CS 341 - Algorithms, Winter 2016. University of Waterloo

Lecture 14: Dynamic programming on graphs

We have seen dynamic programming on "lines" and on trees.

Today we will see dynamic programming on graphs, using two important problems, all-pairs shortest paths and the traveling salesman problem.

Single-source shortest paths with arbitrary edge weights

Input: A directed graph $G=(V,E)$, an edge length l_e for $e \in E$, a vertex s.

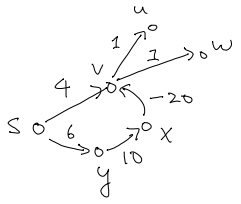
Output: The shortest path length from s to all vertices $t \in V$.

This problem is solved using Dijkstra's algorithm in the special case where the edge lengths are non-negative. Dijkstra's algorithm runs in near-linear time.

It turns out that allowing negative edge lengths makes the problem considerably harder.

First, let's see why Dijkstra's algorithm doesn't work in this more general setting.

In Dijkstra's algorithm, the key is to maintain a set $R \subseteq V$ so that dist[v] is computed correctly for $v \in R$, and then we greedily grow the set R by adding a vertex $v \notin R$ closest to R.



The greedy algorithm does not maintain this invariant anymore.

In the example in the picture, the vertex v is closest to s and Dijkstra's will add v first, but dist[v] $\neq 4$ as the path s,y,x,v is of length -4, because of the (very) negative edge xv.

With this wrong start, Dijkstra's algorithm will add u and w to R.

Only after a while, vertex x is added, and then we realize that there is a shorter path to v through x.

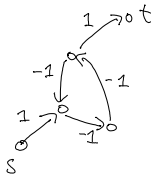
At that time, we know that the distances to u and w are not computed correctly

If we want to fix it, we need to use the new distance to v to update the distance to u and w, but then we can no longer say that each vertex is only "explored" once.

This is where we lost the near-linear time complexity for solving the more general problem.

This is where we lost the near-linear time complexity for solving the more general problem.

Negative cycles



Another issue of having negative edge lengths is that there may be negative cycles. Then, from s to t, we can go around the negative cycle as many times as we want, and so the shortest path distance is not well-defined.

We will study algorithms to solve the following problems:

- Given a directed graph, check if there is a negative length cycle C, i.e. $\sum_{e \in C} l_e < 0$.
- If G has no negative cycles, solve the single-source shortest path problem.

Bellman-Ford Algorithm [kT 6.8]

Intuition

Although Dijkstra's algorithm may not compute all distances correctly in one pass, it will compute the distances to "some" vertices correctly.

In the above example, dist[y] will be computed correctly.

With this, if we do Dijkstra's algorithm again, then we would get dist[x] right.

With one more Dijkstra's, we would then get dist[v] correct and so on.

How many times we need to do?

If the graph has no negative cycles, there is a shortest path with at most n-1 edges.

So, by repeating Dijkstra's algorithm n-1 times, we should have computed all shortest path distances correctly, and the time complexity is about $O(nm)$.

This is basically the Bellman-Ford algorithm.

Recurrence

To formalize the above idea, we will prove that if the graph has no negative cycles (so that shortest path distances are well-defined), then we maintain correctly all shortest paths with at most i edges, and compute it inductively from $i=1$ to $i=n-1$.

Let $D(v,i)$ be the shortest path distance from s to v using at most i edges.

Initially, $D(s,0)=0$ and $D(v,0)=\infty$ for all $v \in V-s$. This is the base case.

Assuming $D(v,i)$ are computed correctly for all $v \in V$ for $i \geq 0$. This is the induction hypothesis.

In the inductive step, we would like to compute $D(v,i+1)$ correctly for all $v \in V$.

A path with at most $i+1$ edges from s to v must be coming from a path with at most i edges from s to u for an in-neighbor u of v.

Therefore, $D(v,i+1) = \min \left\{ D(v,i), \min_{u:uv \in E} \{ D(u,i) + \ell_{uv} \} \right\}$.

As we argued before - if there is no negative cycle, there is a shortest path from s to v with at most $n-1$ edges (if v is reachable from s).

So, it is enough to compute until $i = n-1$, and $dist[v] = D(v, n-1)$.

Time complexity Given $D(v,i)$ for all $v \in V$, it takes in-deg(w) time to compute $D(w,i+1)$.

So, the total time to compute $D(w,i+1)$ for all $w \in V$ takes $O\left(\sum_{w \in V} \text{in-deg}(w)\right) = O(m)$.

And thus the total time complexity is $O(nm)$.

Space complexity A naive implementation requires $\Theta(n^2)$ space.

Notice that to compute $D(w,i+1)$ $\forall w \in V$, we just need $D(v,i)$ $\forall v \in V$ and don't need $D(v,j)$ for $j \leq i-1$. Therefore, we can throw those away and just use $O(n)$ space.

Simple algorithm The algorithm can be made even simpler, matching the intuition that we mentioned in the beginning.

```
dist[s] = 0,   dist[v] = ∞   ∀ v ∈ V - s.
for i from 1 to n-1 do
    for u ∈ V do
        for each edge uv ∈ E do
            if dist[u] + ℓuv < dist[v]
                dist[v] = dist[u] + ℓuv  and  parent[v] = u.
```

This is the Bellman-Ford algorithm.

To see that we don't need two arrays, just notice that using one array could only have the intermediate distances smaller, and they remain to be upper bounds of the true

distances, so using tighter upper bounds would not hurt.

Shortest path tree

By following the edges (parent[v], v), we would like to return a shortest path from s to v.

Now, we have many iterations in the outermost loop, it is no longer clear that we have a tree structure for the shortest paths.

In fact, it is possible to have a directed cycle in the edges (parent[v], v), but they must form a negative cycle.

Lemma If there is a directed cycle C in the edges (parent[v], v), then the cycle C must be a negative cycle, i.e. $\sum_{e \in C} l_e < 0$.

proof Let the cycle C be $v_1, v_2, \ldots, v_k$, with $v_i v_{i+1} \in E \ \forall \ 1 \le i \le k-1$ and $v_k v_1 \in E$.

Assume that $v_k v_1$ is the last edge in the cycle added by the algorithm.

Consider the values dist[v_i] right before the update.

By the definition of parent[], we have $dist[v_{i+1}] \ge dist[v_i] + l_{v_i v_{i+1}}$ for $1 \le i \le k-1$.

Note that when we first set the parent, the inequality holds as an equality, but later dist[v_i] could decrease and it may become an inequality.

Now, when we set parent[v_1] = v_k, it must be because $dist[v_1] > dist[v_k] + l_{v_k v_1}$ at that time.

Adding all these k inequalities, we have $\sum_{i=1}^{k} dist[v_i] > \sum_{i=1}^{k} dist[v_i] + \sum_{e \in C} l_e$,

and this implies that $\sum_{e \in C} l_e < 0$. □

So, the lemma implies that if there are no negative cycles, then there are no cycles in the edges (parent[u], u).

Assuming that every vertex can be reached from vertex s, then every vertex has exactly one incoming edge in (parent[u], u), and there are no directed cycles by the Lemma, and so the edges (parent[u], u) must form a directed tree, i.e. a tree with edges pointing away from s.

Negative Cycles [KT 6.10]

Negative Cycles [KT 6.10]

Ideas

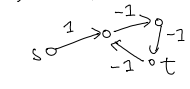
In the Bellman-Ford algorithm, we see that it all worked out when there are no negative cycles.

The Bellman-Ford algorithm still works okay if there are negative cycles.

After k iterations, it will compute the shortest path distance from s to v using at most k edges.

We just used the fact that if there are no negative cycles, then we just need to compute at most $n-1$ iterations (i.e. $k \leq n-1$) to find the shortest path distances.

If there are negative cycles, then we will expect that $D(v,k) \to -\infty$ as $k \to \infty$.

For example, in the graph 

, then $D(t,3) = -2$, $D(t,6) = -5$, $D(t,9) = -8$, and so on.

On the other hand, if there are no negative cycles, then we expect that $D(v,n) = D(v,n-1)$ for all v.

So, intuitively, by comparing $D(v,n)$ and $D(v,n-1)$ for all $v \in V$, we can determine whether the graph has a negative cycle or not.

Assumption

In the following, we assume that every vertex can be reached from s.

For the problem of finding negative cycles, this is without loss of generality since we can restrict our attention to strongly connected components and we learnt from L07 how to identify all strongly connected components in linear time.

Observations

We make the above ideas precise by the following claims.

Claim 1 If the graph has a negative cycle, then $D(v,k) \to -\infty$ when $k \to \infty$ for some $v \in V$.

proof This follows from the definition of $D(v,k)$ and the assumption that every vertex can be reached from s. □

Claim 2 If the graph has no negative cycles, then $D(v,n) = D(v,n-1)$ for all $v \in V$.

proof Any cycle is non-negative, and so we can assume the shortest path has no cycles.

and thus is of length at most $n-1$. □

Claim 3 If $D(v,n) = D(v,n-1)$ for all $v \in V$, then the graph has no negative cycles.

Proof If $D(v,n) = D(v,n-1)$ for all $v \in V$, then $D(v,n+1) = D(v,n)$ for all $v \in V$,

because the recurrence relations are the same.

More precisely, the recurrences are $D(v,n) = \min\{ D(v,n-1), \min_{u:uv \in E}\{ D(u,n-1) + l_{uv}\}\}$ (*).

So, $D(v,n+1) = \min\{ D(v,n), \min_{u:uv \in E}\{ D(u,n) + l_{uv}\}\}$

$= \min\{ D(v,n-1), \min_{u:uv \in E}\{ D(u,n-1) + l_{uv}\}\}$ (because $D(v,n) = D(v,n-1)$ by assumption)

$= D(v,n)$ (because of (*)).

Hence, by induction, $D(v,k) = D(v,n-1)$ for all $k \geq n-1$, and thus $D(v,k)$ is finite when $k \to \infty$ for all $v \in V$.

By claim 1, there are no negative cycles in the graph. □

Notice that the same proof as in Claim 3 shows that as long as $D(v,k+1) = D(v,k)$ $\forall v \in V$, then we can stop in that iteration with all the distances computed correctly.

This provides an early termination rule that is useful in practice (when shortest path have few edges).

Algorithm

Claims 2 and 3 together imply that we can check whether a graph has a negative cycle or not in $O(nm)$ time, by computing $D(v,n)$ $\forall v$ and check whether $D(w,n) < D(w,n-1)$ for some $w \in V$.

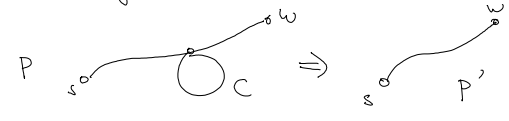
How do we find a negative cycle if it exists (i.e. when $D(w,n) < D(w,n-1)$ for some w)?

Here we assume that we use the $\Theta(n^2)$-space algorithm for computing $D(w,n)$, and that we have stored $parent(w,n) = u$ if $D(w,n) = D(u,n-1) + l_{uv}$.

First, since $D(w,n) < D(w,n-1)$, we know that the path P from s to w with total length $D(w,n)$ must have exactly n edges; as if the path P has at most $n-1$ edges, we would have $D(w,n-1) = D(w,n)$.

A path of length n must have a repeated vertex, and thus a cycle C.

We claim that C must be a negative cycle.



Suppose not, that C is non-negative.

Then, we can skip the cycle C to get a path P' with fewer edges and

length(P') ≤ length(P) since C is non-negative.

But that would imply that $D(w,n-1) \le$ length(P') since P' has at most n-1 edges, and thus $D(w,n-1) \le$ length(P') ≤ length(P) = $D(w,n)$, a contradiction.

So, the cycle C must be negative.

By tracing out the parents using the stored information, we can find P and thus the cycle C.

This gives an $O(mn)$-time algorithm to find a negative cycle, using $\Theta(n^2)$ space.

Space-efficient algorithm: There is also an $O(mn)$-time algorithm using only $O(n)$ space for finding a negative cycle. The details are more involved and we refer the reader to [KT 6.10].

All-Pairs Shortest Paths [DPV 6.6]

Input: A directed graph $G=(V,E)$, an edge length l_e for $e \in E$.

Output: The shortest path length from s to t, for all $s,t \in V$.

We can solve this problem by using Bellman-Ford for all $s \in V$. This would take $O(n^2m)$ time, which could be $O(n^4)$ when $m = \Theta(n^2)$.

It is possible to solve the all-pair shortest path problem in $O(n^3)$ time, using a different recurrence relation.

We will present the Floyd-Warshall algorithm.

Recurrence

Since we are interested in computing all-pairs shortest paths, it makes sense to create more subproblems, to store information for each pair of vertices.

Let the vertex set V be $\{1,2,\ldots,n\}$.

Following Floyd-Warshall, let $D(i,j,k)$ be the shortest path length from i to j only using vertices $\{1,\ldots,k\}$ as intermediate vertices in the graph.

The base cases are $D(i,j,0) = l_{ij}$ if $ij \in E$ and $D(i,j,0) = \infty$ if $ij \notin E$, since $D(i,j,0)$ is asking the shortest path length from i to j without using any intermediate vertex.

(Remark: The choice of the subproblems don't look very natural. A more natural choice is

Perhaps $D'(i,j,l)$, which represents the shortest path length from i to j using at most l edges, as in the Bellman-Ford setting. But it turns out that the Floyd-Warshall subproblems allow faster computation. We leave it as a question to see why $D'(i,j,l)$ won't work as well.)

Assume $D(i,j,k)$ are computed correctly for all $i,j \in V$.

We would like to use them to compute $D(i,j,k+1)$ for all $i,j \in V$.

The only difference between $D(i,j,k+1)$ and $D(i,j,k)$ is that $D(i,j,k+1)$ allows to use vertex $k+1$ as an intermediate vertex while $D(i,j,k)$ is not allowed.

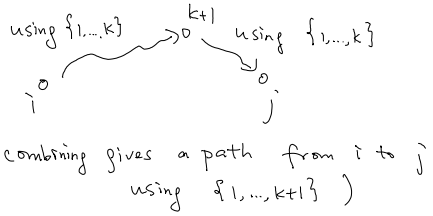
To use vertex $k+1$ as an intermediate vertex for a path between i and j, we need to go from i to $k+1$, and then from $k+1$ to j.

What is the optimal way to do it (using only $\{1,\dots,k+1\}$ as intermediate vertices)?

Of course, we will use a shortest path from i to $k+1$ (using $\{1,\dots,k\}$ as intermediate vertices), and a shortest path from $k+1$ to j (only using vertices in $\{1,\dots,k\}$ as intermediate vertices).

Note that we don't need to use vertex $k+1$ more than once, since there are no negative cycles.

Therefore, $D(i,j,k+1) = \min\{ D(i,j,k),\ D(i,k+1,k) + D(k+1,j,k) \}$, where the first term considers the paths not going through $k+1$, while the second term consider paths that use $k+1$.



Floyd-Warshall algorithm

With the recurrence, it is straightforward to translate into an algorithm.

$D(i,j,0) = \infty$ $\forall ij \notin E$, $D(i,j,0) = l_{ij}$ $\forall ij \in E$.

for k from 1 to n do

  for i from 1 to n do

    for j from 1 to n do

      $D(i,j,k+1) = \min\{ D(i,j,k),\ D(i,k+1,k) + D(k+1,j,k) \}$.

Time complexity: It should be clear that it is $\Theta(n^3)$.

So, to compute all-pair shortest paths, it is always faster than using Bellman-Ford n times, which has time complexity $\Theta(n^2 m)$ where $m = \Omega(n)$ for a connected graph and could be $\Omega(n^2)$.

However, if one is only interested in single-source shortest paths, then Bellman-Ford is

always faster.

Exercise: We leave it as an exercise to return the shortest paths.

Open problem: It has been a long standing open problem whether there is a truly sub-cubic algorithm (i.e. with running time $O(n^{3-\varepsilon})$ for some constant $\varepsilon > 0$, say $O(n^{2.99})$) for computing all pair shortest paths.

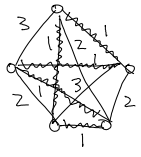
Again, Prof. Timothy Chan is an expert of this problem. invented some $o(n^3)$ algorithms!

Traveling Salesman Problem [DPV 6.6]

Input: An undirected graph $G=(V,E)$, with a non-negative edge length l_{ij} for all $i,j \in V$.

Output: A cycle C that visites every vertex exactly once and minimizes $\sum_{e \in C} l_e$.

The name of the problem comes from the motivation that a salesman wants to find a shortest tour to visit every city and go home.



As we will show in the last part of this course, this problem is NP-hard.

There is a naive algorithm for this problem, by enumerating all possible ordering to visit the cities.

This will take $\Theta(n! \cdot n)$ time. It becomes too slow when $n=13$.

We present a dynamic programming solution that can probably work up to $n=30$.

Recurrence

The difficulty of the problem is that it is not enough to remember only the shortest paths, but also what vertices that we have visited (so that we know what other vertices yet to visit).

The recurrence is unlike everything that we have seen so far, because it has exponentially many subproblems.

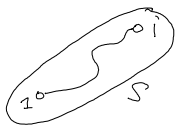
Let $C(i,S)$ be the shortest path to go from 1 to i, with vertices in S on the path.

Note that we don't care about the ordering of vertices in S in the path, and this is where

the speedup over the naive algorithm is coming from.

If we have computed $C(i, V)$ correctly for all $i, j \in V$,

then $\min_i \{ C(i, V) + l_{i1} \}$ is the length of a minimum cycle that visits every vertex exactly once.

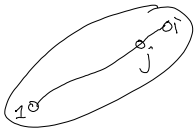


The base cases are $C(i, \{1, i\}) = l_{1i} \quad \forall i \in V$.

Suppose we have computed $C(i, S)$ correctly for all subsets S of size k, i.e. $|S| = k$.

We would like to use them to compute $C(i, S)$ for all subsets S of size $k+1$.

To compute $C(i, S)$ for S with $|S| = k+1$, we just need to try all possibilities of the second last vertex on the path.



Note that the second last vertex must be on the path, and of course the best way to reach the second last vertex j is to use a shortest path from 1 to j that visits every vertex in $S - \{i\}$ once.

Therefore, $C(i, S) = \min_j \{ C(j, S - \{i\}) + l_{ji} \}$.

Algorithm and complexity

We leave it as an exercise to write a pseudo code of the algorithm.

For the time complexity, there are $O(n \cdot 2^n)$ subproblems, each requiring $O(n)$ time to compute, so the total time complexity is $O(n^2 \cdot 2^n)$.

The main drawback is that the space complexity is $\Theta(n \cdot 2^n)$.

Concluding remark: We have seen many examples and structures to design dynamic programming algorithms, from lines to trees to graphs.

With the help of homework problems and supplementary exercises, I hope that you will be familiar with this technique, with which you could solve a much larger class of non-trivial problems than before.

Dynamic programming is an important and useful and general technique to solve problems. The key is to come up with a good recurrence relation with few subproblems.