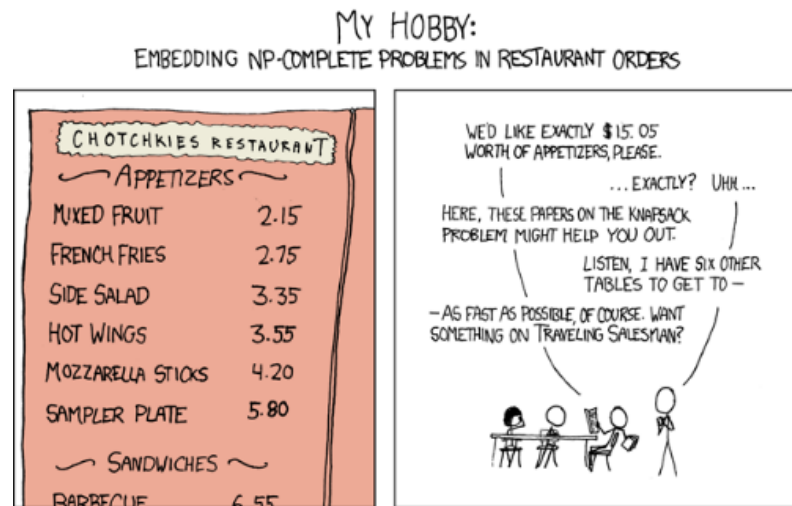


Lecture 24 Many Knapsack Solved ~~All~~ Ways Shortest Path Algorithms



EECS 281: Data Structures & Algorithms

Knapsack Problem Defined

- A thief robbing a safe finds it filled with N items
 - Items have various sizes (or weights)
 - Items have various values
- The thief has a knapsack of capacity M
- Problem: Find maximum value the thief can pack into their knapsack that does not exceed capacity M

Example: Knapsack

- Assume a knapsack with capacity $M = 11$
- There are $N = 5$ different items

	0	1	2	3	4
Size	1	2	5	6	7
Value	1	6	18	22	28

- Return *maxVal*, the maximum value the thief can carry in the knapsack

Variations on a Theme

- Each item is unique (the Standard)
 - Known as the 0-1 Knapsack Problem
 - Must take an item (1) or leave it behind (0)
- Finite amount of each item (explicit list)
- Infinite number of copies of each item
- Fractional amount of item can be taken
- Using weight (w_i) instead of size

Solve Knapsack Problem

Using multiple algorithmic approaches

- Brute-Force
- Greedy
- Divide and Conquer
- Dynamic Programming
- Backtracking
- Branch and Bound

Knapsack: Brute-Force

- Generate possible solution space
 - Given an initial set of N items
 - Consider all possible subsets
- Filter feasible solution set
 - Discard subsets with $setSize > M$
- Determine optimal solution
 - Find $maxVal$ in feasible solution set

Brute-Force Knapsack:

	0	1	2	3	4
Size	1	2	5	6	7
Value	1	6	18	22	28

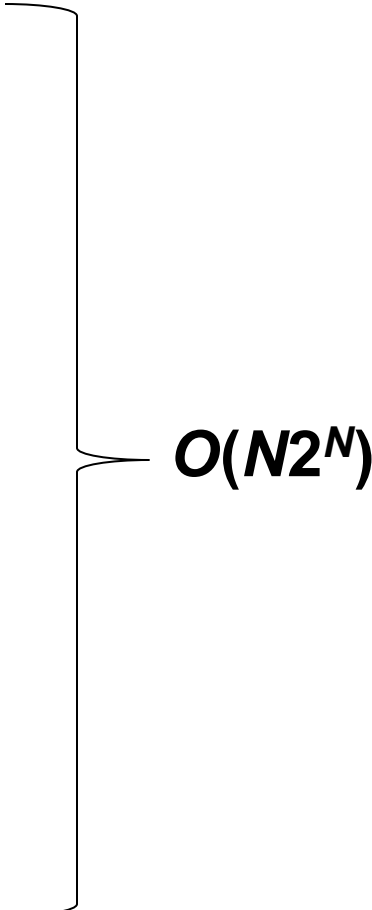
0	1	2	3	4	Val
0	0	0	0	0	0
1	0	0	0	0	1
0	1	0	0	0	6
1	1	0	0	0	7
0	0	1	0	0	18
1	0	1	0	0	19
0	1	1	0	0	24
1	1	1	0	0	25
0	0	0	1	0	22
1	0	0	1	0	23
0	1	0	1	0	28
1	1	0	1	0	29
0	0	1	1	0	40
1	0	1	1	0	41
0	1	1	1	0	46
1	1	1	1	0	47

0	1	2	3	4	Val
0	0	0	0	1	28
1	0	0	0	1	29
0	1	0	0	1	34
1	1	0	0	1	35
0	0	1	0	1	46
1	0	1	0	1	47
0	1	1	0	1	52
1	1	1	0	1	53
0	0	0	1	1	50
1	0	0	1	1	51
0	1	0	1	1	56
1	1	0	1	1	57
0	0	1	1	1	68
1	0	1	1	1	69
0	1	1	1	1	74
1	1	1	1	1	75

Brute-Force Pseudocode

```
1  bool array possSet[1..N] (0:leave,1:take)
2  maxVal = 0
3  for i = 1 to 2N
4      possSet[] = genNextPower(N)
5      setSize = findSize(possSet[])
6      setValue = findValue(possSet[])
7      if setSize <= M and setValue > maxVal
8          bestSet[] = possSet[]
9          maxVal = setValue
10 return maxVal
```


Brute-Force Efficiency

- Generate possible solution space
 - Given an initial set of N items
 - Consider all possible subsets
 - $O(2^N)$
 - Filter feasible solution set
 - Discard subsets with $setSize > M$
 - $O(N)$
 - Determine optimal solution
 - Find $maxVal$ in feasible solution set
 - $O(N)$
- 
- $O(N2^N)$

Greedy Approach

Approaches

- Steal ***highest value*** items first
 - Might not work if large items have high value
- Steal ***lowest size (weight)*** items first
 - Might not work if small items have low value
- Steal **highest value density** items first
 - Might not work if large items have high value density

If greedy is not optimal, why not?

Example: Greedy Knapsack

- Assume a knapsack with capacity $M = 11$
- There are N different items, where
 - Items have various sizes
 - Items have various values

	0	1	2	3	4
Size	1	2	5	6	7
Value	1	6	18	22	28
Ratio	1	3	3.6	3.67	4

Greedy Pseudocode

Input: integers capacity M , $\text{size}[1..N]$, $\text{val}[1..N]$

Output: integer max value size M knapsack can carry

```
1  maxVal = 0, currSize = 0
2  ratio[] = buildRatio(value[], size[])
3  // Sort all three arrays by ratio
4  sortRatio(ratio[], value[], size[])
5  for i = 1 to N //sorted by ratio
6      if size[i] + currSize <= M
7          currSize = currSize + size[i]
8          maxVal = maxVal + value[i]
9
10 return maxVal
```

Greedy Efficiency

- Sort items by ratio of value to size
 - $O(N \log N)$
- Choose item with highest ratio first
 - $O(N)$

$$O(N \log N) + O(N) \Rightarrow O(N \log N)$$

Fractional Knapsack: Greedy

- Suppose that thief can steal a portion of an item
- What happens if we apply a Greedy strategy?
- Is it optimal?

Dynamic Programming

- Enumeration with DP
 - Examples: Fibonacci, Binomial Coefficient, Knight Moves
 - These are constraint satisfaction problems
- Optimization with DP
 - Examples: **Knapsack**, Longest Increasing Subsequence, Weighted Independent Subset
- Both can be solved top-down or bottom-up, optimizations often favor bottom-up

DP Knapsack Approach

- A master thief prepares job for apprentice
 - Alarm code, safe combination, and knapsack
 - List of items in safe
 - Table of items to put in knapsacks, $0 < x \leq M$
- Apprentice finds one extra item in safe
 - Take it or leave it?
 - Remove just enough to fit the new item
 - Q: What should be removed?
 - Q: When should the new item be included?

DP Knapsack Generalization

- Each item will either be taken or left behind
 - If it is too large it is left behind
 - If room can be made to include it, it will be taken only when it improves the haul
- Bottom-Up uses two nested loops
 - Look at items one a time
 - Look at knapsacks from size 0 up to M
 - Build and use a 2D memo

Safe

	0	1	2	3	4
Size	1	2	5	6	7
Value	1	6	18	22	28

Knapsack Size

Items Picked

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Dynamic Programming: Knapsack

```
1  uint32_t knapsackDP(const vector<Item> &items, const size_t m) {
2      const size_t n = items.size();
3      vector<vector<uint32_t>>
4          memo(n + 1, vector<uint32_t>(m + 1, 0)); // +1, +1
5
6      for (size_t i = 0; i < n; ++i) {
7          for (size_t j = 0; j < m + 1; ++j) { // +1 for memo[][m]
8              if (j < items[i].size)
9                  memo[i + 1][j] = memo[i][j];
10             else
11                 memo[i + 1][j] = max(memo[i][j],
12                                     memo[i][j - items[i].size] + items[i].value);
13         } // for j
14     } // for i
15     return memo[n][m];
16 } // knapsackDP()
```

Run time is **$O(MN)$**

Reconstructing the Solution

- Included items improve a smaller solution, excluded items do not
- If a smaller solution plus an item is greater than or equal to a full solution without the item it is included, otherwise it is excluded
- Use completed memo to determine the items taken
- Work backward from (n, m) to $(0, 0)$

Safe

	0	1	2	3	4
Size	1	2	5	6	7
Value	1	6	18	22	28

Knapsack Size

Items Picked

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Knapsack DP Reconstruction

```
18 vector<bool> knapDPReconstruct(const vector<Item> &items,
19                               const vector<vector<uint32_t>> &memo, const size_t m) {
20     const size_t n = items.size();
21     size_t c = m; // current capacity
22     vector<bool> taken(n, false); // included items
23
24     for (int i = n - 1; i >= 0; --i) { // use int for -1 loop exit
25         if (items[i].size <= c) {
26             if (memo[i][c - items[i].size] + items[i].value >= memo[i][c]) {
27                 taken[i] = true;
28                 c -= items[i].size;
29             } // if ..item added
30         } // if ..item fits
31     } // for i..0
32     return taken;
33 } // knapDPReconstruct()
```

Run time is $O(N)$

Knapsack Branch and Bound

- The Knapsack Problem is an optimization problem
- Branch and Bound can be used to solve optimization problems
 - Knapsack is a maximization
 - Initial estimate and current best solution are a “lower bound”
 - Calculate partial solution, remainder is an “upper bound” estimate

Knapsack B&B Elements

- promising(): total weight of items $< M$
- solution(): any collection that is promising
- lower_bound: starts with highest possible underestimate, ends with maximum value taken
 - Can start w/ Greedy 0-1 Knapsack (by value density)
- upper_bound: sum of value of included items, plus an "overestimate" of value that can fit in remaining space
 - Prune if upper_bound $<$ lower_bound
 - Can use Greedy Fractional Knapsack
- Don't need permutations only combinations

Shortest Path Algorithms



EECS 281: Data Structures & Algorithms

Shortest Path Examples

- Highway system
 - Distance
 - Travel time
 - Number of stoplights
 - Krispy Kreme locations
- Network of airports
 - Travel time
 - Fares
 - Actual distance

Weighted Path Length

- Consider an edge-weighted graph $G = (V, E)$.
- Let $C(v_i, v_j)$ be the weight on the edge connecting v_i to v_j .
- A path in G is a non-empty sequence of vertices $P = \{v_1, v_2, v_3, \dots, v_k\}$.
- The weighted path length is given by

$$\sum_{i=1}^{k-1} C(v_i, v_{i+1})$$

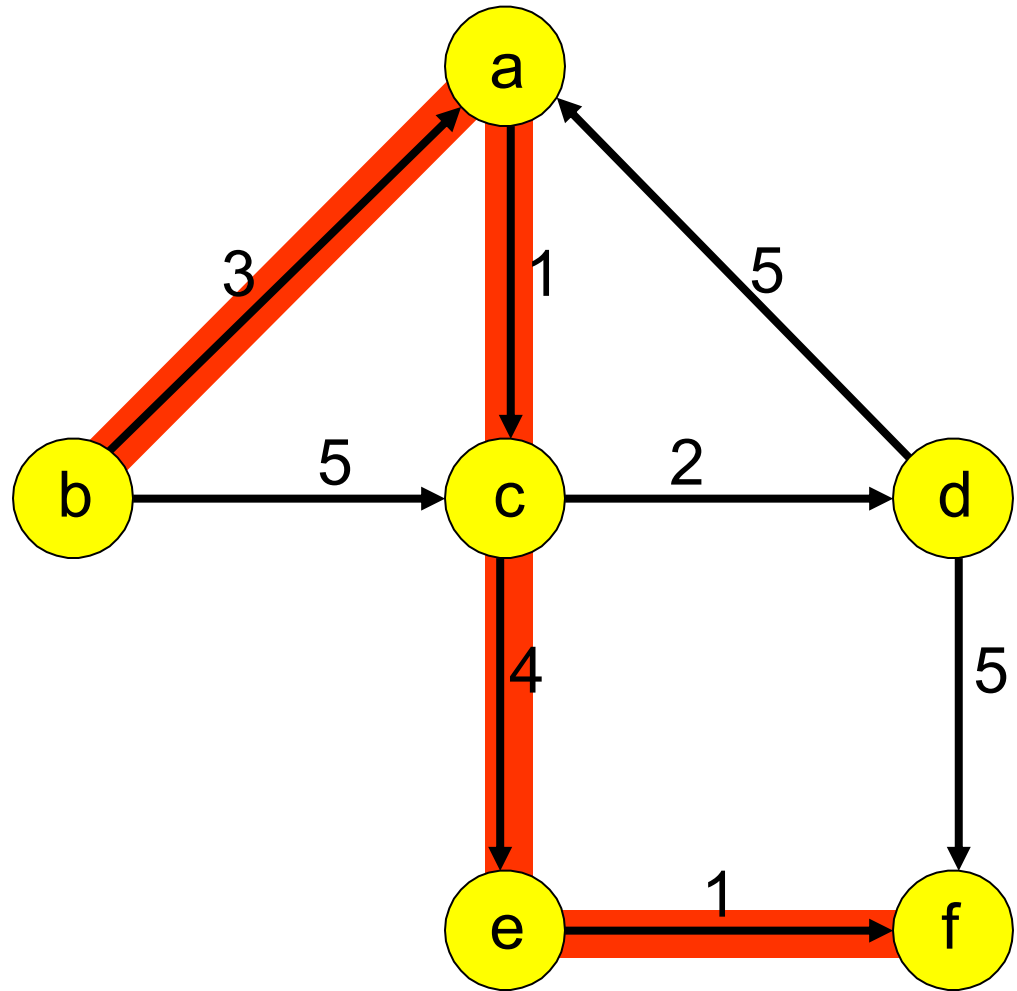
The Shortest Path Problem

Given an edge-weighted graph $G = (V, E)$ and two vertices, $v_s \in V$ and $v_d \in V$, find the path that starts at v_s and ends at v_d that has the smallest weighted path length

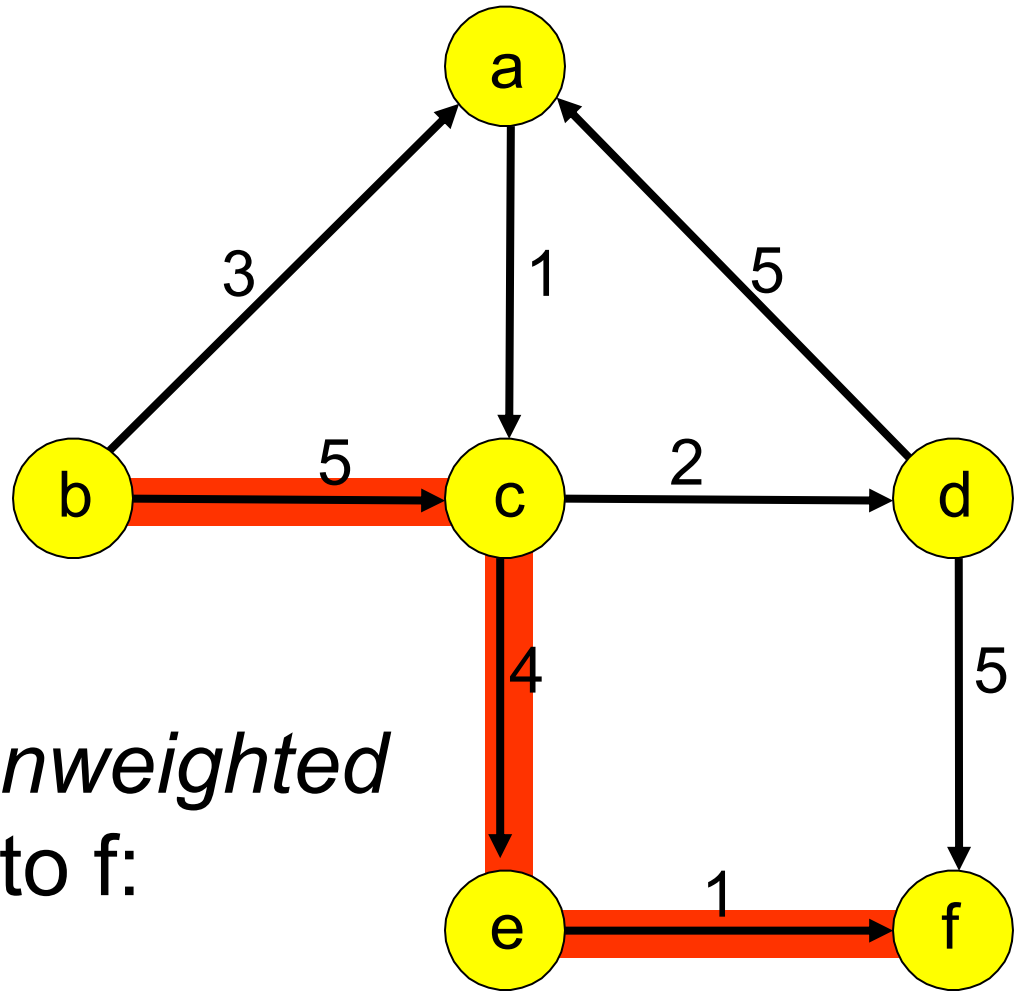
Single-Source Shortest Path

- Given an edge-weighted graph $G = (V, E)$ and a vertex, $v_s \in V$, find the shortest path from v_s to every other vertex in V
- To find the shortest path from v_s to v_d , we must find the shortest path from v_s to all other vertexes in G

The shortest weighted path
from b to f:
{b, a, c, e, f}

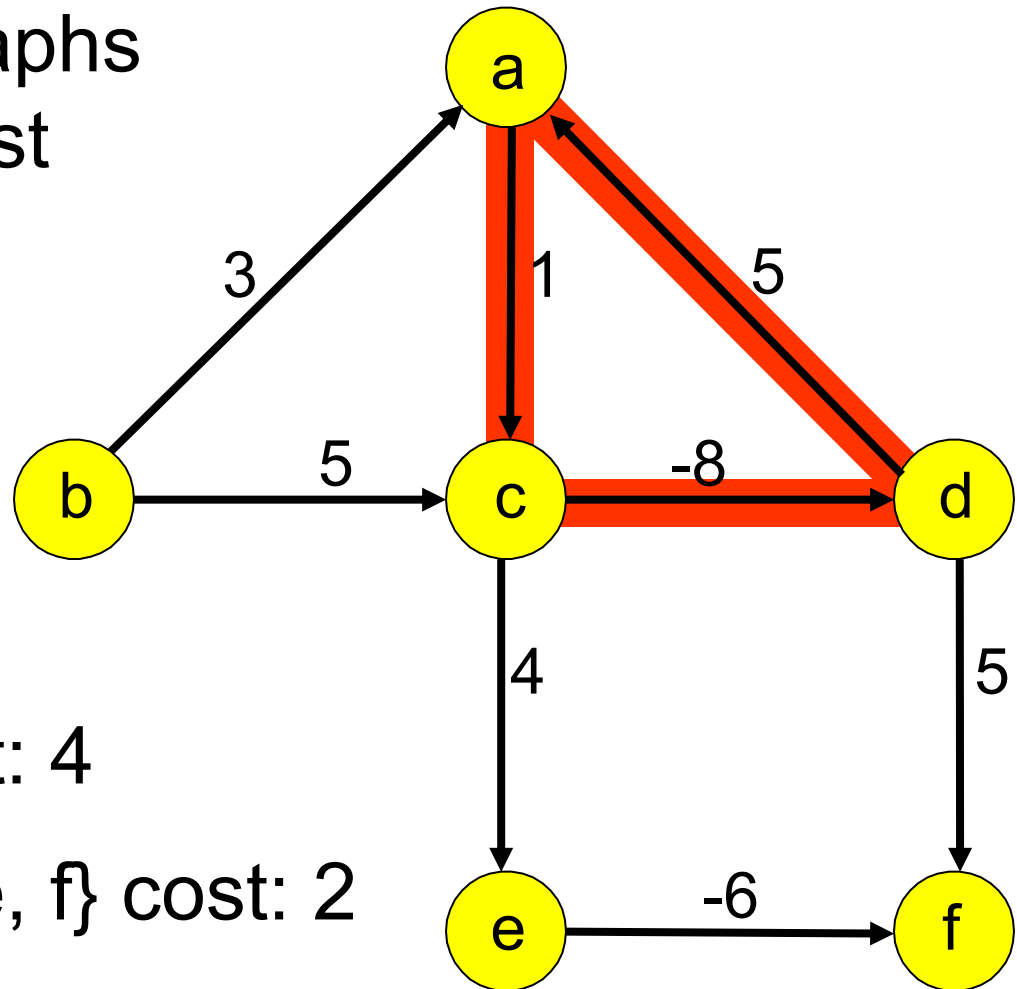


The shortest *weighted* path
from b to f:
{b, a, c, e, f}



A shortest *unweighted*
path from b to f:
{b, c, e, f}

Shortest path problem
undefined for graphs
with negative-cost
cycles



{d, a, c, e, f} cost: 4

{d, a, c, d, a, c, e, f} cost: 2

{d, a, c, d, a, c, d, a, c, e, f} cost: 0

Dijkstra's Algorithm

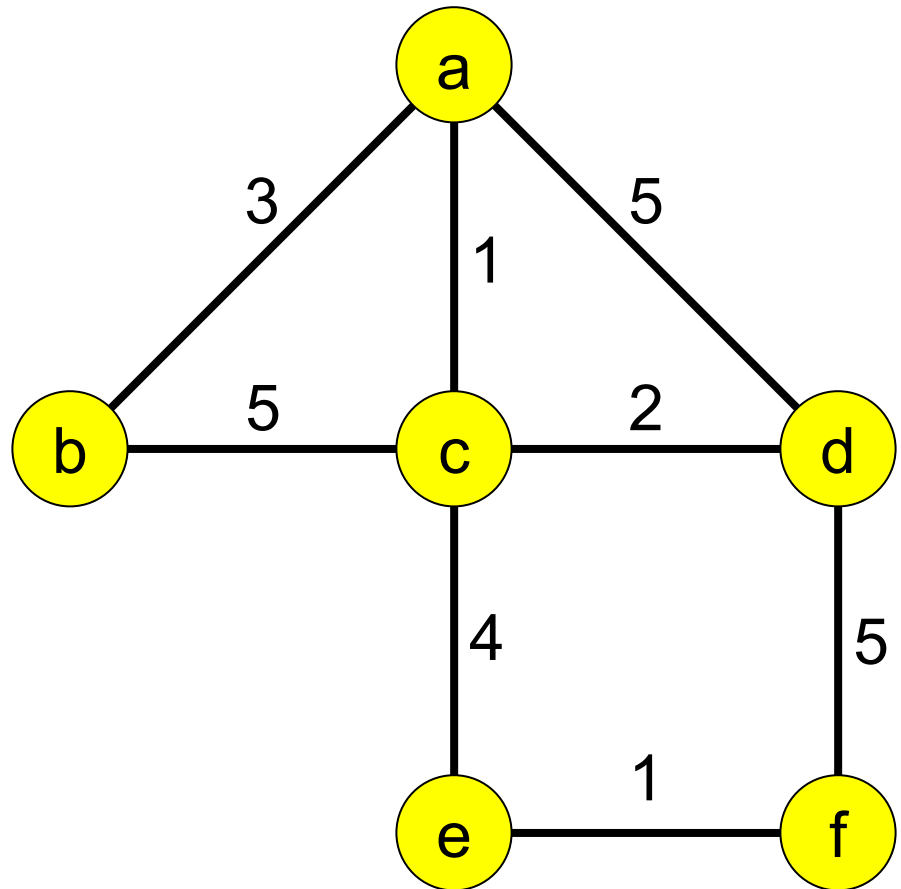
- Greedy algorithm for solving shortest path problem
- Assume non-negative weights
- Find shortest path from v_s to every other vertex

Dijkstra's Algorithm

For each vertex v , need to know:

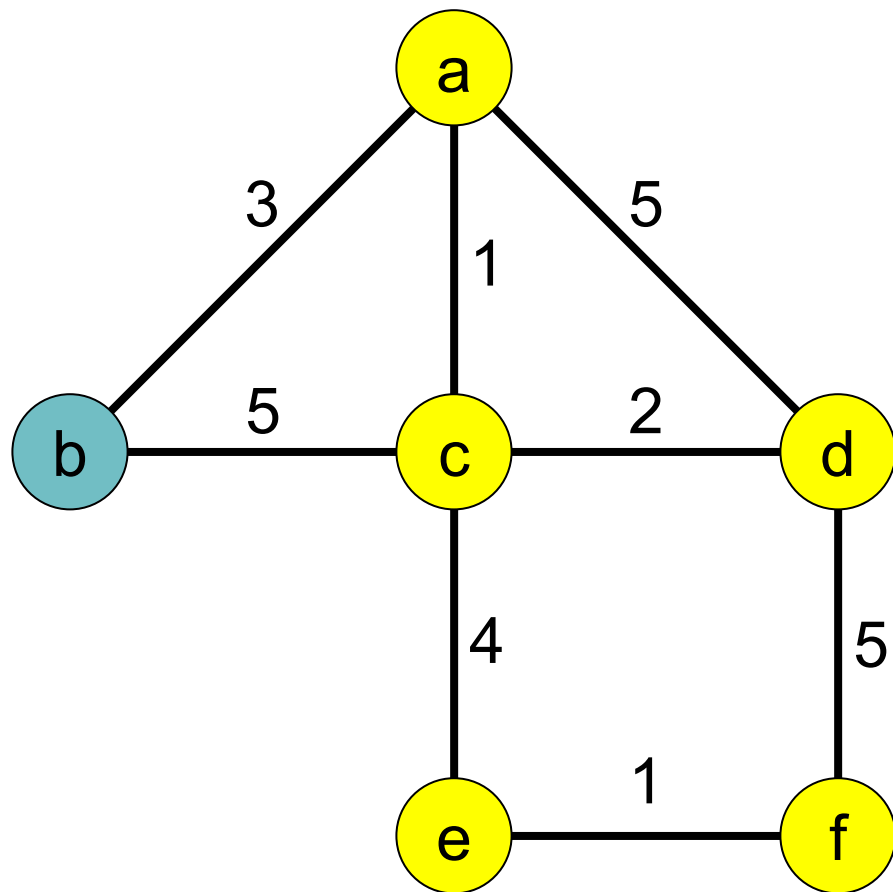
- k_v : Is the shortest path from v_s to v known? (initially false for all $v \in V$)
- d_v : What is the length of the shortest path from v_s to v ? (initially ∞ for all $v \in V$, except $v_s = 0$)
- p_v : What vertex precedes (is parent of) v on the shortest path from v_s to v ? (initially unknown for all $v \in V$)

v	k_v	d_v	p_v
a	F	∞	
b	F	0	
c	F	∞	
d	F	∞	
e	F	∞	
f	F	∞	

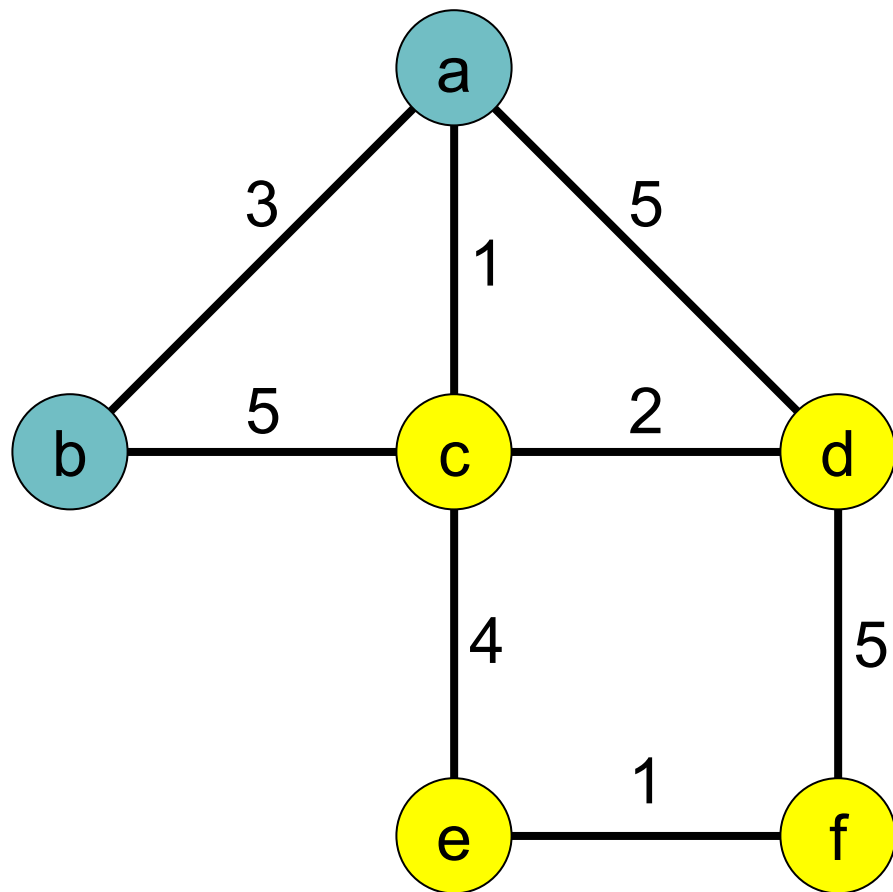


Find shortest paths to b

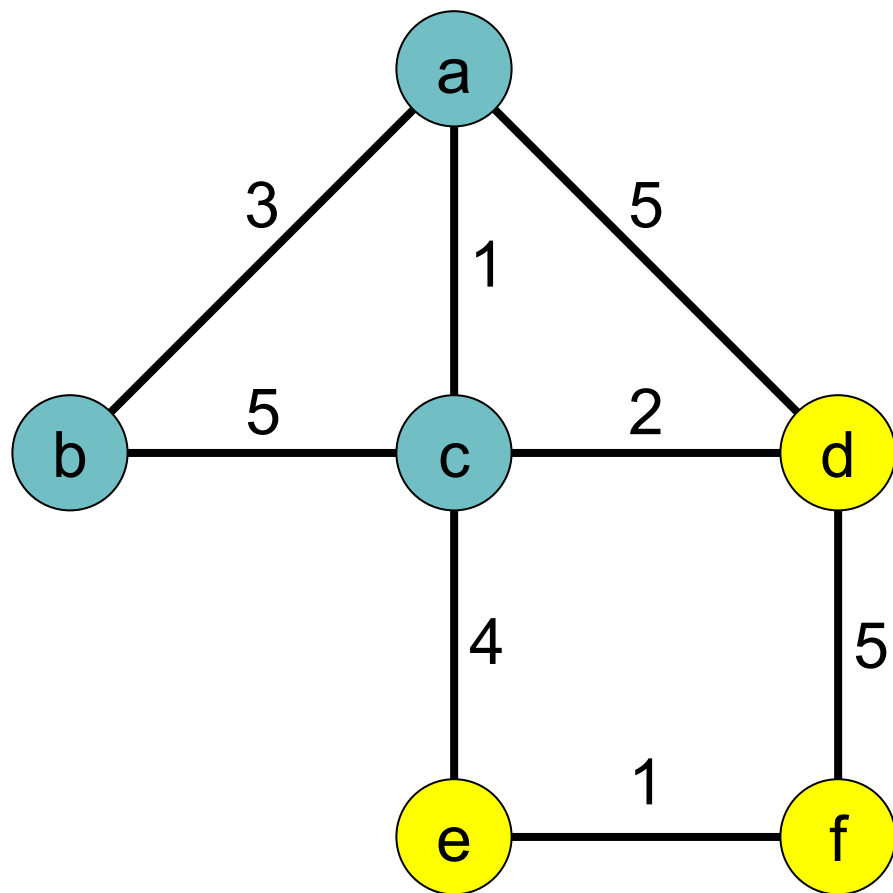
v	k_v	d_v	p_v
a	<i>F</i>	3	<i>b</i>
b	<i>T</i>	0	--
c	<i>F</i>	5	<i>b</i>
d	<i>F</i>	∞	
e	<i>F</i>	∞	
f	<i>F</i>	∞	



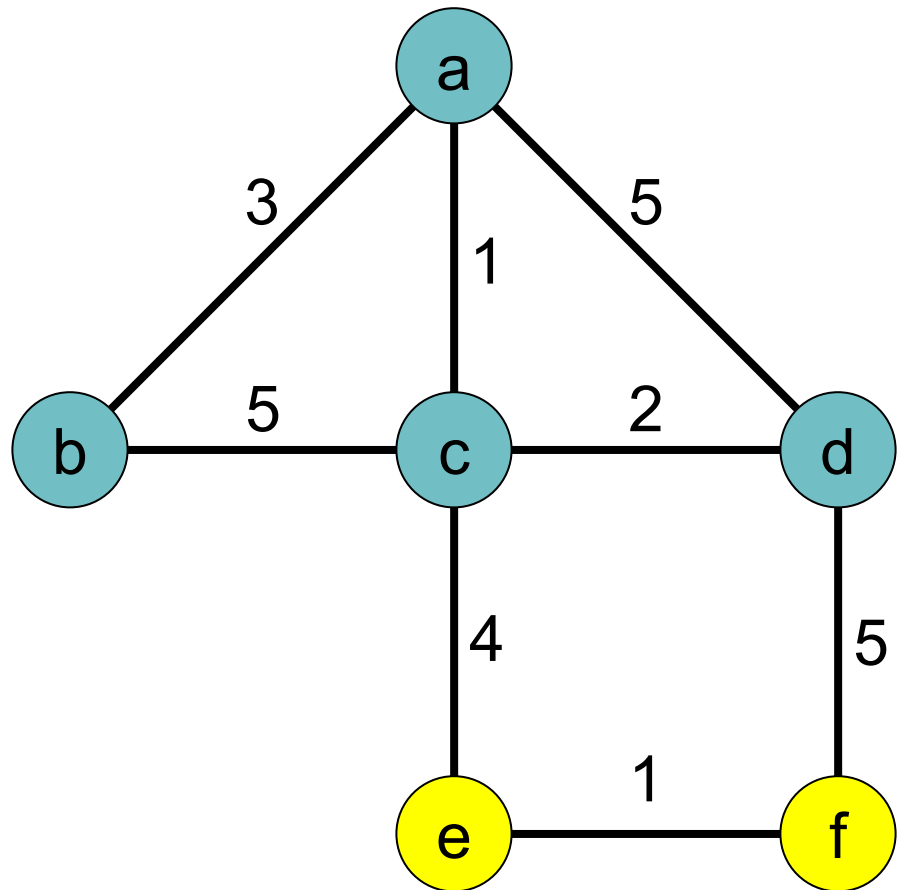
v	k_v	d_v	p_v
a	<i>T</i>	3	b
b	<i>T</i>	0	--
c	<i>F</i>	4	<i>a</i>
d	<i>F</i>	8	<i>a</i>
e	<i>F</i>	∞	
f	<i>F</i>	∞	



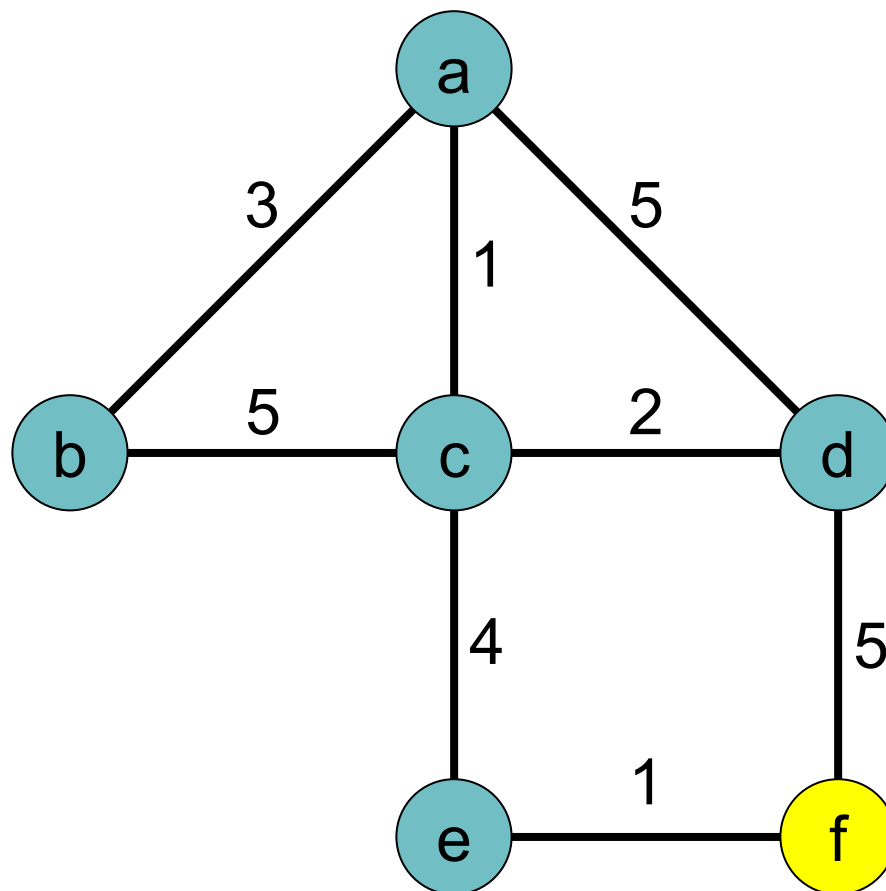
v	k_v	d_v	p_v
a	T	3	b
b	T	0	--
c	T	4	a
d	F	6	c
e	F	8	c
f	F	∞	



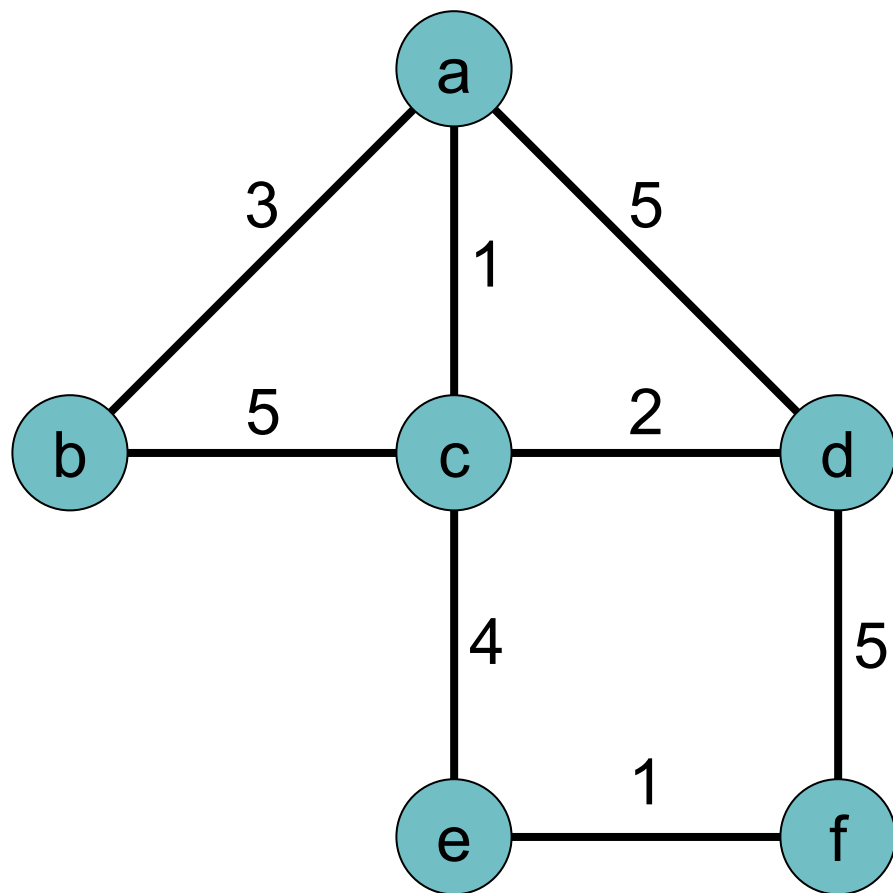
v	k_v	d_v	p_v
a	T	3	b
b	T	0	--
c	T	4	a
d	T	6	c
e	F	8	c
f	F	11	d



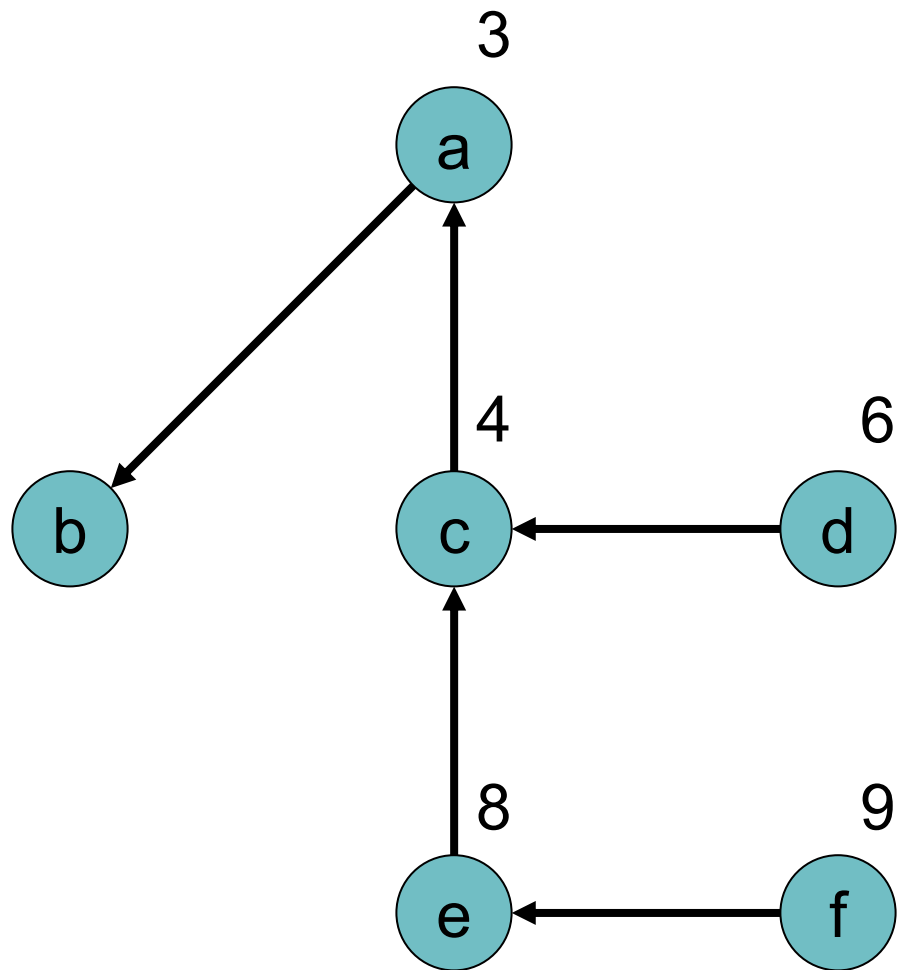
v	k_v	d_v	p_v
a	T	3	b
b	T	0	--
c	T	4	a
d	T	6	c
e	T	8	c
f	F	9	e



v	k_v	d_v	p_v
a	T	3	b
b	T	0	--
c	T	4	a
d	T	6	c
e	T	8	c
f	T	9	e



v	k_v	d_v	p_v
a	T	3	b
b	T	0	--
c	T	4	a
d	T	6	c
e	T	8	c
f	T	9	e



Dijkstra Complexity

- $O(V^2)$ for a simple nested loop implementation, a lot like Prim's
 - Intuition: for each vertex, find the min using linear search
- $O(E \log V)$ for sparse graphs, using heaps
 - E for considering every edge
 - $\log E = O(\log V^2) = O(\log V)$ for finding the shortest edge in heap

Dijkstra's Algorithm

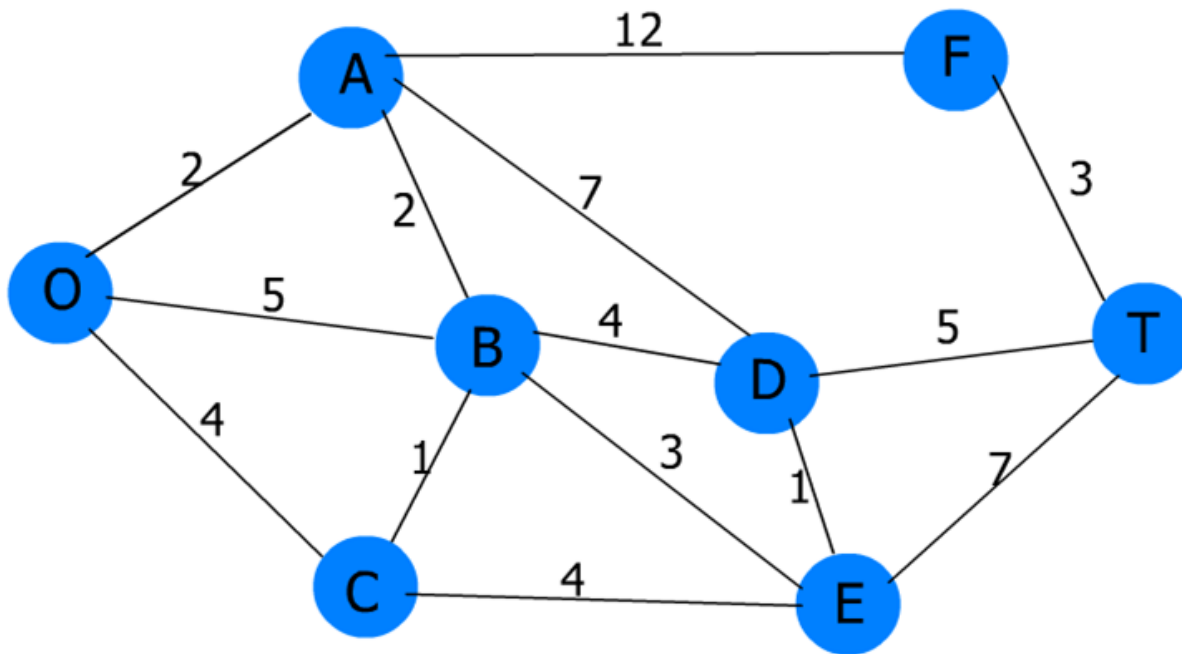
```
1  Algorithm Dijkstra(G, s0)
2    //Initialize
3    n = |V|                                O(1)
4    create_table(n) //stores k,d,p         O(V)
5    create_pq() //empty heap               O(1)
6    table[s0].d = 0                        O(1)
7    insert_pq(0, s0)                       O(1)
```

Dijkstra's Algorithm (cont.)

1	<code>while (!pq.isEmpty)</code>	$O(E)$
2	<code> v0 = getMin() //heap top() & pop()</code>	$O(\log E)$
3	<code> if (!table[v0].k) //not known</code>	$O(1)$
4	<code> table[v0].k = true</code>	$O(1)$
5	<code> for each vi ∈ Adj[v0]</code>	$O(1 + E/V)$
6	<code> d = table[v0].d + distance(vi, v0)</code>	$O(1)$
7	<code> if (d < table[vi].d)</code>	$O(1)$
8	<code> table[vi].d = d</code>	$O(1)$
9	<code> table[vi].p = v0</code>	$O(1)$
10	<code> insert_pq(d, vi)</code>	$O(\log E)$
11		
12	<code>for each v ∈ G(V,E)</code>	$O(V)$
13	<code> //build edge set in T</code>	
14	<code> (v, table[v].p) ∈ T(V, E')</code>	$O(1)$

Exercise

Find the shortest path from O to T



	k_v	d_v	p_v
O			
A			
B			
C			
D			
E			
F			
T			

All-pairs shortest path problem

- Given an edge-weighted graph $G = (V, E)$, for each pair of vertices in V find the length of the shortest weighted path between the two vertices

Solution 1: Run Dijkstra V times

Other solutions:

Use Floyd's Algorithm (dense graphs)

Use Johnson's Algorithm (sparse graphs)

NOTE!

- We usually end up stopping here for lecture material
- You don't need to know Floyd-Warshall for the exam, but it seems a shame to delete perfectly good slides
- If you ever need it for a reference, you've got it!

Solution 2: Floyd's Algorithm

- Floyd-Warshall Algorithm
- Dynamic programming method for solving all-pairs shortest path problem on a dense graph
- Uses an adjacency matrix
- $O(V^3)$ (best, worst, average)

Weighted path length

- Consider an edge-weighted graph $G = (V, E)$, where $C(v, w)$ is the weight on the edge (v, w) .
- Vertices numbered from 1 to $|V|$
(i.e. $V = \{v_1, v_2, v_3, \dots, v_{|V|}\}$)

Weighted path length

- Consider the set $V_k = \{v_1, v_2, v_3, \dots, v_k\}$ for $0 \leq k \leq |V|$
- $P_k(i,j)$ is the shortest path from i to j that passes only through vertices in V_k if such a path exists
- $D_k(i,j)$ is the length of $P_k(i,j)$

$$D_k(i,j) = \begin{cases} |P_k(i,j)| & \text{if } P_k(i,j) \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

Suppose $k = 0$

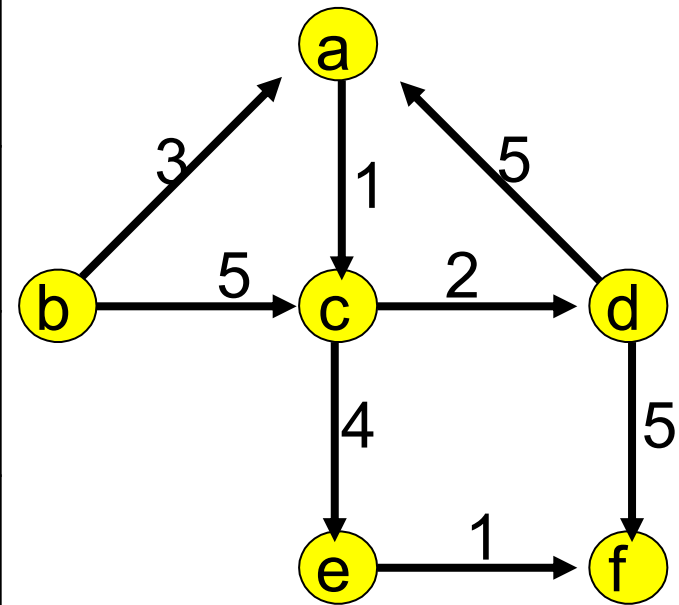
- $V_0 = \emptyset$, so P_0 paths are the edges in G :

$$P_0(i,j) = \begin{cases} \{i,j\} & \text{if } (i,j) \in E \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Therefore D_0 path lengths are:

$$D_0(i,j) = \begin{cases} |C(i,j)| & \text{if } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

		to						
		D ₀	a	b	c	d	e	f
from	a		∞	∞	1	∞	∞	∞
	b		3	∞	5	∞	∞	∞
	c		∞	∞	∞	2	4	∞
	d		5	∞	∞	∞	∞	5
	e		∞	∞	∞	∞	∞	1
	f		∞	∞	∞	∞	∞	∞



Floyd's Algorithm

- Add vertices to V_k one at a time
- For each new vertex v_k , consider whether it improves each possible path
 - Compute $D_k(i,j)$ for each i,j in V
 - Minimum of:
 - $D_{k-1}(i,j)$
 - $D_{k-1}(i,k) + D_{k-1}(k,j)$

$$D_k(i,j) = \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$$

to $V_0 = \{\}$

$V_1 = \{a\}$

from

D_0	a	b	c	d	e	f
a	∞	∞	1	∞	∞	∞
b	3	∞	5	∞	∞	∞
c	∞	∞	∞	2	4	∞
d	5	∞	∞	∞	∞	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

D_1	a	b	c	d	e	f
a	∞	∞	1	∞	∞	∞
b	3	∞	4	∞	∞	∞
c	∞	∞	∞	2	4	∞
d	5	∞	6	∞	∞	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

$$D_k(i,j) = \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$$

to $V_1 = \{a\}$

$V_2 = \{a, b\}$

from

D_1	a	b	c	d	e	f
a	∞	∞	1	∞	∞	∞
b	3	∞	4	∞	∞	∞
c	∞	∞	∞	2	4	∞
d	5	∞	6	∞	∞	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

D_2	a	b	c	d	e	f
a	∞	∞	1	∞	∞	∞
b	3	∞	4	∞	∞	∞
c	∞	∞	∞	2	4	∞
d	5	∞	6	∞	∞	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

$$D_k(i,j) = \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$$

to $V_2 = \{a, b\}$

$V_3 = \{a, b, c\}$

from

D_2	a	b	c	d	e	f
a	∞	∞	1	∞	∞	∞
b	3	∞	4	∞	∞	∞
c	∞	∞	∞	2	4	∞
d	5	∞	6	∞	∞	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

D_3	a	b	c	d	e	f
a	∞	∞	1	3	5	∞
b	3	∞	4	6	8	∞
c	∞	∞	∞	2	4	∞
d	5	∞	6	∞	10	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

$$D_k(i,j) = \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$$

to $V_3 = \{a, b, c\}$

$V_4 = \{a, b, c, d\}$

from

D ₃	a	b	c	d	e	f
a	∞	∞	1	3	5	∞
b	3	∞	4	6	8	∞
c	∞	∞	∞	2	4	∞
d	5	∞	6	∞	10	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

D ₄	a	b	c	d	e	f
a	∞	∞	1	3	5	8
b	3	∞	4	6	8	11
c	7	∞	∞	2	4	7
d	5	∞	6	∞	10	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

$$D_k(i,j) = \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$$

to $V_4 = \{a, b, c, d\}$

$V_5 = \{a, b, c, d, e\}$

from

D_4	a	b	c	d	e	f
a	∞	∞	1	3	5	8
b	3	∞	4	6	8	11
c	7	∞	∞	2	4	7
d	5	∞	6	∞	10	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

D_5	a	b	c	d	e	f
a	∞	∞	1	3	5	6
b	3	∞	4	6	8	9
c	7	∞	∞	2	4	5
d	5	∞	6	∞	10	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

$$D_k(i,j) = \min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$$

to $V_5 = \{a, b, c, d, e\}$ $V_6 = \{a, b, c, d, e, f\}$

from

D_5	a	b	c	d	e	f
a	∞	∞	1	3	5	8
b	3	∞	4	6	8	11
c	7	∞	∞	2	4	7
d	5	∞	6	∞	10	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

D_6	a	b	c	d	e	f
a	∞	∞	1	3	5	6
b	3	∞	4	6	8	9
c	7	∞	∞	2	4	5
d	5	∞	6	∞	10	5
e	∞	∞	∞	∞	∞	1
f	∞	∞	∞	∞	∞	∞

Floyd's Algorithm

```
1  Floyd(G) {  
2    // Initialize  
3    n = |V|;  
4    for (k = 0; k <= n; k++)  
5      for (i = 0; i < n; i++)  
6        for (j = 0; j < n; j++)  
7          d[k][i][j] = infinity;  
8  
9    for (all (v,w) ∈ E)  
10     d[0][v][w] = C(v,w)
```

$O()$

$O()$

$O()$

Floyd's Algorithm

```
11 // Compute next distance matrix
12 for (k = 1; k <= n; k++)
13     for (i = 0; i < n; i++)
14         for (j = 0; j < n; j++)
15             d[k][i][j] = min(d[k-1][i][j],                O( )
16                               d[k-1][i][k] + d[k-1][k][j]);
17 }
```

What About the Paths?

- Can't simply reconstruct them at end
- Add initialization:

```
1   for (i = 0; i < n; i++)
2       for (j = 0; j < n; j++)
3           // If edge doesn't exist, no path
4           if (C(i,j) == infinity)
5               p[0][i][j] = NIL;
6       else
7           p[0][i][j] = j;
```

Updating Paths

- When going through the triple-nested loop of the algorithm, if you ever update a weight, you must also update the path
- See code next page

to

$$V_I = \{a\}$$

from

P_0	a	b	c	d	e	f
a	-	-	c	-	-	-
b	a	-	c	-	-	-
c	-	-	-	d	e	-
d	a	-	-	-	-	f
e	-	-	-	-	-	f
f	-	-	-	-	-	-

P_1	a	b	c	d	e	f
a	-	-	c	-	-	-
b	a	-	a	-	-	-
c	-	-	-	d	e	-
d	a	-	a	-	-	f
e	-	-	-	-	-	f
f	-	-	-	-	-	-

Paths: Primary Loops

```
1   for (k = 1; k < n; k++)
2       for (i = 0; i < n; i++)
3           for (j = 0; j < n; j++)
4               // Compute next distance matrix
5               d[k][i][j] = min(d[k-1][i][j],                O( )
6                               d[k-1][i][k] + d[k-1][k][j]);
7               // Compute next paths matrix
8               if (d[k-1][i][j]
9                   <= d[k-1][i][k] + d[k-1][k][j])
10                  p[k][i][j] = p[k-1][i][j];
11               else                                         O( )
12                  p[k][i][j] = p[k-1][k][j];
```

Worst Case Running Time

- Add vertices to V_k one at a time
 - Outer loop executes $|V|$ times
- For each new vertex, consider whether it improves each possible path
 - Inner loops execute $|V|^2$ times
- Overall $O(|V|^3)$
- Better than running Dijkstra $|V|$ times?

Worst Case Running Time

- Add vertices to V_k one at a time
 - Outer loop executes $|V|$ times
- For each new vertex, consider whether it improves each possible path
 - Inner loops execute $|V|^2$ times
- Overall $O(|V|^3)$
- Better than running Dijkstra $|V|$ times?

Worst Case Running Time

- Add vertices to V_k one at a time
 - Outer loop executes $|V|$ times
- For each new vertex, consider whether it improves each possible path
 - Inner loops execute $|V|^2$ times
- Overall $O(|V|^3)$
- Better than running Dijkstra $|V|$ times?

Worst Case Running Time

- Add vertices to V_k one at a time
 - Outer loop executes $|V|$ times
- For each new vertex, consider whether it improves each possible path
 - Inner loops execute $|V|^2$ times
- Overall $O(|V|^3)$
- Better than running Dijkstra $|V|$ times?

Worst Case Running Time

- Add vertices to V_k one at a time
 - Outer loop executes $|V|$ times
- For each new vertex, consider whether it improves each possible path
 - Inner loops execute $|V|^2$ times
- Overall $O(|V|^3)$
- Better than running Dijkstra $|V|$ times?

Worst Case Running Time

- Add vertices to V_k one at a time
 - Outer loop executes $|V|$ times
- For each new vertex, consider whether it improves each possible path
 - Inner loops execute $|V|^2$ times
- Overall $O(|V|^3)$
- Better than running Dijkstra $|V|$ times?

Worst Case Running Time

- Add vertices to V_k one at a time
 - Outer loop executes $|V|$ times
- For each new vertex, consider whether it improves each possible path
 - Inner loops execute $|V|^2$ times
- Overall $O(|V|^3)$
- Better than running Dijkstra $|V|$ times?