Finding two disjoint shortest paths in $\mathcal{O}(n^3m)$ for directed graphs with positive edge weights

1 Introduction

You are given a directed graph G with positive edge weights, and two pairs of terminal vertices s_1, t_1, s_2, t_2. The goal is to find two vertex disjoint paths P_1, P_2, such that P_i is a shortest path from s_i to t_i for $i \in \{1, 2\}$ (if such paths exist). Let's call this the *2-disjoint shortest path problem* (2DSP).

As of writing this, the best known algorithm for solving this problem has running time $\mathcal{O}(nm^5)$ and is due to Bérczi and Kobayashi [2]. Although it should be said that their work focused mainly on finding any polynomial algorithm rather than trying to optimize it.

In this document, an algorithm with running time $\mathcal{O}(n^3m)$ will be presented. Maybe it can be improved to $\mathcal{O}(n^2m)$ by using fast methods for finding disjoint paths in directed acyclic graphs [3]. I hope the ideas in here can be used to improve the running time even more, perhaps by using algebraic methods similar to those in [1].

2 Solution setup

The first step is to construct the *shortest path DAG* from s_1. This is the graph that contains every edge in G that is part of some shortest path originating from s_1. Since the weights are strictly positive, this will indeed be a DAG. Let's call this DAG D_1, and let D_2 be the shortest path DAG from s_2. We now have two DAGs with the same vertex set but different edge sets, and our goal is to find two disjoint paths on these DAGs. If the DAGs had a mutual topological order (i.e. a way to order the vertices that is a topological ordering with respect to both DAGs), then this could be easily achieved with the standard $O(nm)$ algorithm for finding disjoint paths in a DAG. But D_1, D_2 will not have a mutual topological order in general. However, it turns out that they have something that is not too far off:

Lemma 2.1. *For two vertices $u, v \in G$, denote by $d(u,v)$ the length of the shortest path from u to v. Define the function ψ as*

$$\psi(v) = d(s_1, v) - d(s_2, v)$$

. This function has the following properties:

1. *If (u,v) is an edge in D_1, then $\psi(u) \leq \psi(v)$.*
2. *If (u,v) is an edge D_2, then $\psi(u) \geq \psi(v)$.*
3. *If (u,v) is an edge in $D_1 \cup D_2$ such that $\psi(u) = \psi(v)$, then (u,v) is an edge in both D_1 and D_2.*

Proof. For the first part, take an edge $(u,v) \in D_1$ with weight w. Then

$$\psi(v) - \psi(u) = (d(s_1, v) - d(s_1, u)) - (d(s_2, v) - d(s_2, u))$$

Since the edge is on D_1, it follows that $d(s_1, v) - d(s_1, u) = w$. For the other term, we have that $d(s_2, v) - d(s_2, u) \leq w$, because otherwise there would be a shorter path to u using this edge. Thus, $\psi(v) - \psi(u) \geq 0$, which proves the first part.

The second part is similar to the first. As for the third part, let $(u,v) \in D_1 \cup D_2$ be an edge with weight w. Note that $\psi(u) = \psi(v)$ can only happen if $d(s_i, v) - d(s_i, u) = w$ for $i \in \{1, 2\}$, which is equivalent to this edge being on both D_1 and D_2. $\square$

Now, let $\bar{D_2}$ be D_2 with all edges reversed. We will use this reversed DAG instead of D_2, and consider paths from t_2 to s_2. The lemma above now has the following effects:

- Points 1 and 2 in the lemma says that if (u,v) is an edge on D_1 or $\bar{D_2}$, then $\psi(u) \leq \psi(v)$. So if we were to ignore edges (u,v) where $\psi(u) = \psi(v)$, then sorting the vertices with respect to ψ would actually be a mutual topological ordering.
- Point 3 in the lemma says that the edges (u,v) where $\psi(u) = \psi(v)$ occur in both DAGs D_1 and D_2. But since we are now considering $\bar{D_2}$ instead, we have that the DAGs go in opposite directions along these edges.

The idea of the algorithm is to group up vertices with respect to $\psi(v)$, and use dynamic programming. We will get an "outer" DP problem when the paths move across different groups. This will look a bit like the standard approach for finding disjoint paths in DAGs. When the paths move along the same group however, we get an "inner" DP problem where we have two paths that move in opposite directions along the same DAG.

3 Dynamic programming

In this section, an instance of 2DSP is fixed. Also, since we will be considering $\bar{D}_2$, we are interested in paths from t_2 to s_2. For a graph G and a vertex u, let's denote by $G(u)$ the set of vertices connected by an edge from u. Some details and edge cases concerning the DP will be saved for the next page, in an attempt to make this section slightly more readable.

For two vertices u, v, let $2DSP(u, v)$ be a function that decides if there exist two disjoint shortest paths $u \to t_1$, $v \to s_2$.

Definition 3.1. If $u, v \in G$, then the pair (u, v) is said to be a *terminal state* if

1. $\psi(u) = \psi(v)$, and
2. there exist vertices $u' \in D_1(u)$, $v' \in D_2(v)$, such that $\psi(u') \neq \psi(u)$, $\psi(v') \neq \psi(v)$, and $2DSP(u', v') =$ **True**.

One tiny detail: If $u = t_1$, then the existence of u' is not required, and if $v = s_2$, then the existance of v' is not required.

In addition, denote by $T(p)$ the set of terminal pairs (u, v) such that $\psi(u) = p$.

Definition 3.2. Let $a, b, c, d \in G$ such that $\psi(a) = \psi(b) = \psi(c) = \psi(d)$. Then denote by $2DP(a, b, c, d)$ the function that decides if there exists two disjoint paths $a \to b$, $c \to d$, using edges from D_1.

Now we can state the transitions for $2DSP(u, v)$. The base cases are $2DSP(u, u) =$ **False**, and $2DSP(t_1, s_2) =$ **True**.

The first case is when $\psi(u) < \psi(v)$, here we advance the vertex with the lower ψ:

$$2DSP(u, v) = \bigvee_{w \in D_1(u)} 2DSP(w, v).$$

The second case when $\psi(u) > \psi(v)$ is similar:

$$2DSP(u, v) = \bigvee_{w \in D_2(v)} 2DSP(u, w).$$

The third case when $\psi(u) = \psi(v)$ is the toughest. We need to find a terminal state (u', v') with the same ψ as u and v. Then we need to find two disjoint paths from u to u' and v to v', that move in opposite directions along the DAG. This latter part is made easy though thanks to our definition of $2DP$, it just asks us to check that $2DP(u, u', v', v) =$ **True**.

$$2DSP(u, v) = \bigvee_{(u', v') \in T(\psi(u))} 2DP(u, u', v', v)$$

If the computations of $2DP(a, b, c, d)$ are ignored, the transition in the third case is the most expensive, running in $O(n^4)$.

How do we compute the set of terminal states $T(\psi(u))$? Let's assume that the DP visits states in an order such that (a, b) is visited before (a', b') if $\psi(a)+\psi(b) > \psi(a')+\psi(b')$. This means that when the DP reaches a state (u, v) with $\psi(u) = \psi(v)$, the values of $2DSP(u', v')$ from the definition of terminal states have already been computed. Thus, we can now find all pairs of terminal states and store then for future use. To do this, we can loop over all pairs of neighbours over all pairs of vertices (u, v), which would take $\mathcal{O}(m^2)$. It is also possible to make these transitions sparse with an extra flag, which would improve this part to $\mathcal{O}(nm)$.

How do we compute the values of $2DP(a, b, c, d)$? This will be done in a very simple way. For every pair of sinks (c, d), use the standard DP approach to compute all values $2DP(a, b, c, d)$. This takes $\mathcal{O}(nm)$ for each pair of sinks, so $\mathcal{O}(n^3m)$ in total. In fact, these values can be precomputed before the main DP starts, since the DAG we are dealing with is just the set of edges (u, v) such that $\psi(u) = \psi(v)$, which is known beforehand.

What about states on the form (t_1, v) and (u, s_2)? This is a special case we haven't discussed yet. In these cases, one of the paths has already reached its destination, and we might want to advance the other one even if it has greater ψ. One way to take care of these cases is to consider them as base cases, and precompute them in $\mathcal{O}(nm)$ (we just need to iterate through all such states and run a DFS to check that a path exists). In practice, it was more convenient to deal with these by actually adding them as cases in the DP transition.

What about unreachable vertices? We defined ψ using the distance between vertices, but what if some vertices are unreachable? There are many ways of dealing with this, we can for example add infinity weight edges in such a way that the graph becomes strongly connected. Another way is to define $d(u, v)$ as infinity if v can't be reached from u.

3.1 Summary

This will be a brief summary of the algorithm.

1. Construct the shortest path DAGs D_1 and $\bar{D_2}$. Also, let C be the DAG that only includes edges (u, v) such that $\psi(u) = \psi(v)$.
2. Compute the values $2DP(a, b, c, d)$ on the DAG C, for every pair of sinks (c, d). The running time of this is $\mathcal{O}(n^3 m)$.
3. Loop through all pairs of vertices (u, v) sorted decreasingly by $\psi(u)+\psi(v)$.
   (a) If $\psi(u) \neq \psi(v)$, we use the first and second case transitions on the previous page to compute $2DSP(u, v)$.
   (b) If $\psi(u) = \psi(v)$, we first check if the set of terminal pairs $T(\psi(u))$ has been computed. If not, we loop through all pairs (a, b) with $\psi(a) = \psi(b) = \psi(u)$, and check if it belongs to $T(\psi(u))$. This can be made to run in $\mathcal{O}(nm)$ in total. Then, we use the third case transition to compute $2DSP(u, v)$.

4 Can it be done faster?

Currently, the bottleneck of the algorithm seems to be the following problem:

Definition 4.1. You are given a DAG D, and a set of pairs of vertices T. For every pair of vertices (u, v) you want to know if there exists two disjoint paths $u \to u'$ and $v' \to v$ for some pair $(u', v') \in T$. Let's call this the *2 disjoint opposite paths problem* (2DOP).

- Right now, we are solving this 2DOP problem by computing the value of $2DP(a, b, c, d)$ for all tuples of vertices (a, b, c, d). Perhaps this "all pairs disjoint paths" problem can be solved in $\mathcal{O}(n^4)$ instead of $\mathcal{O}(n^3 m)$?
- In the 2DOP problem, the set of pairs T can be of size n^2 in the worst case. However, it is possible to bring it down to n by adding some extra vertices. For every vertex v, let's add a new vertex v' where there is a directed edge from u to v' for every $(u, v) \in T$. Now, we let T' be the set of all pairs (v', v) of this new graph of size $2n$. This doesn't speed up our current approach, but it does mean that the transitions in the third case of the DP can be made to run in $\mathcal{O}(n^3)$ instead of $\mathcal{O}(n^4)$.
- Maybe some algebraic approach can be used rather than the simple all-pairs DP strategy. It would also be interesting to know if arbitrary instances of 2DOP can show up in the solution of 2DSP, as that would mean that focusing on this subproblem is important here.

References

[1] S. Akmal, V. Vassilevska Williams, N. Wein. *Detecting Disjoint Shortest Paths in Linear Time and More.* https://arxiv.org/abs/2404.15916 (2024)

[2] K. Bérczi and Y. Kobayashi. *The directed disjoint shortest paths problem.* (2017).

[3] T. Tholey. *Linear time algorithms for two disjoint paths problems on directed acyclic graphs* (2012)