Shortest Paths: DAG and Floyd-Warshall

Lecture 18

Part I

The Crucial Optimality Substructure

Shortest distance problems

Optimality substructure:

$$\mathrm{dist}(s,u) = min_{v \in In(u)} \left[\mathrm{dist}(s,v) + \ell(v,u)\right]$$

Shortest distance problems

Optimality substructure:

$$\mathrm{dist}(s,u) = min_{v \in In(u)} \left[\mathrm{dist}(s,v) + \ell(v,u)\right]$$

Bellman-Ford: $d(u) = min_{v \in In(u)} \left[d(v) + \ell(v,u)\right]$

Shortest distance problems

Optimality substructure:

$$\mathrm{dist}(s,u) = min_{v \in In(u)} \left[\mathrm{dist}(s,v) + \ell(v,u)\right]$$

Bellman-Ford: $d(u) = min_{v \in In(u)} \left[d(v) + \ell(v,u)\right]$

1. If v is on the shortest path of u, and $d(v) = \mathrm{dist}(s,v)$, then $d(u) = \mathrm{dist}(s,u)$ in the next iteration.

Shortest distance problems

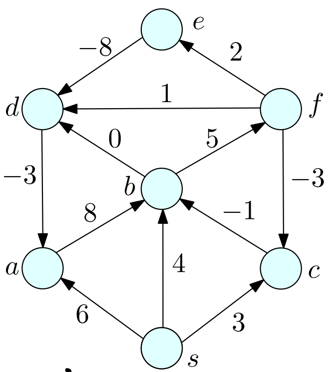
Optimality substructure:

$$\mathrm{dist}(s,u) = min_{v \in In(u)} \left[\mathrm{dist}(s,v) + \ell(v,u)\right]$$

Bellman-Ford: $d(u) = min_{v \in In(u)} \left[d(v) + \ell(v,u)\right]$

1. If v is on the shortest path of u, and $d(v) = \mathrm{dist}(s,v)$, then $d(u) = \mathrm{dist}(s,u)$ in the next iteration.
2. Initialize $d(s) = 0$, all $d(u) = \infty$, converge to the fixed point.

Example



	0	1	2	3	4	5	6
s	0	0	0	0	0	0	0
a	∞	6	6	1	−1	−1	−2
b	∞	4	2	2	2	2	2
c	∞	3	3	3	3	3	3
d	∞	∞	4	2	2	1	1
e	∞	∞	∞	11	9	9	9
f	∞	∞	9	7	7	7	7

$s \xrightarrow{6} a$

$\boxed{d} \xrightarrow{-3} a$

$s \xrightarrow{4} b$

$a \xrightarrow{8} b$

$\boxed{c} \xrightarrow{-1} b$

$s \xrightarrow{3} c$

$f \xrightarrow{-3} c$

$\boxed{b} \xrightarrow{0} d$

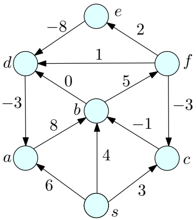
$f \xrightarrow{1} d$

$e \xrightarrow{-8} d$

$f \xrightarrow{2} e$

$b \xrightarrow{5} f$

Example



Parsimonious updates of Dijkstra

Optimality substructure:

$$\mathrm{dist}(s,u) = min_{v \in In(u)} \left[\mathrm{dist}(s,v) + \ell(v,u)\right]$$

Dijkstra: $d(u) = min_{v \in In(u), v \in X} \left[d(v) + \ell(v,u)\right]$

Parsimonious updates of Dijkstra

Optimality substructure:

$$\mathrm{dist}(s,u) = min_{v \in In(u)} \left[\mathrm{dist}(s,v) + \ell(v,u)\right]$$

Dijkstra: $d(u) = min_{v \in In(u), v \in X} \left[d(v) + \ell(v,u)\right]$

1. v in X is known to have $d(v) = d(s,v)$

Parsimonious updates of Dijkstra

Optimality substructure:

$$\mathrm{dist}(s,u) = min_{v \in In(u)} \left[\mathrm{dist}(s,v) + \ell(v,u)\right]$$

Dijkstra: $d(u) = min_{v \in In(u), v \in X} \left[d(v) + \ell(v,u)\right]$

1. v in X is known to have $d(v) = d(s,v)$
2. Only update u adjacent to X. Each edge is only updated once.
3. A good evaluation order saves a lot of work. We will see it again with DAG.

Shortest distance problems

Why didn't we use

$\mathrm{dist}(s, u) = min_v \left[\mathrm{dist}(s, v) + \mathrm{dist}(v, u)\right]$?

Shortest distance problems

Why didn't we use

$$\mathrm{dist}(s,u) = min_v\ [\mathrm{dist}(s,v) + \mathrm{dist}(v,u)]\ ?$$

Bellman-Ford? $d(u) = min_v\ [d(v) + d(v,u)]$?

Shortest distance problems

Why didn't we use

$$\text{dist}(s, u) = min_v \left[\text{dist}(s, v) + \text{dist}(v, u)\right] \text{ ?}$$

Bellman-Ford? $d(u) = min_v \left[d(v) + d(v, u)\right]$?

1. We will need to compute $d(v, u)$, for all v, when we only need distances from s. Extra work.
2. Will be useful for computing all-pair shortest distance. Floyd-Warshall

Part II

Shortest Paths in DAGs

Shortest Paths in a DAG

Single-Source Shortest Path Problems

Input A directed acyclic graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Shortest Paths in a DAG

Single-Source Shortest Path Problems

Input A directed acyclic graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Simplification of algorithms for DAGs

1. No cycles and hence no negative length cycles!
2. Can order nodes using topological sort

Algorithm for DAGs

1. Want to find shortest paths from s. Ignore nodes not reachable from s.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Algorithm for DAGs

1. Want to find shortest paths from s. Ignore nodes not reachable from s.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Observation:

1. shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.

Algorithm for DAGs

```
for i = 1 to n do
        d(s, v_i) = ∞
d(s, s) = 0

for i = 1 to n − 1 do
        for each edge (v_i, v_j) in Out(v_i) do
            d(s, v_j) = min{d(s, v_j), d(s, v_i) + ℓ(v_i, v_j)}

return d(s, ·) values computed
```

Correctness: induction on i and observation in previous slide.
Running time: $O(m + n)$ time algorithm!

Part III

All Pairs Shortest Paths

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.
3. Find shortest paths for *all* pairs of nodes.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Dijkstra's algorithm for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $O(nm)$.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms n times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms n times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$.

Can we do better?

Optimality substructure

Why don't we use

$$\mathrm{dist}(s, u) = min_v \, [\mathrm{dist}(s, v) + \mathrm{dist}(v, u)] \text{ ?}$$

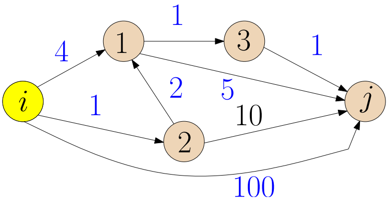
Optimality substructure

Why don't we use

$$\mathrm{dist}(s, u) = min_v\ [\mathrm{dist}(s, v) + \mathrm{dist}(v, u)]\ ?$$

What is a smart recursion?

A naive recursion



A naive recursion

Running Time: $O(n^4)$, Space: $O(n^3)$.

A naive recursion

Running Time: $O(n^4)$, Space: $O(n^3)$.

Worse than Bellman-Ford: $O(n^2 m)$, when $m = O(n^2)$.

A naive recursion

Running Time: $O(n^4)$, Space: $O(n^3)$.

Worse than Bellman-Ford: $O(n^2 m)$, when $m = O(n^2)$.

1. It's wasteful because the intermediate nodes can be any node. As a result, we compute the same path many times.

A naive recursion

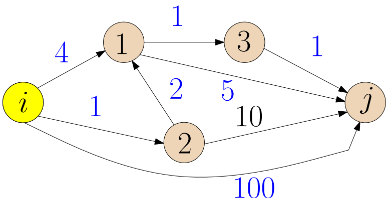
Running Time: $O(n^4)$, Space: $O(n^3)$.

Worse than Bellman-Ford: $O(n^2 m)$, when $m = O(n^2)$.

1. It's wasteful because the intermediate nodes can be any node. As a result, we compute the same path many times.
2. Idea: Restrict the set of intermediate nodes.

All-Pairs: Recursion on index of intermediate nodes

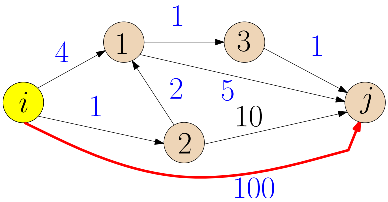
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest index of an *intermediate node* is at most k (could be $-\infty$ if there is a negative length cycle).



$$
\begin{aligned}
dist(i, j, 0) &= \\
dist(i, j, 1) &= \\
dist(i, j, 2) &= \\
dist(i, j, 3) &=
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

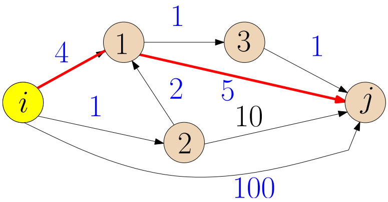
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest index of an *intermediate node* is at most k (could be $-\infty$ if there is a negative length cycle).



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= \\
dist(i, j, 2) &= \\
dist(i, j, 3) &=
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

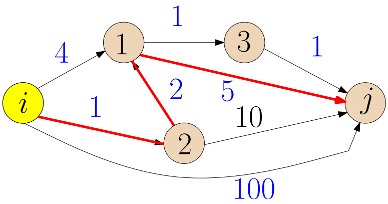
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest index of an *intermediate node* is at most k (could be $-\infty$ if there is a negative length cycle).



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= \\
dist(i, j, 3) &=
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

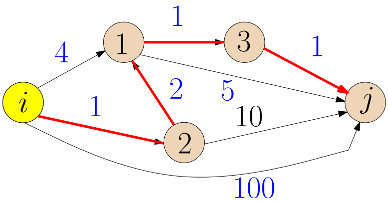
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest index of an *intermediate node* is at most k (could be $-\infty$ if there is a negative length cycle).



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= 8 \\
dist(i, j, 3) &=
\end{aligned}
$$

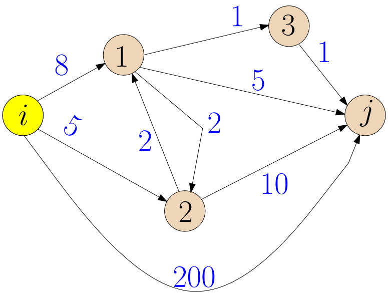
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest index of an *intermediate node* is at most k (could be $-\infty$ if there is a negative length cycle).



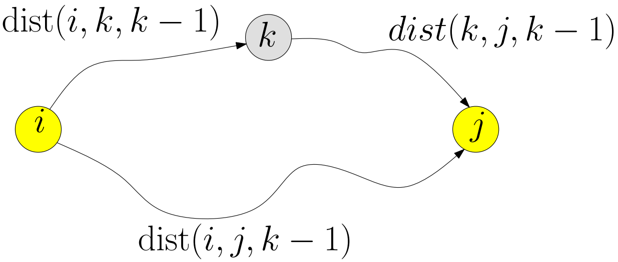
$$
\begin{aligned}
dist(i, j, 0) &= \mathbf{100} \\
dist(i, j, 1) &= \mathbf{9} \\
dist(i, j, 2) &= \mathbf{8} \\
dist(i, j, 3) &= \mathbf{5}
\end{aligned}
$$

For the following graph, **dist(i, j, 2)** is...



- **9**
- **10**
- **11**
- **12**
- **15**

All-Pairs: Recursion on index of intermediate nodes



$$dist(i, j, k) = \min \begin{cases} dist(i, j, k-1) \\ dist(i, k, k-1) + dist(k, j, k-1) \end{cases}$$

Base case: $dist(i, j, 0) = \ell(i, j)$ if $(i, j) \in E$, otherwise ∞

All-Pairs: Recursion on index of intermediate nodes

If i can reach k *and* k can reach j *and* $dist(k, k, k-1) < 0$ then G has a negative length cycle containing k and $dist(i, j, k) = -\infty$.

Recursion below is valid only if $dist(k, k, k-1) \geq 0$. We can detect this during the algorithm or wait till the end.

$$dist(i, j, k) = \min \begin{cases} dist(i, j, k-1) \\ dist(i, k, k-1) + dist(k, j, k-1) \end{cases}$$

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ℓ(i, j)  (* ℓ(i, j) = ∞ if (i, j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min { dist(i, j, k − 1),
                                  dist(i, k, k − 1) + dist(k, j, k − 1)
for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ℓ(i, j)  (* ℓ(i, j) = ∞ if (i, j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
```

$$dist(i,j,k) = \min \begin{cases} dist(i,j,k-1), \\ dist(i,k,k-1) + dist(k,j,k-1) \end{cases}$$

```
for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

Running Time:

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ℓ(i, j)  (* ℓ(i, j) = ∞ if (i, j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min { dist(i, j, k − 1),
                                  dist(i, k, k − 1) + dist(k, j, k − 1) }
for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

$$dist(i,j,k) = \min \begin{cases} dist(i,j,k-1), \\ dist(i,k,k-1) + dist(k,j,k-1) \end{cases}$$

Running Time: $O(n^3)$, Space: $O(n^3)$.

Graph Modeling

Lecture

Part I

An Application to `make`

Make/Makefile

A. I know what make/makefile is.
B. I do NOT know what make/makefile is.

`make` Utility [Feldman]

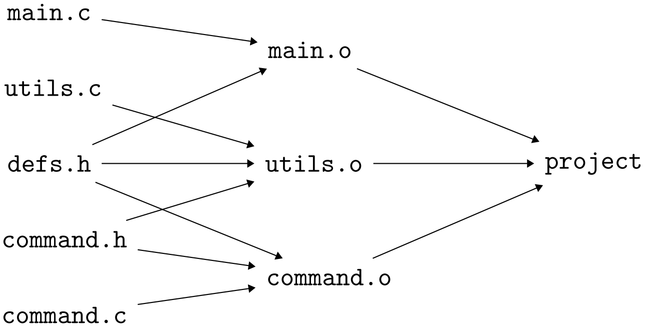
1. Unix utility for automatically building large software applications
2. A makefile specifies
   1. Object files to be created,
   2. Source/object files to be used in creation, and
   3. How to create them

An Example makefile

```
project:  main.o utils.o command.o
    cc -o project main.o utils.o command.o

main.o:  main.c defs.h
    cc -c main.c
utils.o:  utils.c defs.h command.h
    cc -c utils.c
command.o:  command.c defs.h command.h
    cc -c command.c
```

`makefile` as a Digraph



Computational Problems for `make`

1. Is the `makefile` reasonable?
2. If it is reasonable, in what order should the object files be created?
3. If it is not reasonable, provide helpful debugging information.
4. If some file is modified, find the fewest compilations needed to make application consistent.

Algorithms for `make`

1. Is the `makefile` reasonable? Is G a DAG?
2. If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
3. If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
4. If some file is modified, find the fewest compilations needed to make application consistent.
    1. Find all vertices reachable (using **DFS**/**BFS**) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.

Part II

Application to Currency Trading

Why Negative Lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lengths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow

Currency Trading

Input: n currencies and for each ordered pair (a, b) the *exchange rate* for converting one unit of a into one unit of b.

Questions:

1. Is there an arbitrage opportunity?
2. Given currencies s, t what is the best way to convert s to t (perhaps via other intermediate currencies)?

Concrete example:

1. 1 Chinese Yuan $=$ $\mathbf{0.1116}$ Euro
2. 1 Euro $=$ $\mathbf{1.3617}$ US dollar
3. 1 US Dollar $=$ $\mathbf{7.1}$ Chinese Yuan.

Thus, if exchanging $\mathbf{1\ \$} \rightarrow$ Yuan $\rightarrow$ Euro $\rightarrow \mathbf{\$}$, we get: $\mathbf{0.1116 * 1.3617 * 7.1 = 1.07896\$}$.

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.

Create currency trading *directed* graph $G = (V, E)$:

1. For each currency i there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. edge length $\ell(v_i, v_j) =$

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.

Create currency trading *directed* graph $G = (V, E)$:

1. For each currency i there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.

Create currency trading *directed* graph $G = (V, E)$:

1. For each currency i there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if G has a negative length cycle.
2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^{-d} units of j.

Reducing Currency Trading to Shortest Paths

Math recall - relevant information

1. $\log(\alpha_1 * \alpha_2 * \cdots * \alpha_k) = \log \alpha_1 + \log \alpha_2 + \cdots + \log \alpha_k$.
2. $\log x > 0$ if and only if $x > 1$.