451: Dynamic Programming II

G. Miller, K. Sutner
Carnegie Mellon University
2020/09/08

Recall: The Key Steps

Here are the key ingredients in any dynamic programming algorithm.

- Find a suitable notion of (smaller) sub-instance for any given instance.
- Find a recursive way to express the solution of an instance in terms of the solutions of the sub-instance(s).
- For bottom-up: organize this recursive computation into a neat table, possibly with some pre-computation.

Suppose we have a directed graph $G = \langle V, E \rangle$ and a cost function $\mathsf{cost} : E \to \mathbb{N}_{\geq 0}$ on its edges.

The cost of a path $P = x_1, x_2, \ldots, x_k$ is

$$\mathsf{cost}(P) = \sum_i \mathsf{cost}((x_i, x_{i+1})).$$

We will refer to the number of edges on a path as its length. The distance from x to y is

$$\delta(x,y) = \begin{cases} \min\big(\mathsf{cost}(P) \mid P \text{ path } x \to y\big) & \text{if such a path exists,} \\ \infty & \text{otherwise.} \end{cases}$$

These paths should be called minimum cost paths, but are usually referred to as shortest paths.

Claim: If $P : s = x_0, x_1, \ldots, x_k = t$ is a shortest path from s to t, then its initial segments are shortest paths from s to x_i.

Variants

There are two basic variants:

Single-Source: Some source vertex s is fixed, we want to compute $\delta(s,t)$ for some/all targets t.

All-Pairs: We have to compute $\delta(x,y)$ for all $x,y \in V$.

Even in the single-source version, because of the prefix property of shortest paths, we cannot avoid dealing with additional source vertices.

For the time being, we will constrain ourselves to non-negative costs. This is entirely natural for example when dealing with costs related to some geometry: vertices are points in some space, and cost(x, y) is the distance between x and y.

Standard case: the space is "$\mathbb{R}^n$" and the distance is Euclidean.

Negative costs also make perfect sense, for example when describing the change of some potential during a transition in some system from one state to another.

We'll mention negative costs later.

Single Source

Fix some source vertex s and write $\delta(x) = \delta(s, x)$.

Our strategy is to compute

- overestimates $\mathsf{dist}(x)$ for $\delta(x)$,
- that are associated with an actual path.

Thus, $\mathsf{dist}(x)$ may not be optimal, but it is not just some random value; it is always realized by a path.

We say that edge (u, v) requires attention if $\mathsf{dist}(v) > \mathsf{dist}(u) + \mathsf{cost}(u, v)$.

We improve the overestimate for v by relaxing the edge:

$$\mathsf{dist}(v) = \mathsf{dist}(u) + \mathsf{cost}(u, v).$$

Prototype Algorithm

```
// prototype shortest path algorithm

    initialize dist($s$) $= 0$

    while some edge $(u, v)$ requires attention
        relax $(u, v)$
```

Lemma

The prototype algorithm is correct.

Of course, we need to organize the search for the unhappy edge and the data structures. One very elegant solution is Dijkstra's algorithm: explore the neighbors of the currently closest vertex, using a heap to keep track of the latter. With ordinary heaps we get $O((m + n) \log n)$, with Fibonacci heaps even $O(m + n \log n)$.

Termination: dist(x) is always the cost of a real path s to x, so the value of dist(x) can decrease only finitely often (costs are non-negative). Hence an edge can require attention only finitely often and ultimately ceases to do so. Thus, the algorithm terminates.

Correctness: Suppose that upon completion for some path

$$P = s, x_1, \ldots, u, v$$

we have $\delta(v) = \text{cost}(P) < \text{dist}(v)$. Choose P to be of minimal length (not cost!). Then $\text{dist}(u) = \delta(u)$. But after dist$(u)$ is set to $\delta(u)$, the edge (u, v) requires attention:

$$\text{dist}(v) > \delta(v) = \delta(u) + \text{cost}(u, v) = \text{dist}(u) + \text{cost}(u, v).$$

When (u, v) is relaxed, $\text{dist}(v) = \delta(v)$, contradiction.

$\square$

The Obstruction

To turn the prototype algorithm into a dynamic programming method, we need some notion of sub-instance.

The Problem: In general, graphs are notoriously ill-structured, there is no easy way to decompose them into smaller graphs in order to get a recursive approach.

Some families of graphs have nice inductive properties (say, trees or series-parallel graphs), but in general we need a clever idea to come up with useful sub-instances.

Idea: constrain the allowed paths to have length (not cost!) at most k, $k = 0, \ldots, n-1$.

Write $\mathsf{dist}(x, k)$ for the corresponding constrained distance from s to x. Note that this is a realized overestimate, just like before.

Initializing $k = 0$ is easy:

$$\mathsf{dist}(v, 0) = \begin{cases} 0 & \text{if } s = v, \\ \infty & \text{otherwise.} \end{cases}$$

There are $n - 1$ rounds based on the following recursion:

$$\mathsf{dist}(v, k) = \min\big(\mathsf{dist}(v, k-1), \min_{(x,v) \in E} \mathsf{dist}(x, k-1) + \mathsf{cost}(x, v)\big)$$

Comments

In other words, in each round, we try to relax all edges with target v, for all v.

Given a standard adjacency list representation, we can easily precompute predecessor sets $\text{pre}(v) = \{ x \in V \mid (x,v) \in D \}$ in linear time. Since, in each round, we touch every edge only once, the total running time is $O(nm)$.

As usual, we do not obtain the actual shortest path, just its cost. This is easy to fix: maintain a predecessor array $\pi(v)$: the predecessor of v on a currently shortest path from s to v (update whenever an edge with target v is relaxed). To get a path, retrace your steps from the target to the source in linear time.

Negative Costs

What happens if we allow cost$(u, v) < 0$?

One problem now is that the notion of shortest path may no longer be well-defined: if there is a negative cost cycle we are sunk. More precisely, if there is a negative cost cycle that is reachable from the source s, but we won't quibble.

So there are two problems:

- Decide whether there is a negative cost cycle.
- If not, compute shortest paths.

Note that Dijkstra fails in general with negative costs, even if the graph is acyclic.

But BF Works

Recall that we have $n-1$ rounds $k=1,2,\ldots,n-1$, so all simple paths starting at s can be taken into account.

Hence, if there is no negative cost cycle reachable from s, an additional round n will change nothing. On the other hand, if round n changes any distance, this must have been caused by a bad cycle.

Thus we can solve the decision problem, and have the right distances in the good case.

Exercise

Find a negative cost example where Dijkstra fails.

Exercise

Explain carefully why Bellman-Ford works with negative costs.

How about All-Pairs? We could run a single-source algorithm like Dijkstra repeatedly, at a cost of $O(n(n+m)\log n)$. With fancy heaps this can be reduced to $O(n(m+n\log n))$, which is particularly useful for sparse graphs.

At any rate, note that there will be recomputation, we will discover some shortest paths over and over.

So dynamic programming might help.

Again we have to deal with our old problem: graphs don't carry a nice inductive structure.

We need a clever idea to come up with useful sub-instances.

Here is a somewhat brutal approach: let's ignore the topological structure of the graph entirely and simply smash it to pieces:

We restrict intermediate vertices on a path.

Assume $V = [n]$ and set

$$\text{dist}_k(u, v) = \text{length shortest path } u \to v \text{ using only points} \leq k$$

where $1 \leq u, v \leq n$ and $0 \leq k \leq n$. Note that the endpoints of the path are excluded from the restriction, we only worry about the intermediate points being "small."

In other words, we simply smash the graph by temporarily erasing vertices $k + 1, k + 2, \ldots, n$. That's OK since $\text{dist}_n(u, v) = \delta(u, v)$.

Recursion

Initializing $k = 0$ is easy:

$$\mathsf{dist}_0(u, v) = \begin{cases} 0 & \text{if } u = v, \\ \mathsf{cost}(u, v) & \text{if } (u, v) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

The recursion now looks like so:

$$\mathsf{dist}_k(u, v) = \min(\mathsf{dist}_{k-1}(u, v), \mathsf{dist}_{k-1}(u, k) + \mathsf{dist}_{k-1}(k, v))$$

The second term represents paths $i \xrightarrow{<k} k \xrightarrow{<k} j$.

Lemma

Floyd-Warshall runs in time $\Theta(n^3)$ and space $\Theta(n^2)$.

Comment

Note that the code for this is incredibly simple: three nested for-loops with a one-line body, no data structures. Almost impossible to get wrong.

This is an algorithm that breaks our “relax an edge” principle, it works with paths rather than inspecting single edges.

Exercise

Figure out how to detect negative cost cycles using Floyd-Warshall.

Exercise

Show that Floyd-Warshall works with negative costs.

How About Dijkstra?

Running the fancy heap version of Dijkstra n times produces $O(n(m + n \log n))$, which is good for sufficiently sparse graphs where $m \ll n^2$.

But how about negative costs?

Wild & Woolly Idea: Why not simply add some sufficiently large constant C to all edge costs to make them positive?

Exercise

Explain why $\mathsf{cost}'(u, v) = \mathsf{cost}(u, v) + C$ *does not work.*

Constants are too simple-minded, instead we can use a potential function $\Phi : V \to \mathbb{R}$ and define

$$\mathsf{cost}'(u,v) = \mathsf{cost}(u,v) + \Phi(u) - \Phi(v)$$

Now suppose you have a path $P = u, x_1, \ldots, x_k, v$. Then

$$\mathsf{cost}'(P) = \mathsf{cost}(P) + \Phi(u) - \Phi(v)$$

because the sum is telescoping.

So the offset in the modified cost is independent of the actual path taken, only source and target matter. But then we get the same shortest paths. Very neat.

But How?

We need to define and compute Φ. Here goes.

Cone a new vertex $\perp$ onto the old graph: add new edges $(\perp, x)$ of cost 0 for all $x \in V$. Then run Bellman-Ford on this modified graph to determine distances with source $\perp$ and target $x \in V$; the result is $\Phi(x) \leq 0$.

But then necessarily for any old edge (u, v)

$$\Phi(u) + \mathsf{cost}(u, v) \geq \Phi(v).$$

Hence $\mathsf{cost}'(u, v) \geq 0$.

Running time is $O(n(n+m)\log n)$, using plain heaps.
With Fibonacci heaps we get $O(nm + n^2 \log n)$.

Recall that we can interpret the adjacency matrix A of a graphs as a Boolean matrix, an $n \times n$ matrix over the Boolean semiring, the structure

$$\mathbb{B} = \langle \mathbf{2}, +, *, 0, 1 \rangle$$

where $+$ corresponds to disjunction, and $*$ to conjunction. It's not a ring since addition has no inverse.

Then A^k describes paths of length exactly k. To tackle reachability it suffices to compute

$$(A+I)^{n-1} = I + A + A^2 + \ldots + A^{n-1}.$$

Using fast exponentiation, this can be handled in $O(\log n)$ Boolean matrix multiplications (BMM).

We have seen that over better structures than $\mathbb{B}$ we can speed up matrix multiplication to $O(n^\omega)$ for some $\omega < 3$.

To exploit this for BMM, use a trick that will make any type theory person cringe: think of a Boolean matrix as an integer matrix. Then multiply and get back to "Boolean" by applying the sign function everywhere.

Switching the underlying algebraic structure sometimes works wonders.

OK, but reachability is a one-bit answer. Can we use a similar trick to get distances? What do we really need to perform matrix multiplication?

Recall: Semirings

A semiring is an algebraic structure

$$\langle X, \oplus, \otimes, 0, 1\rangle$$

with two binary operations ("addition" and "multiplication") that satisfies the following axioms:

- $\langle X, \oplus, 0\rangle$ and $\langle X, \otimes, 1\rangle$ are monoids, the former is commutative.
- The two operations are distributive: $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ and $(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$.
- 0 is an annihilator or null wrto $\otimes$: $x \otimes 0 = 0 \otimes x = 0$ for all x in M.

A semiring is commutative if $x \otimes y = y \otimes x$.
It is idempotent if $x \oplus x = x$.

Min-Plus Semiring

Here is a slightly exotic structure, the min-plus semiring, aka the tropical semiring.

$$\langle \mathbb{N}_\infty; \min, +, \infty, 0 \rangle$$

Note that "multiplication" here is ordinary addition.

This may look weird, but it is a perfectly good commutative, idempotent semiring.

Now let A be the cost matrix for our graph:

$$A(u,v) = \begin{cases} 0 & \text{if } u = v, \\ \mathsf{cost}(u,v) & \text{if } (u,v) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

Back to Shortest Paths

Since matrix multiplication only requires addition and multiplication (but not subtraction or division, at least the plain version) we can perform matrix multiplication over the min-plus semiring.

Claim: $A^k(u, v)$ is the cost of a shortest path from u to v of length exactly k.

To see this note

$$A^k(u, v) = \min_x \Big(A^{k-1}(u, x) + A(x, v) \Big)$$

To get all paths of length at most k replace A by $A + I$, where I is the identity matrix (multiplicative neutral element on the diagonal, additive neutral element everywhere else).

So it suffices to compute $(A + I)^{n-1}$ using fast exponentiation.

Here is another wild semiring, the Kleene semiring:

$$\langle \mathrm{Reg}_\Sigma; \cup, \cdot, {}^*, \emptyset, a \in \Sigma \rangle$$

all regular languages over an alphabet Σ with operations union, concatenation and Kleene star. Constants are $\emptyset$ and singletons $\{a\}$ for $a \in \Sigma$.

Theorem (Kleene)

Every regular language can be denoted by an expression over this structure, using only the given operations and constants.

These are usually called regular expressions and are critical for tools like `grep`: otherwise we would not have regular expressions and would have to specify our search patterns in terms of finite state machines.

PROOF?

No problem, essentially use the shortest path approach with one modification:

Fix some finite state machine on state set $[n]$ that describes a regular language L. Let $\alpha^k(p,q)$ be a regular expression for the language obtained by setting $p[q]$ as initial[final] state, and erasing all intermediate states $> k$. Then

$$\alpha^k(p,q) = \alpha^{k-1}(p,q) + \alpha^{k-1}(p,k) \cdot \alpha^{k-1}(k,k)^\star \cdot \alpha^{k-1}(k,q)$$

The only difference is that here we cannot ignore "paths" that loop back to k.

The whole language is denoted by

$$\alpha = \sum_{p \in I, q \in F} \alpha^n(p,q)$$

It's Useless

Note one huge problem with this approach: we are dealing with formal expressions, not nice objects like numbers or Booleans. As a consequence, the expressions at level k are roughly 4 times larger than at level $k-1$. We cannot really write them down.

One can try to simplify the expressions a bit, say, $\alpha + \alpha \rightsquigarrow \alpha$ or $\varepsilon \cdot \alpha \rightsquigarrow \alpha$

Alas, Conway has shown that the algebra of these expressions is not finitely axiomatizable, so the rewrite approach won't go very far.

Worse, it is known that the shortest expression may be exponentially larger than the corresponding finite state machine.

TSP

We are given a distance matrix dist $\in \mathbb{N}_\infty^{n\times n}$ and we trying to find a minimum cost tour $1, \pi(2), \pi(3), \ldots, \pi(n), 1$. Here π is a permutation of $[n]$ with fixed point 1.

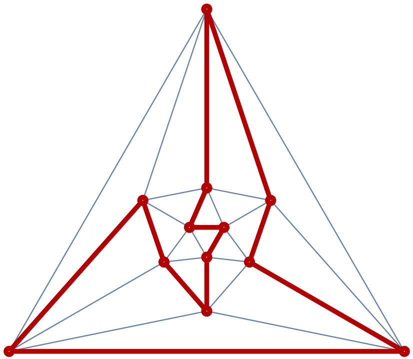
In the general case, the distance matrix is be arbitrary.

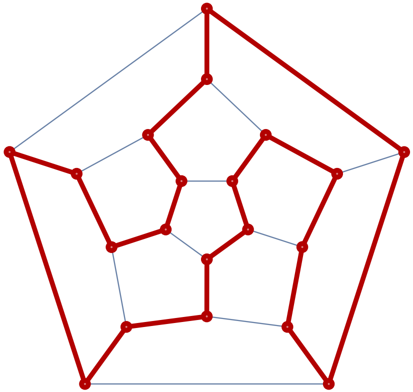
A restricted class of TSP use distances that come from some geometry: say, points in the plane and Euclidean distance.

In this case, the distance matrix is symmetric and obeys the triangle inequalities:

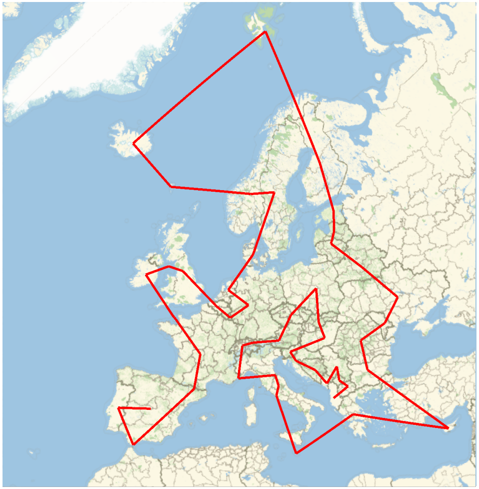
$$\mathsf{dist}(x,y) \leq \mathsf{dist}(x,z) + \mathsf{dist}(z,x)$$

This is useful for heuristics, but is still $\mathbb{NP}$-hard.





Cost is Euclidean distance if there is an edge, ∞ otherwise.



A variant where we leave out the last edge (that closes the cycle).

Pointed Subsets

We would like to use dynamic programming, but it is far from clear what sub-instances we should be considering. We are essentially dealing with a labeled graph, so it might be tempting to use, say, the method of Floyd-Warshall: temporarily remove locations $k+1, \ldots, n$. Alas, there seems to be no reasonable way to extend a tour from constraint k to $k+1$.

A pointed set is a pair (t, X) where $t \in X$. We are interested in pointed sets over $[n]$: $t \in X \subseteq [n]$.

The key observation is that we can interpret a pointed set (t, X) where $1 \in X$ as a partial solution to TSP: we have a good path from 1 to t that includes exactly all points in X.

Note: not a cycle, just a path.

Sub-Instances

Define

$$\mathsf{val}(t, X) = \min(\text{ cost of such a path in } X\ )$$

Now we can set up a recursion like so:

$$\mathsf{val}(t, X) = \begin{cases} \mathsf{cost}(1, t) & \text{if } X = \{1, t\}, \\ \min\big(\, \mathsf{val}(s, X - \{t\}) + \mathsf{cost}(s, t) \mid s \in X - \{1, t\}\,\big) & \text{otherwise.} \end{cases}$$

In the end we compute

$$\min_{t \neq 1} \mathsf{val}(t, [n]) + \mathsf{cost}(t, 1)$$

Claim: There are $n\,2^{n-1}$ pointed sets over $[n]$.

Thus we have $(n-1)2^{n-2}$ pointed sets over $[n]$ containing 1. Exponential, but we are dealing with another $\mathbb{NP}$-complete problem where the obvious brute force attack is around $n\,n!$ Recall Sterling's approximation

$$n! \sim \sqrt{2\pi n}\,(n/e)^n$$

Since we are manipulating sets, it is not so clear how expensive each step in the recursion is. As long as we can represent the sets as reasonably short bit-vectors (say, a few unsigned ints) we may assume constant time.

In that case, the total running time is $O(n^2 2^n)$.