{b_1, \ldots, b_j\}$ denote the boundary nodes of a given region. We assign to every node v a label $L_v : B \to \mathbb{R}$. An entry $L_v(b_i)$ stands for the length of the currently shortest path from b_i to v found by the algorithm. Each node v also has a key $k(v)$ which is used for sorting in the heap. The algorithm behaves now like the labeling algorithm and extracts the elements with the lowest key in our heap. This element v propagates his labels now to every neighbor w of v. Every component of our label vector is updated the usual way. The behavior of this algorithm depends on two choices, the initialization of the labels and the algorithm to set the keys.

Choosing the initialization of the labels

Setting unknown values to infinity This method is the same as used in Dijkstra's algorithm. We assign zero to the distance of the boundary nodes in B to themselves and set all other values to infinity.

Limited initialization We limit a Dijkstra run just on the region which is currently examined. This delivers upper bounds for the distances between the nodes of the region if the region is connected. Even though those values needn't to be the correct values, since shortest paths between two nodes of a region might use arcs outside of the region, we still could use those upper bounds.

Aborted initialization For every boundary node we run Dijkstra's algorithm until we found a shortest path to every node in the region. Then the algorithm is aborted. This delivers correct distances between the boundary nodes of a region.

In practice *aborted initialization* outperforms *limited initialization*. In every case all nodes have to be stored in the heap which have some incorrect label entries.

Choosing the key values

Minimum tentative key Consider that the label of L_v has been changed during the relaxation of an arc (u, v). Let K be the set of values which have been changed in L_v. We update $k(v) = \min(K \cup k(v))$ if v already had a key $k(v)$, else we set $k(v) = \min(K)$. The advantage of this method is that we can prove its good behavior.

Lemma 3.3. *Using the minimum tentative key strategy, every vertex is scanned at most $|B|$ times and the key values of the scanned nodes are non-decreasing.*

Proof. cf. Lemma 3.2 in [KMS09]. □

Minimum total key Another possible way of choosing the key is taking the minimum of all label entries as the key. Using this strategy, labels which do not represent the shortest path value yet will be updated earlier. This is so because nodes that have been scanned already will still have their old key value after they have been processed again. So they are ordered into the heap before every node which has not been visited yet. Hence we don't have to correct many labels which were based on a wrong estimate of an entry. Even though we can't prove an upper bound for the number of times a node is added to the heap, the minimum total key strategy outperforms the minimum tentative key strategy in practice.

Domination values We want to propagate those labels which are most likely correct. For this we define a kind of quality based key value. We store to every node v the domination value, which is the number of improvements in the label from v since the last propagation of v. The key now consists of this domination value and the minimum total key, where the domination value has priority.

4 Partition

We have already seen in Lemma 2.1 that every partition can be used for Dijkstra's algorithm with arc flags. The question arises which partition leads to the best speed-ups. First of all the number of separator arcs should be small, as we have already seen in Section 3.2. We should also balance the number of nodes in every region. Every flag vector has the same importance if we balance this. We should also try to minimize the number of almost-full flag vectors as they are not useful for the algorithm.

Most of our parition algorithms use a layout of our graph. We define $L : V \to \mathbb{R}^2$ to be a *layout* and identify every node with its position in the layout. We can assume that every graph lies inside a bounding-box.

Rectangular grid We divide the bounding-box into a $x \times y$-grid. Let (l, t) denote the top-left position and (r, b) denote the bottom-right position. The regions $G_{i,j}$ are defined as the rectangular $[l + i \cdot \frac{(r-l)}{x}, l + (i+1) \cdot \frac{(r-l)}{x}] \times [b + j \cdot \frac{(t-b)}{y}, b + (j+1) \cdot \frac{(t-b)}{y}]$ for $0 \leq i < x$ and $0 \leq j < y$. This method ignores every other information about the graph and therefore it is not surprising that it is slower in practice compared to other partitions.

Quad-Trees We define a tree. The leaves of the tree represent our regions. We start with the root node v_0 which represents the the bounding-box r_0. We now divide every region r_i which is represented by the node v_i into four quadrants, as long as r_i contains more nodes than a given bound. Every quadrant is assigned a subregion and a node, which is a child of v_i. This method guarantees almost balanced regions.

kd-Trees Similar to Quad-Trees we divide a bounding-box into two parts. Every part will be divided recursively until every region has less than a given upper bound. We divide so, that the position is given by the median of the points inside. The division is alternated between the axes, it is divided by the x-axis,y-axis,x-axis, ...

Multi-way arc separator Up to now all partition algorithms used a given layout. A way to get a region can also be to cut the graph in parts with a minimal separator set. This can be done recursively. Using such an approach can also balance the number of nodes in every region. For details see [KK98].

5 Conclusion

Arc flags are a fast and easy way to improve Dijkstra's algorithm. As the results in [KMS09] show, Dijkstra's algorithm improves to a factor of more than 1000 times faster running time on certain instances. Despite this improvement less overhead is stored. 225 regions proved to be sufficient for graphs which occur in practice. The additional bits needed can be lowered by compressing the flag vectors. Further studies on the quality of the region could still improve the algorithm.

References

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34:596–615, 1987.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[KMS09] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *Shortest Paths: Ninth DIMACS Implementation Challenge*, 2009.

[Lau04] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In Martin Raubal, Adam Sliwinski, and Werner Kuhn, editors, *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230, 2004.