Shortest Path Algorithms and Greed algorithms

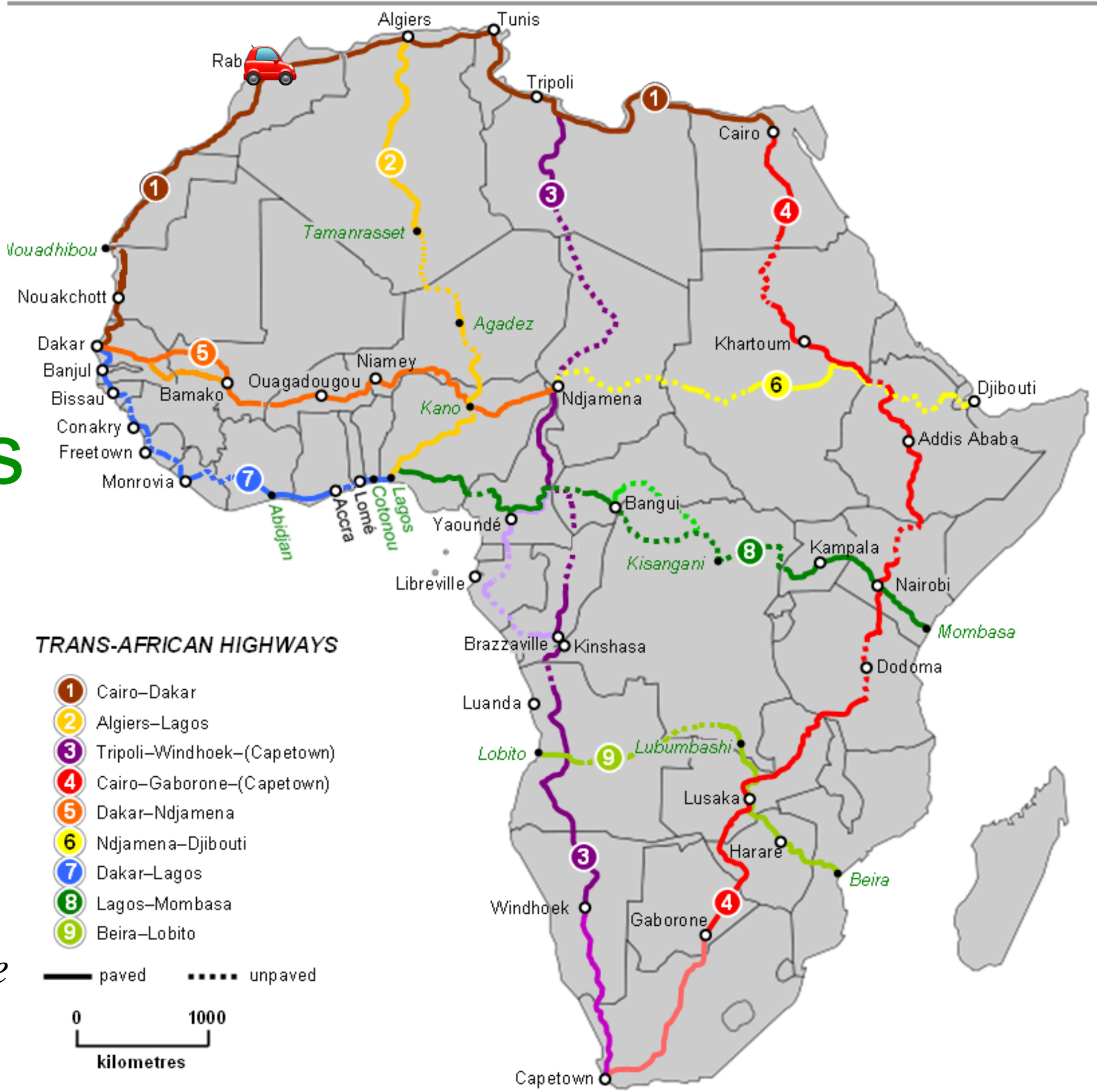
Shortest Path Algorithms

© 2017 Linda Sellie

Single Source Shortest Path Algorithms

Find shortest distance between two points on the graph.

Edge weight can be negative (for some algorithms) but we *don't allow negative weight cycles*



© 2017 Linda Sellie

By I, Rexparry sydney, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2427121

Single Source Shortest Path Algorithms

$w(p) = \text{weight of path } p = \langle v_0, v_1, \ldots, v_k \rangle = \sum_{i=1}^{k} w(v_{i-1}, v_i) = \text{ sum of weights on path } p$

shortest-path weight u to v :

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\sim\!>} v & \text{if there exists a path } u \sim\!> v \\ \infty & \text{otherwise} \end{cases}$$

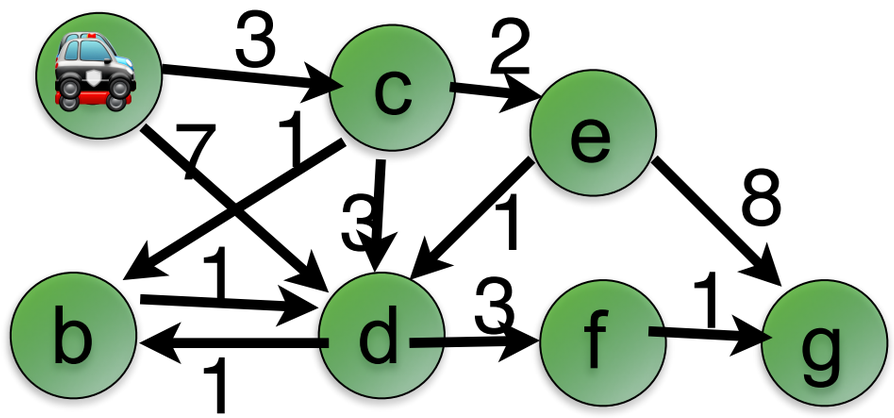
Travel from *a* to *g*

p = a, d, f , g w(p) = 11

p' = a, c, e, g w(p') = 13

p'' = a, c, e, d, g w(p'') = 10

p''' = a, c, b, d, g w(p''') = 9



$\delta(a,g) = 9$

© 2017 Linda Sellie

Observation: A shortest* path doesn't have a cycle

reasons:

negative weight cycles are not well defined

$p_{a\,j}$ = a, h, i, a h i, h, i, ..., j $w(p_{a\,j}) = -\infty$

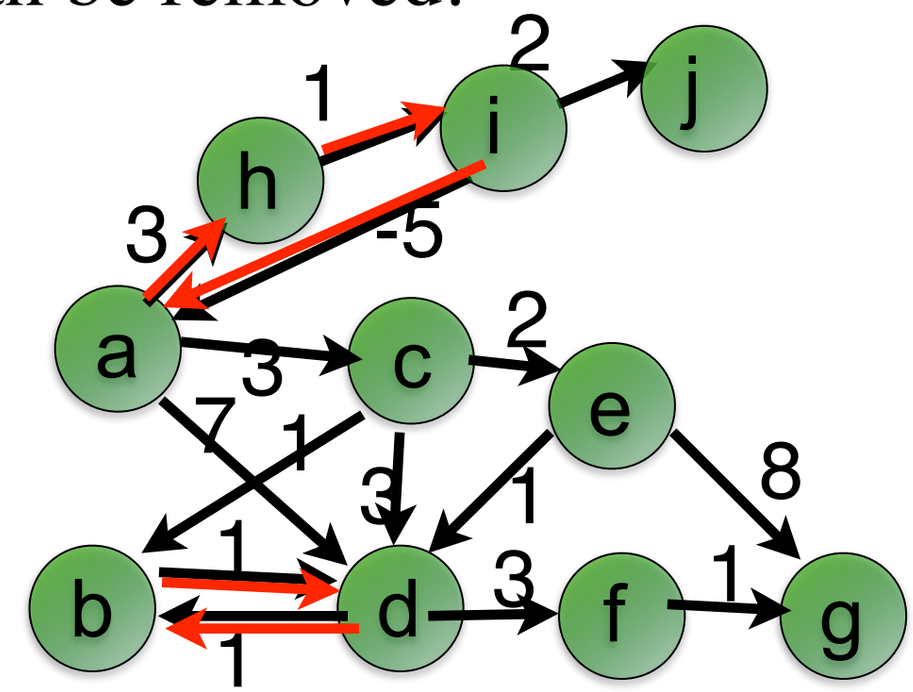
A positive weight cycle can be removed

$p_{a\,g}$ = a, c, b, d, b, d, f, g $w(p_{a\,g}) = 11$

$p'_{a\,g}$ = a, c, b, d, f, g $w(p'_{a\,g}) = 9$

*A zero weight cycle doesn't matter and can be removed.

So we assume they don't have them…



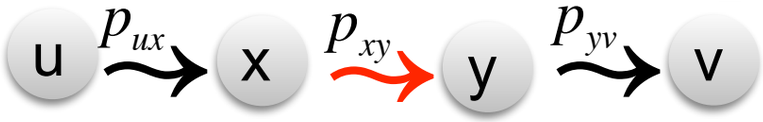
© 2017 Linda Sellie

Optimal Substructure

Lemma: Any subpath of a shortest path is a shortest path

Proof: Cut-and-paste.

Proof by contradiction, **Suppose not**



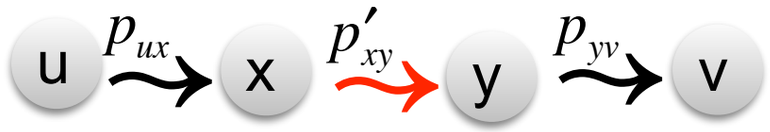
Suppose this path, p, is a shortest path from u to v

$$\delta(u,v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$$

Suppose there is a shorter path from x to y, $x \overset{p'_{xy}}{\rightsquigarrow} y$

then $w(p'_{xy}) < w(p_{xy})$

construct p':



Then $w(p') = w(p_{ux}) + w(p'_{xy}) + w(p_{yv})$

$$< w(p_{ux}) + w(p_{xy}) + w(p_{yv})$$

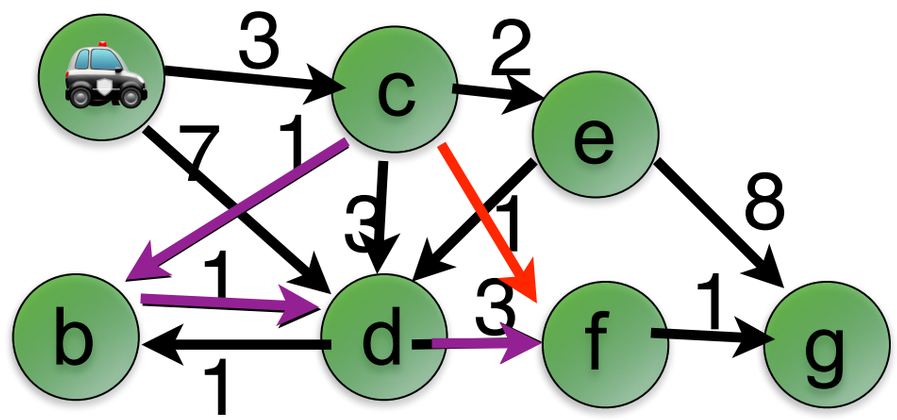
$$= w(p)$$ ◆◆

Dynamic Programming might apply

Greedy Algorithm might apply

p''' = a, c, b, d, f, g w(p''') = 9

p''' = a, c, f, g w(p''') = 5



© 2017 Linda Sellie

Output of single source shortest path algorithms

For every vertex, v:

$v.d$ shortest path estimate - initially ∞

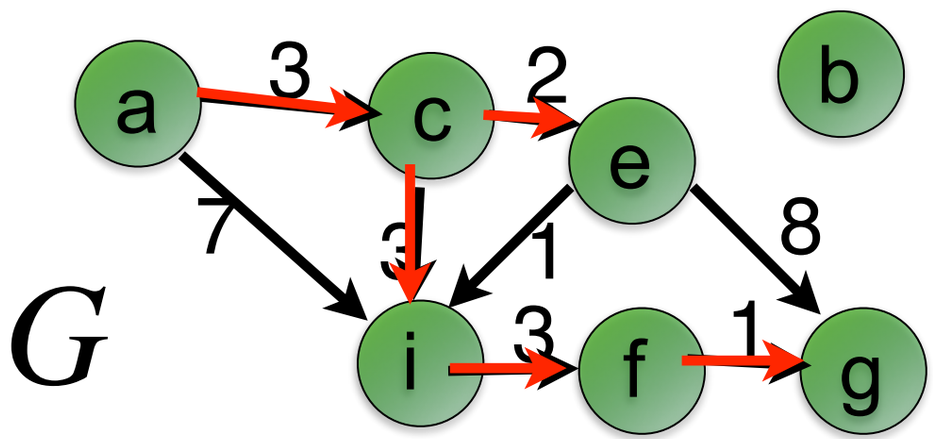
Predecssor subgraph $G_\pi = (V_\pi, E_\pi)$

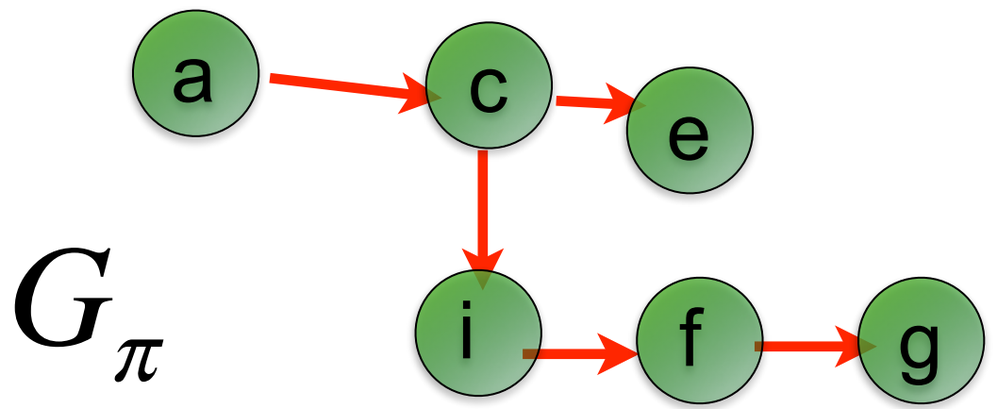
$V_\pi = \{v \in V \mid v.\pi \neq NIL\} \cup \{s\}$ $\quad E_\pi = \{(v.\pi, v) \in E \mid v \in V_\pi - \{s\}\}$

$v.\pi$ predecessor on path can change during the algorithm

Creates a tree (shortest-paths tree)

If no predecssor $v.\pi$ = NIL





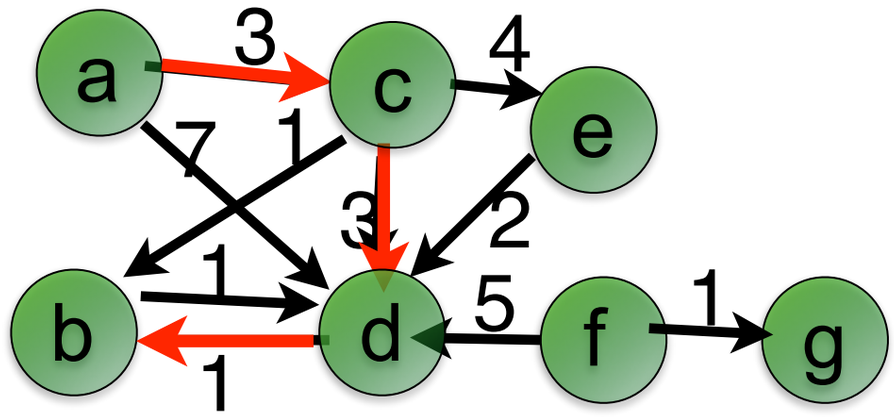
a	b	c	i	e	f	g
0	∞	3	6	5	9	10

$E_\pi = \{(a,c),(c,e),(c,i),(e,f),(f,g)\}$

$V_\pi = \{a,c,i,e,f,g\}$

Initialization

INIT-SINGLE-SOURCE(G, s)
 for each v∈G.V
  v.d = ∞
  v.π = NIL
 s.d = 0



a	b	c	d	e	f	g
0	7	3	6	∞	∞	∞

Relaxation of an edge

RELAX(u, v, w)
 if v.d > u.d + w(u, v)
  v.d = u.d + w(u, v)
  v.π = u

if you have found a shorter path to v by going from s to u and u to v,, **update the distance from s to v**

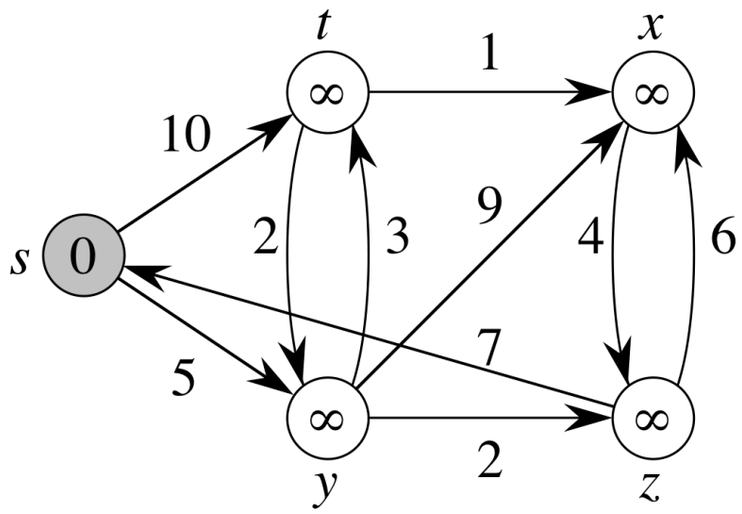
RELAX(a,c,w)
RELAX(a,d,w)
RELAX(d,b,w)
RELAX(c,d,w)
RELAX(d,b,w)

© 2017 Linda Sellie

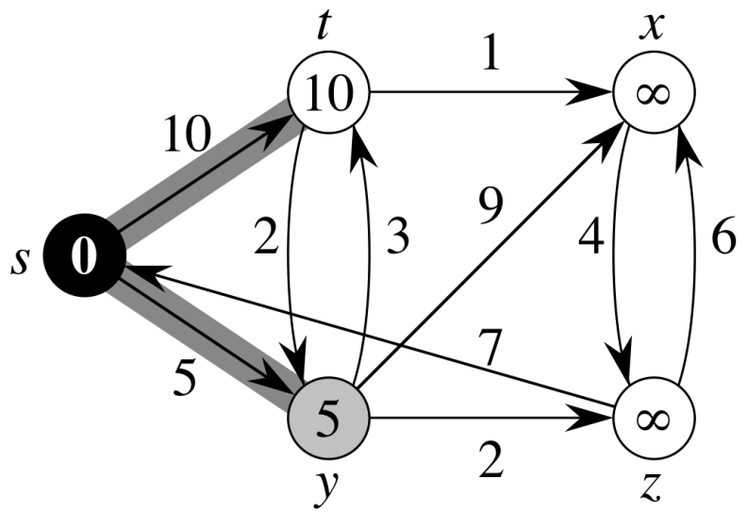
```
DIJKSTRA(G,w,s)
  INIT-SINGLE-SOURCE(G,s)
  S = ∅
  Q = G.V
  WHILE Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each v ∈ G.Adj[u])
        RELAX(u,v,w)
```

```
INIT-SINGLE-SOURCE(G, s)
 for each v∈G.V
    v.d = ∞
    v.π = NIL
  s.d = 0
```

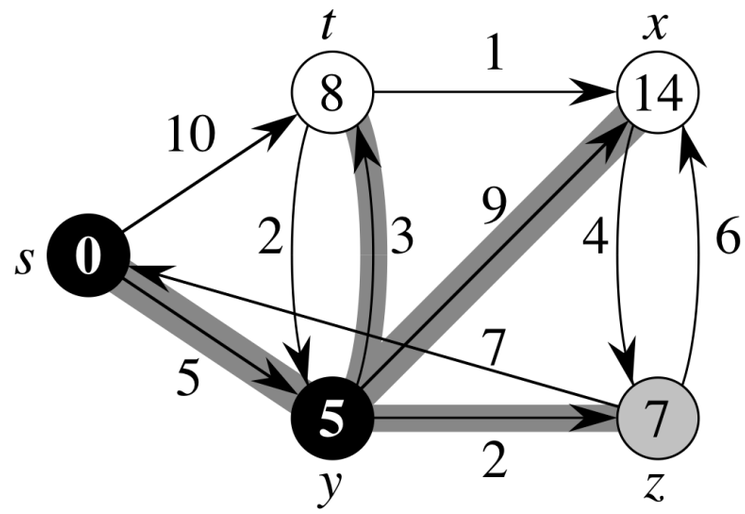
```
RELAX(u, v, w)
 if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```



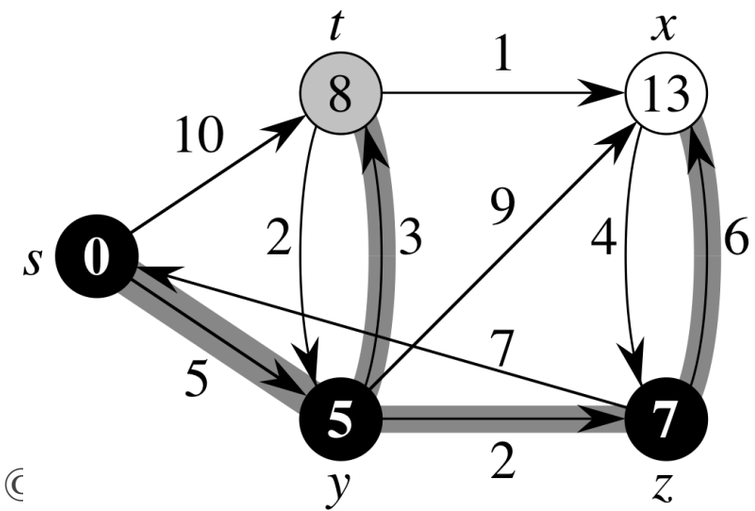
(a)

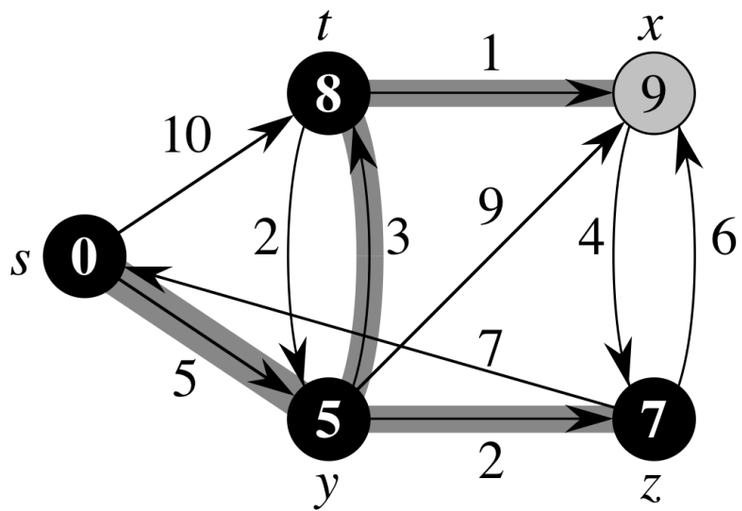


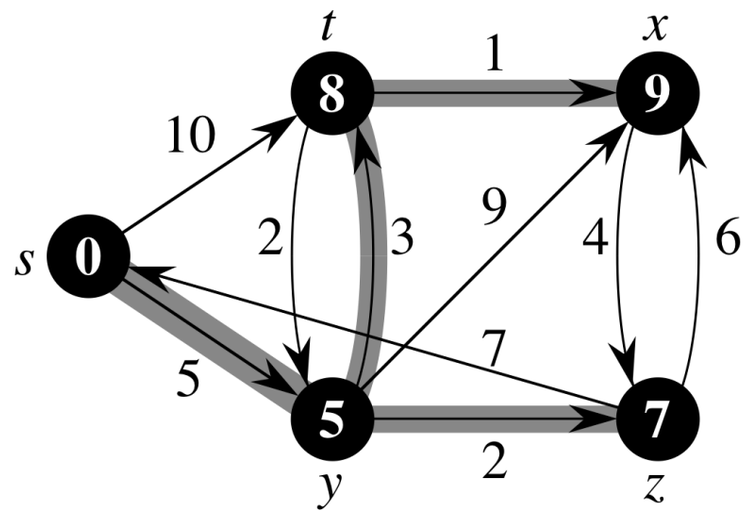
(b)



(c)







©

Dijkstra Single Source Shortest Path Algorith

NO negative-weight edges allowed!

Computes v.d and v.π for all v∈ V

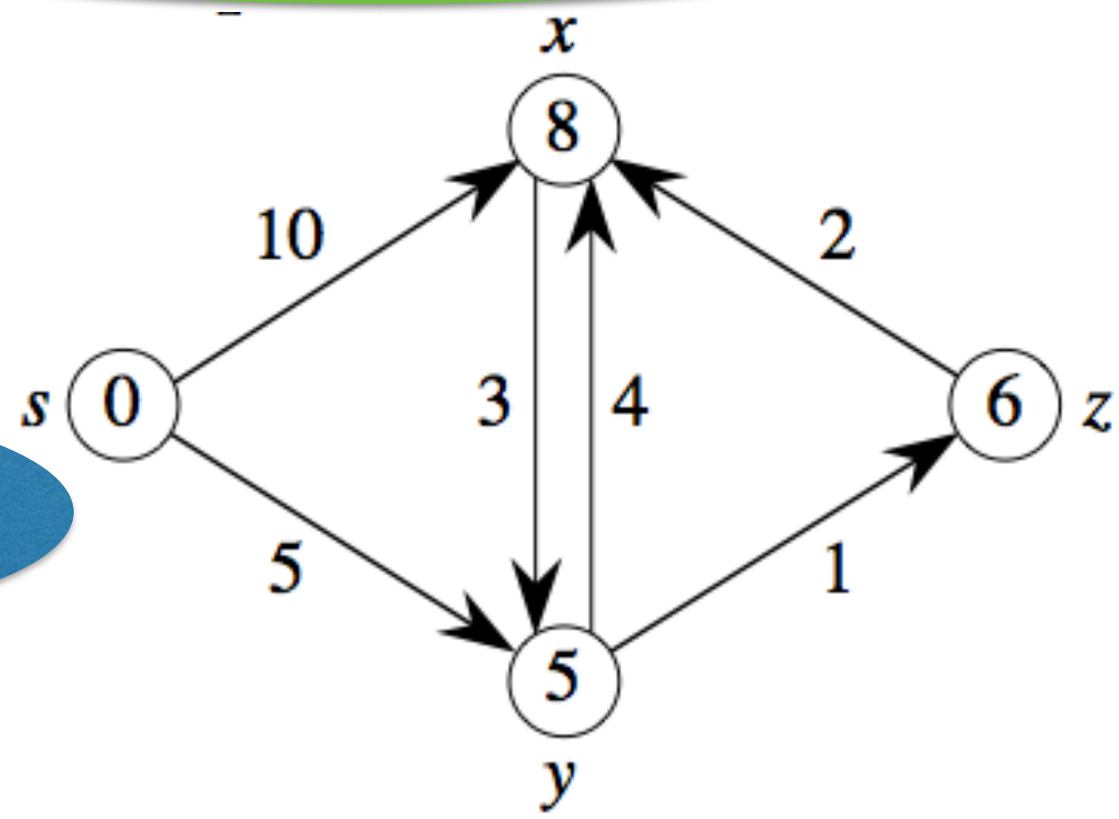
```
DIJKSTRA(G, w, s)
   INIT-SINGLE-SOURCE(G, s)
   S = ∅
   Q = G.V
   WHILE Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each v ∈ G.Adj[u])
         RELAX(u, v, w)   // How long does this take!!!!!
                          // Hint: it is NOT O(1)!
```

Running Time?

min-heap $\Theta(E \lg V + V \lg V)$

Looks like Prim's algorithm - we always choose the smallest to extend our current solution

Use shortest-path weights as keys i.e. use u.d



Greedy - always extending the graph by choosing the "lightest edge" (adding the "closest vertex in V-S to add to S

© 2017 Linda Sellie

Correctness

Loop Invariant:

At the start of every iteration of the while loop, $v.d = \delta(s,v)$ for all $v \in S$

Initialization:

Initially S = $\varnothing$, so trivially true

Termination:

Initially Q = $\varnothing$ ☞ $S = V$ ☞ $v.d = \delta(s,v)$ for all $v \in V$

© 2017 Linda Sellie

Loop Invariant:

At the start of every iteration of the while loop, $v.d = \delta(s,v)$ for all $v \in S$

Maintenance:

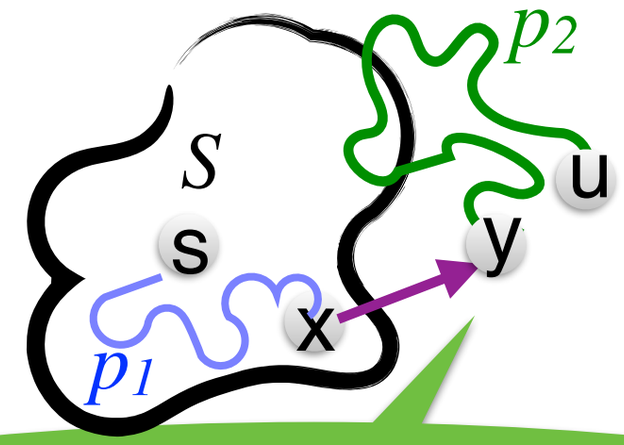
Need to show that $u.d = \delta(s,u)$ when u is added to S in the loop

Suppose not! Suppose u is added to S, but $u.d \neq \delta(s,u)$

WLOG let u be the first vertex added to S that does not have the optimal path

$u \neq s$ $s \in S$ Let p be a shortest path $s \overset{p}{\rightsquigarrow} u$

Decompose p into subpaths: $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$



Claim: $y.d = \delta(s,y)$ when u was added to S

Proof: $x.d = \delta(s,x)$ when u was added to S

relaxed (x, y, w) when x was added to S. ■

Since y is on shortest path $s \rightsquigarrow u$, and all edges are non-negative ☞

$$\delta(s,y) \leq \delta(s,u) \quad \text{☞} \quad y.d = \delta(s,y) \leq \delta(s,u) \leq u.d$$

But, u and y were in Q, and we chose u so,

$$u.d \leq y.d \text{ ☞ } u.d = y.d \text{ ☞ } \delta(s,y) = \delta(s,u)$$ ◗◖

More Shortest Path Algorithms

Goal: $O(V^3)$ for all graphs

Greedy Algorithms

© 2017 Linda Sellie

No negative edges.
We already know how to do this!
Run DIJKSTRA's algorithm from every vertex!
$O(VE \lg V + V^2 \lg V)$
Dense Graph?
Sparse Graph?

All Pairs Shortest Path Algorithm

Goal: $O(V^3)$ for all graphs

Given a set of cities, can you find the shortest distance between all pairs of cities?

© 2017 Linda Sellie

Steps for Developing a Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution

   all subpaths of a shortest path are shortest paths
2. Recursively define the value of an optimal solution

3. Compute the value of an optimal solution

4. Construct an optimal solution from computed information

© 2017 Linda Sellie

We will allow negative edges but assume there are not any negative weight cycles

Recall:
Optimal substructure!
subpaths of shortest paths are shortest paths

All Pairs Shortest Path Algorithm

Given G = (V, E), and $w : E \rightarrow \mathbb{R}$, |V| = n

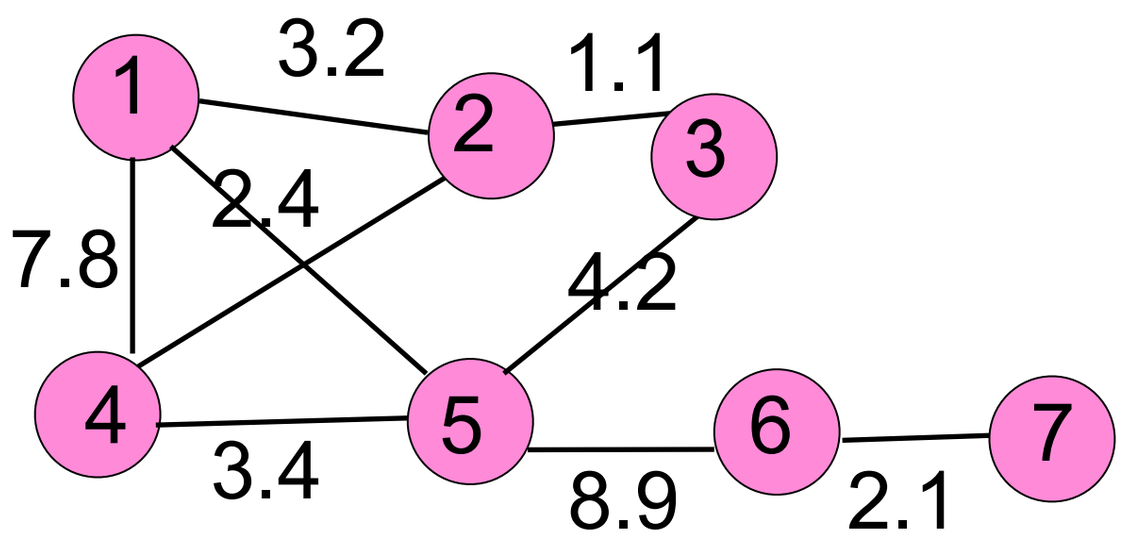
WLOG, number the vertices: 1, 2,…, n

$W = (w_{ij})$

We will represent the graph using a matrix instead of an adjacency list

	1	2	3	4	5	6	7
1	0	3.2	∞	7.8	2.4	∞	∞
2	3.2	0	1.1	2.4	∞	∞	∞
3	∞	1.1	0	∞	4.2	∞	∞
4	7.8	2.4	∞	0	3.4	∞	∞
5	2.4	∞	4.2	3.4	0	8.9	∞
6	∞	∞	∞	∞	8.9	0	2.1
7	∞	∞	∞	∞	∞	2.1	0

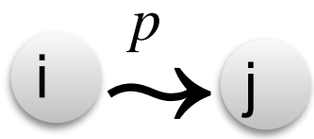
$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of } (i,j) & \text{if } i \neq j, (i,j) \in E \\ \infty & \text{if } i \neq j, (i,j) \notin E \end{cases}$$



© 2017 Linda Sellie

Using the Optimal Substructure

Consider a shortest path, ***p***, from ***i*** to ***j***

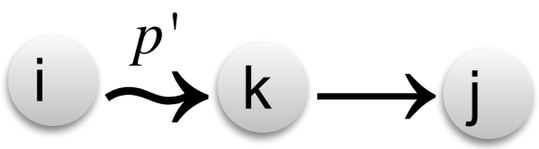


(We are assuming there are no negative-weight cycles in ***p***)

If ***i = j***, then ***p*** has weight ***0***

Otherwise, we know the path has length at least ***1***

and we can decompose ***p***



By the previous lemma, we know that ***p'*** is optimal so

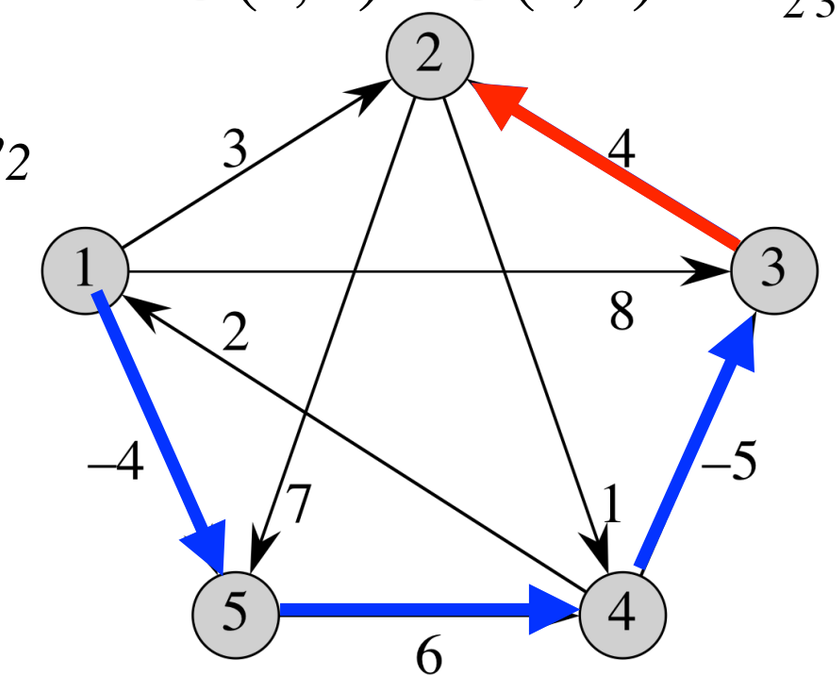
$$\delta(i,j) = \delta(i,k) + w_{kj}$$

As before, we define $\delta(i,j)$ to be the optimal solution

p = v_1, v_5, v_4, v_3, v_2

p' = v_1, v_5, v_4, v_3

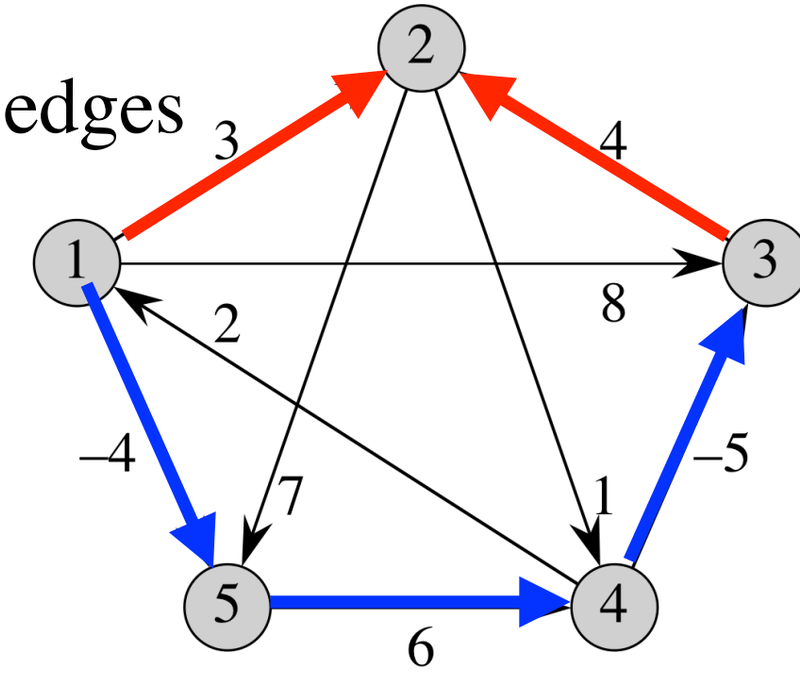
$$\delta(1,2) = \delta(1,3) + w_{23}$$



© 2017 Linda Sellie

Recursive Solution

$l_{i,j}^{(m)}$ = weight of shortest i ~> j path that contains $\leq m$ edges



case 1 $m = 0$

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

case 2 $m \geq 1$

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\} \right)$$

$$= \min_{1 \leq k \leq n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\} \text{ since } w_{jj} = 0$$

$$l^0(1,2) = \infty \qquad l^0(1,1) = 0$$

$$l^1(1,2) = l^0(1,1) + w_{12} = 0 + 3$$

$$l^2(1,2) = l^1(1,2)$$

$$l^3(1,2) = l^2(1,2) = 3$$

$$l^4(1,2) = l^3(1,3) + w_{23} = -3 + 4$$

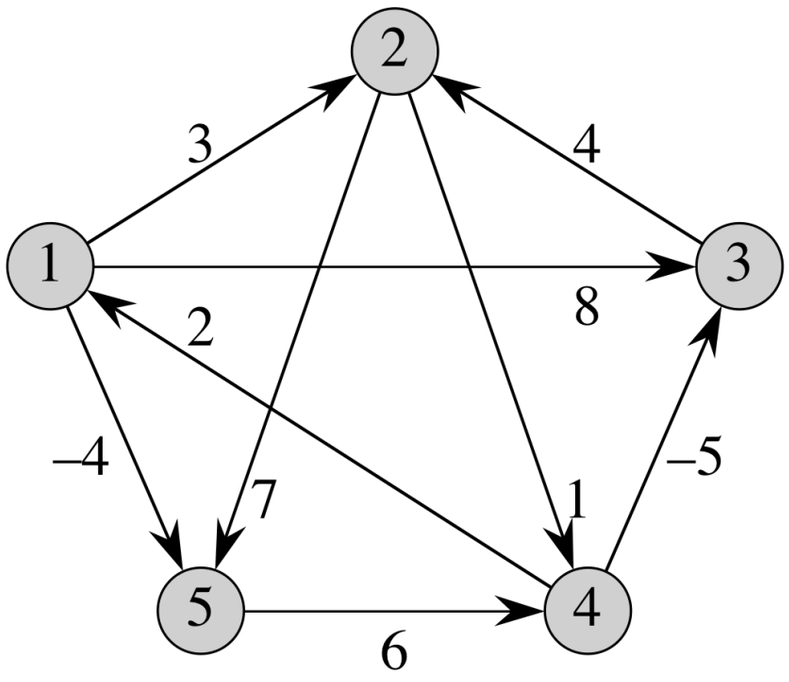
$$l^5(1,2) = l^4(1,2)$$

Observe that if $m = 1$ $l_{ij}^{(1)} = w_{ij}$

the original adjacency matrix!

Observe that $\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = l_{ij}^{(n+2)} = \cdots$

© 2017 Linda Sellie




```
EXTEND(L, W, n)
  let L' = (l'ij) be a new n×n matrix
  for i = 1 to n
      for j = 1 to n
          l'ij = ∞
          for k = 1 to n
              l'ij = min(l'ij, lik + wkj)
  return L'
```

$$W = L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$l_{35}^{m} = \min\{l_{31}^{m-1} + w_{15}, l_{32}^{m-1} + w_{25}, l_{33}^{m-1} + w_{35}, l_{34}^{m-1} + w_{45}, l_{35}^{m-1} + w_{55}\}$$

© 2017 Linda Sellie

Dynamic Programming Solution (Bottom up)

$L^{m-1}, W \rightarrow L^{m}$

```
EXTEND(L, W, n)
  let L' = (l'_ij) be a new n×n matrix
  for i = 1 to n
      for j = 1 to n
          l'_ij = ∞
          for k = 1 to n
              l'_ij = min(l'_ij, l_ik + w_kj)
  return L'
```

Running Time?

$\Theta(n^3)$

Can we do this faster?
Inspired by matrix multiplication?

Compute each $L^{(m)}$

```
SLOW-APSP(W, n)
  L^(1) = W
  for m = 2 to n - 1
      let L^(m) be a new n×n matrix
      L^(m) = EXTEND (L^(m-1), W, n)
  return L^(n-1)
```

Running Time?

$\Theta(n^4)$

© 2017 Linda Sellie

Dynamic Programming Solution (Bottom up)

$L^{m-1}, W \rightarrow L^m$

```
EXTEND(L, W, n)
  let L' = (l'_ij) be a new n×n matrix
  for i = 1 to n
      for j = 1 to n
          l'_ij = ∞
          for k = 1 to n
              l'_ij = min(l'_ij, l_ik + w_kj)
  return L'
```

Running Time?

$\Theta(n^3)$

Compute some $L^{(m)}$

```
FASTER-APSP(W, n)
  L^(1) = W
  m = 1
  while m < n - 1
      let L^(2m) be a new n×n matrix
      L^(2m) = EXTEND (L^(m), L^(m), n)
      m = 2m
  return L^(n-1)
```

Running Time?

$\Theta(n^3 \lg n)$

© 2017 Linda Sellie

Floyd-Warshall
NP-Completeness

© 2017 Linda Sellie

Steps for Developing a Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.

   all subpaths of a shortest path are shortest paths

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution.

4. Construct an optimal solution from computed information

© 2017 Linda Sellie

We will allow negative edges but assume there are not any negative weight cycles

Recall: Optimal substructure! subpaths of shortest paths are shortest paths

All Pairs Shortest Path Algorithm

Given G = (V, E), and $w : E \rightarrow \mathbb{R}$, |V| = n

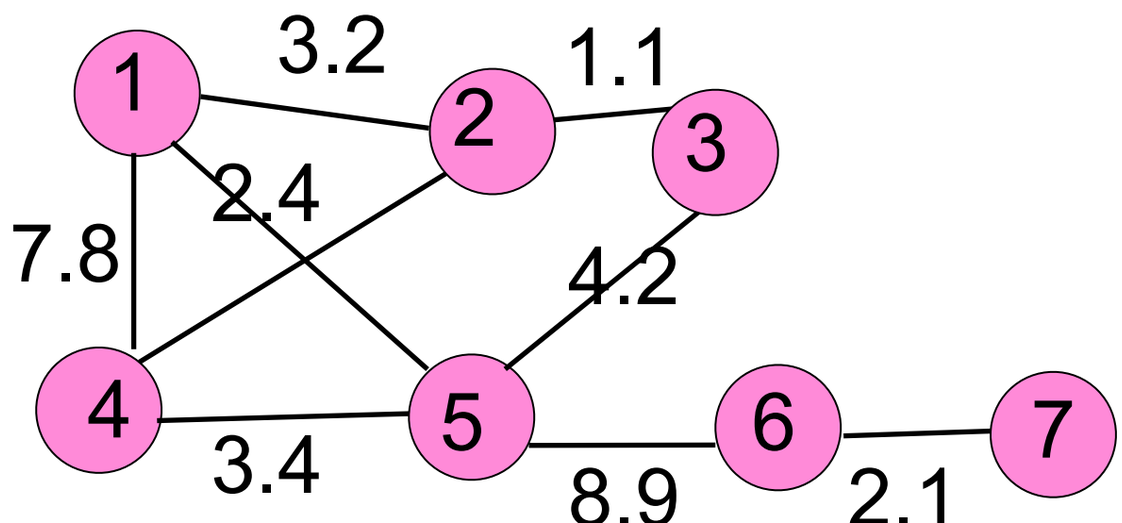
WLOG, number the vertices: 1, 2,…, n

$W = (w_{ij})$

We will represent the graph using a matrix instead of an adjacency list

	1	2	3	4	5	6	7
1	0	3.2	∞	7.8	2.4	∞	∞
2	3.2	0	1.1	2.4	∞	∞	∞
3	∞	1.1	0	∞	4.2	∞	∞
4	7.8	2.4	∞	0	3.4	∞	∞
5	2.4	∞	4.2	3.4	0	8.9	∞
6	∞	∞	∞	∞	8.9	0	2.1
7	∞	∞	∞	∞	∞	2.1	0

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of } (i,j) & \text{if } i \neq j, (i,j) \in E \\ \infty & \text{if } i \neq j, (i,j) \notin E \end{cases}$$

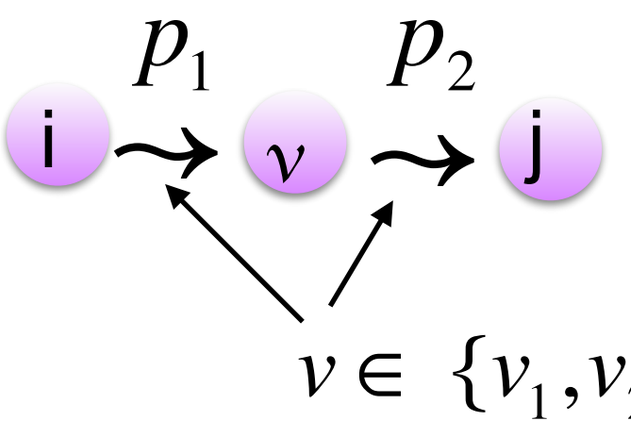


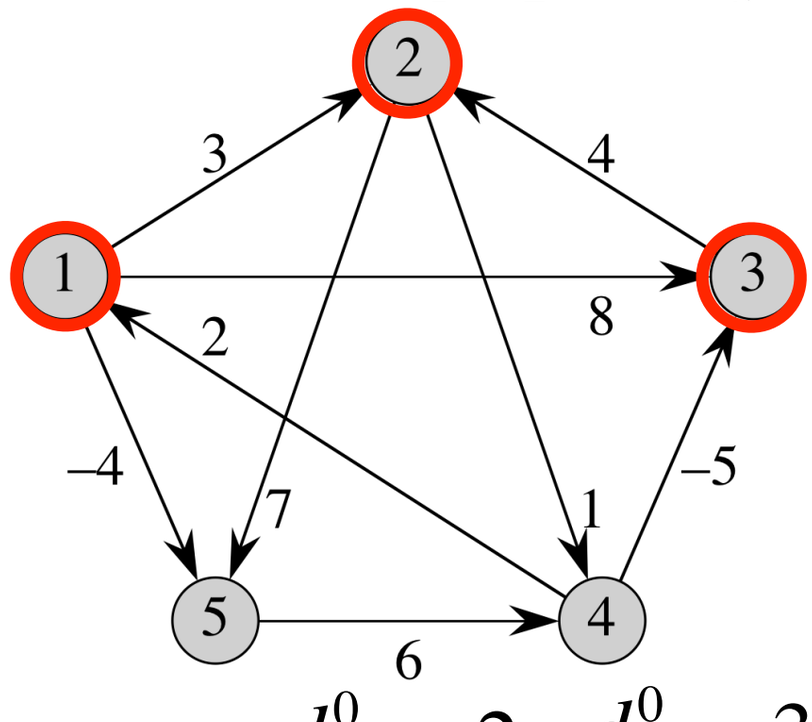
© 2017 Linda Sellie

Floyd-Warshall Algorithm

$d_{ij}^{(k)}$ = shortest path from i to j with intermediate vertices in $\{v_1, v_2, \ldots, v_k\}$

i and j might not be in $\{v_1, v_2, \ldots, v_k\}$





graph from CLRS

intermediate vertices are in $\{V_1, V_2, \ldots, V_k\}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k > 0 \end{cases}$$

$d_{4,2}^{0} = \infty \quad d_{4,1}^{0} = 2 \quad d_{1,2}^{0} = 3$

$d_{4,2}^{1} = d_{4,1}^{0} + d_{1,2}^{0} = 3 + 2$

$d_{4,2}^{2} = d_{4,2}^{1}$

$d_{4,2}^{3} = d_{4,3}^{2} + d_{3,2}^{2} = -1$

$d_{ij}^{(0)} = w_{ij}$ since there cannot have any intermediate vertices with < 1 edge

Find $D^n = \left(d_{ij}^{(n)}\right)$ since the number of vertices $\leq n$

© 2017 Linda Sellie

Example

Running Time? $\Theta(n^3)$

FLOYD-WARSHALL(W, n)

let $D^{(0)} = W$

for k = 1 **to** n

let $D^{(k)} = (d^{(k)}_{ij})$ be a new n×n matrix

for i = 1 **to** n

for j = 1 **to** n

$d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})$

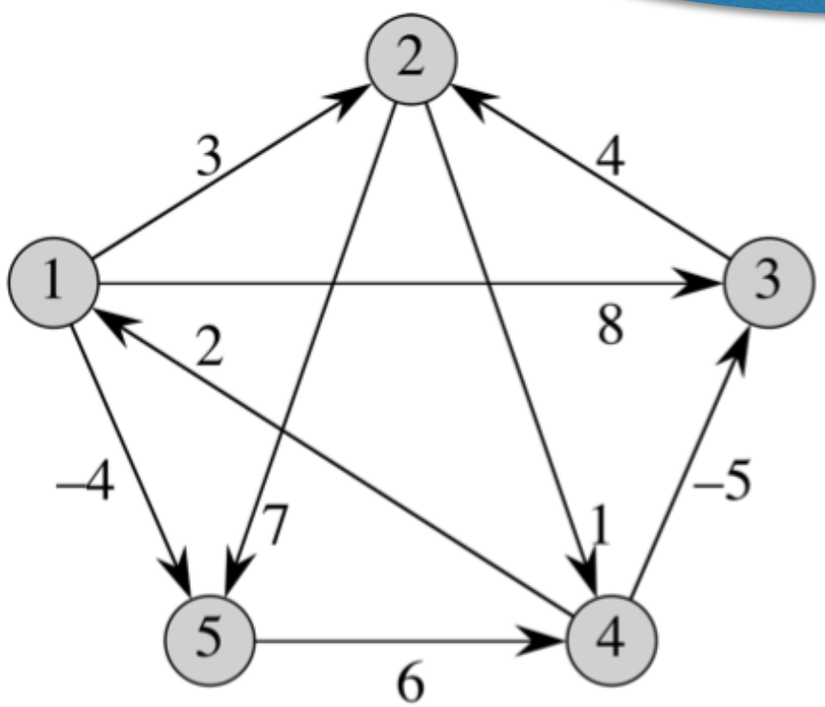
return $D^{(n)}$

Does the path improve if it goes through k?

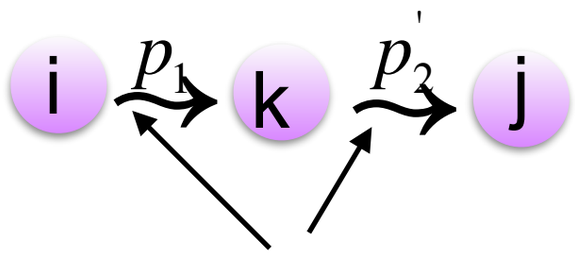
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$



$$d^{(k)}_{ij} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d^{(k-1)}_{ij},\ d^{(k-1)}_{ik} + d^{(k-1)}_{kj}) & \text{if } k \geq 0 \end{cases}$$



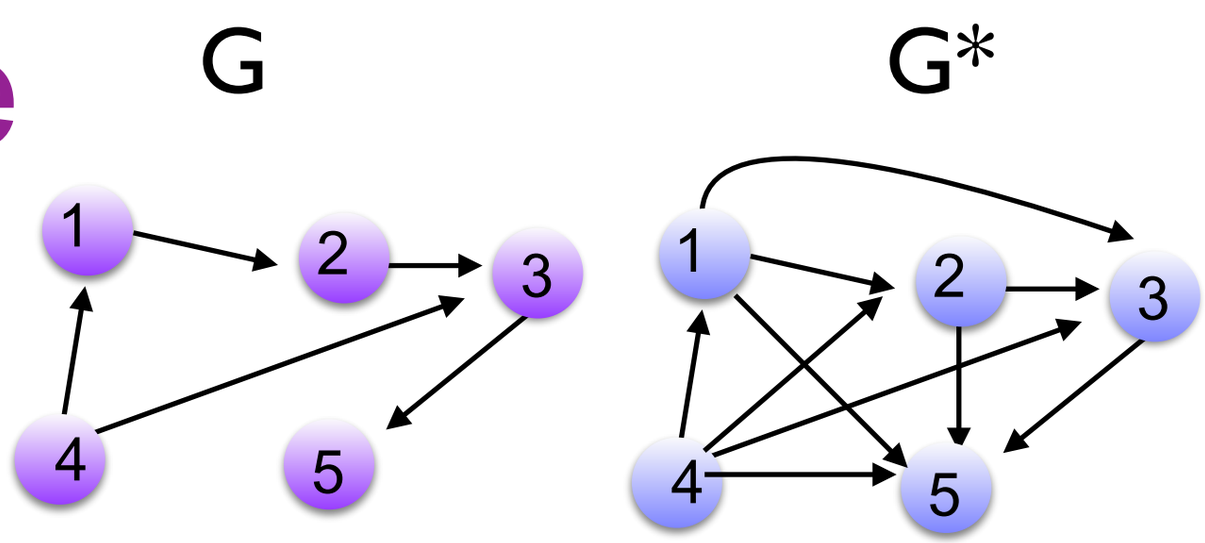
Transitive Closure

Given G = (V, E), a directed graph

Compute G* = (V, E*)

E* = {(i,j): there is a path i→j in G}

Modify FLOYD-WARSHALL to compute G*?



$$\min \rightarrow \vee \ (OR)$$
$$+ \rightarrow \wedge \ (\text{AND})$$

$$t_{ij}^{(k)} = \begin{cases} 1 & \text{if there is a path i to j with intermediate vertices in } \{1, 2, 3, ..., k\} \\ 0 & \text{otherwise} \end{cases}$$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

$$k \geq 1 \qquad t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right)$$

$T^{(0)}$

1	1	0	0	0
0	1	1	0	0
0	0	1	0	1
1	0	1	1	0
0	0	0	0	1

© 2017 Linda Sellie

```
TRANSITIVE-CLOSURE(G,n)
  n = |G.V|
  let T(0) = (tij(0)) be a new n×n matrix
  for i = 1 to n
    for j = 1 to n
      if i ==j or (i,j)∈G.E
        tij(0) = 1
      else tij(0) = 0
  for k = 1 to n
    let T(k) = (tij(k)) be a new n×n matrix
    for i = 1 to n
      for j = 1 to n
        tij(k) = tij(k-1) ∨(tik(k-1)∧ (tkj(k-1)))
return T(n)
```

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

$T^{(0)}$

1	1	0	0
0	1	1	0
0	0	1	0
1	0	0	1

$T^{(1)}$

1	1	0	0
0	1	1	0
0	0	1	0
1	1	0	1

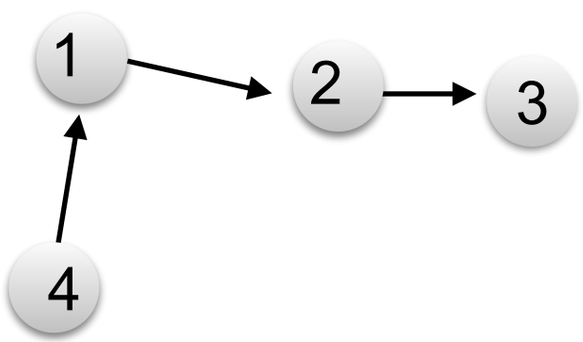
$T^{(2)}$

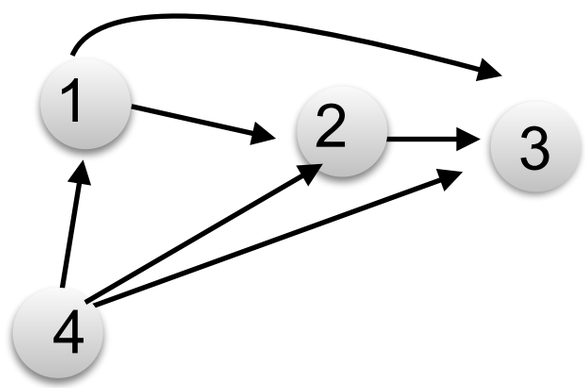
1	1	1	0
0	1	1	0
0	0	1	0
1	1	1	1

$T^{(3)}$ and $T^{(4)}$

$\min \rightarrow \vee$ (*OR*)

$+ \rightarrow \wedge$ (AND)





Running Time?

$\Theta(n^3)$

Greedy Algorithms

Similar to Dynamic Programming
Used to find the optimal solution!

https://upload.wikimedia.org/wikipedia/commons/a/a5/Gold_Bars.jpg

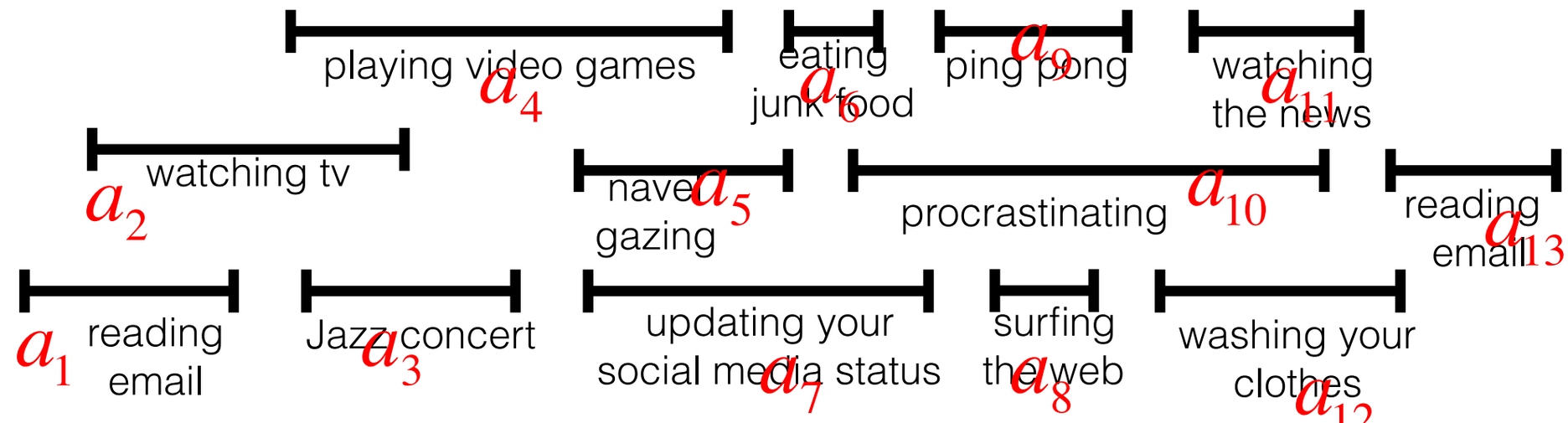
Choose the best at the current moment - no long term planning…

Sometimes this actually works!

Example:

Activity selection

Activity Selection



Which activities will you choose?

$$S = \{a_1, a_2, \ldots, a_n\}$$

a_i, uses resources during $[s_i, f_i)$

You would not like to miss doing any of these …

Unfortunately each activity cannot be combined with another activity or started late or started early.

What is the maximum number you can choose? Each activity is equally attractive to you.

Can you come up with an algorithm to find the largest number of non-overlapping activities?

We could solve this with dynamic programing…

Does it have optimal substructure?

- How could we find a subproblem which solved optimally could be used to find the optimal solution?

- Suppose we know our optimal solution includes actively a_k. Then we can schedule activities before a_k starts and after a_k finishes

- Let S_{0k} be all the activities that end before a_k starts

- Let $S_{k(n+1)}$ be all the activities that start after a_k ends

- Let A_{0k} be an optimal set of activities from the set S_{0k}. (And similarly let $A_{k(n+1)}$ be an optimal set of activities from the set $S_{k(n+1)}$)

- The optimal number of activities is then $|A_{0k}| + |A_{k(n+1)}| + 1$

We have to prove this works

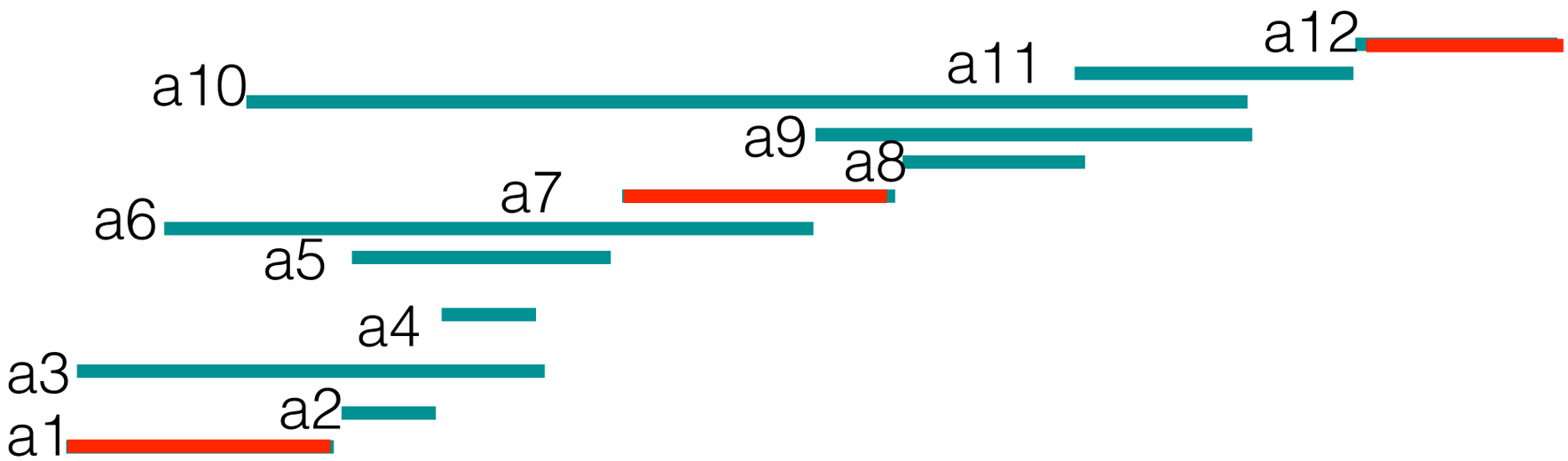
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
s_i		1	4	1	5	4	2	7	10	9	3	12	15	17
f_i	0	4	5	6	6	7	9	10	12	14	14	15	17	

$S_{17} = \{a_2, a_4, a_5\}$

$A_{17} = \{a_2, a_4\}$

$S_{7\,12} = \{a_8, a_{11}\}$

$A_{7\,12} = \{a_8, a_{11}\}$



Notation:

$S = \{a_1, a_2, \ldots, a_n\}$ a_i, uses resources during $[s_i, f_i)$

$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$

= all activities that start *after* a_i finishes and before a_j starts

A_{ij} = maximum-size set of mutually compatible activities in S_{ij}

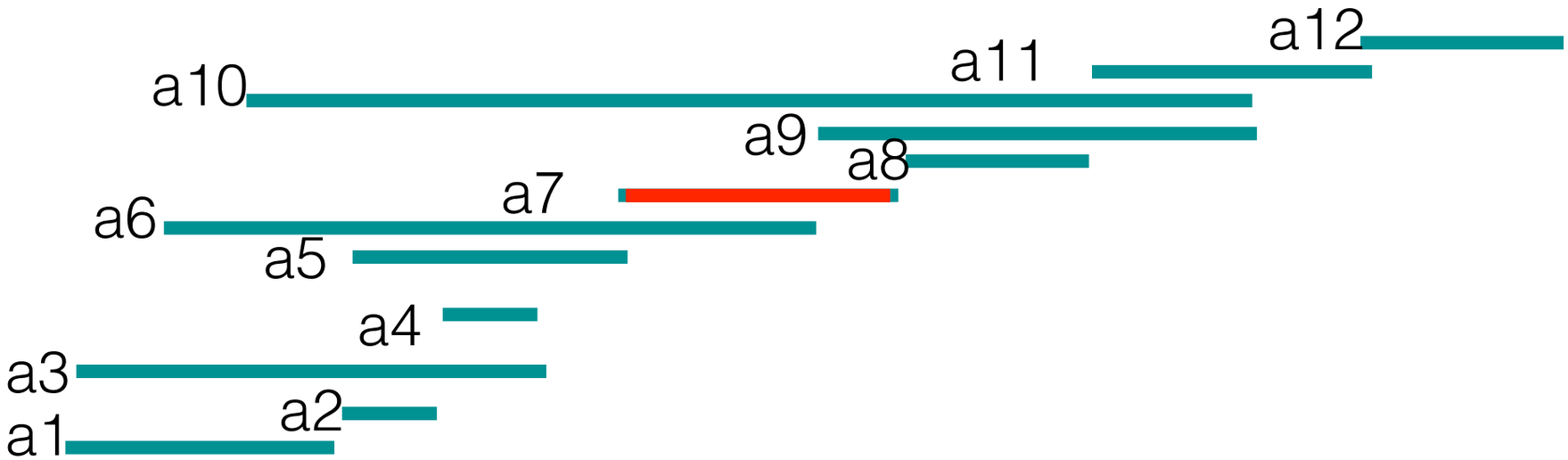
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
s_i		1	4	1	5	4	2	7	10	9	3	12	15	17
f_i	0	4	5	6	6	7	9	10	12	14	14	15	17	

$A_{0\ 13} = \{a_1, a_2, a_4, a_7, a_8, a_{11}, a_{12}\}$

If $a_7 \in A_{0\ 12}$ find $A_{0\ 7}$ and $A_{7\ 13}$

$S_{0\ 7} = \{a_1, a_2, a_3, a_4, a_5\}$ $A_{0\ 13} \cap S_{0\ 7} = A_{0\ 7} = \{a_1, a_2, a_4\}$

$S_{7\ 13} = \{a_8, a_{11}, a_{12}\}$ $A_{0\ 13} \cap S_{7\ 13} = A_{7\ 13} = \{a_8, a_{11}, a_{12}\}$



Note that:

if $a_k \in A_{ij}$, we have two subproblems:

$A_{ik} = A_{ij} \cap S_{ik}$ = activites in A_{ij} that finish before a_k starts

$A_{kj} = A_{ij} \cap S_{kj}$ = activites in A_{ij} that finish before a_k finishes

So, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ ☞ $|A_{ij}| = |A_{ik}| \cup |\{a_k\}| \cup |A_{kj}|$

Optimal Substructure

Claim: For an optimal solution that includes a_k

the optimal solution A_{ij} must include optimal subproblems for S_{ik}, S_{kj}

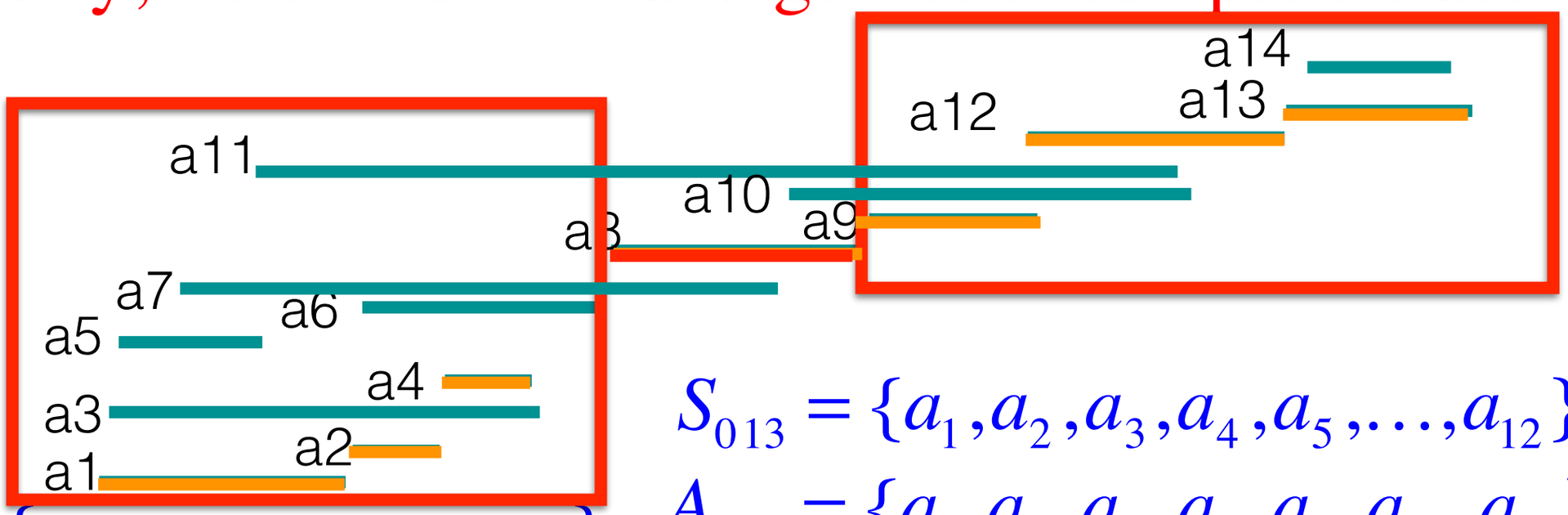
Proof: Cut and paste

Suppose not, suppose $Z_{ik} = A_{ij} \cap S_{ik}$ is not optimal in S_{ik}

Suppose there exists an A'_{ik} such that $|A'_{ik}| > |Z_{ik}|$

Then $|A'_{ik}| + |A_{kj}| + 1 > |Z_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ ⋈

Similarly, we cannot find a larger set of compatible activities in S_{kj}



$S_{013} = \{a_1, a_2, a_3, a_4, a_5, \ldots, a_{12}\}$

$A_{013} = \{a_1, a_2, a_4, a_8, a_9, a_{12}, a_{13}\}$

$S_{08} = \{a_1, a_2, a_3, a_4, a_5, a_6\}$

$|A_{08}| > |Z_{08}| = |\{a_1, a_2, a_4\}|$

Recursive Solution:

c[i,j] = size of optimal solution

$$\text{if } a_k \in A_{ij}$$

c[i,j] = $\left|A_{ij}\right| = \left|A_{ik}\right| \cup \left|\{a_k\}\right| \cup \left|A_{kj}\right|$

c[i,j] =c[i,k] + 1 + c[k,j]

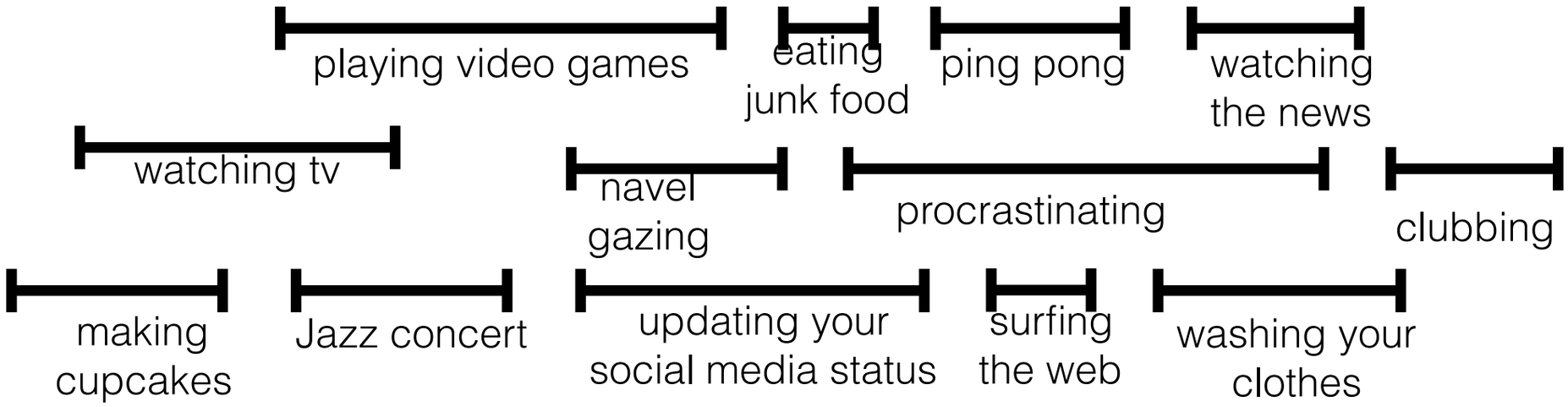
$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1 & \text{if } S_{ij} \neq \varnothing \end{cases}$$

… exponential…

polynomial with dynamic programming i.e. memoization/bottom up

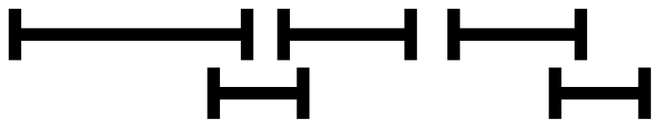
or… try a greedy approach!

What is a greedy approach?



Choose earliest starting time (of feasible options)

Shortest duration?



Earliest finishing time? Algorithm?

Sort activities based on finishing time (earliest finishing time till latest finishing time)

Keep choosing activities bases on finishing time which is feasible

i	1	2	3	4	5	6	7	8	9	10	11	12
$\mathbf{s_i}$	1	4	1	5	4	2	7	10	9	3	12	10
$\mathbf{f_i}$	4	5	6	6	7	9	10	12	14	14	15	15

REC-ACTIVITY-SELECTOR(s, f, k, n)

m = k + 1

while m ≤ n and s[m] < f[k] **//** find the first activity in S_k to finish

m = m + 1

if m ≤ n

return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)

else return ∅

Initial call

REC-ACTIVITY-SELECTOR(s, f, 0, n)

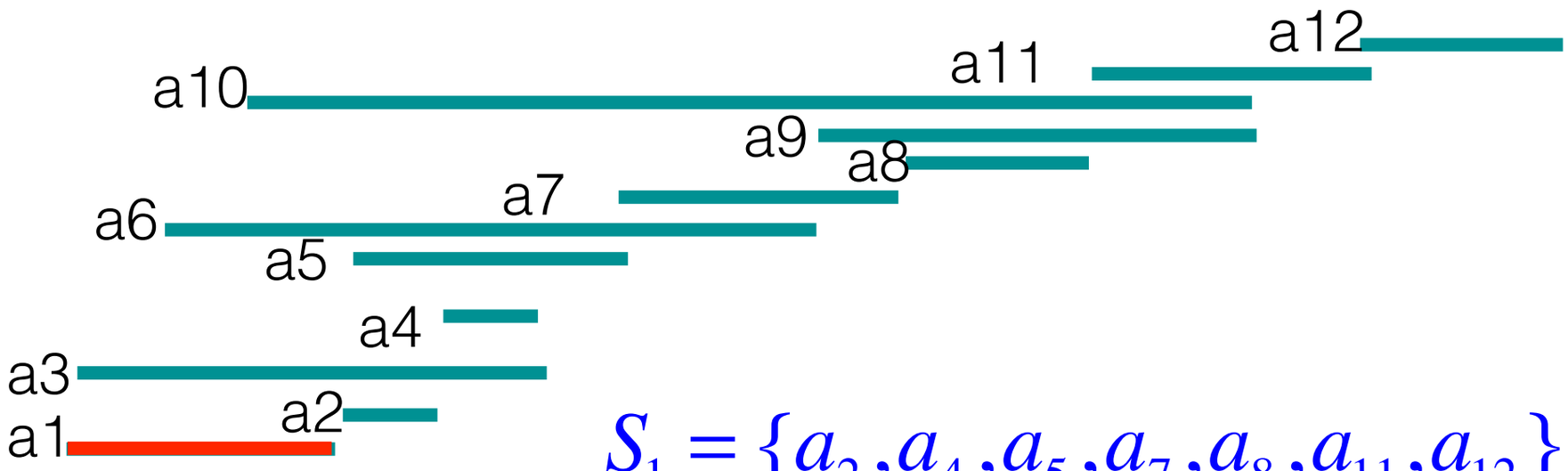
Running Time? $O(n)$

$O(n) + O(n \log n)$

Is the algorithm correct?

Greedy Choice Property:

We create a globally optimal solution by making locally optimal (greedy) choices



$$S_1 = \{a_2, a_4, a_5, a_7, a_8, a_{11}, a_{12}\}$$

Notation

Let $S_k = \{a_i \in S \mid s_i \geq f_k\}$ = all activities that start after a_k finishes

Notation

Let $S_k = \{a_i \in S \mid s_i \geq f_k\}$ = all activities that start after a_k finishes

Making the greedy choice:

Choosing a_1 reduce the problem to S_1 - only one subproblem to solve

Correctness

Consider any nonempty subproblem S_k let $a_m \in S_k$ have the earliest finishing time in S_k. Then a_m is in some max-size subset of mutually compatible activities in S_k

proof

If S_k is nonempty and $a_m \in S_k$ has the earliest finishing time

Let A_k be an optimal solution to S_k if $a_m \in A_k$ we are done

If $a_m \notin A_k$ find the activity $a_j \in A_k$ with the earliest finishing time

Create $A'_k = A - \{a_j\} \cup \{a_m\}$

Activities in A'_k are disjoint since $f_m \leq f_j$

$|A'_k| = |A_k|$ thus A'_k is and optimal solution and it contains a_m ⋈

i	1	2	3	4	5	6	7	8	9	10	11	12
s_i	1	4	1	5	4	2	7	10	9	3	12	10
f_i	4	5	6	6	7	9	10	12	14	14	15	15

GREEDY-ACTIVITY-SELECTOR(s, f)

```
n = s.length
A = {a_1}
k =1
for m = 2 to n
   if s[m] ≥ f[k]
      A = A ∪ {a_m}
      k = m
return A
```

Running Time? $O(n)$

$O(n) + O(n \log n)$

Greedy Technique and Dynamic Programming

- Both use optimal substructure (optimal solution contains optimal solution for subproblems)

- Dynamic programming solution requires the optimal solution to the subproblems to make a choice

- Greedy solution makes a choice without knowing solution to subproblems (shortsighted!)

- Any problem that can be solved by a greedy algorithm can be solved by dynamic programming

- You must always prove your greedy algorithm works

Steps:

When might we use a greedy strategy

1. we have optimal substructure
2. we can determine recursively solution
3. we can show that if we make a greedy choice, only one problem remains
4. we can prove greedy choice property holds
5. we can devise algorithm

Streamlined steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.

2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.

3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

Dynamic Programming

1. optimal substructure
2. overlapping subproblems
3. recursively solve problem - choice depends on *knowing* to solution to all subproblems
4. use memomization or bottom-up

Greedy algorithm requirements

1. optimal substructure
2. greedy choice property - can make the right choice *without knowing* the solution to subproblems