

# Welcome to CS101 Final Discussion!

Week 16

2021/12/28

20:30~22:00

# HW7-Question 1

**Question 1.** A union tree, used in a disjoint set with only union-by-size optimization, has height 6. The number of nodes contained in that tree **can not** be A.

(A) 63

(B) 64

(C) 65

(D) 80

The worst case tree of height  $h$  must result from taking union of two worst case trees of height  $h-1$ .

## Worst-Case Scenario

Thus, suppose we have a worst-case tree of height  $h$

– The number of nodes is  $\sum_{k=0}^h \binom{h}{k} = 2^h = n$

– The sum of node depth is  $\sum_{k=0}^h k \binom{h}{k} = h2^{h-1}$

– Therefore, the average depth is  $\frac{h2^{h-1}}{2^h} = \frac{h}{2} = \frac{\lg(n)}{2}$

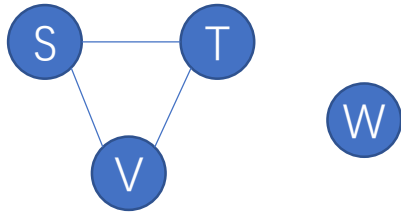
– The height and average depth of the worst case are  $O(\ln(n))$

## HW7-Question 2

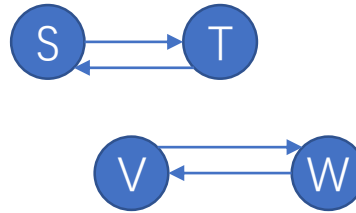
Question 2. Which of the following statements is true? C

- (A) If a graph with  $n$  vertices has  $n - 1$  edges, it must be a tree.
- (B) A directed graph with  $n$  vertices has at least  $2n$  edges to ensure the whole graph is strongly connected.
- (C) Both the time complexity of DFS and that of BFS on graph are  $\Theta(|V| + |E|)$ .
- (D) Every edge is visited exactly once in one iteration of DFS on a connected, undirected graph.

A:



B:



# HW7-Question 4

Question 4. Which of the following statements is false? C

- (A) In a directed simple graph, the maximal number of edges is  $|V|(|V| - 1)$ .
- (B) Undirected graph  $G = (V, E)$  is stored in an adjacency matrix  $A$ . The degree of  $V_i$  is  $\sum_{j=1}^{|V|} A[i][j]$ .
- (C) A DFS of a directed graph always produces the same number of tree edges, i.e. independent of the order in which the vertices are considered for DFS.
- (D) In BFS, let  $d(v)$  be the minimum number of edges between a vertex  $v$  and the start vertex. For any two vertices  $u, v$  in the fringe,  $|d(u) - d(v)|$  is always less than 2.

C:



D:

**Fringe:** the elements in the queue of BFS, or the elements in the stack of DFS. (the elements to be visited.)

# HW7-Disjoint set practice

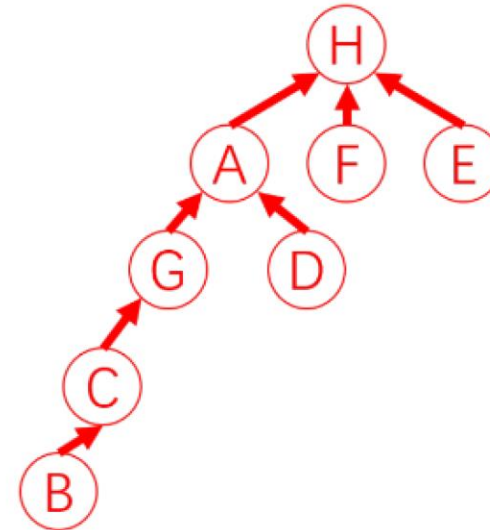
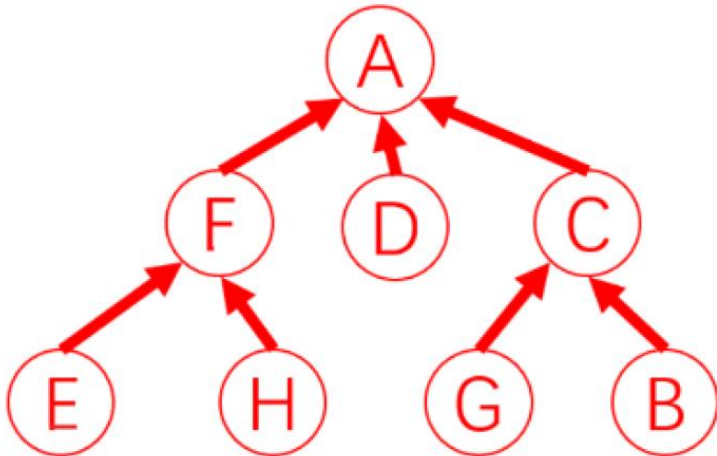
## 2: (2×2') Disjoint set practice

Given the following set of operations on a disjoint set, show the final disjoint set tree for each of the following optimization strategies.

$set\_union(A, D)$ ,  $set\_union(C, B)$ ,  $set\_union(F, E)$ ,  $set\_union(G, C)$ ,  
 $set\_union(D, G)$ ,  $find(A)$ ,  $set\_union(H, E)$ ,  $set\_union(E, G)$ ,  $find(E)$

(1)(2') Only with union-by-size optimization. (When two trees have the same height, the set specified first in the union will be the root of the merged set.)

(2)(2') Only with path compression. (The set specified first in the union will always be the root of the merged set.)



# HW7-DFS went wrong!

## 4: (4') DFS went wrong!

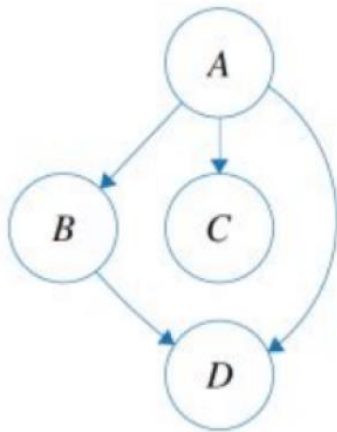
The following algorithm which runs DFS on a directed graph, but it contains an fatal error.

```
Create a stack.  
Choose the initial vertex and mark it as visited.  
Put the initial vertex onto the stack.  
while the stack is not empty:  
    Pop a vertex V from the top of the stack.  
    for each neighbor of V:  
        if that neighbor is not marked as visited:  
            Mark that neighbor as visited.  
            Push that neighbor onto the stack.
```

Please give a graph as an counterexample and briefly explain why this algorithm is wrong.

*Note: In this problem, we say a node is "visited" whenever it is marked as visited.*

This is incorrect because it actually marks all neighbors of the initial node. If this happens:



If A follows the alphabetical order, then the result should be A, B, D, C. But in this case it will be A, B, C, D.

## Breadth-first traversal

Breadth-first traversal on a graph:

- Choose any vertex, mark it as visited and push it onto queue
- While the queue is not empty:
  - Pop the top vertex  $v$  from the queue
  - For each vertex adjacent to  $v$  that has not been visited:
    - Mark it visited, and
    - Push it onto the queue

This continues until the queue is empty

- If there are no unvisited vertices, the graph is connected

## Depth-first traversal

Use a stack:

- Choose any vertex
  - Mark it as visited
  - Place it onto an empty stack
- While the stack is not empty:
  - If the vertex on the top of the stack has an unvisited adjacent vertex  $v$ ,
    - Mark  $v$  as visited
    - Place  $v$  onto the top of the stack
  - Otherwise, pop the top of the stack

# A quick review

- Minimum Spanning Trees
  - Prim
  - Kruskal
- Greedy
  - Interval scheduling
  - Scheduling to minimize lateness
- MST: the spanning tree which minimizes the weight.
  - Given a connected graph with  $n$  vertices, a spanning tree is defined as a subgraph that is a tree and includes all the  $n$  vertices ( $n-1$  edges).

# Minimum Spanning Tree

- Prim's algorithm:
  - Start with an **arbitrary** vertex to form a minimum spanning tree on one vertex.
  - At each step, add the edge with least weight that connects **the current minimum spanning tree (fringe)** to a new vertex.
  - Continue until we have  $n-1$  edges and  $n$  vertices.
- Time complexity
  - With adjacency list:  $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$  as  $|E| = O(|V|^2)$
  - Use **binary heap** to find the shortest edge:  $O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln |V|)$
  - Use Fibonacci heap:  $O(|E| + |V| \ln(|V|))$



# Minimum Spanning Tree

- Kruskal's algorithm:
  - Sort the edges by weight.
  - Go through the edges from least weight to greatest weight, add the edges to the spanning tree so long as the addition does not create a cycle.
  - Repeatedly add more edges until  $|V|-1$  edges have been added.
- Time complexity
  - Without disjoint set: we need  $O(|V|)$  to determine if two vertices are connected. Total:  $O(|E| \ln(|E|) + |E| \cdot |V|) = O(|E| \cdot |V|)$
  - With **disjoint set**: constant time to determine if two vertices are connected. Total:  $O(|E| \ln(|E|)) = O(|E| \ln(|V|))$

## HW8-Question3

ABCD

**Question 3.** *You are given a connected undirected graph  $G$  with  $m$  distinct edges (distinct costs), in adjacency list representation. You are also given the edges of a minimum spanning tree  $T$  of  $G$ . This question asks how quickly you can recompute the MST if we change the cost of a single edge. Which of the following are/is true? (RECALL: The disjoint set data structure has run-time  $O(\alpha(n))$ , which is effectively a constant)*

- (A) Suppose  $e \in T$  and we increase the cost of  $e$ . Then, the new MST can be recomputed in  $O(m)$  deterministic time.*
- (B) Suppose  $e \notin T$  and we increase the cost of  $e$ . Then, the new MST can be recomputed in  $O(m)$  deterministic time.*
- (C) Suppose  $e \in T$  and we decrease the cost of  $e$ . Then, the new MST can be recomputed in  $O(m)$  deterministic time.*
- (D) Suppose  $e \notin T$  and we decrease the cost of  $e$ . Then, the new MST can be recomputed in  $O(m)$  deterministic time.*

## HW8-Question4

D

**Question 4.** Consider the following algorithm that attempts to compute a minimum spanning tree of a connected undirected graph  $G$  with distinct edge costs. First, sort the edges in decreasing cost order (i.e., the opposite of Kruskal's algorithm). Initialize  $T$  to be all edges of  $G$ . Scan through the edges (in the sorted order), and remove the current edge from  $T$  if and only if it lies on a cycle of  $T$ .

Which of the following statements is true?

- (A) The output of the algorithm will never have a cycle, but it might not be connected.
- (B) The algorithm always outputs a spanning tree, but it might not be a minimum cost spanning tree.
- (C) The output of the algorithm will always be connected, but it might have cycles.
- (D) The algorithm always outputs a minimum spanning tree.

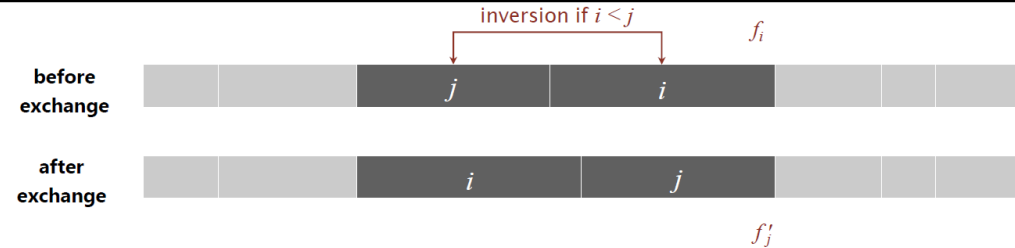
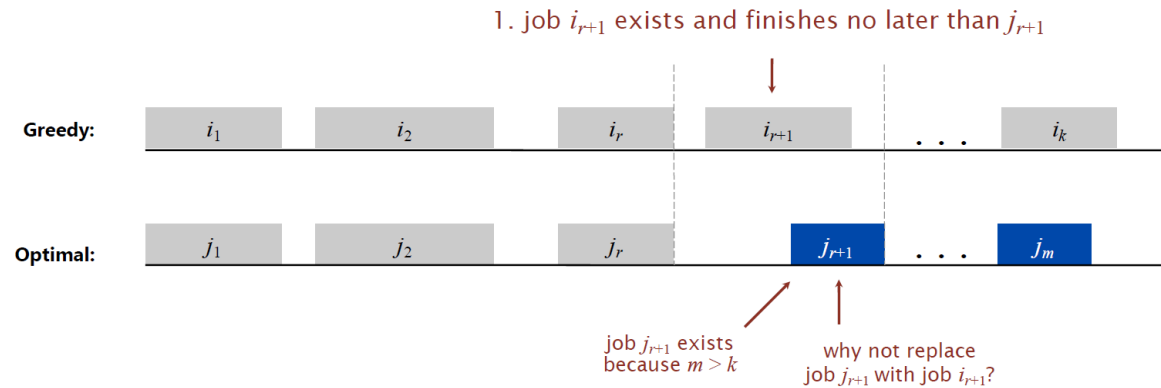
# Greedy Algorithm

- Definition:

- A greedy algorithm is an algorithm which has:
  - A set of **partial solutions** from which a solution is built
  - An objective function which assigns a value to any partial solution
- Then given a partial solution, we
  - Consider possible extensions of the partial solution
  - Discard any extensions which are not feasible
  - Choose that extension which **minimizes** the object function

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let  $i_1, i_2, \dots, i_k$  denote set of jobs selected by greedy.
- Let  $j_1, j_2, \dots, j_m$  denote set of jobs in an optimal solution with  $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$  for the largest possible value of  $r$ .



**Key claim.** Exchanging two adjacent, inverted jobs  $i$  and  $j$  reduces the number of inversions by 1 and does not increase the max lateness.

Pf. Let  $\ell$  be the lateness before the swap, and let  $\ell'$  be it afterwards.

- $\ell'_k = \ell_k$  for all  $k \neq i, j$ .
- $\ell'_i \leq \ell_i$ .
- If job  $j$  is late,  $\ell'_j = f'_j - d_j$  ← definition  
 $= f_i - d_j$  ←  $j$  now finishes at time  $f_i$   
 $\leq f_i - d_i$  ←  $i < j \Rightarrow d_i \leq d_j$   
 $\leq \ell_i$ . ← definition

Then, please see the solution of the following problems, they're very helpful!

HW8-Problem3

HW8-Problem4

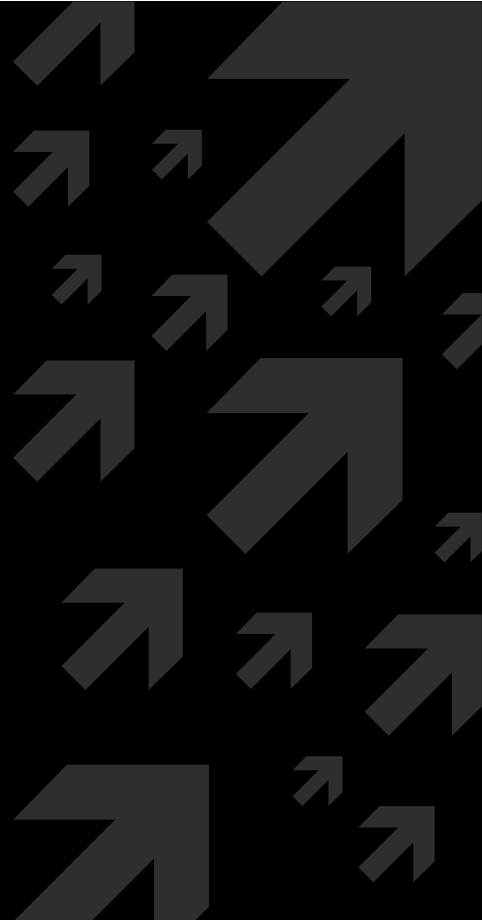
HW8-Problem5

# CS101 Final Review

Topological Sort, Shortest Paths(Dijk/BF/A\*/FW)

# 1. Topological Sort

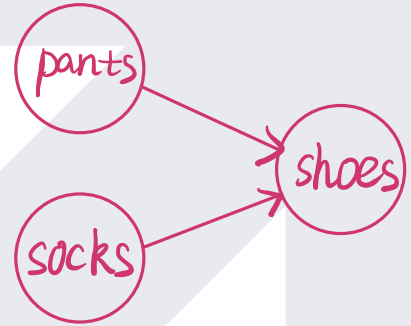
Definition/Algorithm/Notes





# Toposort - Definition

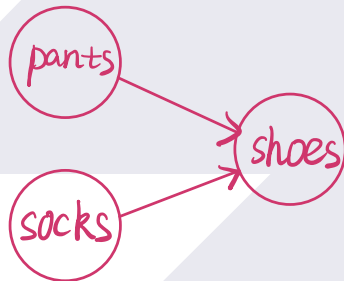
- Directed **A**cyclic **G**raph
- Tasks (**vertices**) with dependencies (**edges**)
  - Task A must be done before task B if ...
  - **Vertex A must appear before vertex B** in topological sorting if ...
  - ... if there exists a **path A** → ... → **B**



# Toposort - Algorithm

```
TopologicalSort(G):  
    Q ← empty queue  
    T ← empty list to store topological sorting  
    In ← list to store in-degree of vertices  
    Push vertices with 0 in-degree into Q  
    While Q is not empty :  
        Pop vertex u from Q  
        Add u to T  
        For each neighbor v of u in G :  
            In[v] ← In[v] - 1  
            if In[v] = 0 :  
                push v into Q  
    Return T
```

# Toposort - Notes



- $G$  must be a **DAG**
- Time complexity:  $O(|V|+|E|)$
- This algorithm returns one feasible topological sorting of  $G$ , but ...
- ... but  $G$  might have different topological sortings!
- Can **detect cycle** in directed graphs!

pants, socks, shoes ✓  
socks, pants, shoes ✓

# Toposort - Homework

**Question 1.** Which of the following statements about *topological sort* is/are true?

(A) Any sub-graph of a DAG has a topological sorting. ✓

(B) Any directed tree has a topological sorting. ✓

} DAG

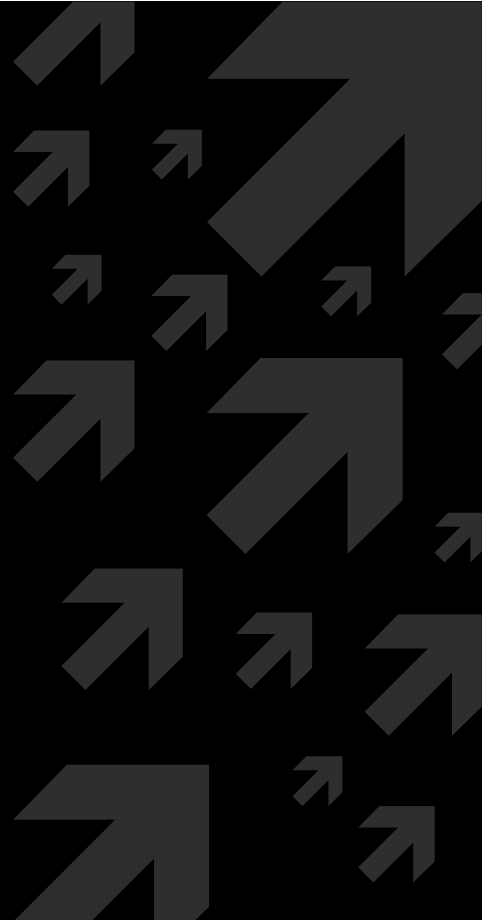
(C) Implementation of topological sort requires  $O(|V|)$  extra space. ✓ in-degree / queue

(D) Since we have to scan all vertices to find those with zero in-degree in each iteration, the run time of topological sort is  $\Omega(|V|^2)$ . ✗  $O(|V|+|E|)$  Using queue

# 2.

## Shortest Paths

Dijkstra/Bellman-Ford/A\* Search/Floyd-Warshall



# Shortest Path

→ Initialization

```
Init-Single-Source(G, s):  
    for each vertex v in G:  
        dis[v] ← ∞  
    dis[s] ← 0  
  
Init-Multiple-Sources(G):  
    dis ← |V|×|V| arrays initialized to ∞  
    for each vertex v in G:  
        dis[v][v] ← 0  
    for each edge e = (u, v) in G:  
        dis[u][v] ← w(u, v)
```

→ Relaxation (apply update rule)

```
Relax(e = (u, v)):  
    if dis[v] > dis[u] + w(u, v) :  
        dis[v] = dis[u] + w(u, v)  
        pre[v] = u
```

- SSST: A\* Search
- SSMT: Dijkstra/Bellman-Ford
- MSMT(All-pairs): Floyd-Warshall

# Shortest Path – Dijkstra

- **Greedy**: like Prim's algorithm for MST
- SSMT: **Single source** multiple terminals
- Only work for **non-negative-weighted** graphs
- Time complexity:  **$O(|E|\log|V|)$**  with binary heap

# Shortest Path – Dijkstra



```
Dijkstra( $G = (V, E)$ ,  $s$ ):  
  Init-Single-Source( $G$ ,  $s$ )  
   $Q \leftarrow$  priority queue initialized by  $v$   
  For  $i \leftarrow 1$  to  $|V|$ :  
    Pop  $u$  with minimum  $\text{dis}[u]$  from  $Q$   
    For each neighbor  $v$  of  $u$  in  $G$ :  
      Relax( $u$ ,  $v$ )
```



# Dijkstra – Homework

Question 2. Which of the following statements about Dijkstra's algorithm is/are true?

- (A) Dijkstra's algorithm can find the shortest path in any DAG.  $\times$   $w < 0$
- (B) If we use Dijkstra's algorithm, whether the graph is directed or undirected does not matter.  $\checkmark$
- (C) If we implement Dijkstra's algorithm with a binary min-heap, we may change keys of internal nodes in the heap.  $\checkmark$  relax / update
- (D) We prefer Dijkstra's algorithm with binary heap implementation to the naive adjacency matrix implementation in a dense graph where  $|E| = \Theta(|V|^2)$ .  $\times$

with adjacency  
dis[i] in one  
lge (u, v). Heap

Implementation	pop()	relax()	total
Adjacency Matrix	$O( V )$	$O(1)$	$O( V ^2)$
Adjacency List	$O( V )$	$O(1)$	$O( V ^2)$
Binary Heap	$O(\log  V )$	$O(\log  V )$	$O( E  \log  V )$
Fibonacci Heap	$O(\log  V )$	$O(1)$	$O( V  \log  V  +  E )$

$|E| = \Theta(|V|^2) \Rightarrow O(|V|^2 \log |V|)$

# Shortest Path – Bellman-Ford

- Dynamic Programming
- SSMT: **Single source** multiple terminals
- Work for **negative-weighted** graphs!
- Time complexity:  **$O(|V||E|)$**
- Can detect negative cycle!

# Shortest Path – Bellman-Ford

$s \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_i \rightarrow \dots \rightarrow t$

```
Bellman-Ford(G = (V, E), s):  
  Init-Single-Source(G, s)  
  For i ← 1 to |V|-1:  
    For each edge e=(u,v) in E:  
      Relax(e)  
  
  For each edge e=(u,v) in E:  
    IF  $dis[v] > dis[u] + w(u, v)$  :  
      There exists negative cycles in G
```

## Loop invariant :

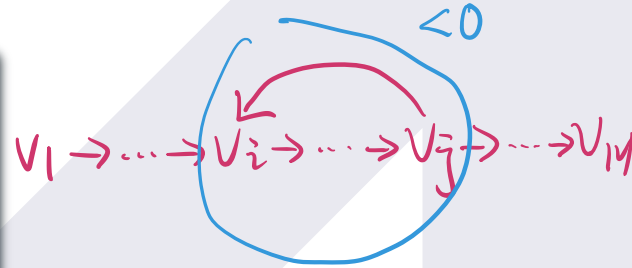
After we relax each edge for  $i$  times ( $i$  outer iterations), we can get **shortest paths of length  $\leq i$**

## Need $|V|-1$ iterations because:

Longest (simple) path is of length  $|V|-1$  at most

# Shortest Path – Bellman-Ford

```
bellman-ford( $G = (V, E), s$ ):  
  init-single-source( $G, s$ )  
  for  $i = 1$  to  $|V|-1$ :  
    for each edge  $e=(u,v)$  in  $E$ :  
      relax( $e$ )  
  
  for each edge  $e=(u,v)$  in  $E$ :  
    if  $\text{dis}[v] > \text{dis}[u] + w(u, v)$  :  
      There exists negative cycles in  $G$ 
```



**Negative cycle exists if...**  
we can still relax some edge for  
the  $|V|-th$  time!

# Shortest Path – Bellman-Ford



```
Bellman-Ford( $G = (V, E)$ ,  $s$ ):  
  Init-Single-Source( $G$ ,  $s$ )  
  For  $i \leftarrow 1$  to  $|V|-1$ :  
    For each edge  $e=(u,v)$  in  $E$ :  
      Relax( $e$ )  
  For each edge  $e=(u,v)$  in  $E$ :  
    if  $\text{dis}[v] > \text{dis}[u] + w(u, v)$  :  
      There exists negative cycles in  $G$ 
```

# Bellman-Ford – Homework

Question 3. Which of the following statements about *Bellman-Ford's algorithm* is/are true?

- (A) *Bellman-Ford's algorithm can find the shortest path for negative-weighted directed graphs without negative cycles while Dijkstra's algorithm may fail.* ✓
- (B) *The run time of Bellman-Ford's algorithm is  $O(|V||E|)$ , which is more time-consuming than Dijkstra's algorithm with heap implementation.* ✓
- (C) *Topological sort can be extended to determine whether a graph has a cycle while Bellman-Ford's algorithm can be extended to determine whether a graph has a negative cycle.* ✓
- (D) *Topological sort can find the critical path in a DAG while Bellman-Ford's algorithm can find the single-source shortest path in a DAG.* ✓

# Shortest Path – Floyd-Warshall

- Dynamic Programming
- MSMT: **All-pairs** shortest path
- Work for **negative-weighted** graphs!
- Time complexity:  $O(|V|^3)$
- Can detect negative cycle! (See your HW)

# Shortest Path – Floyd-Warshall

```
Floyd-Warshall(G = (V, E)):  
  Init-Multiple-Sources(G)  
  For k ← 1 to |V|:  
    For i ← 1 to |V|:  
      For j ← 1 to |V|:  
        dis[i][j] ← min(dis[i][j], dis[i][k]+dis[k][j])
```

## Loop invariant :

After  $k$  out-most iterations, we can get shortest paths which **pass through  $v_1, v_2, \dots, v_k$**

$v_1, v_2, \dots, v_k, \dots, v_{|V|}$



# Floyd-Warshall – Homework

```
for (k = 0; k < V; k++) {  
    for (i = 0; i < V; i++) {  
        for (j = 0; j < V; j++) {  
            if (dist[i][k] + dist[k][j] < dist[i][j])  
                dist[i][j] = dist[i][k] + dist[k][j];  
        }  
    }  
}
```

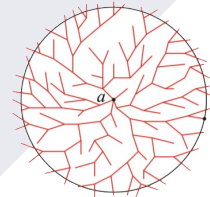
```
for (int i = 0; i < V; i++) _____  
    if (dist [ i ] [ i ] < 0) _____  
        return true; _____
```

# Shortest Path – A\* Search

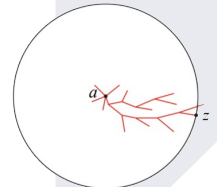
- SSST: **Single Source Single Terminal** (find shortest path from  $s$  to  $t$ )
- Heuristic  $h(v, t)$ : a (estimated) lower bound for shortest path from  $v$  to  $t$
- Compare with Dijkstra:

→ Equivalent to Dijkstra if  $h(v, t) = \begin{cases} 0, & v = t \\ 1, & v \neq t \end{cases}$

→ Often faster than Dijkstra because...



Dijkstra



A\*

# Shortest Path – A\* Search

## → Admissible heuristic

- $h(v,t) \leq \text{dis}(v,t)$  (lower bound of actual shortest distance)
- e.g. Hamming(L0)/Manhattan(L1) distance in N-Puzzle Problem
- Ensures tree search to be optimal

## → Consistent heuristic

- $h(u,t) \leq w(u,v) + h(v,t)$  (triangle inequality)
- Ensures graph search to be optimal
- e.g. Euclidean(L2) distance in real world shortest path
- Consistent heuristic is also admissible

# Shortest Path – A\* Search

```
A*-Tree-Search(G = (V, E), s, h):  
  Init-Single-Source(G, s)  
  Q ← priority queue initialized with s  
  While Q is not empty:  
    Pop u from Q with minimum dis[u]+h(u)  
    For each neighbor v of u:  
      Relax(u, v)  
      If v is not in Q:  
        Push v into Q
```

Requires admissible heuristic

```
A*-Graph-Search(G = (V, E), s, h):  
  Init-Single-Source(G, s)  
  Q ← priority queue initialized with s  
  While Q is not empty:  
    Pop u from Q with minimum dis[u]+h(u)  
    Mark u as visited  
    For each unvisited neighbor v of u:  
      Relax(u, v)  
      If v is not in Q:  
        Push v into Q
```

Requires consistent heuristic

# A\* Search – Homework

## 4: (4\*2') A\*-CSCS

After seeing the A\* search algorithm, in this question we explore a new search procedure using a dictionary for the closed set, A\* graph search with Cost Sensitive Closed Set (A\*- CSCS).

Application:  
Admissible h

```
function A*-CSCS-GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
```

```
  closed ← an empty dictionary
```

```
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
```

```
  loop do
```

```
    if fringe is empty then return failure
```

```
    node ← REMOVE-FRONT(fringe, strategy)
```

```
    if GOAL-TEST(problem, STATE[node]) then return node
```

```
    if STATE[node] is not in closed or COST[node] < closed[STATE[node]] then
```

```
      closed[STATE[node]] ← COST[node]
```

```
      for child-node in EXPAND(node, problem) do
```

```
        fringe ← INSERT(child-node, fringe)
```

```
      end
```

```
  end
```

*closed*  $\Leftrightarrow$  visited  
*fringe*  $\Leftrightarrow$  priority Q

# Thanks!

**Any questions?**

Good Luck!

Contact me via [lianyh@shanghaitech.edu.cn](mailto:lianyh@shanghaitech.edu.cn) if you have any doubt



# CS101 Algorithms and Data Structures

## Final Review

Section DP&NPC

Dec.28<sup>th</sup> 2021

---

# Dynamic Programming



# Dynamic Programming

---

- Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and **storing their solutions** using a memory-based data structure.
- Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup. So the next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby **saving computation time**.
- The example you have learned:
  - Weighted interval scheduling
  - Knapsack problem

# Dynamic Programming

---

- The general outline of a correctness proof for a dynamic programming algorithm is as following:
- 1. Identify optimal substructure.
- 2. Find a recursive formulation for the value of the optimal solution.
- 3. Prove the correctness

Having written out your recurrence, you will need to prove it is correct. Typically, you would do so by going case-by-case and proving that each case is correct. In doing so, you will often use a “cut-and-paste” argument to show why the cases are correct.

# Knapsack Problem

- We have  $n$  items with weights and values:

Item:					
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- And we have a knapsack:



Capacity: 10



Capacity: 10

Item:



Weight:

6

2

4

3

11

Value:

20

8

14

13

35



## • Unbounded Knapsack:

- Suppose I have **infinite copies** of all of the items.
- What's the **most valuable way to fill the knapsack?**



Total weight: 10  
Total value: 42

## • 0/1 Knapsack:





- Suppose I have **only one copy** of each item.
- What's the **most valuable way to fill the knapsack?**



Total weight: 9  
Total value: 35

# Some notation

---

Item:				...	
Weight:	$W_1$	$W_2$	$W_3$		$W_n$
Value:	$V_1$	$V_2$	$V_3$		$V_n$



Capacity:  $W$

# Optimal substructure

---

- Sub-problems:
  - Unbounded Knapsack with a smaller knapsack.



- First solve the problem for small knapsacks



- Then larger knapsacks



- Then larger knapsacks

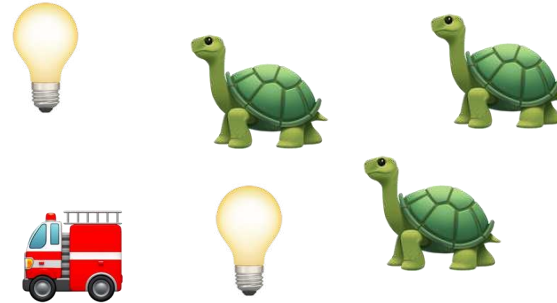
# Optimal substructure



item i

- Suppose this is an optimal solution for capacity  $x$ :

Say that the optimal solution contains at least one copy of item i.

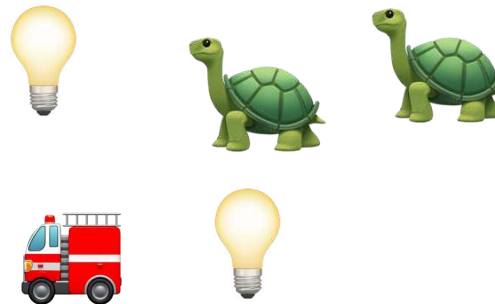


Weight  $w_i$   
Value  $v_i$



Capacity  $x$   
Value  $V$

- Then this optimal for capacity  $x - w_i$ :



Capacity  $x - w_i$   
Value  $V - v_i$

If I could do better than the second solution, then adding a turtle to that improvement would improve the first solution.

# Recursive relationship

- Let  $K[x]$  be the optimal value for capacity  $x$ .

$$K[x] = \max_i \{ \text{🎒} + \text{🐢} \}$$

The maximum is over  
all  $i$  so that  $w_i \leq x$ .

Optimal way to  
fill the smaller  
knapsack

The value of  
item  $i$ .

$$K[x] = \max_i \{ K[x - w_i] + v_i \}$$

- (And  $K[x] = 0$  if the maximum is empty).
  - That is, there are no  $i$  so that  $w_i \leq x$ .



---

# **Reduction, P/NP and NP-complete**

# Decision Problems

---

- Def: assignment of inputs to No(0) or Yes(1)
- Inputs are either **No instances** or **Yes instances** (i.e. satisfying instances)

Problem	Decision
Independent Set	Given a graph, does it contains a subset of $k$ (or more) vertices such that no two are adjacent?
3-SAT	Given a CNF formula, dost it have a satisfying truth assignment?
3-coloring	Given an undirected graph, can the nodes be colored black, white, and blue so that no adjacent nodes have the same color?

# Reduction Review

---

- Suppose you want to solve problem  $A$ .
- One way to solve is to convert  $A$  into a problem  $B$  you know how to solve.
- Solve using an algorithm for  $B$  and use it to compute solution to  $A$ .
- This is called a reduction from problem  $A$  to problem  $B$  ( $A \rightarrow B$ )
- Because  $B$  can be used to solve  $A$ ,  $B$  is **at least as hard** ( $A \leq B$ )
- If you can solve  $B$ , you can also solve  $A$ .

# P and NP

---

- **P**: the set of decision problems for which there is an algorithm  $A$  such that for **every** instance  $I$  of size  $n$ ,  $A$  on  $I$  runs in  $\text{poly}(n)$  time and solves  $I$  correctly.
- **NP**: the set of decision problems for which there is an algorithm  $V$ , a “certifier”, that takes as input **an** instance  $I$  of the problem, and a “certificate” bit string of length **polynomial** in the size of  $I$ , so that:
  - $V$  always runs in **polynomial** time, and the input and output should be in the **polynomial** size of  $I$
  - if  $I$  is a YES-instance, then there is **some** certificate  $c$  so that  $V$  on input  $(I, c)$  returns **YES**, and
  - if  $I$  is a NO-instance, then **no matter** what  $c$  is given to  $V$  together with  $I$ ,  $V$  will always output **NO** on  $(I, c)$ .
- **For NP Problem, an instance can be verified in polynomial time.**
- You can think of the certificate as a proof that  $I$  is a YES-instance. If  $I$  is actually a NO-instance then no proof should work.

## P, NP, and EXP

---

**P.** Decision problems for which there exists a poly-time algorithm.

**NP.** Decision problems for which there exists a poly-time certifier.

**EXP.** Decision problems for which there exists an exponential-time algorithm.

**Proposition.**  $P \subseteq NP$ .

**Pf.** Consider any problem  $X \in P$ .

- By definition, there exists a poly-time algorithm  $A(s)$  that solves  $X$ .
- Certificate  $t = \varepsilon$ , certifier  $C(s, t) = A(s)$ . ▀

**Proposition.**  $NP \subseteq EXP$ .

**Pf.** Consider any problem  $X \in NP$ .

- By definition, there exists a poly-time certifier  $C(s, t)$  for  $X$ , where certificate  $t$  satisfies  $|t| \leq p(|s|)$  for some polynomial  $p(\cdot)$ .
- To solve instance  $s$ , run  $C(s, t)$  on all strings  $t$  with  $|t| \leq p(|s|)$ .
- Return *yes* iff  $C(s, t)$  returns *yes* for any of these potential certificates. ▀

**Fact.**  $P \neq EXP \Rightarrow$  either  $P \neq NP$ , or  $NP \neq EXP$ , or both.

# P vs NP

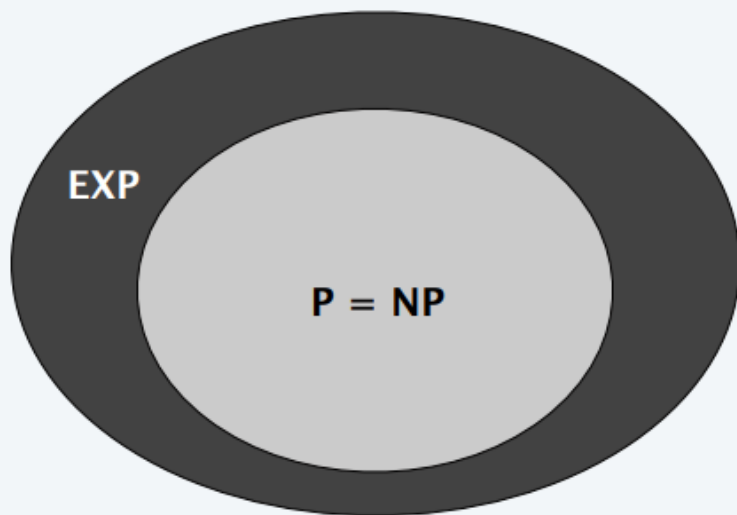
---

- NP  $\neq$  Not Polynomial Time Problem
- NP = Nondeterministic Polynomial Time Problem
- $P = NP$ ?  $P \neq NP$ ? This is an open question.
- Why do we care? If can show a problem is hardest problem in NP, then problem cannot be solved in polynomial time if  $P \neq NP$ .
- How to define hardest? How do we relate difficulty of problems? Reductions!

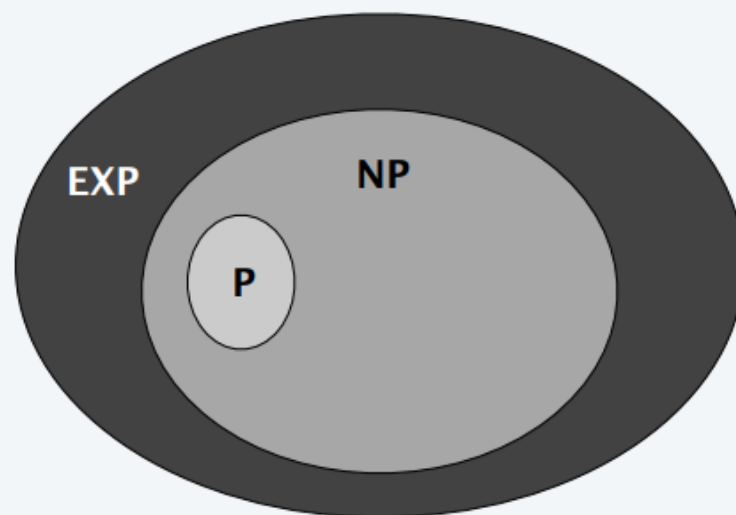
# P vs NP

Does  $P = NP$ ? [Cook 1971, Edmonds, Levin, Yablonski, Gödel]

Is the decision problem as easy as the certification problem?



If  $P = NP$



If  $P \neq NP$

# NP-completeness

---

- Out of all the NP problems, there's a subset of NP problems called **NP-complete (NPC)** problems that are the “hardest” NP problems.
- To determine whether  $P=NP$ , it suffices to know whether  $P=NPC$ .
  - If the hardest problems can be solved in polytime, then all NP problems can be solved in polytime. i.e.  $P=NP$ .
- So the study of P vs NP focuses on NPC problems.



# NP-completeness

---

- Def A problem  $A$  is **NP-complete (NPC)** if the following are true.
  - $A \in NPC$ .
  - Given any other problem  $B \in NP$ ,  $B \leq_p A$ .
- Thus, a NP-complete problem is an NP problem that can be used to solve any other NP problem.
  - It's a “hardest” NP problem.



# How to prove NPC?

---

- Given two NP problems  $A$  and  $B$ , suppose  $A$  is NP-complete, and  $A \leq_P B$ . Then  $B$  is also NP-complete.
- Proof Let  $C$  be any NP problem. Then  $C \leq_P A$ , since  $A$  is NP-complete.
  - Since  $A \leq_P B$ , then by Theorem 1, we have  $C \leq_P A \leq_P B$ .
  - Since also  $B \in NP$ , then  $B$  is NPC.
- To prove a problem  $B$  is NP-complete
  - First, you have to prove  $B \in NP$ , but that's usually not hard.
  - Second, take a problem  $A$  you know is NPC, and prove  $A \leq_P B$ .
    - To prove  $A \leq_P B$ , you need to give a polytime reduction from  $A$  to  $B$ .
    - $\Rightarrow$ : To say  $B$  is harder than  $A$ , which has been known as a NPC problem.
    - $\Leftarrow$ : Because  $B$  is NPC as well,  $A$  and  $B$  are “**equivalently**” hard. This means  $B$  can also be reduced to  $A$ .

# Knapsack Problem

---

- We have  $n$  items, each with weight  $a_j$  and value  $c_j$  ( $j = 1, \dots, n$ ). All  $a_j$  and  $c_j$  are positive integers.
- The question is to find a subset of the items with total weight at most  $b$  such that the corresponding profit is at least  $k$  ( $b$  and  $k$  are also integers). Show that **Knapsack** is NP-complete by a reduction from **Subset Sum**.
- Subset Sum Problem: Given  $n$  natural numbers  $w_1, \dots, w_n$  and an integer  $W$ , is there a subset that adds up to exactly  $W$ ?

# Knapsack Problem

---

- 1. First, prove Knapsack  $\in NP$
- Given a subset of the items, it is obvious that we can check whether the total weight is at most  $b$  and the corresponding profit is at least  $k$ , which can be done in polynomial time. Thus the Knapsack problem is in NP.

# Knapsack Problem

---

- 2. Second, prove  $\text{Subset Sum} \leq_P \text{Knapsack}$
- Give a polytime reduction from **Subset Sum** to **Knapsack**:

For any instance of subset sum problem with sum  $W$ ,

We can construct an instance of the Knapsack problem with

$$a_i = c_i = S_i \text{ and } b = k = W$$

which has a satisfying result iff it is satisfiable of subset sum problem.

# Knapsack Problem

---

- 2. Second, prove  $\text{Subset Sum} \leq_P \text{Knapsack}$

- We now prove  $(W, S)$  is a yes-instance of subset sum problem if and only if  $(a, c, b, k)$  is a yes-instance of the constructed Knapsack problem:

- “ $\Rightarrow$ ”:

if  $(W, S)$  is a yes-instance of subset sum problem, we can let  $\{x_1, x_2, \dots, x_k\}$  be a result of the problem with  $\sum S_i = W$ . Then we have  $\sum a_i = \sum c_i = b = k$  which means the constructed Knapsack problem is also a yes-instance.

- “ $\Leftarrow$ ”

# Knapsack Problem

---

- 2. Second, prove  $\text{Subset Sum} \leq_P \text{Knapsack}$
- We now prove  $(W, S)$  is a yes-instance of subset sum problem if and only if  $(a, c, b, k)$  is a yes-instance of the constructed Knapsack problem:
  - “ $\Rightarrow$ ”
  - “ $\Leftarrow$ ”:

if  $(a, c, b, k)$  is a yes-instance of the constructed Knapsack problem with a result  $\{x_1, x_2, \dots, x_k\}$ , then  $\sum S_i = \sum a_i \leq b$  and  $\sum S_i = \sum c_i \geq c$ . Then we have  $\sum S_i = W$ , which is also a yes-instance of subset sum problem.



---

**The last thing you must know!**

# Agenda

---

- Recent Deadlines:
  - Weekly Homework 12: Jan.3<sup>th</sup>, 23:59 (Monday)
  - Programming Homework 4: Jan.2<sup>th</sup>, 23:59 (Sunday)

# Agenda

---

- Exam Time:
  - **Jan.5<sup>th</sup>, Wednesday**, 08:00-10:00, 120 minutes in total
- Covers:
  - The lectures from **lec15 to lec28** (*excluding lec15.1 Red-Black Trees*)
- Policy:
  - Closed book. No cheating sheet.
  - No electronic devices. For example, calculators, PC, smartphones and so on.
  - **And any other actions that violate the rule of the exam will be judged as plagiarism and zero score immediately.**

# Agenda

---

- Exam Contents:
  - 10 T/F (20 points)
  - 5 Single Choice (15 points)
  - 5 Multiple Choices (25 points)
  - 4 General Problems (40 points)



上海科技大学  
ShanghaiTech University

---

**Thanks for your hard work!**

**Wish you have a better score in final exam!**

CS101 Staff Team