

COMP90038

Algorithms and Complexity

Lecture 23: Revision

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

What is examinable?

- **Data structures:** Stacks, queues, trees. Binary search trees, AVL trees, heaps (priority queues). Graphs: directed, undirected, weighted.
- **Sorting algorithms:** Selection sort, insertion sort, (shellsort), mergesort, quicksort, heapsort, distribution counting.
- **Search algorithms:** Sequential search, binary search, BSTs, AVL trees, 2–3 trees, hashing, *k*th-largest element.
- **Graph algorithms:** DFS, BFS, topsort, (cyclicity, connectedness), Warshall, Floyd, Prim, Dijkstra.

What is examinable?

- **String manipulation algorithms:** Brute-force string search, Horspool (Huffman encoding, Boyer-Moore, Knuth-Morris-Pratt and Rabin-Karp are not examinable)
- **Other algorithms:** Closest pairs, knapsack, ...
- **Algorithmic techniques:** Brute force, decrease-and-conquer, divide-and-conquer, transform-and-conquer (presorting, representation change), dynamic programming, greedy methods, time/space tradeoffs.
- **Algorithm analysis:** Asymptotics, the Master Theorem for divide-and-conquer.

Priority queues and heaps

- A **priority queue** is a set of elements, each one has an associated **priority** which determines its ejection order
- A **heap** is a **partially ordered binary tree**
 - It must be a **complete tree** (filled from top to bottom, left to right)
 - Each child has a **priority** which is **less or equal** than its parent (max-heap)
 - We represent the heap as an array, starting from position 1, such that the children of node i will be nodes $2i$ and $2i + 1$

Heapsort

- It iterates the sequence: Build the heap – eject the root – build the heap – eject the root ...

swap the root with the last item

- An exercise: sort [8 45 11 97 1 78 82]

1	2	3	4	5	6	7
8	45	<u>11</u>	97	1	<u>78</u>	<u>82</u> ←
8	45	<u>82</u>	97	1	<u>78</u>	<u>11</u>
8	<u>45</u>	82	<u>97</u>	<u>1</u>	78	11
8	<u>97</u>	82	<u>45</u>	<u>1</u>	78	11
<u>8</u>	<u>97</u>	<u>82</u>	45	1	78	11 →
<u>97</u>	<u>8</u>	<u>82</u>	45	1	78	11
97	<u>8</u>	82	<u>45</u>	<u>1</u>	78	11
97	<u>45</u>	82	<u>8</u>	<u>1</u>	78	11

Build the heap

1	2	3	4	5	6	7
<u>97</u>	45	82	8	1	78	<u>11</u> →
<u>11</u>	<u>45</u>	<u>82</u>	8	1	78	<u>97</u>
82	45	<u>11</u>	8	1	<u>78</u>	<u>97</u>
<u>82</u>	45	78	8	1	<u>11</u>	<u>97</u>
<u>11</u>	<u>45</u>	78	8	1	<u>82</u>	<u>97</u>
<u>78</u>	45	<u>11</u>	8	<u>1</u>	<u>82</u>	<u>97</u>
<u>1</u>	<u>45</u>	<u>11</u>	8	<u>78</u>	<u>82</u>	<u>97</u>

1	2	3	4	5	6	7
45	<u>1</u>	11	<u>8</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>8</u>	1	11	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u> ? ?
11	1	<u>8</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>8</u>	1	<u>11</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>1</u>	<u>8</u>	<u>11</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>1</u>	<u>8</u>	<u>11</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>45</u>	<u>8</u>	<u>11</u>	<u>1</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>1</u>	<u>8</u>	<u>11</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>
11	<u>8</u>	<u>1</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>1</u>	<u>8</u>	<u>11</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>
<u>1</u>	<u>8</u>	<u>11</u>	<u>45</u>	<u>78</u>	<u>82</u>	<u>97</u>

Transform and Conquer

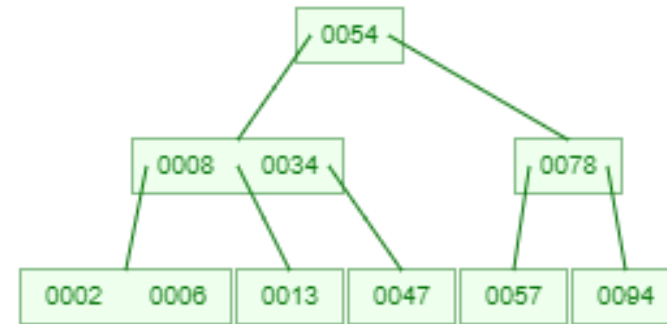
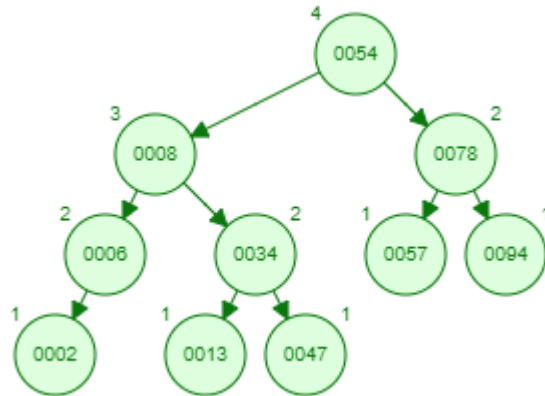
- The idea is to make a change, such that the problem can be solved faster. Two main approaches:
 - Instance simplification, e.g., pre-sorting
 - Representational change, e.g., use a binary search tree
- A **binary search tree** is a binary tree, which must satisfy:
 - For the root r , each element in the **left subtree is smaller** than r , and each element in the **right subtree is larger** than r
- Balanced trees are preferable as in the worst case search requires $\Theta(\log n)$

AVL Trees and 2-3 Trees

- These are approaches to keep the tree balanced
- An AVL tree rotates its nodes, such that the balance factor remains below 2 or -2
- A 2-3 tree allows a maximum of 2 keys and three children per node.

Example

- Build an AVL tree and a 2-3 tree with the data [8 6 54 78 94 13 57 47 2 34]



- <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

Sorting by counting

- Lets go through this example carefully:
 - The keys are: [1 2 3 4 5]
 - The data is: [5 5 1 5 4 1 2 3 5 5 1 5 5 3 5 1 3 5 4 5]
- Lets count the appearances of each key:

Key	1	2	3	4	5
Occurrences	4	1	3	2	10

- Lets add up the occurrences

Occurrences	4	1	3	2	10
	0+4	4+1	5+3	8+2	10+10
Cumulation	4	5	8	10	20

Sorting by counting

- Lets sort the data:


Key	1	2	3	4	5
Cumulation	4	5	8	10	20
P[20]					19
P[10]				9	
P[19]					18
P[8]			7		

[illegible]

Horspool's algorithm

- Lets go through this example carefully:
 - The pattern is 'ACGT' (A=1, T=2, G=3, C=4 $\rightarrow P[.] = [1\ 4\ 3\ 2]$, $m = 4$)
 - The string is GACCGCGTGAGATA**ACGT**CA
- This algorithm creates the table of shifts:

```
function FINDSHIFTS( $P[.]$ ,  $m$ )  
  for  $i \leftarrow 0$  to  $\text{alphasize} - 1$  do  
     $\text{Shift}[i] \leftarrow m$   
  for  $j \leftarrow 0$  to  $m - 2$  do  
     $\text{Shift}[P[j]] \leftarrow m - (j + 1)$ 
```



	A	T	G	C
After first loop	4	4	4	4
j=0	3	4	4	4
j=1	3	4	4	2
j=2	3	4	1	2

Horspool's algorithm

- We append a **sentinel** at the end of the data to guarantee completion
 - The string is now GACCGCGTGAGATA**ACGT**CA**ACGT**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
STRING	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	A	C	G	T
T[.]	3	1	4	4	3	4	3	2	3	1	3	1	2	1	1	4	3	2	4	1	2	1	1	4
FAILED (C!=T, SHIFT BY C)	A	C	G	T																				
FAILED (C!=T, SHIFT BY C)			A	C	G	T																		
FAILED (G!=A, SHIFT BY T)					A	C	G	T																
FAILED (A!=T, SHIFT BY A)									A	C	G	T												
FAILED (A!=T, SHIFT BY A)												A	C	G	T									
FOUND AT 17															A	C	G	T						

- Any change in P will change the shift table.

How many comparisons have to make? (and Brute force)

Hashing

- **Hashing** is a standard way of implementing the abstract data type “dictionary”, a collection of <attribute name, value> pairs.
- The challenges in implementing a **hash table** are:
 - Design an easy (cheap) to compute hash function that distribute the keys evenly.
 - Handling of same addresses (collisions) for different key values
- We examined three approaches
 - Separate Chaining
 - Linear probing
 - Double Hashing

An exercise

- With the hash functions $h(k) = k \bmod 11$ and $s(k) = 3 + k \bmod 7$, draw the hash table that results after inserting in the given order: [35 20 26 62 48]

Index	0	1	2	3	4	5	6	7	8	9	10
Separate Chaining			35		26			62		20	
					48						
Linear Probing			35		26	48		62		20	
Double Hashing	48		35		26			62		20	



48 collision, 再往后移9个?

Recap

- **Dynamic programming** is a bottom-up problem solving technique. The idea is to divide the problem into smaller, overlapping ones. The results are tabulated and used to find the complete solution.
 - Solutions often involves recursion.
- Dynamic programming is often used on **Combinatorial Optimization** problems.
 - We are trying to find the **best** possible **combination** subject to some **constraints**
- Discussed a few problems
 - Coin row problem
 - Knapsack problem
 - Transitive closure of a matrix
 - All pairs shortest paths

The coin row problem

- You are shown a group of coins of different denominations ordered in a row.
- **You can keep some of them**, as long as you **do not pick two adjacent ones**.
 - Your objective is to **maximize your profit** , i.e., you want to take the largest amount of money.
- The solution can be expressed as the recurrence:

$$S(n) = \max (c_n + S (n - 2) , S (n - 1)) \text{ for } n > 1$$

$$S(1) = c_1$$

$$S(0) = 0$$

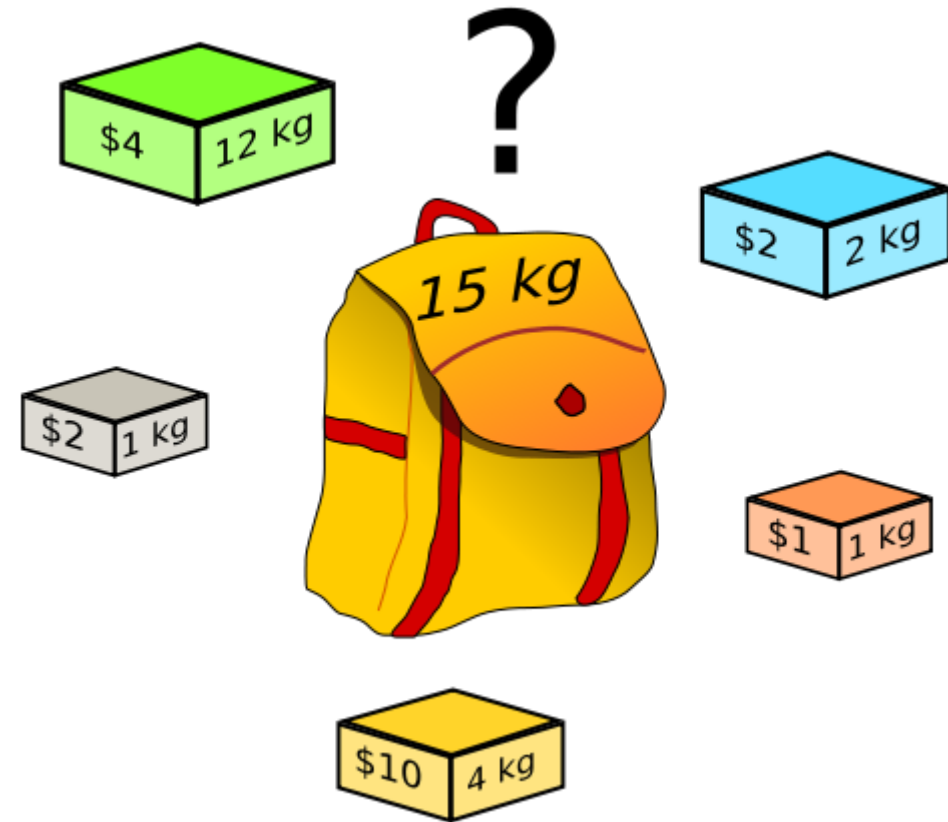
The coin row problem

- Try to solve the problem $C = [50\ 10\ 20\ 20\ 50\ 50\ 20\ 5\ 5\ 10]$

Step	0	50	10	20	20	50	50	20	5	5	10
0	0										
1		50									
2	0	50	50								
3		50	50	70							
4			50	70	70						
5				70	70	120					
6					70	120	120				
7						120	120	140			
8							120	140	140		
9								140	140	145	
10									140	145	150

The knapsack problem

- We also talked about the **knapsack problem**:
- Given a list of n items with:
 - Weights $\{w_1, w_2, \dots, w_n\}$
 - Values $\{v_1, v_2, \dots, v_n\}$
- and a knapsack (container) of capacity W
- Find the **combination** of items with the **highest value** that would **fit into the knapsack**
- All values are positive integers



The knapsack problem

- The algorithm was based on a recursion, with a base **state**:

$$K(i, w) = 0 \text{ if } i = 0 \text{ or } w = 0$$

- And a general case:

$$K(i, w) = \begin{cases} \max(K(i-1, w), K(i-1, w - w_i) + v_i) & \text{if } w \geq w_i \\ K(i-1, w) & \text{if } w < w_i \end{cases}$$

- Another exercise:
 - The knapsack capacity $W = 10$
 - The values are $\{8, 24, 13, 19\}$
 - The weights are $\{7, 4, 6, 5\}$

The knapsack problem

			j	0	1	2	3	4	5	6	7	8	9	10
v	w	i												
		0		0	0	0	0	0	0	0	0	0	0	0
8	7	1		0	0	-1	-1	0	0	0	-1	-1	-1	8
24	4	2		0	-1	-1	-1	24	24	-1	-1	-1	-1	24
13	6	3		0	-1	-1	-1	-1	24	-1	-1	-1	-1	37
19	5	4		0	-1	-1	-1	-1	-1	-1	-1	-1	-1	43

```

function MFKNAP(i, j)
  if i < 1 or j < 1 then
    return 0
  if  $F(i, j) < 0$  then
    if  $j < w(i)$  then
      value = MFKNAP(i - 1, j)
    else
      value = max(MFKNAP(i - 1, j),  $v(i) + \text{MFKNAP}(i - 1, j - w(i))$ )
     $F(i, j) = \text{value}$ 
  return  $F(i, j)$ 

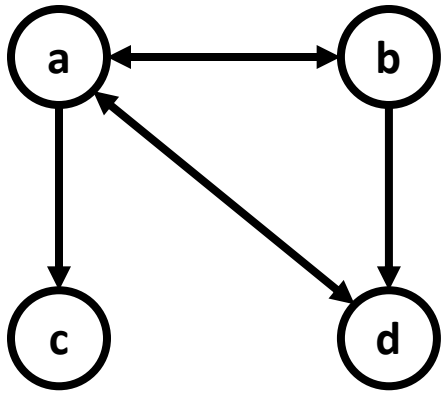
```

Warshall's algorithm

- Warshall's algorithm computes the **transitive closure** of a directed graph, by building a path through the following the rules:
 - step from i to j using only nodes $[1 \dots k-1]$, or
 - step from i to k using only nodes $[1 \dots k-1]$, and then step from k to j using only nodes $[1 \dots k-1]$.
- We examined the simplest version of the algorithm.

```
for  $k \leftarrow 1$  to  $n$  do
  for  $i \leftarrow 1$  to  $n$  do
    if  $A[i, k]$  then
      for  $j \leftarrow 1$  to  $n$  do
        if  $A[k, j]$  then
           $A[i, j] \leftarrow 1$ 
```

Warshall's Algorithm



$$k = 1, i = 2, j = 2$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$k = 1, i = 2, j = 3$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$k = 1, i = 4, j = 2$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$\begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{array}{c} a \quad b \quad c \quad d \end{array} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$k = 1, i = 4, j = 3$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$$k = 1, i = 4, j = 4$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$k = 2, i = 1, j = 1$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

??

Floyd's Algorithm

- Floyd's algorithm solves the **all-pairs shortest-path** problem for weighted graphs with **positive weights**, by building a path following these rules:
 - step from i to j using only nodes $[1 \dots k-1]$, or
 - step from i to k using only nodes $[1 \dots k-1]$, and then step from k to j using only nodes $[1 \dots k-1]$.
- We examined a simple version updating D :

```
function FLOYD( $W[\cdot, \cdot], n$ )  
     $D \leftarrow W$   
    for  $k \leftarrow 1$  to  $n$  do  
        for  $i \leftarrow 1$  to  $n$  do  
            for  $j \leftarrow 1$  to  $n$  do  
                 $D[i, j] \leftarrow \min(D[i, j], D[i, k] + D[k, j])$   
    return  $D$ 
```

Floyd's Algorithm

$$\begin{bmatrix} \infty & 5 & 1 & 2 \\ 4 & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty \\ 3 & \infty & \infty & \infty \end{bmatrix}$$

$$\begin{bmatrix} \infty & 5 & 1 & 2 \\ 4 & 9 & 5 & 10 \\ \infty & \infty & \infty & \infty \\ 3 & 8 & 4 & 5 \end{bmatrix}$$

k=1

$$\begin{bmatrix} 9 & 5 & 1 & 2 \\ 4 & 9 & 5 & 6 \\ \infty & \infty & \infty & \infty \\ 3 & 8 & 4 & 5 \end{bmatrix}$$

k=2

$$\begin{bmatrix} 9 & 5 & 1 & 2 \\ 4 & 9 & 5 & 6 \\ \infty & \infty & \infty & \infty \\ 3 & 8 & 4 & 5 \end{bmatrix}$$

k=3

$$\begin{bmatrix} 5 & 5 & 1 & 2 \\ 4 & 9 & 5 & 6 \\ \infty & \infty & \infty & \infty \\ 3 & 8 & 4 & 5 \end{bmatrix}$$

k=4

Greedy algorithms

- A **greedy algorithm** takes the **locally best** choice among all feasible ones. Such choice is **irrevocable**.
 - Greedy algorithm can provide good **approximations**.
- We applied this idea to two graph problems :
 - Prim's algorithm for finding **minimum spanning trees**
 - Dijkstra's algorithm for **single-source shortest path**

Prim's Algorithm

- We examined the complete algorithm, that uses priority queues:

```
function PRIM( $\langle V, E \rangle$ )
```

```
  for each  $v \in V$  do
```

```
     $cost[v] \leftarrow \infty$ 
```

```
     $prev[v] \leftarrow nil$ 
```

```
  pick initial node  $v_0$ 
```

```
   $cost[v_0] \leftarrow 0$ 
```

```
   $Q \leftarrow \text{INITPRIORITYQUEUE}(V)$ 
```

▷ priorities are cost values

```
  while  $Q$  is non-empty do
```

```
     $u \leftarrow \text{EJECTMIN}(Q)$ 
```

```
    for each  $(u, w) \in E$  do
```

```
      if  $weight(u, w) < cost[w]$  then
```

```
         $cost[w] \leftarrow weight(u, w)$ 
```

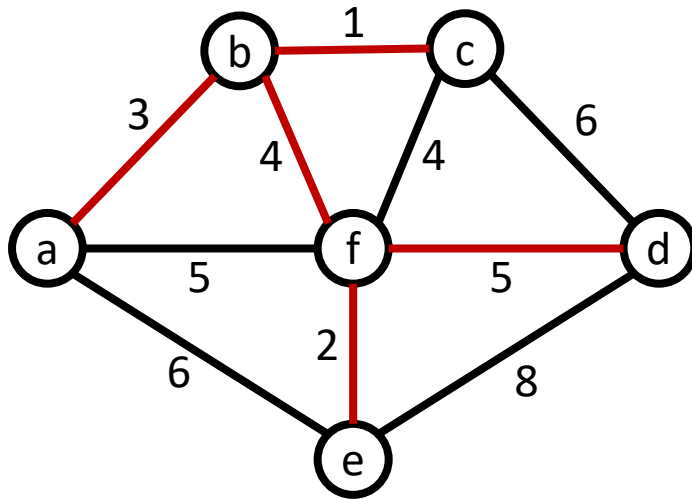
```
         $prev[w] \leftarrow u$ 
```

```
         $\text{UPDATE}(Q, w, cost[w])$ 
```

▷ rearranges priority queue

Another example

- Let's work with the following graph, but starting from b:



Tree T		a	b	c	d	e	f
	cost	∞	∞	∞	∞	∞	∞
	prev	nil	nil	nil	nil	nil	nil
	cost	∞	0	∞	∞	∞	∞
	prev	nil	nil	nil	nil	nil	nil
b	cost	3		1	∞	∞	4
	prev	b		b	nil	nil	b
b,c	cost	3			6	∞	4
	prev	b			c	nil	b
b,c,a	cost				6	6	4
	prev				c	a	b
b,c,a,f	cost				5	2	
	prev				f	f	
b,c,a,f,e	cost				5		
	prev				f		
b,c,a,f,e,d	cost						
	prev						

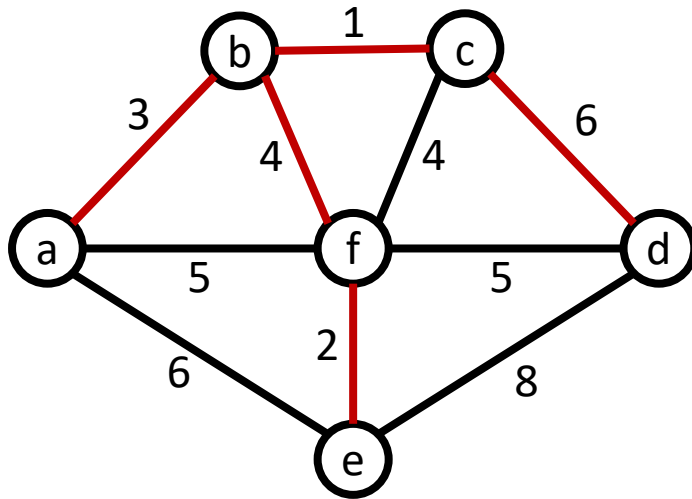
Dijkstra's Algorithm

- **Dijkstra's algorithm** finds all shortest paths **from a fixed start node**. Its complexity is the same as that of Prim's algorithm.

```
function DIJKSTRA( $\langle V, E \rangle, v_0$ )
  for each  $v \in V$  do
     $dist[v] \leftarrow \infty$ 
     $prev[v] \leftarrow nil$ 
   $dist[v_0] \leftarrow 0$ 
   $Q \leftarrow \text{INITPRIORITYQUEUE}(V)$  ▷ priorities are distances
  while  $Q$  is non-empty do
     $u \leftarrow \text{EJECTMIN}(Q)$ 
    for each  $(u, w) \in E$  do
      if  $dist[u] + weight(u, w) < dist[w]$  then
         $dist[w] \leftarrow dist[u] + weight(u, w)$ 
         $prev[w] \leftarrow u$ 
         $\text{UPDATE}(Q, w, dist[w])$  ▷ rearranges priority queue
```

Another example

- Let's work with this graph again, but starting from b:



Tree T		a	b	c	d	e	f
	cost	∞	∞	∞	∞	∞	∞
	prev	nil	nil	nil	nil	nil	nil
	cost	∞	∞	∞	∞	∞	∞
	prev	nil	nil	nil	nil	nil	nil
b	cost	3		1	∞	∞	4
	prev	b		b	nil	nil	b
b,c	cost	3			7	∞	4
	prev	b			c	nil	b
b,c,a	cost				7	9	4
	prev				c	a	b
b,c,a,f	cost				7	6	
	prev				c	f	
b,c,a,f,e	cost				7		
	prev				c		
b,c,a,f,e,d	cost						
	prev						

- Toby will be back for Wednesday.
- Thanks for your attention.
- Please do not forget to fill in the SES.