Dijkstra's algorithm and shortest paths in graphs

SHORTEST PATH

Input: A directed graph with edge weights. A start node s
Goal: The distances and shortest paths from s to all other nodes.

(Distance = minimal sum of weights on directed paths from s to the node.)

Two important cases:

1. All weights are > 0.

2. We allow negative weights.

Two other cases:

1. Directed graphs.

2. Undirected graphs.

We will focus on the first case.

Special case: If all weights are = 1 we can use the BFS algorithm. But if we have weights not equal to 1 it's easy to find cases when BFSs doesn't work.

Dijkstra's algorithm

Def: If u ∈ V then d(u) = lenght of shortest path s →u in G. (By lenght we mean the sum of the weights along the path.)

Idea: We compute the distances from s to all other nodes in a certain order. Let S be a set such that at each step in the algorithm the distance from s to the nodes in S are computed correctly. Then we expand this set with one node at each step.

We start with S = {s} and d(s) = 0. All neighbors to s are given temporary distance d(v) = w(s,v). We give all other nodes a temporary distance d(v) = ∞.

At each step we consider the neighbors to S, i.e. nodes v ∉ S such that there is a node u ∈ S and edge (u,v) ∈ E.

We chose the imediate neighbor with minimal temporary distance and put it into S. The temporary distance will be now be a permanent (real) distance.

For each neighbor q ∉ S to v we see if d(q) > d(v) + w(v,q). If this is the case we set d(q) = d(v) + w(v,q) as new temporary distance.

When S = V the algorithm ends and all distances are computed.

Pseudo code:

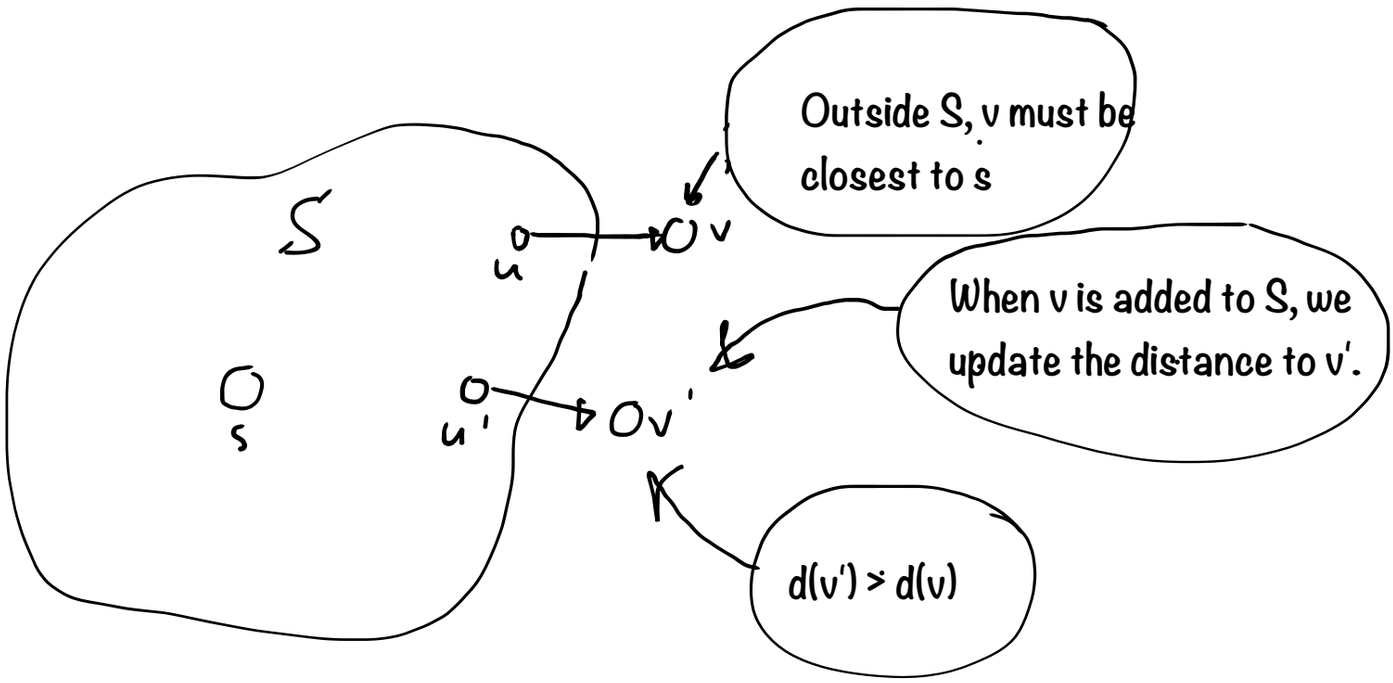
```
Set S = {s}, d(s) = 0
For all neighbors u to s, set d(u) = w(s,u)
For all other nodes, set d(u) = ∞
While S ≠ V
    Chose v ∉ S with d(v) minimal
    Set S = S ∪ {v}
    For all neighbors q to v such that q ∉ S
        If d(q) > d(v) + w(v,q)
            Set d(q) = d(v) + w(q,v)
        End if
    End for
End while
```

Complexity: It depends on what method you use to find the v with d(v) minimal. Without a clever method this will take $O(|V|)$ steps. Since we have to do this $O(|V|)$ times we get a complexity $O(|V|^2)$. The step " For all neighbors q to v ..." will take $O(|E|)$ steps if we use adjacency lists. For dense graphs this is $O(|V|^2)$. In any case, $O(|V|^2)$ is an upper bound on the complexity.

Correctness: We will sketch an argument for the correctness.
At the center of the algorithm we have to loop "While S ≠ V ..."
We can define an invariant I for this loop.

I : For all nodes u in S, the values d(u) tell the correct distances from s to u in G. For all nodes v not in S the values d(v) tell the length of the shortest path from s to v only using nodes in S (except v). (And d(v) = ∞ tells that there are no such paths.)

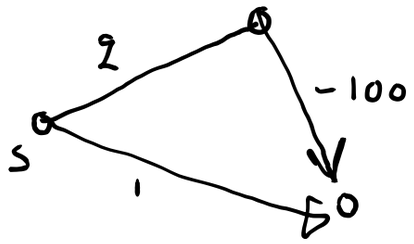
I is obviously true when the loop starts and it can be shown that I is always true.
The loop ends after |V| - 1 steps. Then all distances are computed.



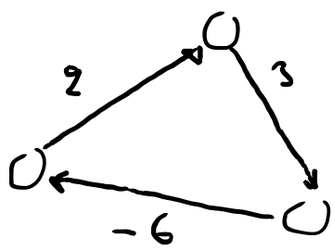
Comment: The algorithm is stated for directed graphs. It actually works equally well (without modifications) for undirected graphs.

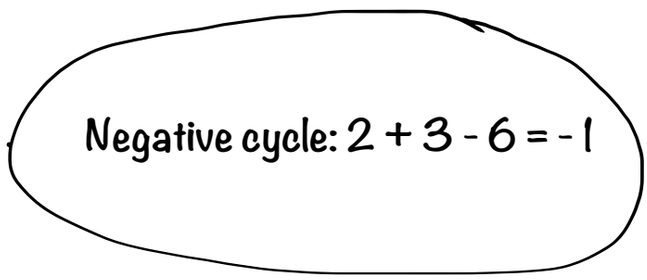
Negative weights

Dijkstra's algorithm does not work for graphs with negative weights.



There are other algorithms that sometimes work for negative weights. The crucial question is if a graph contain negative cycles or not.





If a graph does not contain any negative cycles there are algorithms for finding shortest paths. Examples are Bellman - Ford's algorithm and Warshall's algorithm. They both work i time $O(|V|^3)$.

Obs: When we talk about paths we mean a sequence of nodes connected by edges such that no node occur more than one time.

This is Bellman - Ford's algorithm:

Bellman-
Ford($G = \langle V, E \rangle$,$s \in V$, $w : E \rightarrow$)
(1) **foreach** $u \in V$
(2) $d[u] \leftarrow \infty$
(3) $d[s] \leftarrow 0$
(4) **for** $i = 1$ **to** $|V| - 1$
(5) **foreach** $(u, v) \in E$
(6) **if** $d[v] > d[u] + w(u, v)$
(7) $d[v] \leftarrow d[u] + w(u, v)$
(8) $p[v] \leftarrow u$
(9) **foreach** $(u, v) \in E$
(10) **if** $d[v] > d[u] + w(u, v)$
(11) **return** Negative Cycle!
(12) **return** d, p

The time complexity is O(|V| |E|), which makes it slower than Dijkstra's algorithm. The advantages are that the algoritm can handle negative weights and detect negative cycles if there are any.

There is no efficient algorithm for finding shortest paths in graphs with negative cycles. In fact, it can be shown that the problem is NP-complete. (More about this later.)

All this goes for directed graphs. If you have an undirected graph with negative weights but no negative cycles there are algorithms for finding shortest paths but they are surprisingly complicated.

Flow problems

We will study the so called flow-problem. We start with a directed graph with two special nodes s (source) and t (sink) given. We assume that there is at least one directed path from s to t.

On each edge e we have a capacity c(e). It is a number ≥ 0

Flow: · A flow is given by a number f(e) for each edge. The numbers must satisfy two conditions.

1. $0 \leq f(e) \leq c(e)$ for all e.

Let In(v) be the set of all edges going in to v and Out(v) be the set of all edges going out from v.

2. $\sum_{In(v)} f(e) = \sum_{Out(v)} f(e)$ for all v except s and t

Value of flow: We set $val(f) = \sum_{Out(s)} f(e)$

(It's obvious that this is equal to $\sum_{In(t)} f(e)$)

Some more notation: If $e = (u,v)$ we write $f(e) = f(u,v)$

If $X \subseteq V$ and $Y \subseteq V$ we set $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$.

Cut: A cut is a partition of V into two disjoint parts X, Y such that $s \in X$ and $t \in Y$.

It is not hard to prove that if (X,Y) is a cut, then $val(f) = f(X,Y) - f(Y,X)$

Two questions:

1. What is the maximal possible value for a flow in G?

2. How do we find a maximum flow?

The capacity of a cut

Def: Given a cut (X,Y) we define $C(X,Y) = \sum_{x \in X} \sum_{y \in Y} c(x,y)$

We see that for any cut (X,Y) we have $val(f) \leq C(X,Y)$

This is because $val(f) = f(X,Y) - f(Y,X) \leq f(X,Y)$
and, since $f(e) \leq c(e)$, we have $f(X,Y) \leq c(X,Y)$

Let MC be the minimal capacity for cuts in G. Then we get $val(f) \leq MC$.

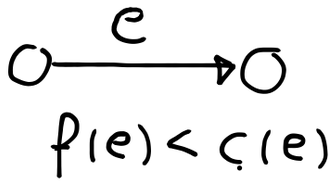
The famous Max flow- min cut theorem says that if f^* is a maximum flow we actually have $val(f^*) = MC$.

We will soon prove this theorem.

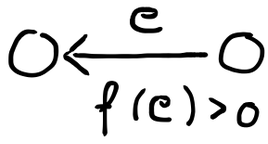
Ford - Fulkerson's algorithm

We start with $f(e) = 0$ for all edges and build up a flow in several steps.
We look for so called augmenting paths.

Unsaturated forward edge:



Unsaturated backward edge:



An augmented path is a path consisting of unsaturated paths (forward and/or backward) going from s to t.

If we can find an augmenting path we can increase the flow. How much?

For each edge e in an augmenting path P we define a number $\triangle(e)$. If e is a forward edge (in the path) we set $\triangle(e) = c(e) - f(e)$. If e is a backward edge we set $\triangle(e) = f(e)$. Then we let δ be the minimum value of $\triangle(e)$ for all edges in P.

We now increase the theflow by setting $f'(e) = f(e) + \delta$ for all forward edges and $f'(e) = f(e) - \delta$ for all backward edges.

We will now try to show the following:
We have reached a maximum flow if and only if there are no augmenting paths left in G.

It's obvious that if we have reached a maximum flow there can be no augmenting paths left. The other direction is the difficult one.

We assume that there are no augmenting paths left. We will construct a special type of cut: Starting at s we try to build paths using unsatureted edges. Let X' be set of all nodes that can be reached from s by such paths. Let Y' be V - X'. Then we have s ∈ X' and t ∈ Y'. So (X',Y') is a cut.

For this cut, and this flow, we have val(f) = C(X',Y') !

Why? Let (x,y) be a directed edge going from X' to Y'. Since x but not y can be reached by unsaturated edges we must have f(x,y) = c(x,y). If (y,x) is a directed edge going from Y' to X' we must have f(y,x) = 0. This means that val(f) = f(X',Y') - f(Y',X') = f(X',Y) = C(X',Y').

We know that if f* is a maximum flow and MC is the size of a minimum cut then val(f) ≤ val(f*) and MC ≤ C(X',Y'). But since val(f) = C(X',Y') we get val(f) = val(f*) = MC = C(X',Y').

This gives us

1. We see that f is amaximum flow and the Ford-Fulkerson algorithm works.

2. We have proved the Max flow - Min cut theorem.

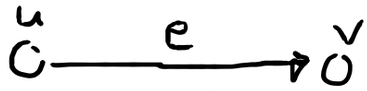
Implementing the Ford - Fulkerson algorithm

When we implement the algorithm we use a so called Residual Graph. Let us assume that we are given a directed graph G with capacities on the edges.

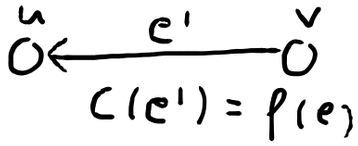
We want to find augmenting paths in G. It can be shown that it's better not just to look for any augmenting paths. Instead we should use BFS and look for shortest augmenting paths. This variant of Ford - Fulkerson's algorithm is called Edmond - Karp's algorithm.

We want to use BFS in standard form. The problem is that BFS looks for "forward" paths. To get around this problem we use so called residual graphs.

Let G be a directed graph and a flow f (not necessarily maximal). Given G and f we construct a new graph with some new edges and capacities:



For each edge we define a new capacity $c'(e) = c(e)-f(e)$ and we define a new edge and capacity:



This gives us the residual graph Gf.

We then use the following algorithm:

Ford Fulkerson's algorithm in pseudocode

c[u,v] is the capacity from u to v, f[u,v] is the flow, cf[u,v] is the residual capacity.

```
foreach edge (u,v) in the graph do
        f[u,v]:=0; f[v,u]:=0
        cf[u,v]:=c[u,v]; cf[v,u]:=c[v,u]
while there is a path p from s to t in the residual flow graph do
        r:=min(cf[u,v]: (u,v) is in p)
        foreach edge (u,v) in p do
        f[u,v]:=f[u,v]+r; f[v,u]:= -f[u,v]
        cf[u,v]:=c[u,v] - f[u,v]; cf[v,u]:=c[v,u] - f[v,u]
```

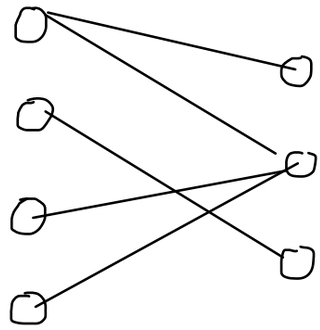
We can use the flow algorithm to solve another problem: The matching problem for bipartite graphs.

BIPARTITE MATCHING

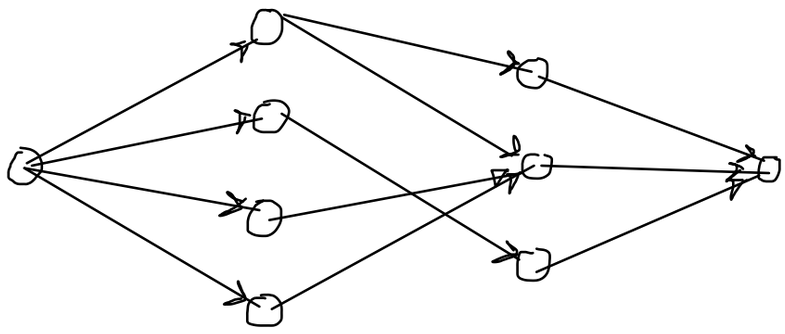
Input: A bipartite graph (X Y, E).
Goal: A matching of maximal size.

A matching is a set of edges with no pair of edges having a node in common.

Start with:



Then we add some nodes and edges and directions:



Give all edges capacity 1 and run the flow algorithm. When the algorithm ends the edges from the original graph that are saturated (i.e. f(e) = 1) gives us a maximal matching.