COMP20003 Workshop Week 11

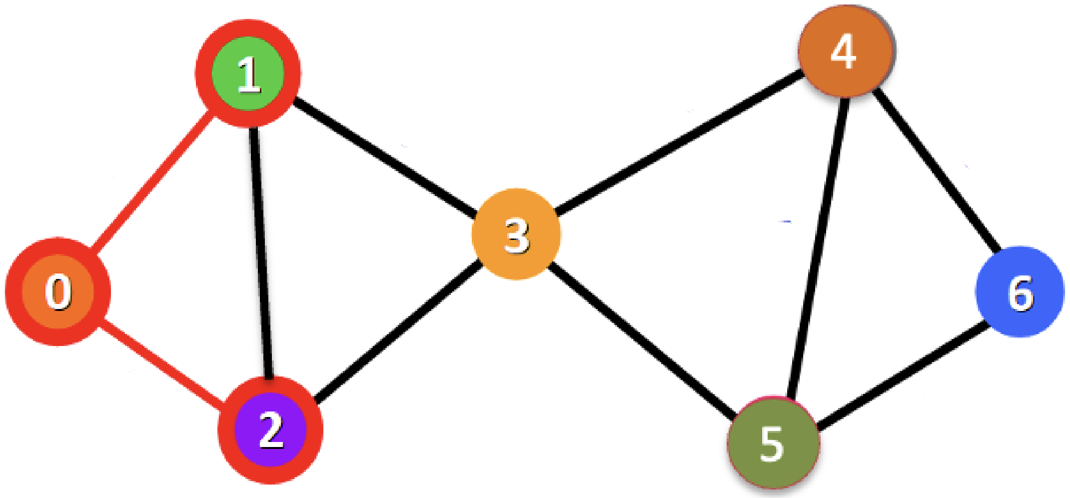
Graph Algorithms & Assignment 3

- SSSP (1S*D) and Dijkstra's Algorithm
- APSP (*S*D) with Floyd-Warshall Algorithm
- Uniform-Cost Search and A*

Lab:

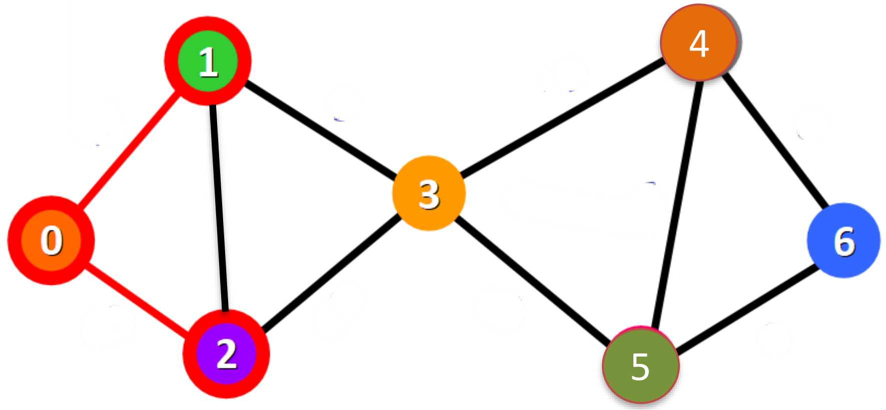
- Assignment 3: Q&A
- Implementing Floyd-Warshall Algorithm

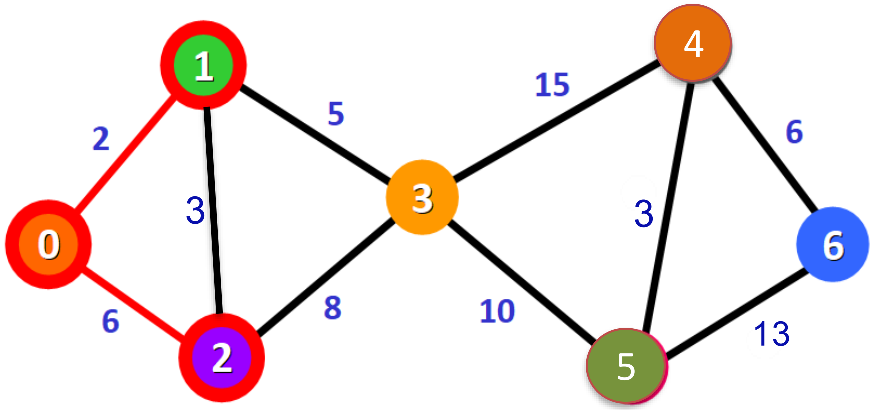
Review: Graph Search So Far



- BFS and DFS both explore all nodes reachable from a source, but *only BFS guarantees the shortest paths* in unweighted graphs.

shortest path in unweighted graphs vs weighted graphs





Task	*Find the shortest paths (SP) from 0 to all other vertices*	
a SP from 0 to 6	0 → 1 → 3 → 4 → 6 length = 4	0 → 1 → 3 → 5 → 4 → 6 length = 24
How?	BFS(0) Using a queue to explore nodes in order of their *depth from the root*. ➔ Nodes enqueued first are dequeued (visited) first.	Dijkstra(0) Using a min priority queue to explore nodes in order of their *distance from the root*. ➔ Nodes with smaller tentative distances are dequeued (visited) first.
Example		

Using Dijkstra's Algorithm for the SSSP on weighted graphs

Dijkstra's Algorithm from s

```
set dist[v]= ∞, prev[v]=-1 for each v
set dist[s]= 0
set PQ= min PQ of all (v, dist[v])
while (PQ not empty)
  u= deleteMin(PQ)      // found SP to u
  for each neighbour v of u
    if (dist[u]+w(u,v)<dist[v]):
      update dist[v], pred[v], PQ
```

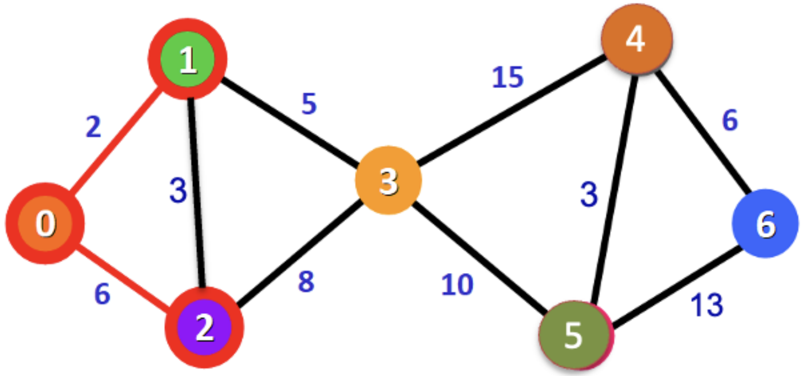
Start with

dist[] = { 0, ∞, ∞, ∞, ∞, ∞, ∞}

prev[] = {-1, -1, -1, -1,-1,-1,-1,-1};

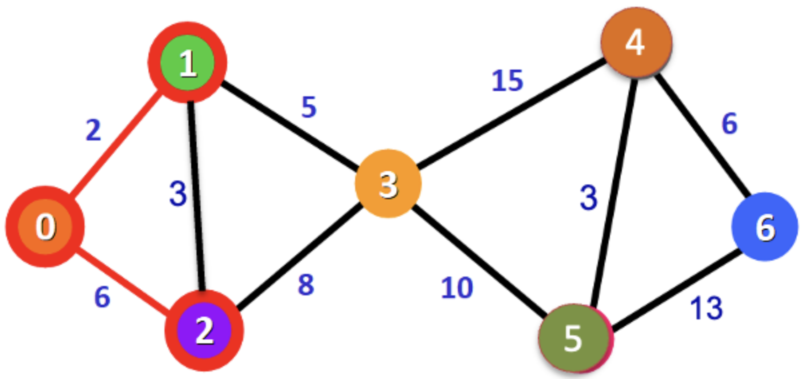
PQ = { (0,**0**), (1,**∞**), (2, **∞**), (3, **∞**), (4, **∞**), (5, **∞**), (6, **∞**) }

loop while PQ is not empty:
- u= node removed from PQ (having smallest dist)
- for each v in the adjacency list of u
  - if a shorter path is found:
    - update dist[v],
    - set prev[v]= u,
    - change the priority of v in PQ to new dist[v]



Note:
At the end, prev[] is used to reconstruct the shortest path by backtracking from the destination node to the source.

Example: tracing Dijkstra's Algorithm and Interpreting Its Outputs



	0	1	2	3	4	5	6
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
0							

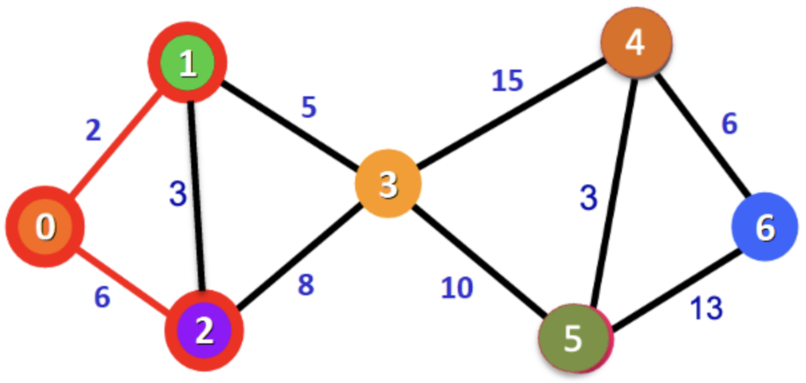
current content of PQ (with 7 elements)

Removed from PQ, shortest path found

dist[1]= ∞: tentative distance from 0

prev[4]= nil (or -1): node that precedes 4 in the tentative path 0→4

Example: tracing Dijkstra's Algorithm and Interpreting Its Outputs



	0	1	2	3	4	5	6
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
0		**2,0**	6,0	∞,nil	∞,nil	∞,nil	∞,nil
1			**5,1**	7,1	∞,nil	∞,nil	∞,nil
2				**7,1**	∞,nil	∞,nil	∞,nil
3					22,3	**17,3**	∞,nil
5					**20,5**		30,5
4							**26,4**
6							

Length of Shortest Paths:

- from 0 to 5 = ?
- from 0 to 6 = ?
- from 3 to 6 = ?

The actual SP:

- from 0 to 5 = ?
- from 0 to 6 = ?

Dijkstra's algorithm & Complexity when using Adjacency Lists

```
set dist[u]= ∞, prev[u]=nil for all u
set dist[s]= 0
set PQ= minPQ from all V with dist[] as priority
while (PQ not empty)
    u= deleteMin(PQ)
    visit  u
    for all (u,v) in G:
     if (dist[u]+w(u,v)<dist[v]):
         update dist[v] and prev[v]
         decrease priority of v in PQ
```

θ(V)

θ(V)

O(log V) x V steps = O(V logV)

using Adjacency Lists: O(log V) x E steps = O(E logV)

Programming note:

"**update dist[v] and pred[v]**": `dist[v]= dist[u]+w(u,v),prev[v]= u`

"**decrease priority of v in PQ**" includes:

- locate **v** in **PQ** (can be done in `O(1)`, but a bit tricky)
- change the priority of **v** to **dist[v]** then **upheap** (done in `O(logV)`)

Total Complexity if using adjacency matrix:

- *count complexity for each node and edge (as above) , and add up* ➔ ***O((V+E) logV)***

Dijkstra's Algorithm: Notes

Complexity:

- O((V+E) log V) if using *adjacency lists*
  → O(V log V) for sparse graphs, $O(V^2 \log V)$ for dense graphs

- $O((V^2) \log V)$ if using *adjacency matrix*
  → $O(V^2 \log V)$ for *all* graphs

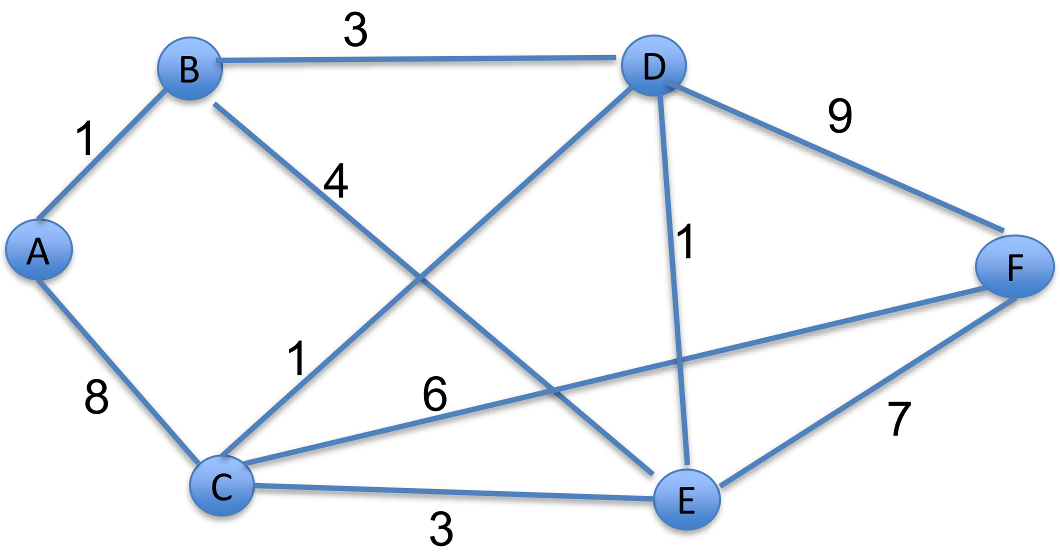
Conditions:

- all weights must be *non-negative*
- graphs can be weighted/unweighted(ie. all edges have weight 1), directed/undirected, cyclic/acyclic

If there are negative weights (but with no negative cycle):

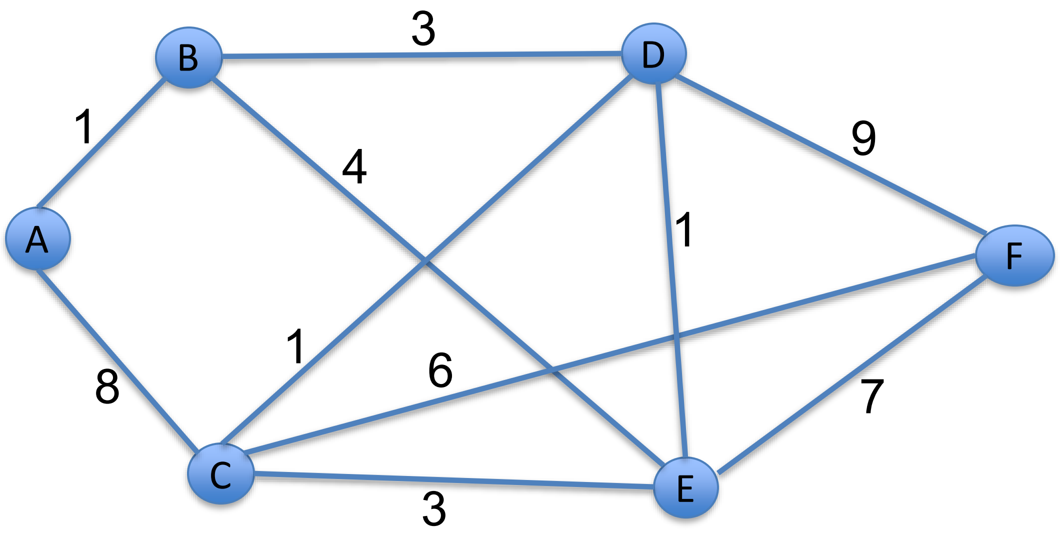
- Dijkstra's Algorithm is not applicable
- use Bellman-Ford algorithm instead (this algorithm has the same complexity)

W11.4: run Dijkstra's Algorithm for this graph



Find a shortest path:
- From A to B
- From A to C
- From A to F
- From A to any other node

Tracing Dijkstra's Algorithm

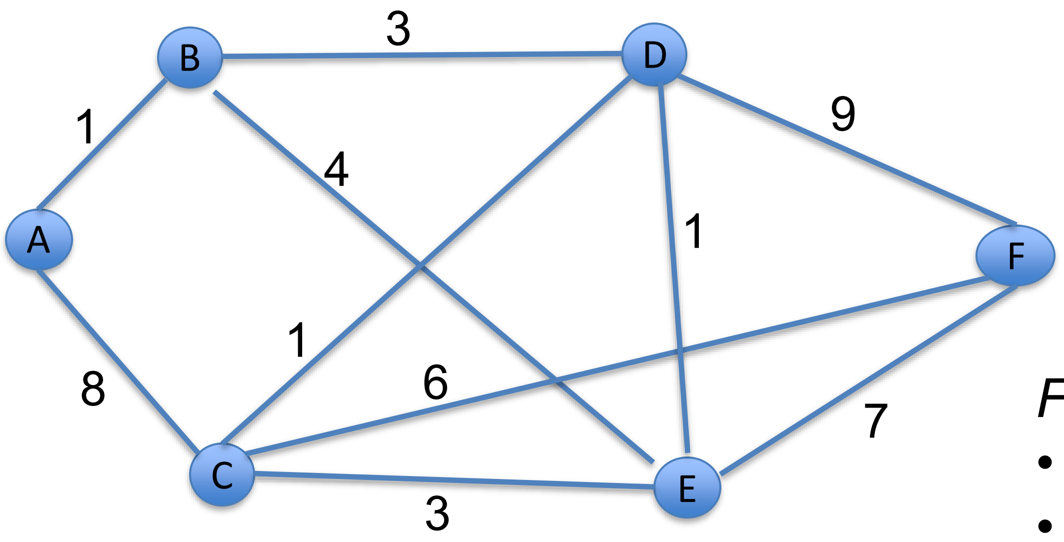


Removed from PQ, shortest path found

dist[3], prev[3]
ie. dist("D"), prev("D")

	A	**B**	**C**	**D**	**E**	**F**
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
A						

Dijkstra's Algorithm: tracing

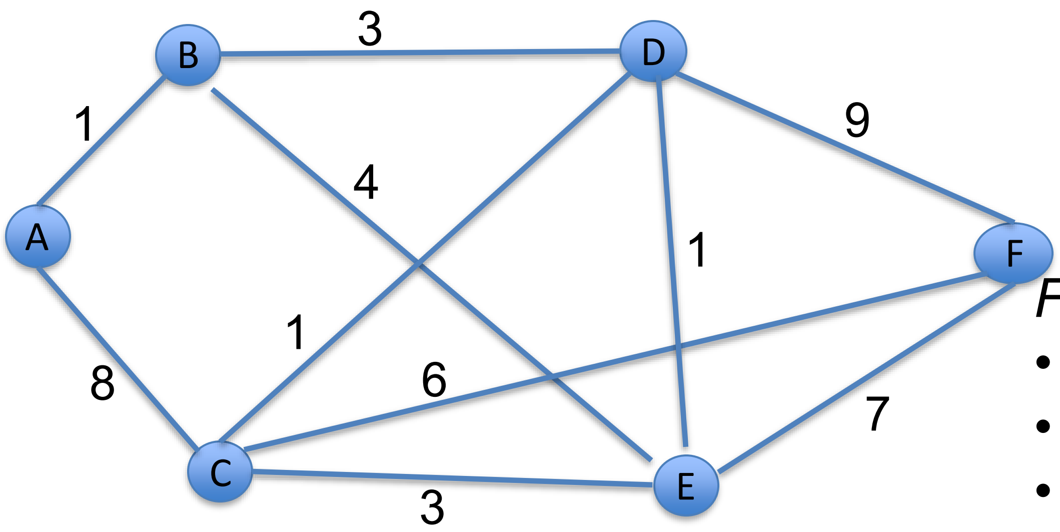


Find a shortest path:

- From A to B
- From A to C
- From A to F
- From A to any other node

	A	B	C	D	E	F
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
A		**1,A**	8,A	∞,nil	∞,nil	∞,nil
B			8,A	**4,B**	5,B	∞,nil
D						

Dijkstra's Algorithm: full tracing



Find a shortest path:

- From A to B
- From A to C
- From A to F
- SP A->F=

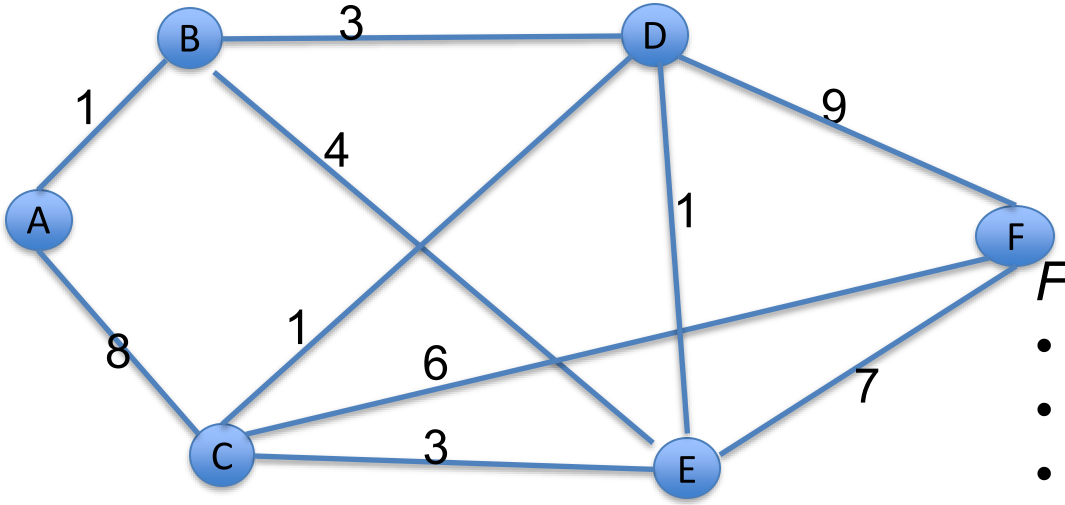
The dist at A is 0, there is an edge A->C with length 8, so we can reach C from A with distance 0+8, and 8 is better than previously-found distance of ∞

done	A	B	C	D	E	F
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
A		**1,A**	8,A	∞,nil	∞,nil	∞,nil
B			8,A	**4,B**	5,B	∞,nil
D			5,D		5,B	13,D
C					**5,B**	11,C
E						**11,C**
C						

Update this cell because now we can reach C from D with distance 4 (of D) + 1 (of edge D→C), and 5 is **better** than 8

At this pointy, we can reach E from D with distance 4 (of D) + 1 (of edge D→E), but new distance 5 is **not better** than the previously found 5, so no update!

Dijkstra's Algorithm: Interpreting the result



Find a shortest path:

- From A to B
- From A to C
- From A to F
- SP A->F=

What's the found shortest path from A to F?
distance= 11, path=A→B→D→ C→F

done	A	B	C	D	E	F
	0, nil	∞,nil	∞,nil	∞,nil	∞,nil	∞,nil
A		**1,A**	8,A	∞,nil	∞,nil	∞,nil
B			8,A	**4,B**	5,B	∞,nil
D			**5,D**		5,B	13,D
C					**5,B**	11,C
E						**11,C**
C						

pred[B]= A:
A→B→D→ C→F

pred[D]= B:
B→D→ C→F

pred[C]= D:
D→ C→F

pred[F]= C, that is we came to F from C: C→F

the shortest distance from A to F is 11

Floyd-Warshall Algorithm – APSP (APSP == *S*D)

The Task:

- Given a weighted graph G=(V,E,w(E))
- Find shortest path (path with min weight) *between all pairs of vertices. (*S*D)*

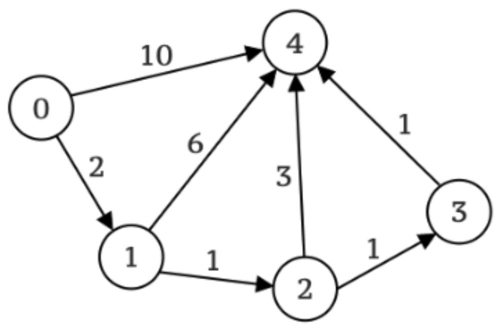
?:

- can we use Dijkstra's Algorithm for the task?
- why Floyd-Warshall's ?

Floyd-Warshall Algorithm

use `dist=` adjacency matrix of `G`, for the initial shortest path length

$$
\texttt{dist[s][t]} = \begin{cases} \texttt{w(s,t)} & \text{if there is and edge from } \texttt{s} \rightarrow \texttt{t} \\ \texttt{0} & \text{if } \texttt{s==t} \\ \infty & \text{otherwise} \end{cases}
$$



	0	1	2	3	4
0		2			10
1			1		6
2				1	3
3					1
4					

example path from 0 to 4
At the start:
path from 0→4 has dist(0,4)= 10
Using node 1: found new paths
0→1→4 dist(0,4) == 8
0 → 1 → 2 dist(0,2) == 3
Using node 2:
. . .

IDEA: If we use node `i` as an intermediate stepstone, we can find some new paths by

```
for each pair (s,t) :
    if path s->i->t is shorter than path s->t   // found a shorter path
        update dist[s][t] with new path length
```

Using all possible `i` in that way to create all possible paths!

Algorithm:

```
for (i=0; i<V; i++)        // for each node i: try it as a stepstone for new paths
  for each pair (s,t)  // for (s=...) for (t=...
    if  (dist[s][i]+dist[i][t] < dist[s][t])    // if new path via i is shorter
        dist[s][t]= dist[s][i]+dist[i][t]       // ... take it!
```

Floyd-Warshall Algorithm

Main algorithm:

D= adjacency matrix

for (i=0; i<V; i++)

for (j=0; j<V; j++)

for (k=0; k<V; k++)

if (D[s]**[i]**+D**[i]**[t]<D[s][t])

D[s][t]= D[s]**[i]**+D**[i]**[t];

init: dist D = adjacency matrix

for each node i: try it as a stepstone for new paths

for each pair (s,t)

if s → **i** → t is shorter than current s→t

update path s→t

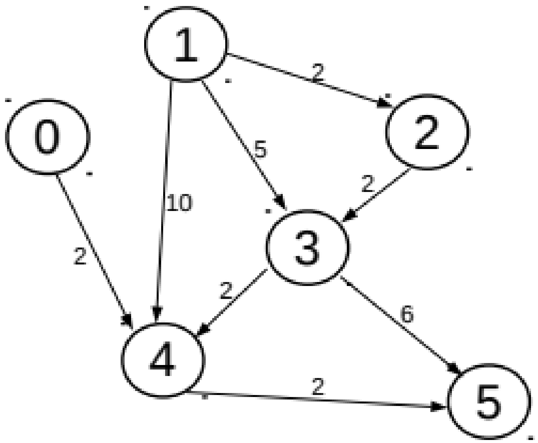
Conditions= ?

- directed or undirected
- weighted (for unweighted, set edge weight to 1)
- negative weights are OK, but **no negative cycles** (similar to the conditions for the Bellman-Ford Algorithm)

Data structures / Graph representation = `adjacency matrix` or `adjacency lists`? why?

Complexity = $\Theta(V^3)$

Step-by-step Example: Tracing FWA for a graph



Trace the Floyd-Warshall algorithm.
Step i= 0, 1, 2, 3, 4, 5

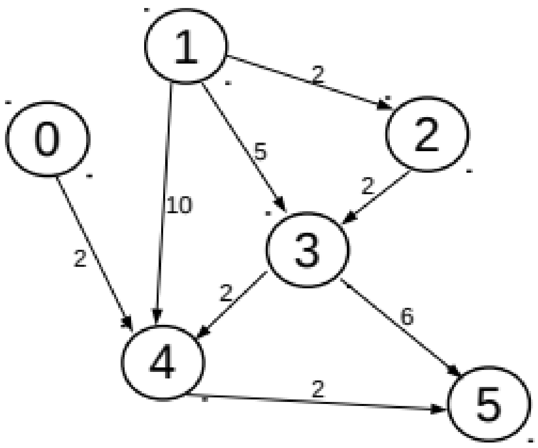
empty cell for ∞
(note `A[s][s]` should be zero)

--------------------TO (`t`) ----------------------

FROM (`s`)

	0	1	2	3	4	5
0	0				2	
1		0	2	5	10	
2			0	2		
3				0	2	6
4					0	2
5						0

Tracing FWA



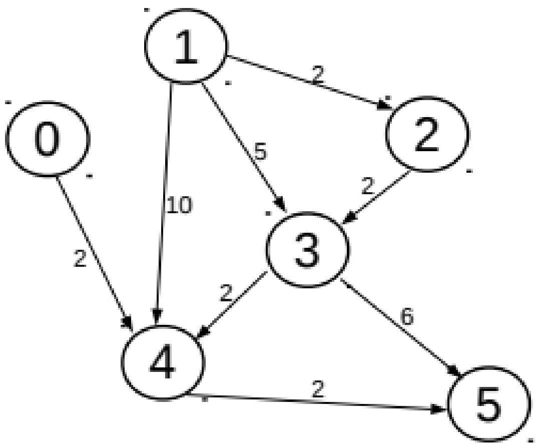
Notes:

- when 0, 1, or 5 is used as an intermediate, no change is possible (why?)

	0	1	2	3	4	5
0	0				2	
1		0	2	5	10	
2			0	2		
3				0	2	6
4					0	2
5						0

Tracing FWA

i= 2 as the stepstone: use rows 2 and column 2 as references



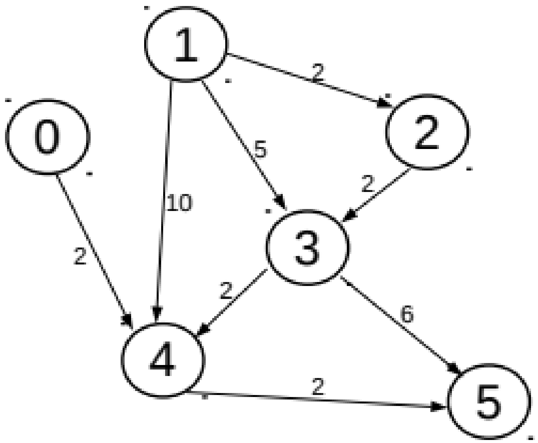
column 2 gives paths “from s to 2”

Only this cell need to be considered. Why?

row 2 gives paths “from 2 to t”

	0	1	2	3	4	5
0	0				2	
1		0	2		10	
2			0	2		
3				0	2	6
4					0	2
5						0

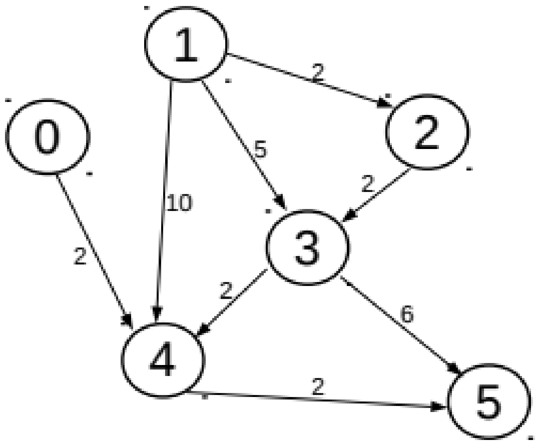
Tracing FWA



i= 2 as the stepstone

	0	1	2	3	4	5
0	0				2	
1		0	2	4	10	
2			0	2		
3				0	2	6
4					0	2
5						0

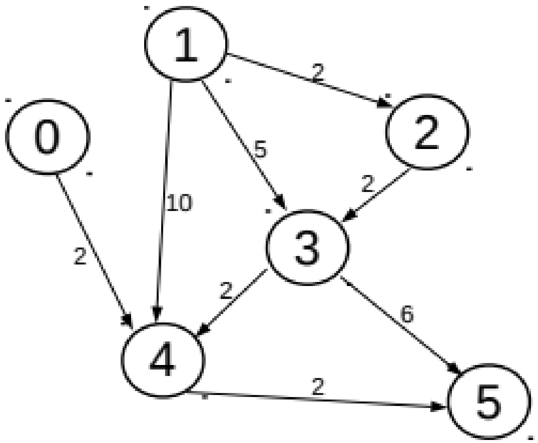
Tracing FWA



i= 3 as the stepstone

	0	1	2	3	4	5
0	0				2	
1		0	2	4	10	
2			0	2		
3				0	2	6
4					0	2
5						0

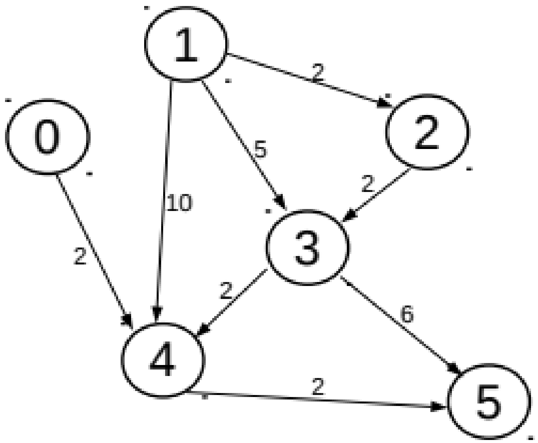
Tracing FWA



i= 4 as the stepstone

	0	1	2	3	4	5
0	0				2	
1		0	2	4	6	10
2			0	2	4	8
3				0	2	6
4					0	2
5						0

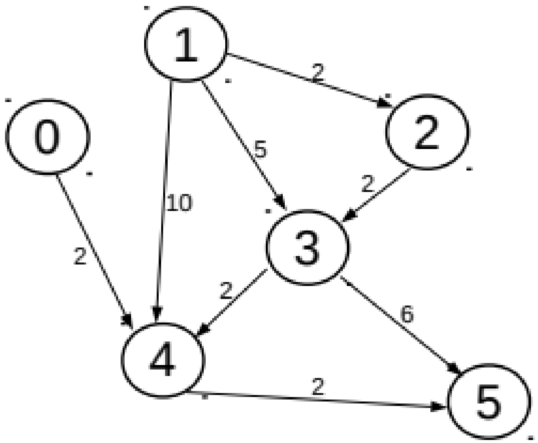
Tracing FWA



i= 4 as the stepstone, done

	0	1	2	3	4	5
0	0				2	4
1		0	2	4	6	8
2			0	2	4	6
3				0	2	4
4					0	2
5						0

Tracing FWA



i= **4** as the stepstone, done

	0	1	2	3	4	5
0	0				2	4
1		0	2	4	6	8
2			0	2	4	6
3				0	2	4
4					0	2
5						0

Floyd-Warshall Algorithm: How to retrieve the path between s-->t ?

in addition to matrix `dist[s][t]=` shortest path length from `s` to `t`, also maintain

matrix `next[s][t]=` the choice made for the pair `(s,t)`

`=` the first stop on the path `s→t`

at the end, use `next[][]` to track the shortest path for any desirable pair `(s,t)`

Algorithm:

```
for each node i in V:    // for (i=0; i<V; i++): use all nodes as stepstones!
  for each pair (s,t):  // for (s=...) for (t=...
    if  (dist[s][i]+dist[i][t] < dist[s][t]){  // if new path is shorter
         dist[s][t]= dist[s][i]+dist[i][t]       // ... take it, update dist
         next[s][t]= i                          // ... and update next
    }
```

Peer Activity: All Pairs Shortest Paths

What graph conditions justify running Dijkstra's algorithm on each vertex over using Floyd-Warshall's algorithm?

a. None: Floyd-Warshall > repeated Dijkstra's for all graphs
b. Sparse graph ($|E| \approx |V|$)
c. Dense graph ($|E| \approx |V|^2$)

Do Peer Activities then fill in Big-O complexity for the APSP task
(supposing to apply Dijkstra for the APSP task)

	Dijkstra	Floyd-Warshall
General		$\Theta(V^3)$
Sparse		
Dense		

UCS: Uniform-Cost Search

- Typically used for AI, when having implicit graphs
- The Task: 1S1D – finding shortest path from the root to (any) winning node
- Can be done with modified Dijkstra's (edge weight==1) or just BFS algorithm:
  - first enqueue only the initial state (the source node, instead of all nodes)
  - when exploring a node after dequeuing it:
    - check if it's a Destination (a winning node), exit if yes
    - enqueue all unseen new neighbours (ie. new states that can get from the current state)

- Note: In the case of typical AI search, the graph is a DAG → **no** need to **check for** being **visited**

Example UCS for AI

start is the initial state

- Q contains only the initial node n at the start

if (n is a winning state):
break with SUCCESS

make a child node newNode

enqueue newNode if valid

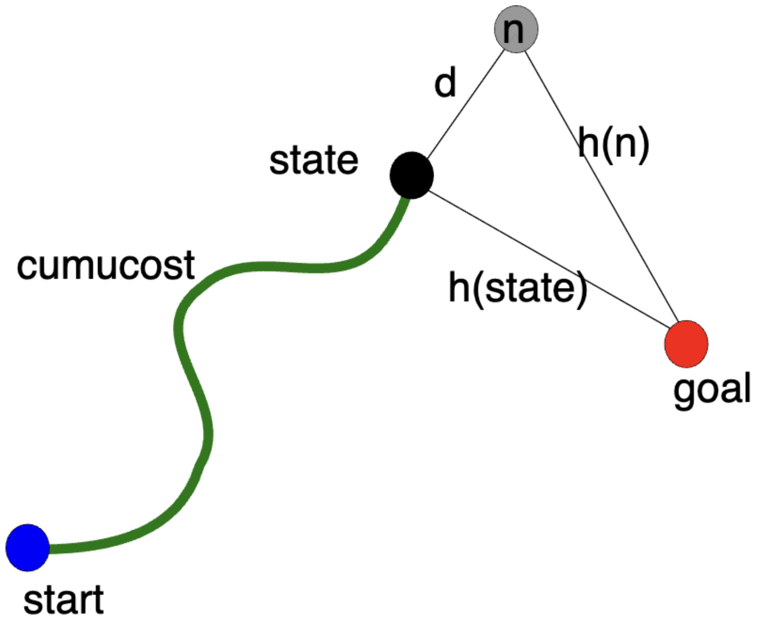
Algorithm 1 AI Impassable Gate Algorithm

```
procedure FindSolution(start, showSolution)
    n ← CreateInitNode(start)
    numPieces ← GetNumberOfPieces(start)
    Enqueue(n)
    while queue ≠ empty do
        n ← queue.Dequeue
        exploredNodes ← exploredNodes + 1
        if WinningCondition(n) then
            solution ← SaveSolution(n)
            solutionSize ← n.depth
            break
        end if
        for each move action a ∈ {up, down, left, right} × {0, . . . , num
            pieceMoved ← ApplyAction(n, newNode, a)
            generatedNodes ← generatedNodes + 1
            if pieceMoved is false then
                Free(newNode)
                continue
            end if
            queue.Enqueue(newNode)
        end for
    end while
end procedure
```

at the end: delete all nodes (inside and outside PQ)

A* search for 1S1D

A* is UCS guided by a heuristic: it picks the node with the cheapest path so far plus a smart guess to the goal.



Foundation: A* is basically Dijkstra's algorithm, but enhanced with a heuristic guide.

Node Selection: A* expands the node with the smallest total estimated cost, f(n), calculated as:

$$f(n)= g(n)+h(n)$$

g(n): cost from start to current node (known cumucost).

h(n): estimated cost from current node to goal (heuristic).

Optimality Guarantee: For A* to find the optimal path, the heuristic h(n) must be ***admissible*** (it must **never overestimate the true cost to the goal**).

Complexity: In the worst case, its time complexity is the same as Dijkstra's (O(E+VlogV)), but it is often much faster due to the targeted search guided by the heuristic.

A3: understanding, Q&A

Key points:

- Optimization in Algorithm 2 is interesting and simple, do it
- Algorithm 3 is very interesting, but more complicated. Why not try it after finish Algorithm 2?
- Keeping tracks of all malloc for free-ing later
- Smartly using queue
- Stuffs to do with a node *after* dequeuing it

Spend reasonable time for answering the questions.
Try to build informative graphs!

Additional Notes

- Attend Week 12 lecture: A* search, Computational Complexity, P & NP (probably unexaminable, but very interesting!)
- Check out https://clementmihailescu.github.io/Pathfinding-Visualizer/ - a pretty interactable visualization tool for all the traversals and searches.

  The above is a fork of the project:

  https://qiao.github.io/PathFinding.js/visual/

Next Workshop:

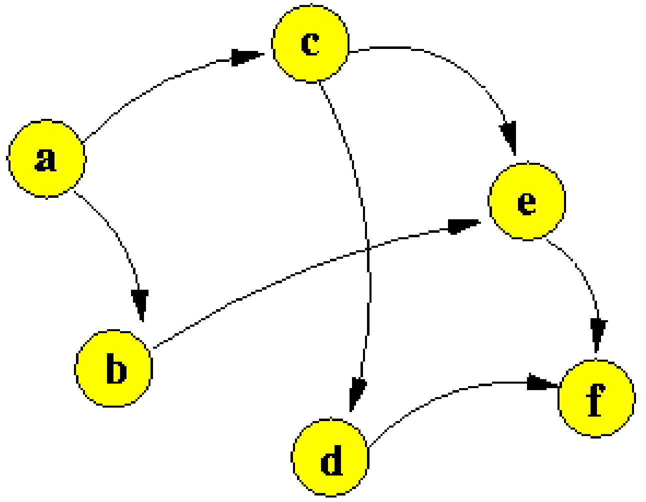
- Review: past years' exam papers
- Perhaps finish A3 before that?

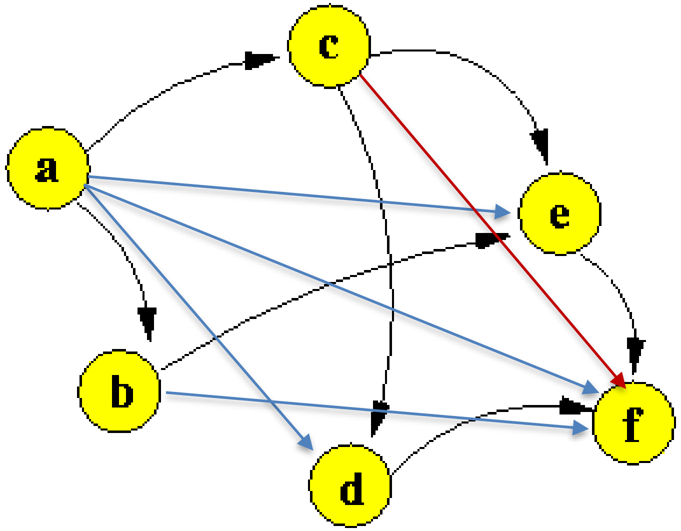
Additional Slides

Transitive Closure of digraphs

Transitive Closure of a di-graph G:

- is a graph G′ where there is a link from i→j if there is a path from i→j in G.
- has an adjacency matrix A where $A_{ij}=1$ iif j is reachable from i in G.





Related Tasks:

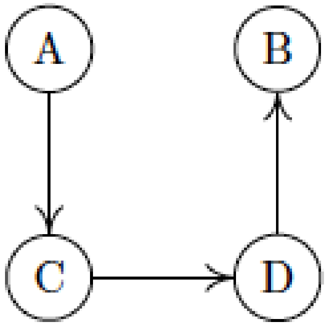
Compute the transitive closure for digraph

Find APSP for a weighted graph

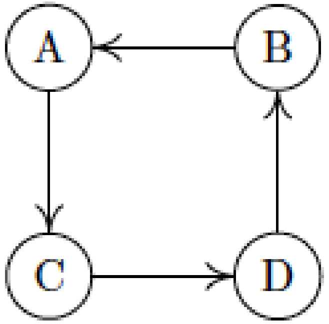
Examples: Transitive Closure of digraphs

Draw the transitive closure of the following two graphs:

(a)



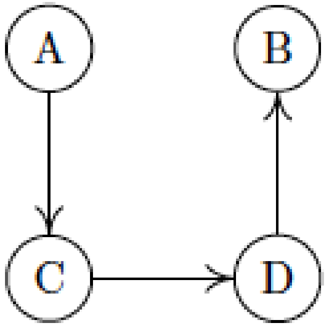
(b)



Warshall's Algorithm for Transitive Closure: very similar to Floyd-Warshall's

Input: adjacent matrix A

Main argument: transitiveness: if there are paths i→k and k→j, then there is path i→j which uses k as an intermediate stepstone.



FROM \ TO	A	B	C	D
A				
B				
C				
D				

Warshall Algorithm

```
for (i=0; i<V; i++)
  // using i as intermediate
  for (s=0; s<V; s++)
    for (t=0; t<V; t++)
      if (A_si && A_it)
         A_st= 1;
```

Note: The Warshall's algorithm is a simplified version of FWA for using with unweighted digraphs. Tracing is similar, but simpler. DIY with the above graph.

Graph Search: some interesting tasks

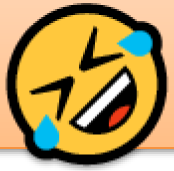
- Find a longest path in a weighted (acyclic) graph
- Find an Euler cycle
- Find a Hamiltonian cycle
- …

An **Euler trail** is a way to pass through every edge exactly once. If it ends at the initial vertex then it is an *Euler cycle*. Note that an Euler trail might pass through a vertex more than once.

A **Hamiltonian path** is a path that passes through every vertex exactly once (NOT every edge). If it ends at the initial vertex then it is a *Hamiltonian cycle*. Note that a Hamiltonian path may not pass through all edges.

Q: Why an Euler trail, but not a Hamiltonian path, must pass through every edges exactly once?

A: Because Euler and Edge share the same starting letter E



P, NP, NP: Naïve & Intuitive Understanding

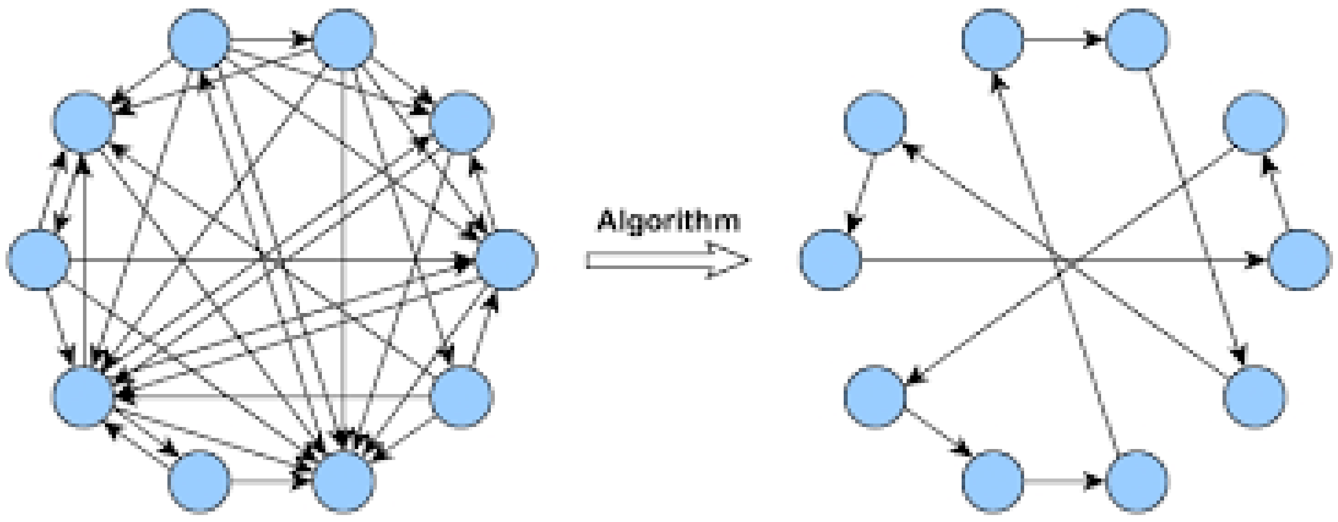
Decision Problem: Problem with YES/NO answer. A decision problem is:

- P: iif it can be **solved** in *polynomial time* by an algorithm.
- NP: iif the 'yes'-answers can be ***verified*** in polynomial time ($O(n^k)$ where *n* is the problem size, and *k* is a constant).
- NP-Complete: iif it's NP and, so far, there is no polynomial time algorithm for solving.

Notes:

- Some of the above concepts are for intuitive understanding, and are not the definitions.
- For more on P, NP and related interesting theoretical topics: attend live lecture W12.

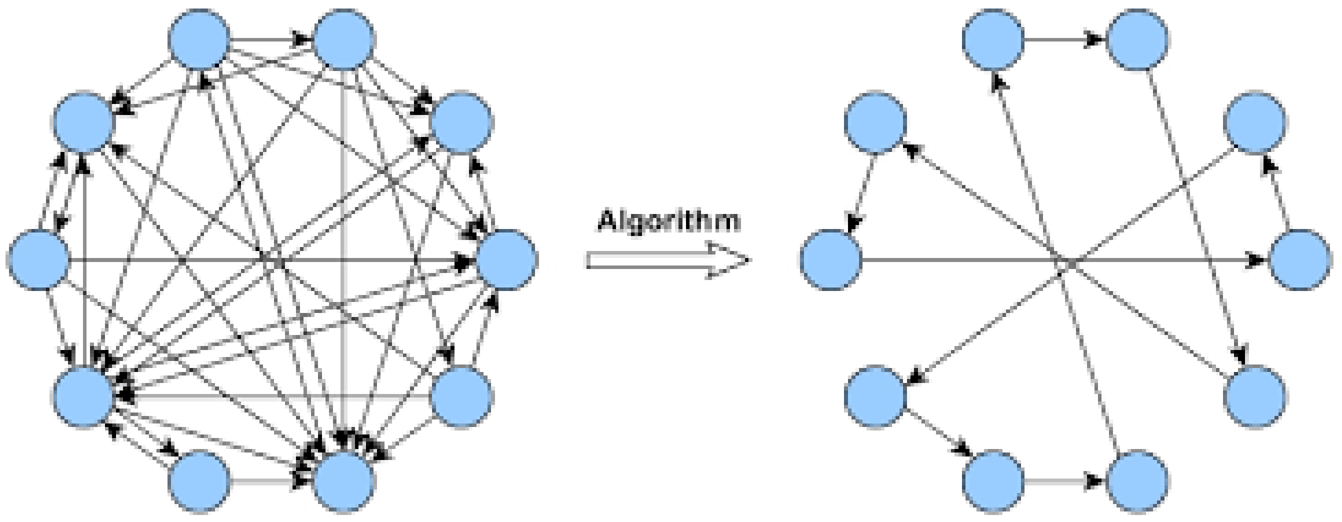
Graph Search can be a NP-complete task: Hamiltonian cycle



Can we just run DFS or BFS? The complexity would be O(V+E), right?

Picture from: https://archive.codeplex.com/?p=findinghamiltoniancycle

Graph Search can be a NP task: Hamiltonian cycle



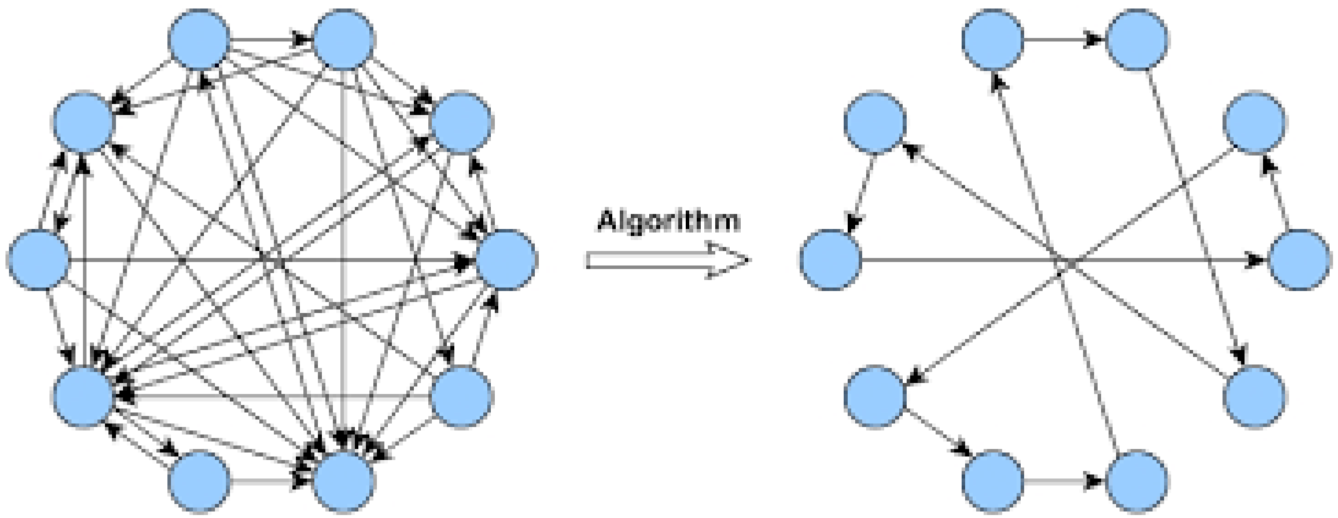
Yes, we can do the path finding in DFS or BFS manner, but the complexity is no longer O(V+E). *Why?*

The complexity of this task is O(?)

The task belongs to the NP-Complete class.

Picture from: https://archive.codeplex.com/?p=findinghamiltoniancycle

Graph Search can be a NP task: Hamiltonian cycle



Yes, we can do the path finding in DFS or BFS manner, but the complexity is no longer O(V+E). *Why?*

The complexity of this task is O(?)

The task belongs to the NP-Complete class. *What's that?*

Picture from: https://archive.codeplex.com/?p=findinghamiltoniancycle

Compare Dijkstra’s Algorithm with the UCS for AI

Note: The UCS used in Assignment 3 is BFS, not Dijkstra’s. The next 2 slides is for understanding only, and should not be used for Assignment 3.

Dijkstra(G=(V,E,W),s)

```
set dist[0..V-1]= ∞
    pred[0..V-1]= nil
set dist[s]= 0
set PQ= makePQ(V,dist)
while (PQ not empty) {
  u= deleteMin(PQ)

  for all (u,v) in G {
    if (dist[u]+w(u,v) < dist[v]){
       update dist[v], pred[v]
       decrease weight of v in
               PQ to dist[v]
    }
  }
}
```

// note: BFS can be used instead!

How to Modify for AI games?

- no such arrays
- dist/cost and prev/parent should be kept in node

- PQ contains only s at the start

if (u is winning): break with SUCCESS

// discover all possible children of u

for each such child:

  node= make new node for the child,
     with updating cost and parent

  if (node seen or can be eliminated):
     delete node & continue

  enPQ(node)

at the end: delete all nodes (inside and outside PQ)

Dijkstra(G=(V,E,W),s) — Modified(G, s): UCS for AI == Algorithm 1

```
set dist[0..V-1]= ∞
    pred[0..V-1]= nil
set dist[s]= 0

set PQ= makePQ(V,dist)

while (PQ not empty) {
  u= deleteMin(PQ)

  for all (u,v) in G {
    if (dist[u]+w(u,v)<dist[v]){
      update dist[v], pred[v]
      decrease weight of v in
               PQ to dist[v]
    }
  }
}
```

- no such arrays
- dist/cost and prev/parent should be kept in node

- PQ contains only s at the start

if (u is winning): break with SUCCESS

make a child node

enqueue if valid

Algorithm 1 AI Impassable Gate Algorithm

```
procedure FindSolution(start, showSolution)
    n ← CreateInitNode(start)
    numPieces ← GetNumberOfPieces(start)
    Enqueue(n)
    while queue ≠ empty do
        n ← queue.Dequeue
        exploredNodes ← exploredNodes + 1
        if WinningCondition(n) then
            solution ← SaveSolution(n)
            solutionSize ← n.depth
            break
        end if
        for each move action a ∈ {up, down, left, right} × {0, ..., num
            pieceMoved ← ApplyAction(n, newNode, a)
            generatedNodes ← generatedNodes + 1
            if pieceMoved is false then
                Free(newNode)
                continue
            end if
            queue.Enqueue(newNode)
        end for
    end while
end procedure
```

at the end: delete all nodes (inside and outside PQ)