CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 14: Dynamic Programming on Graphs

We use dynamic programming to design algorithms for graphs, including shortest paths problems and TSP.

Single-Source Shortest Paths with Arbitrary Edge Lengths

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

Output: The shortest path distances from s to every vertex $v \in V$.

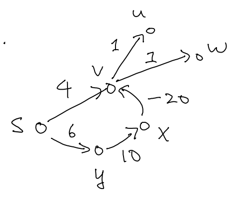
Dijkstra's Algorithm

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

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

Let's first see why Dijkstra's algorithm does not work in this more general setting.

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



With the presence of negative edge lengths, however, this greedy algorithm does not maintain this invariant anymore.

In the example, the vertex v is closest to s and is added to R first, but dist[v] $\neq 4$ as the path s,y,x,v is of length -4 because of the negative edge xv

The correctness of Dijkstra's algorithm crucially uses that the length of a prefix of a path cannot be shorter than the length of the path, and this doesn't hold anymore with the prescence of negative edges.

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

Only after vertex x is added and explored later, we realize that there is a shorter path from s to v via x.

Then, we know that the distances to u and w are not computed correctly.

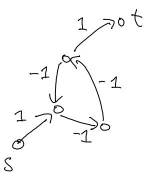
To fix it, we need to use the new distance to v to update the distance to u and w, but then we can no longer say that each vertex is only explored once.

We can extend this example so that this update process needs to be done many times.

This is where we could not maintain the near-linear time complexity for solving this more general problem.

Negative Cycles

Another issue of having negative edges is that there may exist negative cycles.



In the example, 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 even well-defined.

In the following, we will study algorithms to solve the following problems.

① If G has no negative cycles, solve the single-source shortest paths problem.

② Given a directed graph G, check if there exists a negative cycle C, i.e. $\sum_{e \in C} l_e < 0$.

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, e.g. the first vertex on a shortest path.

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

Then, if we do the update on every edge again, then we would get dist[x] right for sure.

Then, with one more update phase on every edge, then we would get dist[v] correct and so on.

How many times we need to do?

If the graph has no negative cycles, then any shortest walk must be a simple path, which has at most $n-1$ edges.

So, by repeating the updating phases at most $n-1$ times, we should have computed all shortest path distances correctly, with time complexity about $O(nm)$.

This is basically the Bellman-Ford algorithm.

Dynamic Programming

To formalize the above idea, we design an algorithm using dynamic programming to compute the shortest path distance from s to every vertex $v \in V$ using at most i edges, from $i=1$ (base case) to $i=n-1$.

Then we will show that this is equivalent to the Bellman-Ford algorithm we see in textbooks.

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

Answers: For each $v \in V$, $D(v,n-1)$ is the shortest path distance from s to v when the graph has no negative cycles.

Base cases: $D(s,0)=0$ and $D(v,0)=\infty$ for all $v \in V-s$.

Recurrence: To compute $D(v,i+1)$, note that a path with at most $i+1$ edges from s to v must be coming from a path using at most i edges from s to u for an in-neighbor u of v.

Since we are to compute the shortest path distance from s to v using at most $i+1$ edges, we should use a shortest path using at most i edges from s to u.

$$\begin{tikzcd} s \arrow[r, squiggly] & u \arrow[r] & v \end{tikzcd}$$

We try all possibilities of u and get the recurrence relation

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

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 time to compute $D(w,i+1)$ for all $w \in V$ is $O(\sum_{w \in V} \text{in-deg}(w)) = O(m)$.

We do this for $1 \le i \le n-1$, thus the total time complexity is $O(nm)$.

Space Complexity A direct implementation requires $\Theta(n^2)$ space, to store all the values $D(v,i)$.

Note that to compute $D(w,i+1)$ $\forall w \in V$, we just need the values $D(v,i)$ $\forall v \in V$ but don't need

$D(v,j)$ for $j \le i-1$, and so we can throw these away and only 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] = $\infty$   $\forall v \in V - s$.
for i from 1 to n-1 do
    for each edge $uv \in E$ do
        if dist[u] + $l_{uv}$ < dist[v]
            dist[v] = dist[u] + $l_{uv}$  and  parent[v] = u.
```

This is the Bellman-Ford algorithm. The simplification is that we don't need to use two arrays.

To see why we don't need two arrays, note that using one array could only have the intermediate

distances smaller, and they remain to be upper bounds on the true distances, so using

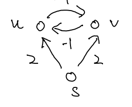
tighter upper bounds would not hurt (and may speed up in practice).

Shortest Path Tree

As in Dijkstra's algorithm, we would like to return a shortest path from s to v by following the edges (parent[v],v).

In Bellman-Ford, there are many iterations in the outerloop, and it is not clear whether these edges still form a tree.

Actually, it is possible to have a directed cycle in the edges (parent[v], v),



but the following lemma shows that these directed cycles must be negative cycles.

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 directed 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 C formed in the algorithm, i.e.

the cycle C formed when v_k becomes the parent of v_1, while parent$[v_i] = v_{i-1}$ already for $2 \le i \le k$.

Consider the values dist$[v_i]$ right before v_k becomes the parent of v_1.

Since v_{i-1} is the parent of v_i, we have dist$[v_i] \ge$ dist$[v_{i-1}] + l_{v_{i-1}v_i}$ for $2 \le i \le k$.

(Note that at the time when we set parent$[v_i] = v_{i-1}$, the inequality holds as an equality, but later dist$[v_{i-1}]$ could decrease and it may become an inequality.

Note also that we cannot have dist$[v_i] <$ dist$[v_{i-1}] + l_{v_{i-1}v_i}$ as otherwise parent$[v_i]$ would be updated.)

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$[v_i] + \sum_{e\in C} l_e$, which implies that $\sum_{e\in C} l_e < 0$. □

The lemma implies that if there are no negative cycles, then there are no directed cycles in the edges (parent$[v]$,v).

Assuming that every vertex can be reached from vertex s, then every vertex has exactly one incoming edge in (parent$[v]$,v), and there are no directed edges by the lemma.

So, the edges (parent$[v]$,v) must form a directed tree, i.e. a tree with edges pointing away from s.

To conclude, when there are no negative edges, the edges (parent$[v]$,v) form a shortest path tree from s.

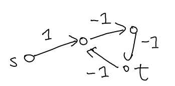
Negative Cycles [KT 6.10]

Ideas: We can extend the dynamic programming algorithm to identify a negative cycle if it exists.

Even with negative cycles, after k iterations of the dynamic programming algorithm, the same recurrence relation proves that we compute the shortest path distance from s to v using at most k edges $\forall v \in V$.

To solve the single-source shortest paths problem, we only used the assumption that there are no negative cycles to prove that the algorithm can stop after $n-1$ iterations and conclude that distances are computed correctly.

If there are are negative cycles, then we would expect that $D(v,k) \to -\infty$ as $k \to \infty$ for some $v \in V$.

For example, in the graph  , we have $D(t,3) = -1$, $D(t,6) = -4$, $D(t,9) = -7$, 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 $v \in V$, and this implies that $D(v,\infty) \not\to -\infty$ as $k \to \infty$ for all $v \in V$.

So, intuitively, by checking if $D(v,n) = D(v,n-1)$ $\forall v \in V$, we can determine if there is a negative cycle or not.

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

For the problem of finding a negative cycle, 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 SCCs in linear time.

Observations

We make the above ideas precise by the following claims.

Claim 1 If the graph has a negative cycle, then $D(v,k) \to -\infty$ as $k \to \infty$ 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, so we can assume that any shortest walk from s to v has no cycles, and thus it 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 $v \in V$, as the recurrences are the same.

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

$$\begin{aligned} \text{So, } D(v,n+1) &= \min\left\{ D(v,n), \min_{u:uv\in E} \{ D(v,n) + l_{uv} \} \right\} \\ &= \min\left\{ D(v,n-1), \min_{u:uv\in E} \{ D(u,n-1) + l_{uv} \} \right\} \quad (\text{because } D(v,n) = D(v,n-1)\ \forall v \text{ by assumption}) \\ &= D(v,n) \quad (\text{because of } (*)). \end{aligned}$$

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

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

Note 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 $D(v,l) = D(v,k)$ $\forall v \in V$ $\forall l > k$, and so we can stop in that iteration with all distances computed correctly.

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

Algorithms

Checking: Claim 2 and 3 together imply that a graph has no negative cycles iff $D(v,n-1) = D(v,n)$ $\forall v \in V$.

Since we can compute $D(v,n)$ and $D(v,n-1)$ $\forall v \in V$ in $O(mn)$ time, this implies an $O(mn)$ time algorithm for checking.

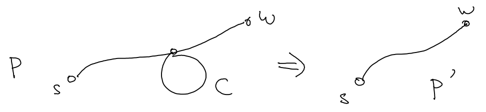
Finding: The next question is: if $D(w,n) < D(w,n-1)$ for some w, how do we find a negative cycle?

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

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

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 a non-negative cycle.

Then, we can skip the cycle C to get a path P' with fewer edges than that in P and length$(P') \leq$ length(P) since C is a non-negative cycle.

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

So, the cycle C must be a negative cycle.

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 implementation There is also an $O(mn)$-time algorithm using only $O(n)$ space.

The details are more involved and we refer 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 running Bellman-Ford for each $s \in V$.

This would take $O(n^2 m)$ time, which could be $\Theta(n^4)$ when $m=\Theta(n^2)$.

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

Here we present the Floyd-Warshall algorithm.

Dynamic Programming

In the Floyd-Warshall algorithm, more subproblems are used to store information for each pair of vertices.

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

Let $D(i,j,k)$ be the length of a shortest path from vertex i to vertex j using only vertices $\{1,\ldots,k\}$ as intermediate vertices in the path.

(Another perhaps more natural choice is $D'(i,j,k)$ which denotes the length of a shortest path from i to j using at most k edges similar to that in the Bellman-Ford algorithm.

We leave it as a question to think about $D'(i,j,k)$ doesn't work as well as the Floyd-Warshall subproblems.)

Answers: $D(i,j,n)$ $\forall i,j \in V$.

Base cases: $D(i,j,0) = l_{ij}$ if $ij \in E$ and $D(i,j,0)=\infty$ if $ij \notin E$.

This is because $D(i,j,0)$ is asking for the shortest path length from i to j without using intermediate vertices.

Recurrence: Assume $D(i,j,k)$ are computed correctly $\forall i,j \in V$ for some k.

We would like to compute $D(i,j,k+1)$ $\forall i,j \in V$.

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

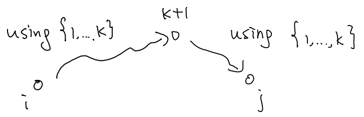
To use vertex $k+1$ as an intermediate vertex for a path between i and j, the path has to go from i to $k+1$ and then from $k+1$ to j.

What is the optimal way to do it using only vertices $\{1,2,\dots,k+1\}$ as intermediate vertices?

Of course, we should use a shortest path from i to $k+1$ using $\{1,2,\dots,k\}$ as intermediate vertices, and a shortest path from $k+1$ to j using $\{1,2,\dots,k\}$ as intermediate vertices.

Note that we don't need to use vertex $k+1$ more than once, as 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 vertex $k+1$.



Floyd-Warshall algorithm

```
D(i,j,0) = ∞  ∀ij∉E.   D(i,j,0) = l_ij  ∀ij∈E.   // base cases
for k from 0 to n-1 do   // allowing more and more intermediate vertices
    for i from 1 to n do          } // going through all pairs
        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 is clear that the runtime is $O(n^3)$.

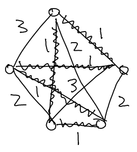
Exercise: Given $s,t \in V$, return a shortest path from s to t.

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

Traveling Salesman Problem [DPV 6.6]

Input: A directed graph $G=(V,E)$, with an edge length l_{ij} for all $i,j \in V$.

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



It is one of the most famous problem in combinatorial optimization.

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

There is a naive algorithm for this problem, by enumerating all possible orderings to visit the vertices. 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 $n=30$.

Dynamic Programming

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 what other vertices yet to visit.

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

Subproblems: Let $C(i,S)$ be the length of a 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.)

Answers: $\min_{i \in V} \{ C(i,V) + l_{i1} \}$ - from 1 to i using all vertices in V once, then come back to 1.

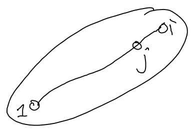
Base cases: $C(i, \{1,i\}) = l_{1i}$ for all $i \in V$.

Recurrence: Suppose we have computed $C(i,S)$ for all subsets S of size k.

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

To compute $C(i,S)$ for S of size $k+1$, we try all possibilities of the second last vertex on the path.

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



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

Algorithm: Exercise.

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 of this algorithm 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 interesting problems.