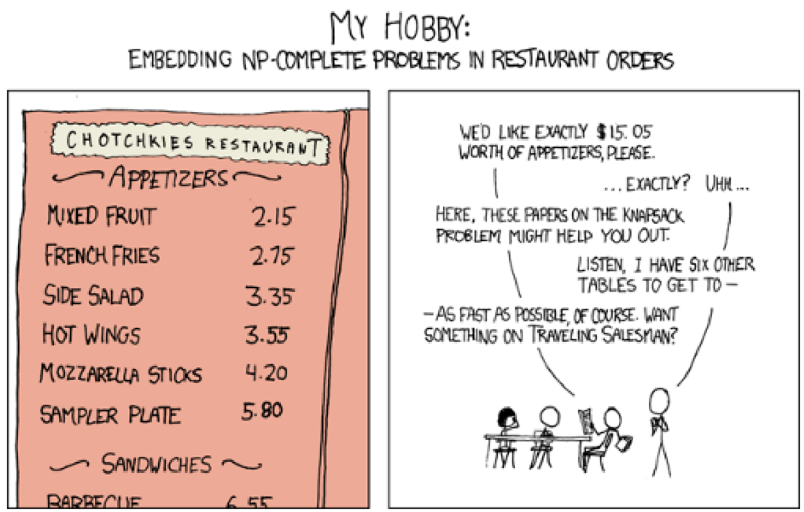
Lecture 24
Knapsack Solved ~~All~~ Many Ways
Shortest Path Algorithms



EECS 281: Data Structures & Algorithms

http://xkcd.com/287

The Knapsack Problem

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

	1	2	3	4	5
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

The Knapsack Problem

Data Structures & Algorithms

Knapsack: Brute-Force

Data Structures & Algorithms

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:

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

1	2	3	4	5	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

1	2	3	4	5	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

```
bool array possSet[1..N] (0:leave,1:take)
maxVal = 0
for i = 1 to 2^N
  possSet[] = genNextPower(N)
  setSize = findSize(possSet[])
  setValue = findValue(possSet[])
  if setSize <= M and setValue > maxVal
    bestSet[] = possSet[]
    maxVal = setValue
return maxVal
```

Brute-Force Efficiency

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

$\boldsymbol{O(N2^N)}$

Knapsack: Brute-Force

Data Structures & Algorithms

Knapsack: Greedy

Data Structures & Algorithms

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

	1	2	3	4	5
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, size[1..N], val[1..N]

Output: integer max value size M knapsack can carry

```
maxVal = 0, currentSize = 0
ratio[] = buildRatio(value[], size[])
// Sort all three arrays by ratio
sortRatio(ratio[], value[], size[])
for i = 1 to N //sorted by ratio
    if size[i] + currSize <= M
        currSize = currSize + size[i]
        maxVal = maxVal + value[i]

return maxVal
```

Greedy Efficiency

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

$$\mathbf{O}(N \log N) + \mathbf{O}(N) \Rightarrow \mathbf{O}(N \log N)$$

Fractional Knapsack: Greedy

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

Knapsack: Greedy

Data Structures & Algorithms

Knapsack: Dynamic Programming

Data Structures & Algorithms

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

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

Knapsack Size

Safe Item #

	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

```
uint32_t knapsackDP(const vector<Item> &items, const size_t m) {
  const size_t n = items.size();
  vector<vector<uint32_t>>
    memo(n + 1, vector<uint32_t>(m + 1, 0));  // +1, +1

  for (size_t i = 0; i < n; ++i) {
    for (size_t j = 0; j < m + 1; ++j) {  // +1 for memo[][m]
      if (j < items[i].size)
        memo[i + 1][j] = memo[i][j];
      else
        memo[i + 1][j] = max(memo[i][j],
            memo[i][j – items[i].size] + items[i].value);
    }  // for j
  }  // for i
  return memo[n][m];
}  // 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

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

Knapsack Size

Safe Item #

	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

```
vector<bool> knapDPReconstruct(const vector<Item> &items,
              const vector<vector<uint32_t>> &memo, const size_t m) {
  const size_t n = items.size();
  size_t c = m;  // current capacity
  vector<bool> taken(n, false);  // included items

  for (int i = n – 1; i >= 0; --i) {  // use int for -1 loop exit
    if (items[i].size <= c) {
      if (memo[i][c - items[i].size] + items[i].value >= memo[i][c]) {
        taken[i] = true;
        c -= items[i].size;
      }  // if ..item added
    }  // if ..item fits
  }  // for i..0
  return taken;
}  // knapDPReconstruct()
```

Run time is ***O*(*N*)**

Knapsack: Dynamic Programming

Data Structures & Algorithms

Knapsack: Branch and Bound

Data Structures & Algorithms

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

Knapsack: Branch and Bound

Data Structures & Algorithms

Shortest Path Algorithms

EECS 281: Data Structures & Algorithms

Dijkstra’s Algorithm

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 $P = \{v_1, v_2, v_3, \ldots, v_k\}$.
- The weighted path length is given by

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

The Shortest Path Problem

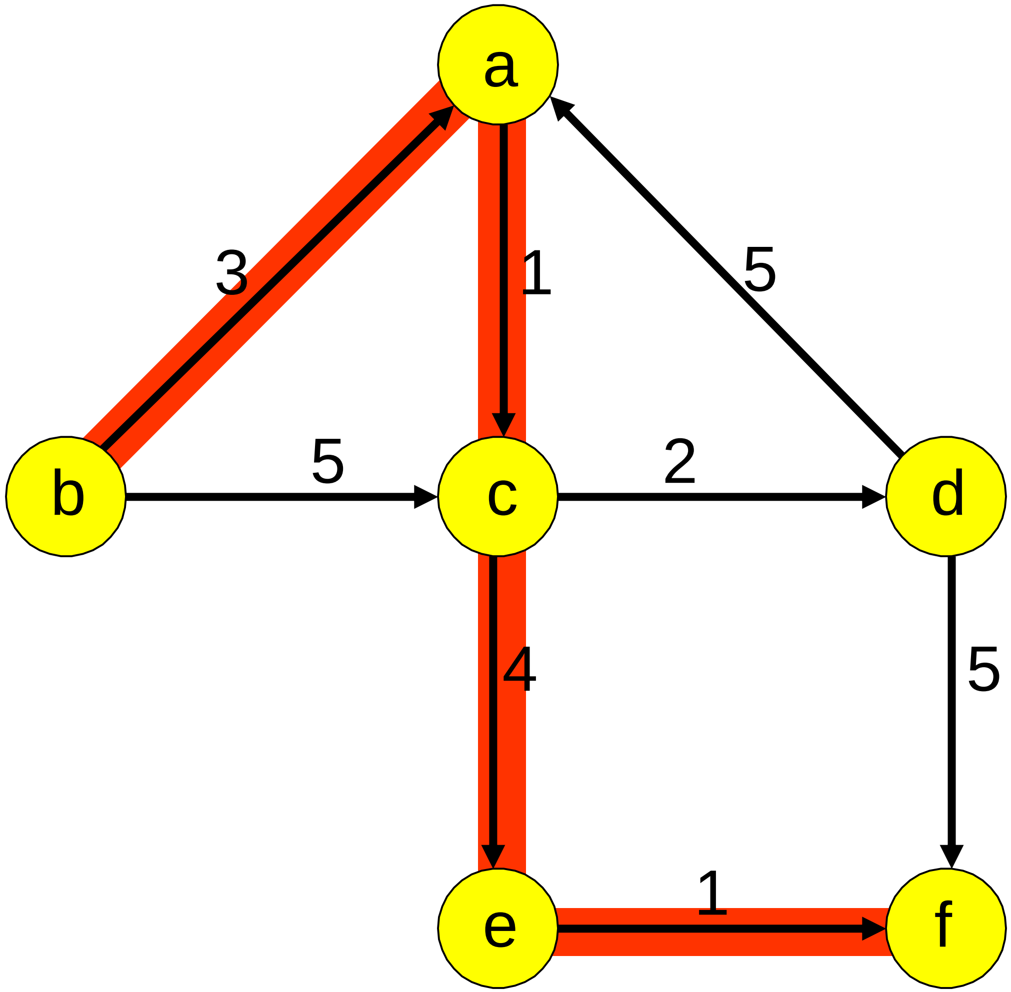
Given an edge-weighted graph $G = (V, E)$ and two vertices, $v_s \in V$ and $v_d \in V$, find the path that starts at v_s and ends at v_d that has the smallest weighted path length

Single-Source Shortest Path

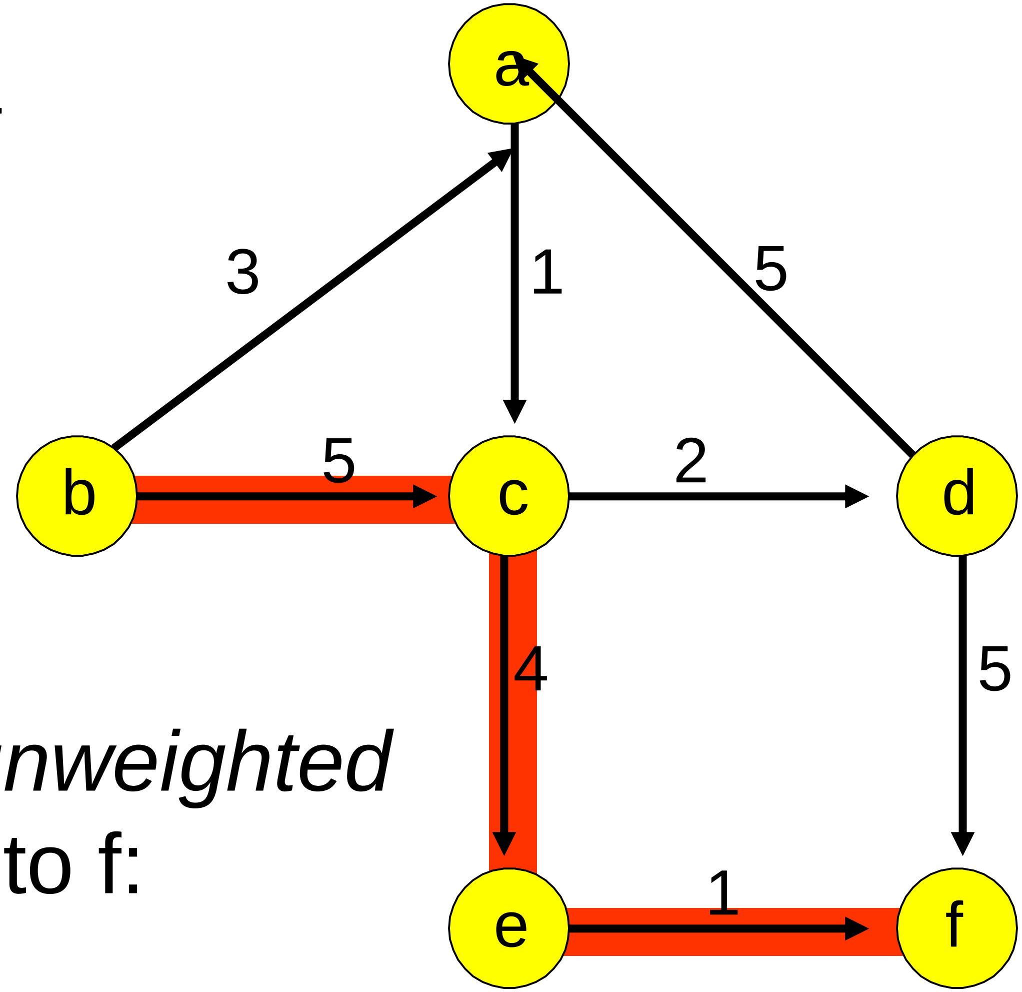
- Given an edge-weighted graph $G = (V, E)$ and a vertex, $v_s \in V$, find the shortest path from v_s to every other vertex in V

- 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}

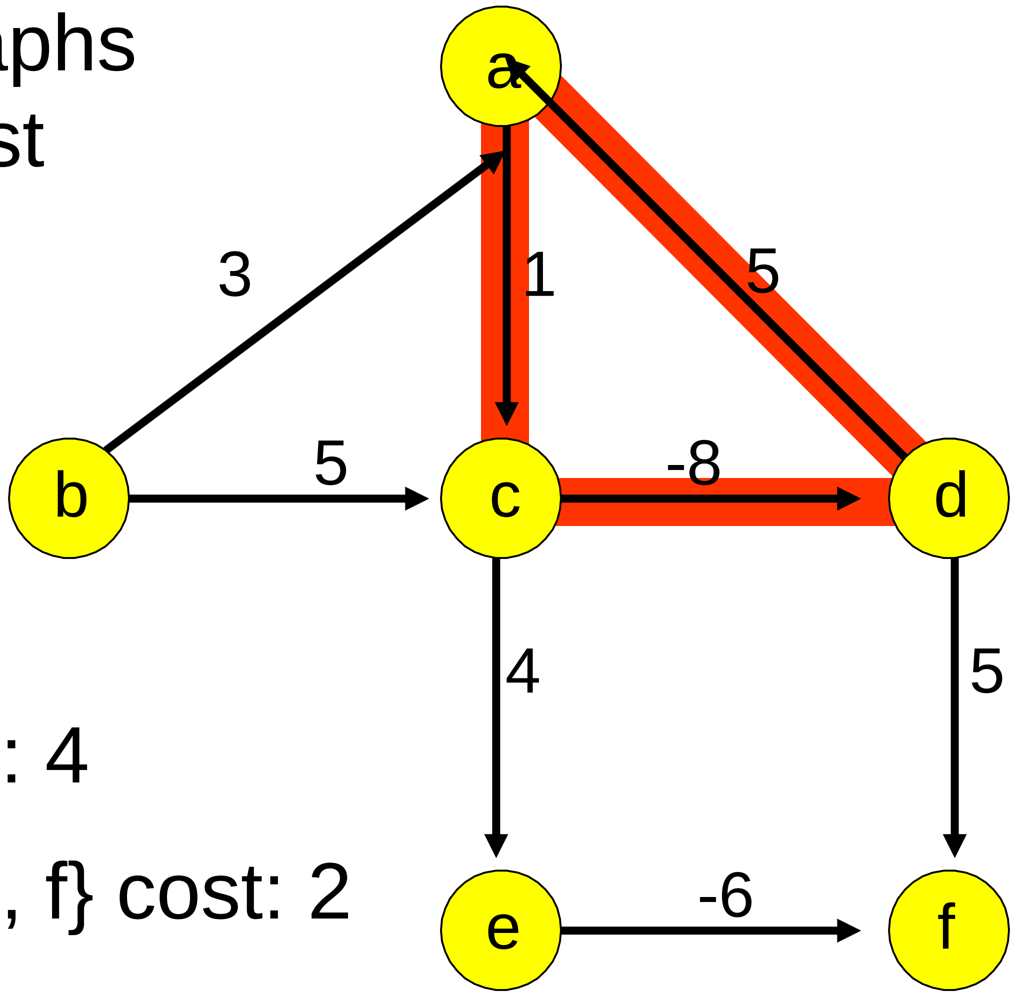


The shortest *weighted* path from b to f: {b, a, c, e, f}



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

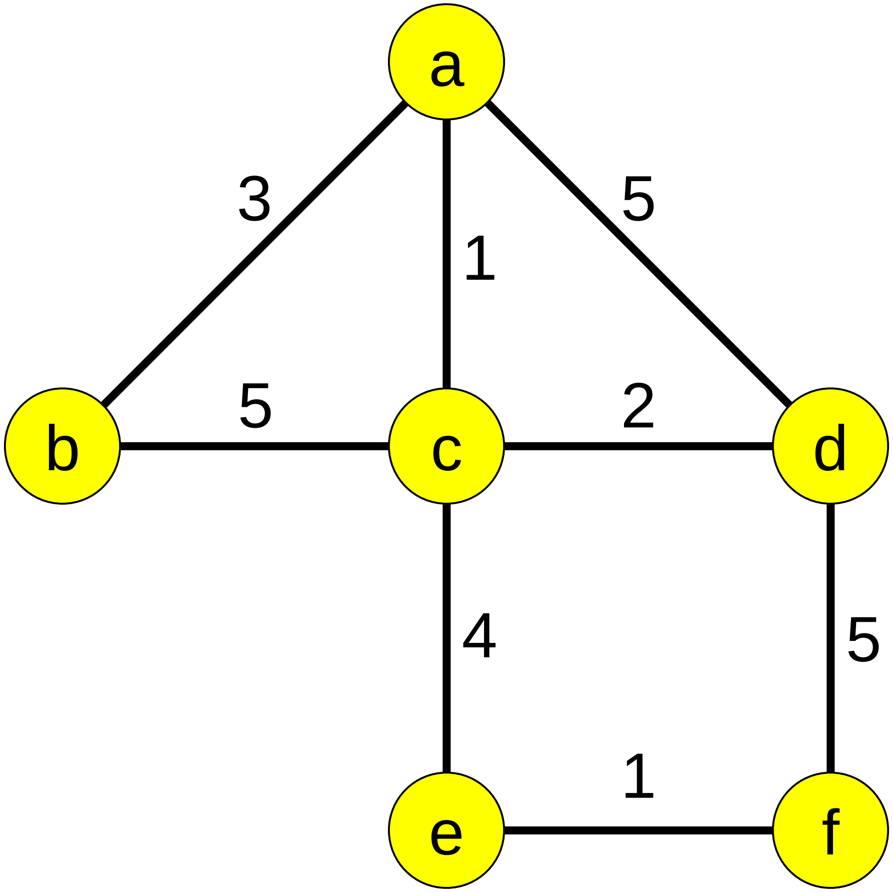
- Greedy algorithm for solving shortest path problem
- Assume non-negative weights
- Find shortest path from v_s to every other vertex

Dijkstra’s Algorithm

For each vertex v, need to know:

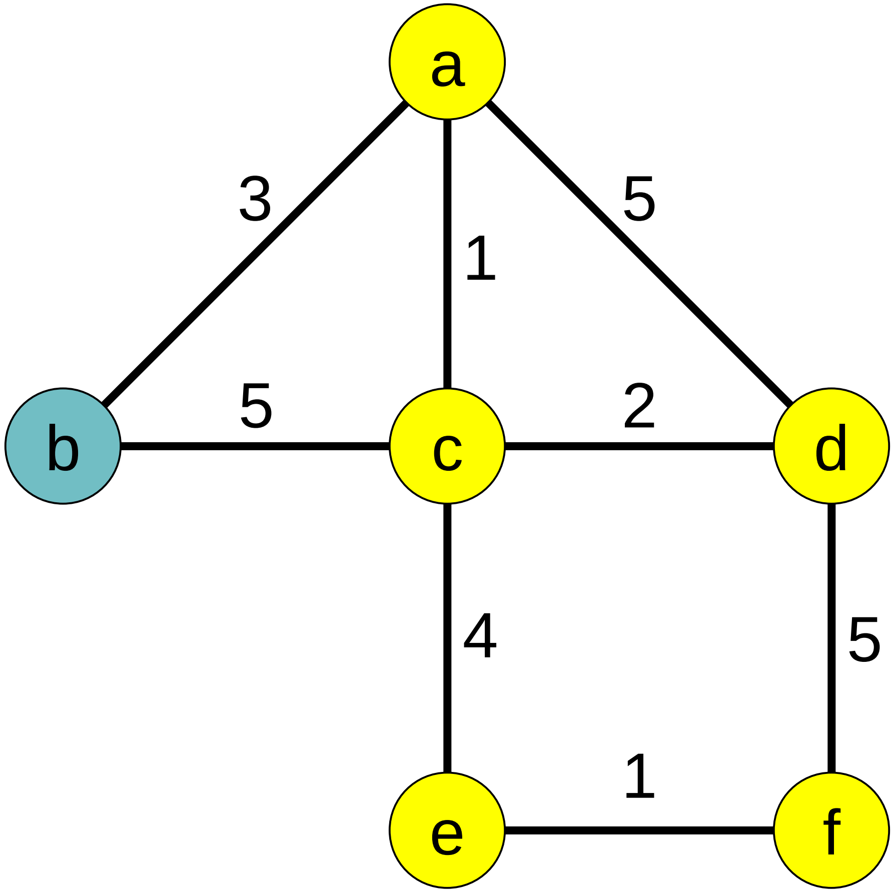
- k_v: Is the shortest path from v_s to v known? (initially false for all $v \in V$)
- d_v: What is the length of the shortest path from v_s to v? (initially ∞ for all $v \in V$, except $v_s = 0$)
- p_v: What vertex precedes (is parent of) v on the shortest path from v_s to v? (initially unknown for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

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

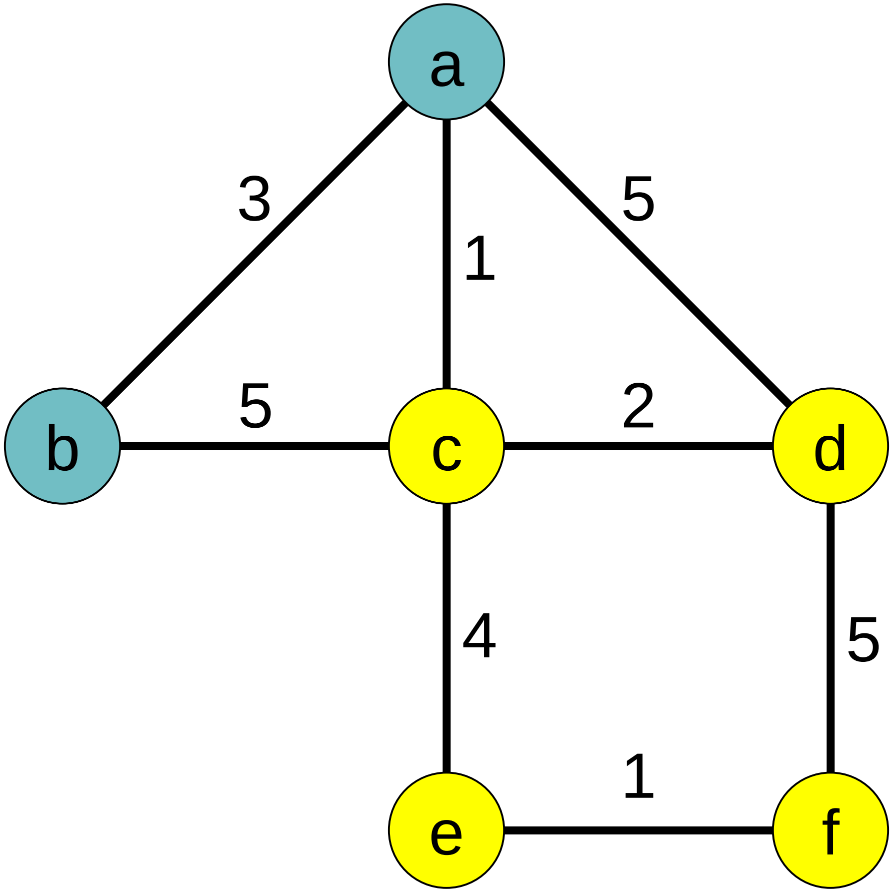


Find shortest paths to *b*

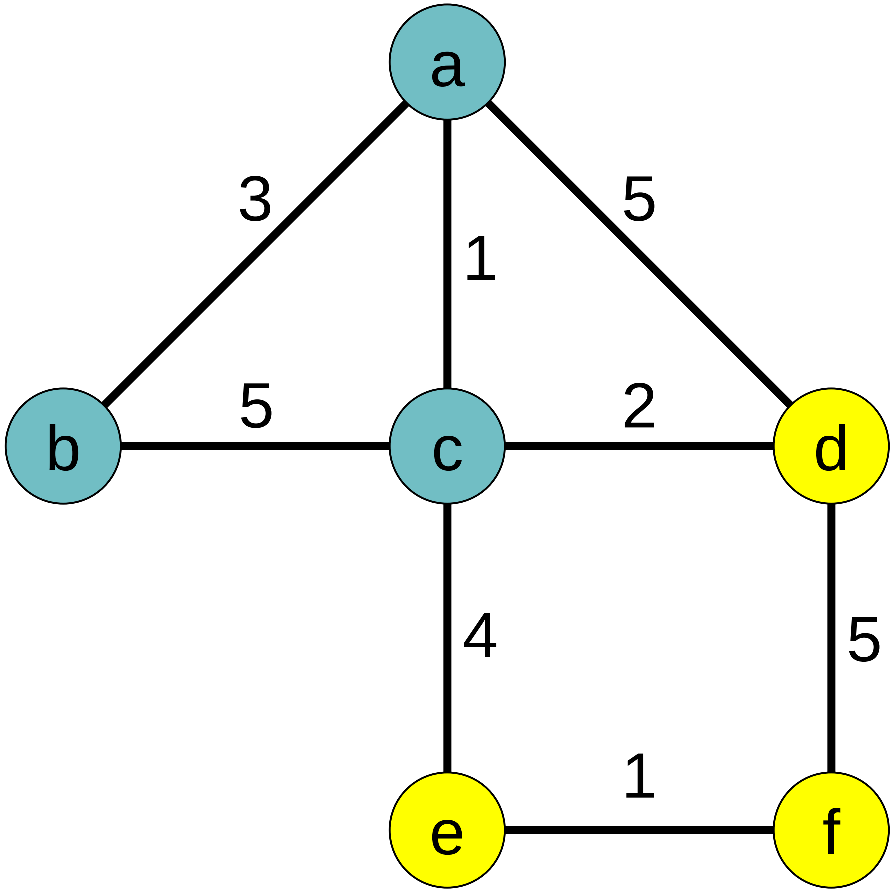
v	k_v	d_v	p_v
a	***F***	3	*b*
b	***T***	0	--
c	***F***	5	*b*
d	***F***	∞	
e	***F***	∞	
f	***F***	∞	



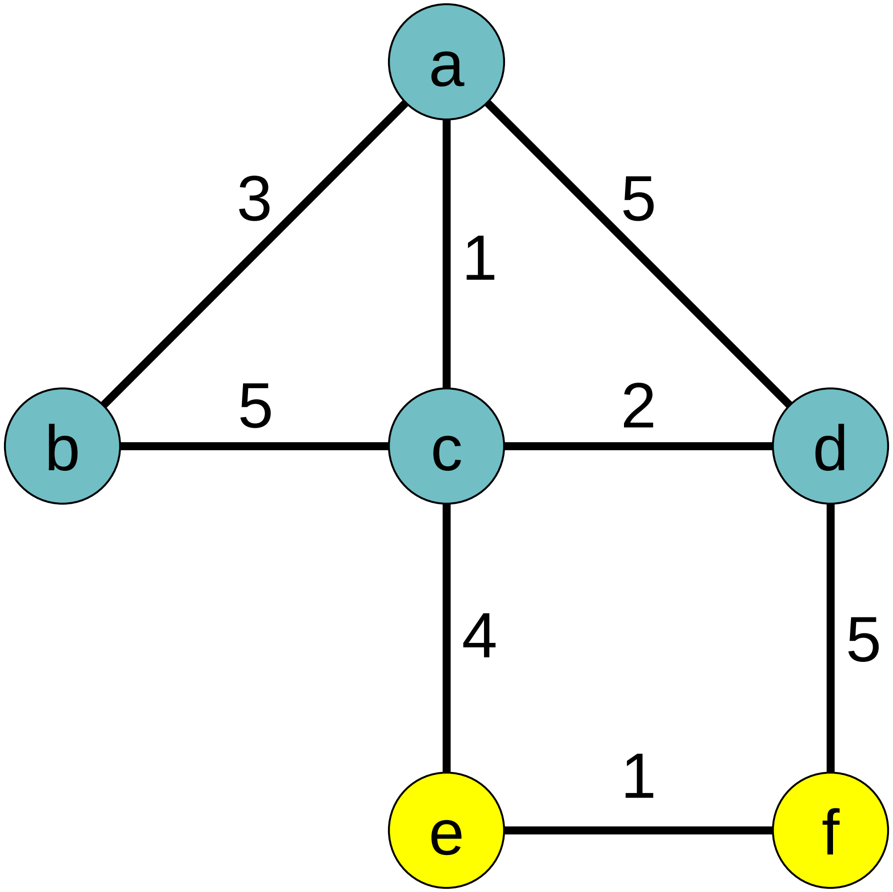
v	k_v	d_v	p_v
a	***T***	3	*b*
b	***T***	0	--
c	***F***	4	*a*
d	***F***	8	*a*
e	***F***	∞	
f	***F***	∞	



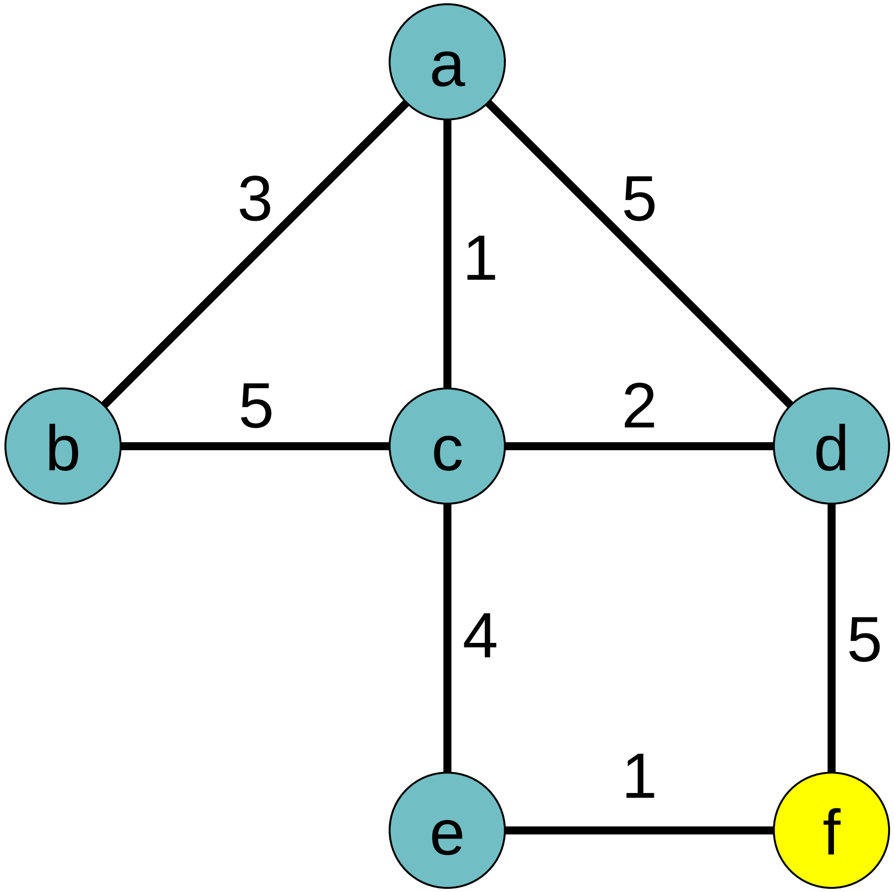
v	k_v	d_v	p_v
a	***T***	3	*b*
b	***T***	0	--
c	***T***	4	*a*
d	***F***	6	*c*
e	***F***	8	*c*
f	***F***	∞	



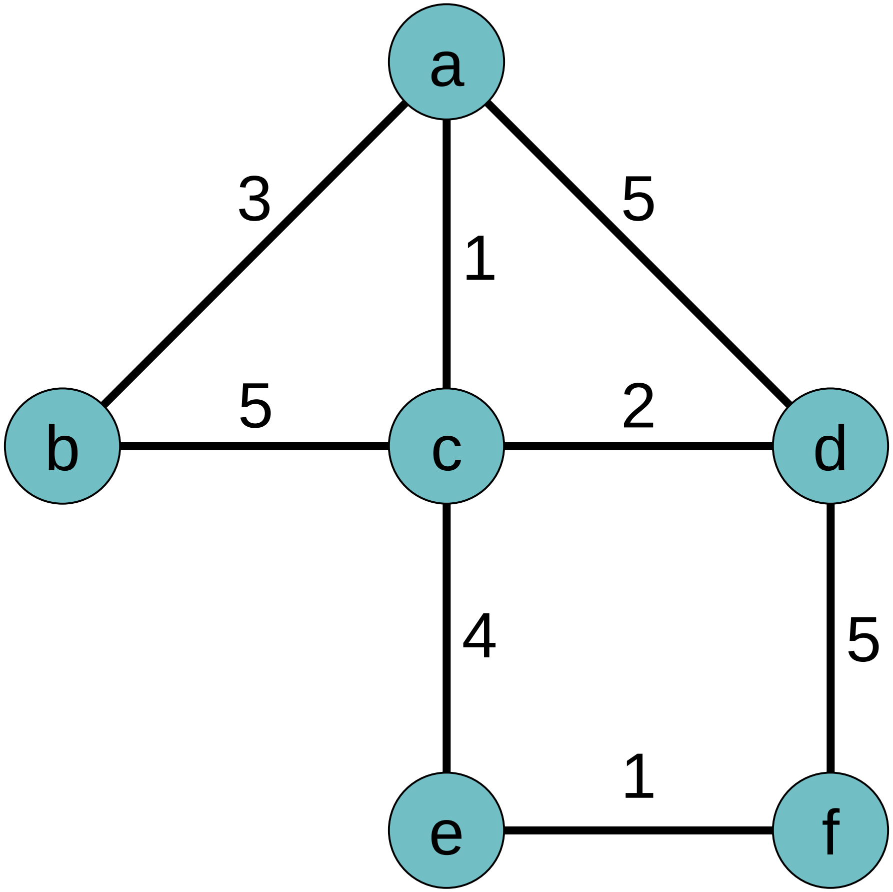
v	k_v	d_v	p_v
a	***T***	3	*b*
b	***T***	0	--
c	***T***	4	*a*
d	***T***	6	*c*
e	***F***	8	*c*
f	***F***	11	*d*



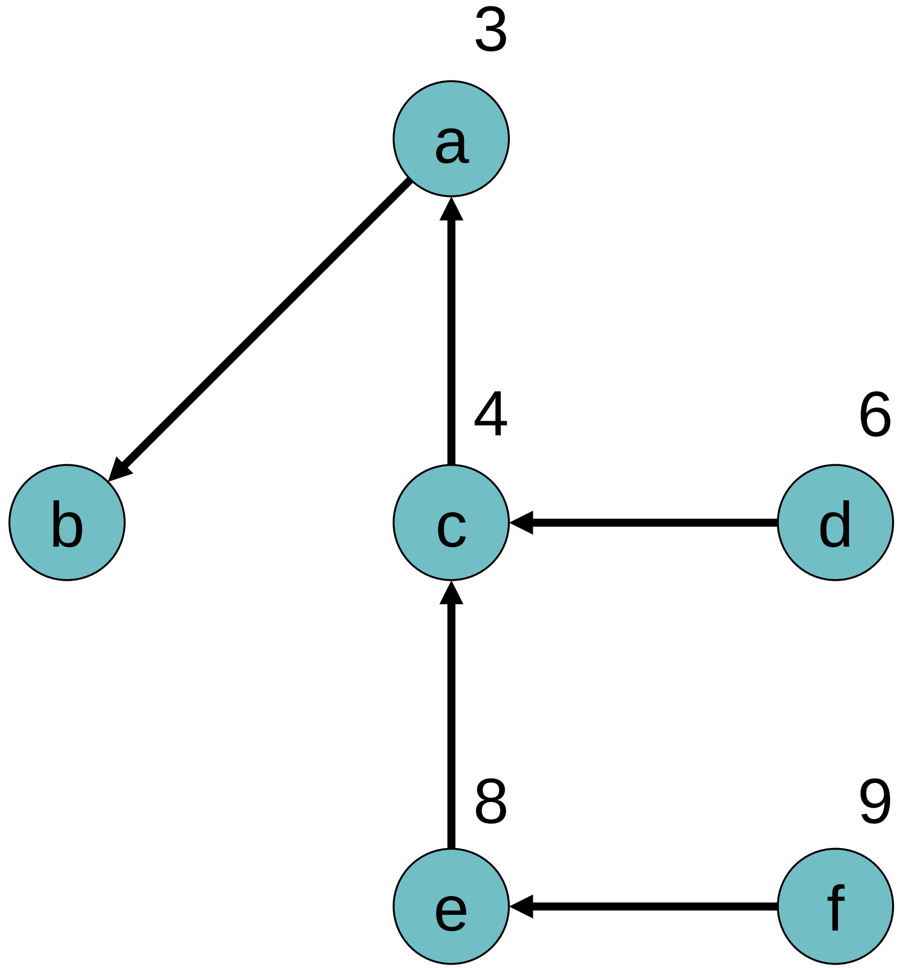
v	k_v	d_v	p_v
a	***T***	3	*b*
b	***T***	0	--
c	***T***	4	*a*
d	***T***	6	*c*
e	***T***	8	*c*
f	***F***	9	*e*



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
- CLRS 24.3 has a good explanation

Dijkstra’s Algorithm

```
Algorithm Dijsktra(G, s0)
  //Initialize
  n = |V|                                O(1)
  create_table(n) //stores k,d,p         O(V)
  create_pq() //empty heap               O(1)
  table[s0].d = 0                        O(1)
  insert_pq(0, s0)                       O(1)
```

Dijkstra’s Algorithm (cont.)

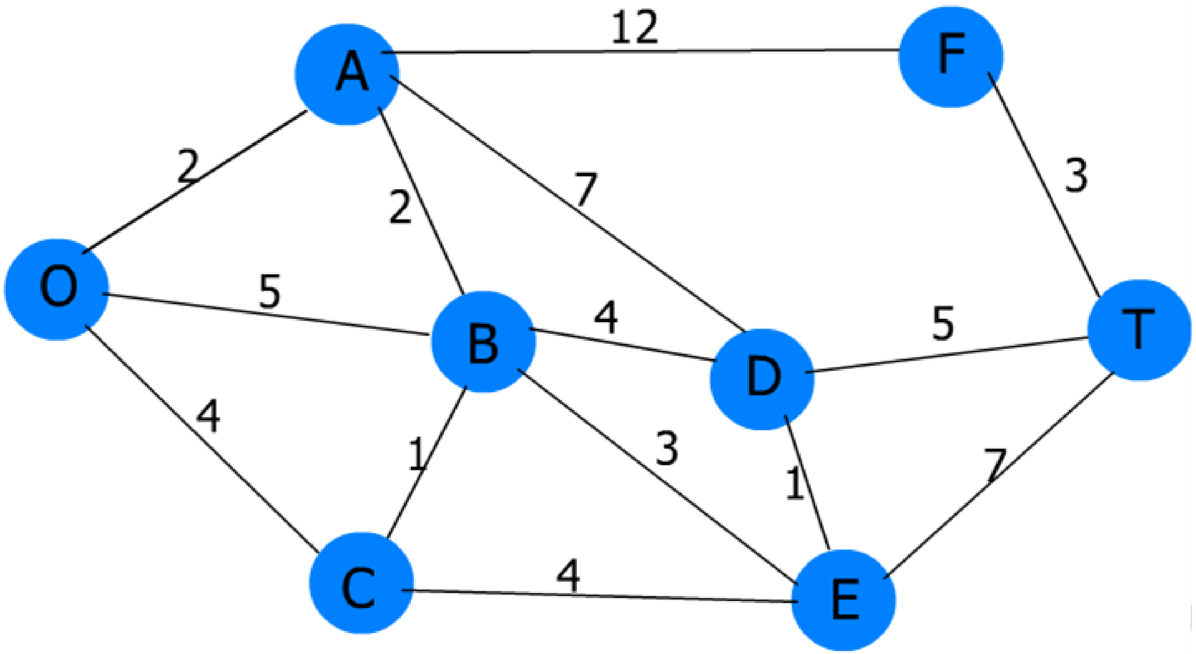
```
while (!pq.isempty)                                  O( E)
  v0 = getMin() //heap top() & pop()                 O( log E)
  if (!table[v0].k) //not known                      O( 1)
    table[v0].k = true                               O( 1)
    for each vi ∈ Adj[v0]                            O( 1 + E/V)
      d = table[v0].d + distance(vi, v0)             O( 1)
      if (d < table[vi].d)                           O( 1)
        table[vi].d = d                              O( 1)
        table[vi].p = v0                             O( 1)
        insert_pq(d, vi)                             O( log E)

for each v ∈ G(V,E)                                  O( V)
      //build edge set in T
      (v, table[v].p) ∈ T(V, E’)                     O( 1)
```

Dijkstra’s Algorithm

Data Structures & Algorithms

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 <u>dense</u> graph
- Uses an adjacency matrix
- $\boldsymbol{O(V^3)}$ (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. $V = \{v_1, v_2, v_3, \ldots, v_{|V|}\}$)

Weighted path length

- Consider the set $V_k = \{v_1, v_2, v_3, \ldots, v_k\}$ 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

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

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

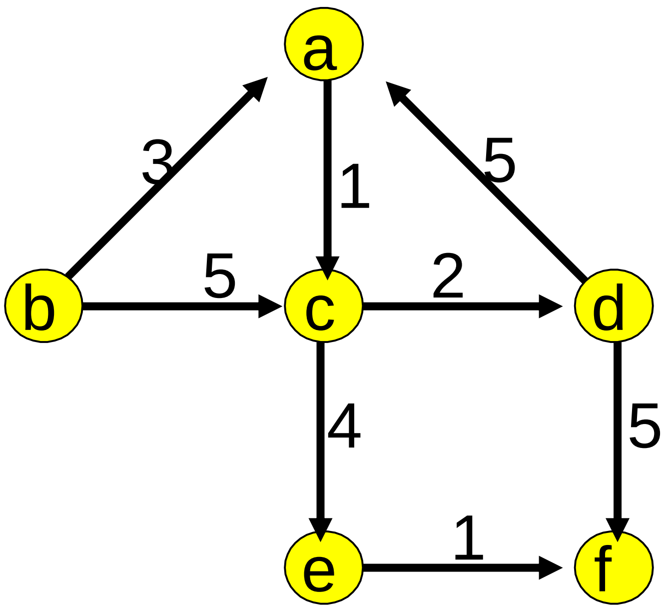
- Therefore D_0 path lengths are:

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

to

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

from



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:
    - $D_{k-1}(i,j)$
    - $D_{k-1}(i,k) + D_{k-1}(k,j)$

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

$V_0 = \{\}$

to

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

from

$V_1 = \{a\}$

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

$V_1 = \{a\}$

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

from

$V_2 = \{a, b\}$

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

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

$V_2 = \{a, b\}$

to

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

from

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

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	∞	∞	∞	∞	∞	∞

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

$V_3 = \{a, b, c\}$ $V_4 = \{a, b, c, d\}$

to

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	∞	∞	∞	∞	∞	∞

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	∞	∞	∞	∞	∞	∞

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

$V_4 = \{a, b, c, d\}$ $V_5 = \{a, b, c, d, e\}$

to

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	∞	∞	∞	∞	∞	∞

from

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	∞	∞	∞	∞	∞	∞

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

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

to

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	∞	∞	∞	∞	∞	∞

from

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

```
Floyd(G) {
  // Initialize
  n = |V|;                                   O( )
  for (k = 0; k <= n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        d[k][i][j] = infinity;               O( )

  for (all (v,w) ∈ E)
    d[0][v][w] = C(v,w)                      O( )
```

Floyd’s Algorithm

```
  // Compute next distance matrix
  for (k = 1; k <= n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        d[k][i][j] = min(d[k-1][i][j],
                         d[k-1][i][k] + d[k-1][k][j]);
}
```

O( )

What About the Paths?

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

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
     // If edge doesn’t exist, no path
     if (C(i,j) == infinity)
       p[0][i][j] = NIL;
     else
       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

$V_1 = \{a\}$

to

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

from

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

```
for (k = 1; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      // Compute next distance matrix
      d[k][i][j] = min(d[k-1][i][j],                              O( )
                       d[k-1][i][k] + d[k-1][k][j]);
      // Compute next paths matrix
      if (d[k-1][i][j]
          <= d[k-1][i][k] + d[k-1][k][j])
        p[k][i][j] = p[k-1][i][j];
      else                                                        O( )
        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?