COMP20003 Workshop Week 11

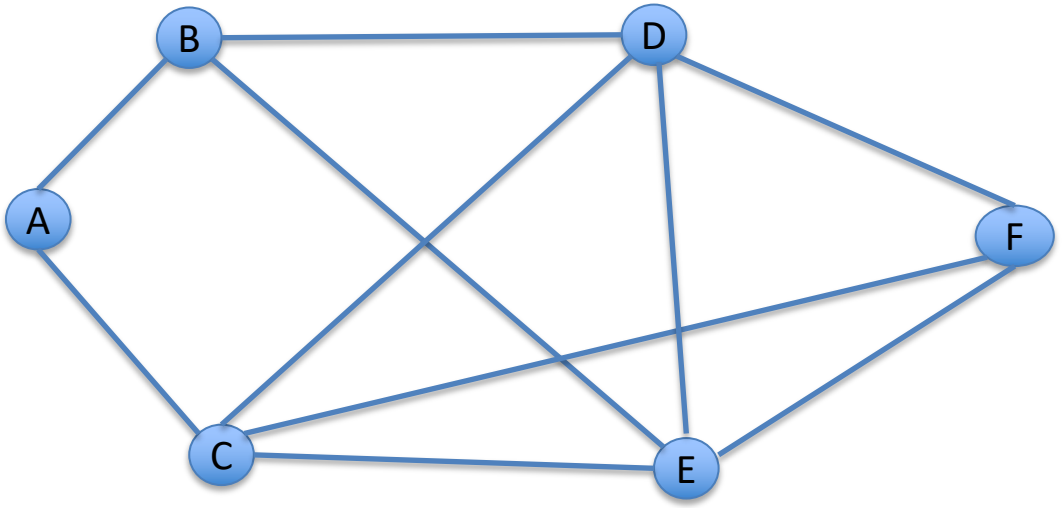
Graph Algorithms & Assignment 3

- SSSP and Dijkstra's Algorithm
- APSP with Floyd-Warshall Algorithm
- Uniform-Cost Search with the Dijkstra's Algorithm

Lab:

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

Review: find shortest paths from A to any other node in unweighted graph



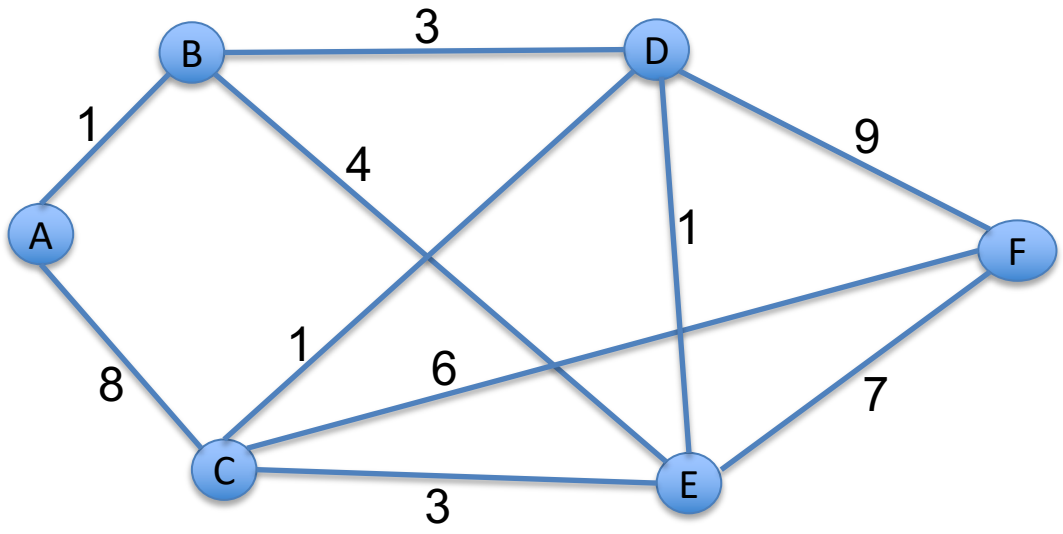
Input:

- *V= {0:A, 1:B, 2:C, 3:D, 4:E, 5:F}*
- *adjacency matrix or adjacency lists*

Output:

Algorithm:

Using Dijkstra's Algorithm for the SSSP on weighted graph



Dijkstra's Algorithm from `s`

```
set dist[v]= ∞, prev[v]=nil for each v
set dist[s]= 0
set PQ= makePQ(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
```

Input:

- *V= {0:A, 1:B, 2:C, 3:D, 4:E, 5:F}*
- *adjacency matrix or lists with weight w(u,v)*

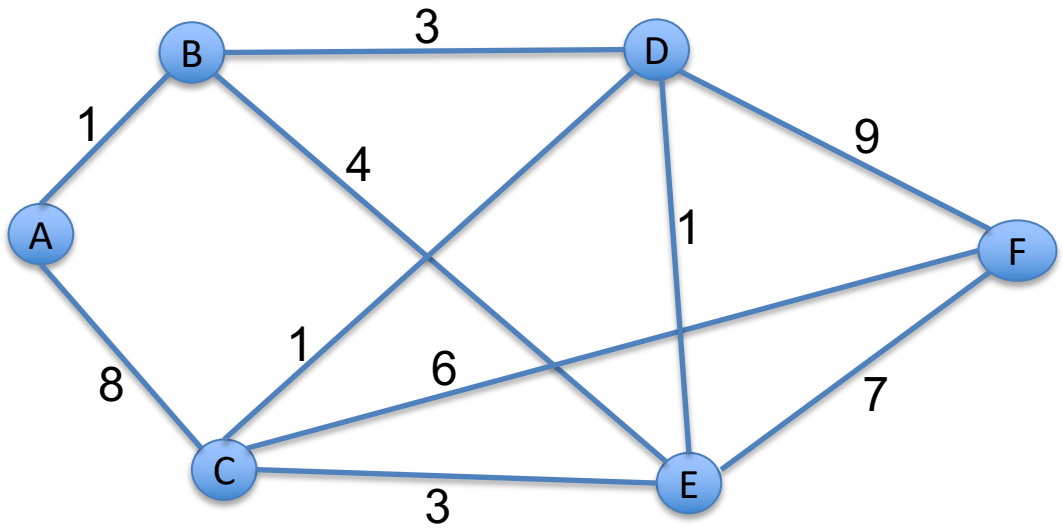
Output: shortest path from one source (A) to all others

- `dist[]= {0, 1, 5, 4, 5, 11}`
- `pred[]= {-, 0, 3, 1, 1, 2 }`

Algorithm: Dijkstra's

- *same as BFS, but using a min priority queue (PQ) instead of a queue (Q)*
- *the PQ use distance-found-so-far as the priority value*
- *at the start* `dist[]= {0, ∞, ∞, ∞, ∞}` *and these values are all pushed into the PQ*
- *loop:*
  - *remove the node with min dist from PQ*
  - *use edges from this node to update any of* `dist[]` *to a smaller value if applicable*

Tracing Dijkstra's Algorithm



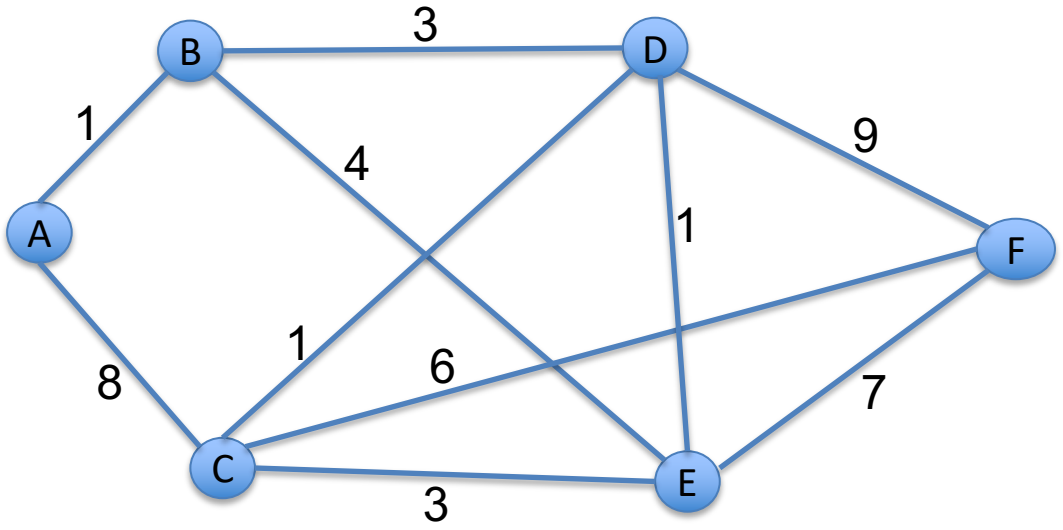
Dijkstra's Algorithm from `s`

```
set dist[v]= ∞, prev[v]=nil for each v
set dist[s]= 0
set PQ= makePQ(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
```

Removed from PQ, shortest path found

`dist[3], pred[3]`

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



Find a shortest path:

- From A to B, A to C, ...
- From A to 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

Dijkstra's Algorithm: Notes

Complexity:

- O((V+E) log V) if using adjacency lists

Condition:

- all weights must be non-negative
- can be used for weighted/unweighted, directed/undirected, cyclic/acyclic

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

- use Bellman-Ford algorithm

Floyd-Warshall Algorithm - APSP

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

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

IDEA: We use a particular node `i` as an intermediate stepstone in finding path from s to t:

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

Using all possible `i` means examining all possible paths!

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!
```

Floyd-Warshall Algorithm

Main algorithm:

```
initialize: dist= adjacency matrix
for each vertex i do
   for each pair (s,t) do
      if dist(s,i)+dist(i,t) < dist(s,t):
         update dist(s,t)
```

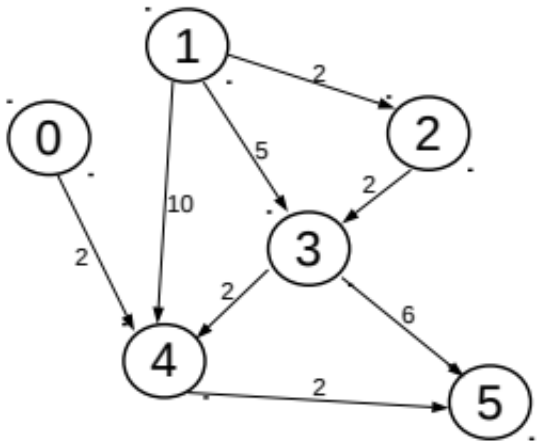
Conditions= ?

- directed or undirected
- weighted (for unweighted, set edge weight to 1)
- negative weights possible?

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

Complexity = O(?) or Θ (?)

Tracing FWA for a graph



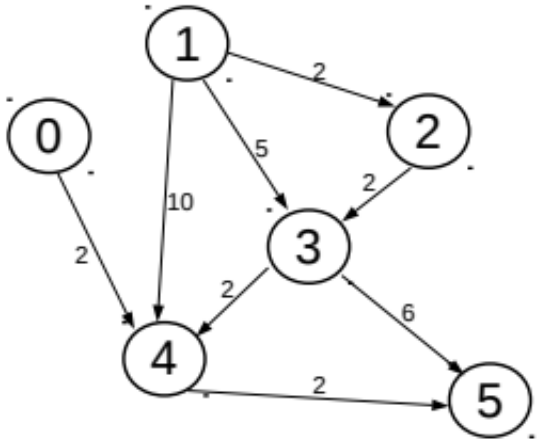
Trace the Floyd-Warshall algorithm.
Step i= 0, 1, 2, 3, 4, 5

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

Run FWA *manually*



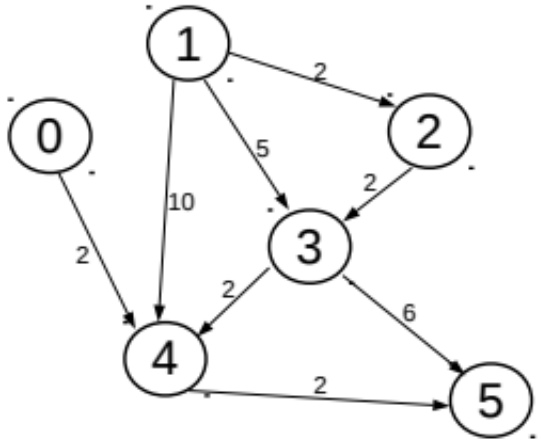
Notes:

- 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

Run FWA *manually*

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



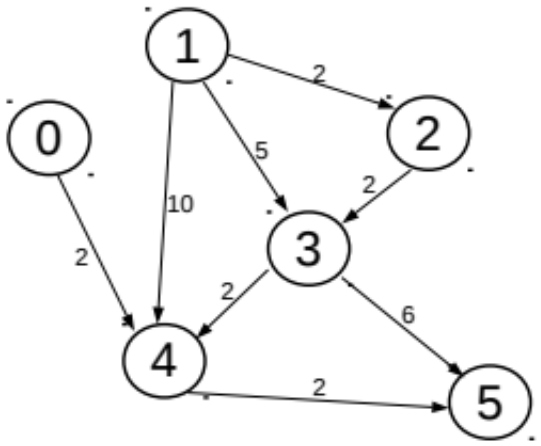
column **2** gives paths "from s to **2**"

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: last round



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 first stop after s on the path from `s` to `t`

$$\texttt{next[s][t]} = \begin{cases} \texttt{t} & \text{if there is and edge from } \texttt{s}\rightarrow\texttt{t} \\ \texttt{nil} & \text{otherwise} \end{cases}$$

Using all possible `i` means examining all possible paths!

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]= ???                        // ... and update next
     }
```

APSP – comparing Dijkstra & Floyd-Warshall: do Peer Activity

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		
Sparse		
Dense		

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 algorithm

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]
    }
  }
}
```

Modified(G, s) 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 neighbors of u

for each such neighbor:

  if reaching memory quota: break with FAILURE

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

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

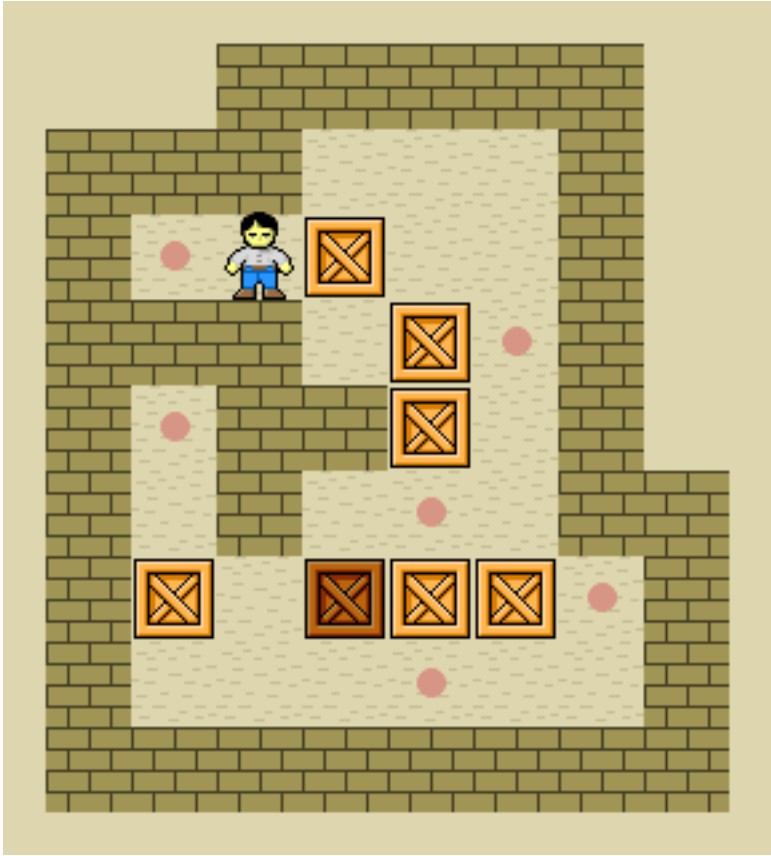
  enPQ(node)

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

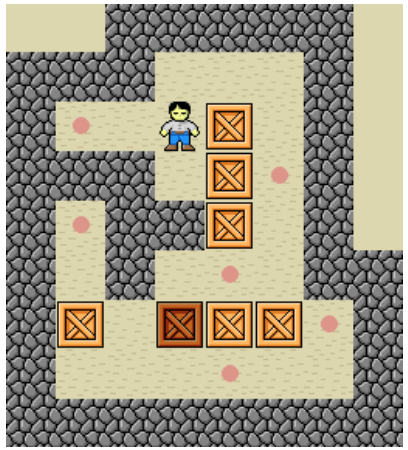
A3: The Sokoban

A process of playing a game can be viewed as a graph:

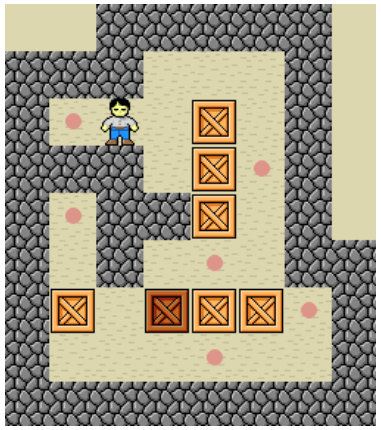
- a particular state of the game is a node
- a move or possible move is an edge, which transfer a node to a new node



Games & Implicit Graphs

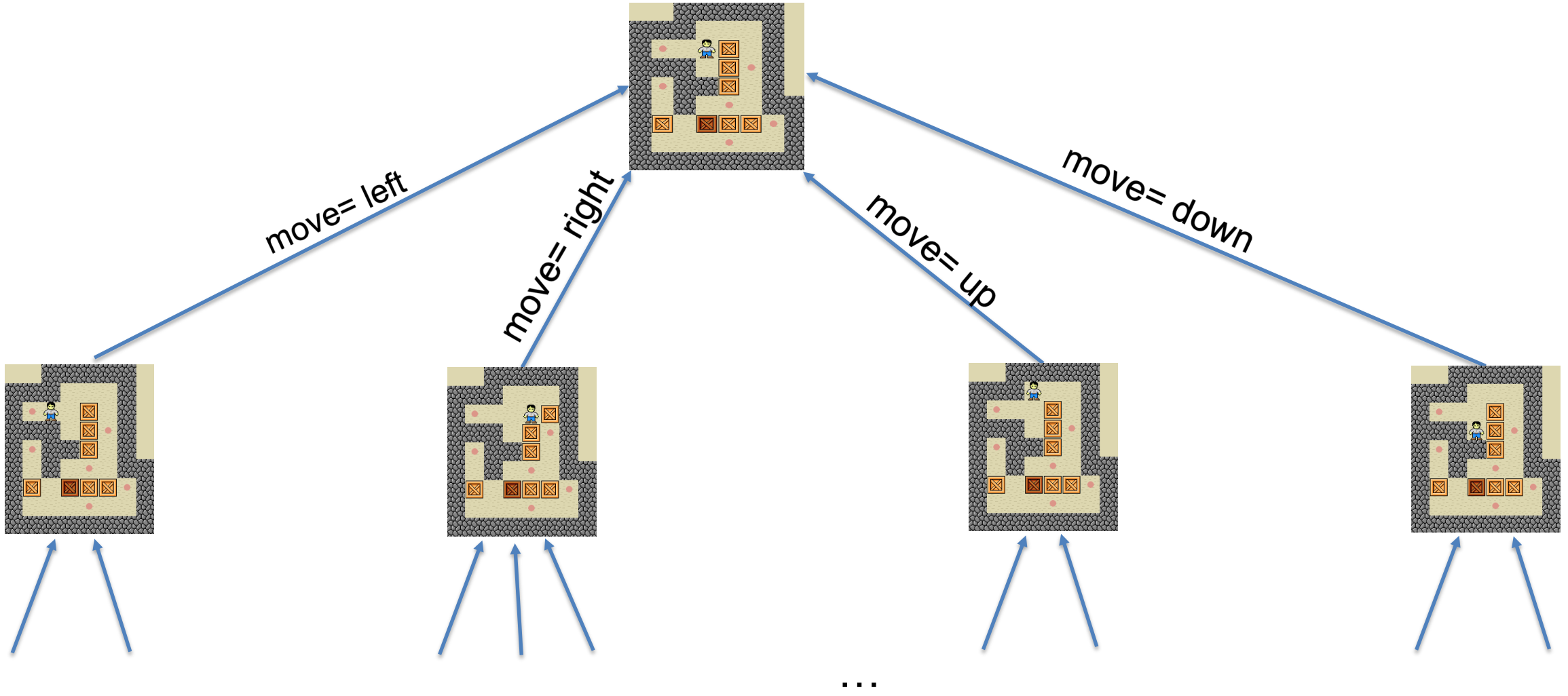


go left

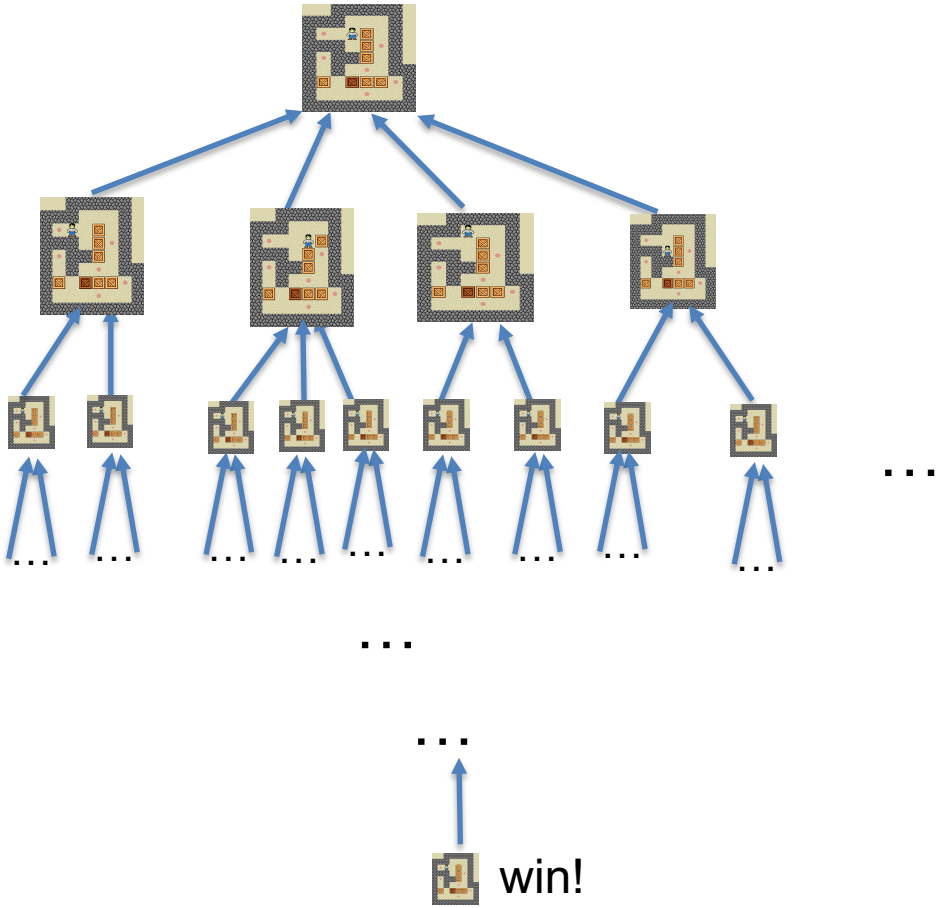


We represent the game implicitly as a graph:

- A particular configuration of the game board is called a *state*
- When a move is performed, the board goes from one state to another state
- So: **A state is a vertex, and a move is an edge of the graph**



We look at the state and generate all possible actions from that state, then turn into edges.
With implicit graphs we usually don't generate the whole graph.
We generate (children) nodes when exploring a (parent) node.
We just need to find a shortest path from the root to a winning node (that we know).



Challenges:

- how to efficiently store a state
- depth d of the search might be high
- number of nodes= ?
- Complexity= ?
- How to avoid generating a same state?
- How to keep the memory usage reasonable?
- How to keep track of the nodes and free them?
- ...

A3: understanding, Q&A

Understanding the marking rubric and expectations

Notes on Programming Styles (2 marks)

Some other notes for implementation:

- What's required?
- What's in a node, how to create a node
- How to get all possible moves
- How to control memory usage, incl. delete all nodes

What needed for experimentation, how to get data for it?

How to start the dead-end detection?

Command:

```
valgrind --suppressions=sokoban.supp --leak-check=full --show-reachable=yes
        ./sokoban -s test_maps/capability1 [play_solution]
```

Additional Notes

- Attend online Week 12 lecture: A* search, Computational Complexity, P & NP
- 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 no need for A3 problems?

Additional Slides

How to start the program of A3

To start the base code, run

```
make
./sokoban –h
```

Then, in the sub-directory `test_maps`, you can see the maps for testing.
If you want to play the game with, say, map `capacity12`, run:

```
./sokoban test_maps/capacity12
```

then it brings up the game, and you can play using the arrow keys to control your Sokoban.

If you want to use your ai code to play instead, run

```
./sokoban –s test_maps/capacity12 play_solution
```

but of course, that works only after you fill in the function `find_solution` in the file `src/ai/ai.c`
Your programming task in this assignment is, basically, fill in that function. Moreover, you need to make sure that that function won't create memory errors or leaks. Note that if you run the base code, there are already some memory problems. That's normal, because some parts of the function `find_solution` were already filled for you, and these parts might create some memory that you need to free later.

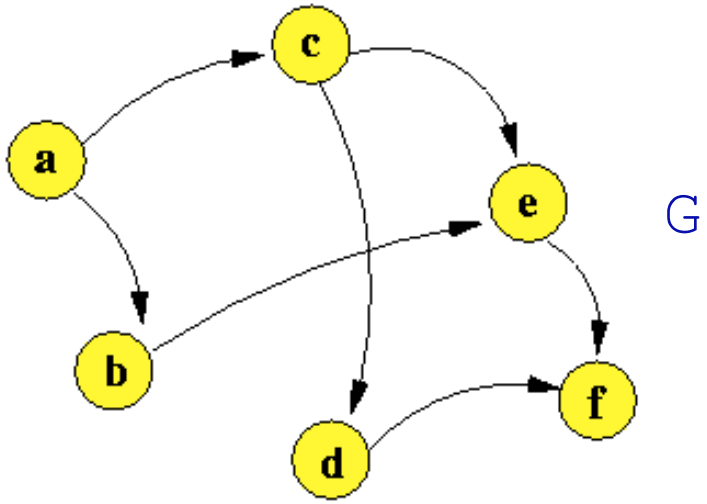
If you want to see the memory errors/leaks of you ai code, run

```
valgrind --suppressions=sokoban.supp --leak-check=full --show-reachable=yes
         ./sokoban –s test_maps/capability1
```

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.

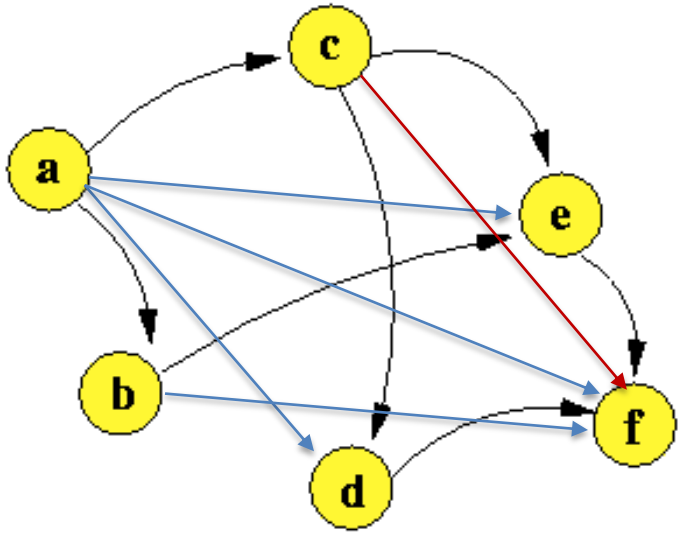


G

Related Tasks:

Compute the transitive closure for digraph

Find APSP for a weighted graph

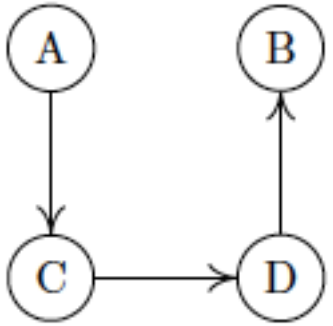


G′

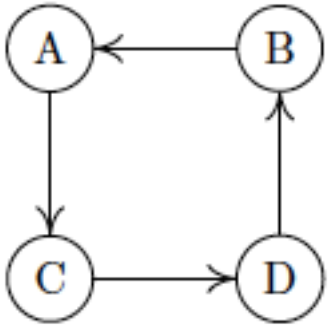
Examples: Transitive Closure of digraphs

Draw the transitive closure of the following two graphs:

(a)



(b)

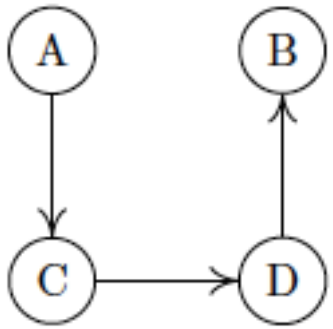


FROM \ TO	A	B	C	D
A				
B				
C				
D				

Warshall's Algorithm: for Transitive Closure

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.



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 (Asi && Ait)
         Ast= 1;
```

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: Currently Naïve & Intuitive Understanding

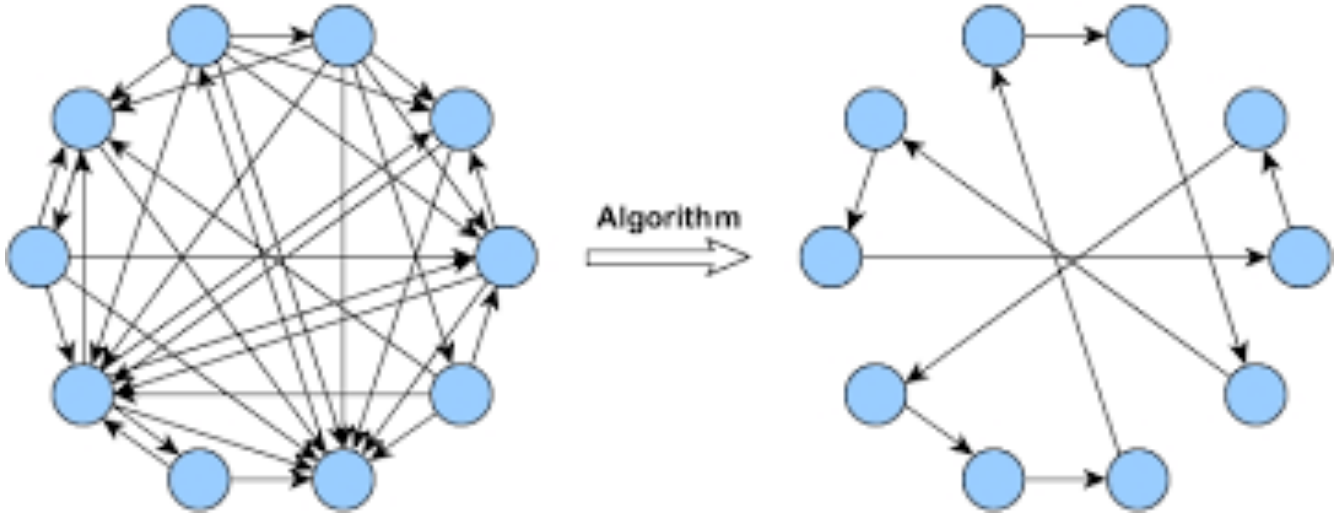
Decision Problem: Problem with YES/NO answer. A decision problem is:

- 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:

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

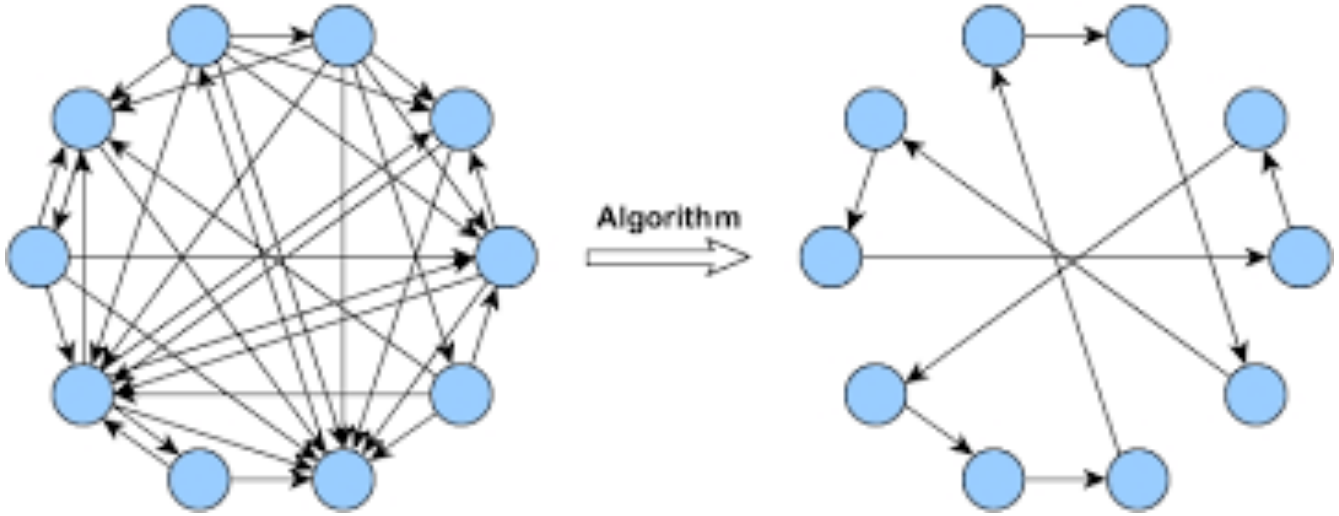
Graph Search can be a NP-complete task: Hamiltonian cycle



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



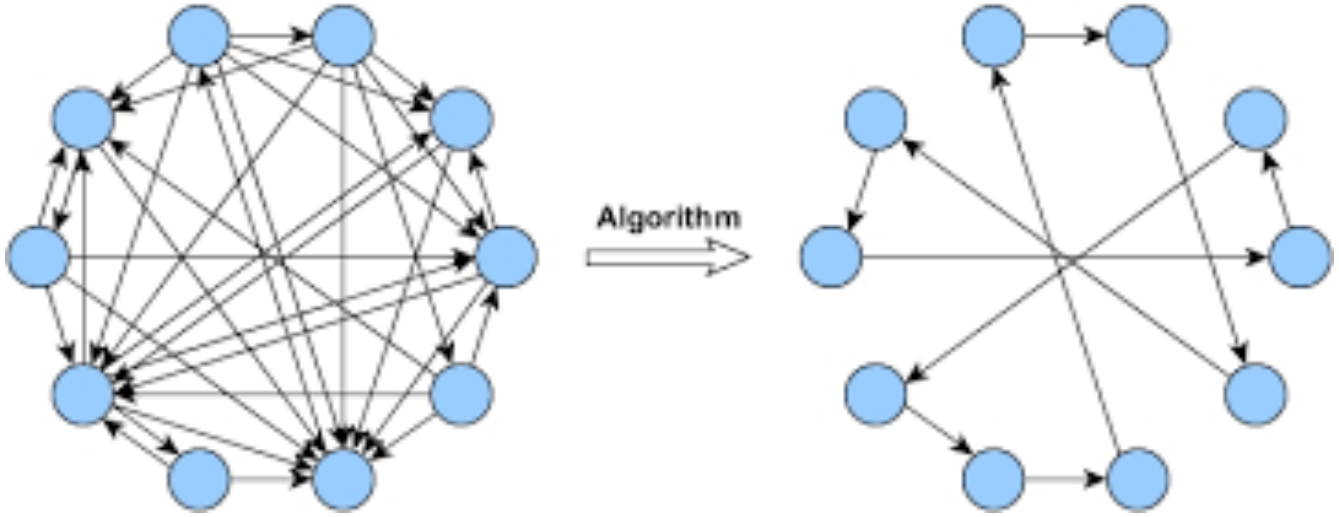
Yes, we can do the path finding in DFS or BFS manner, but the complexity is no longer O(V+E). *Why?*

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?*

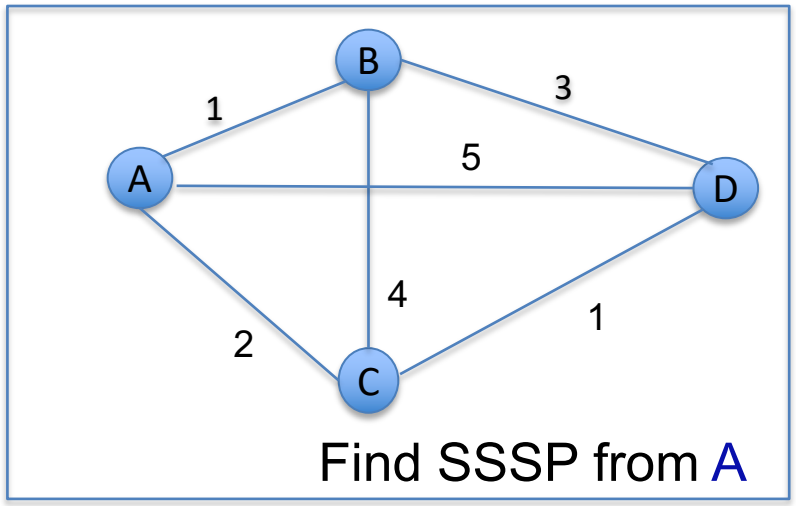
Picture from: https://archive.codeplex.com/?p=findinghamiltoniancycle

Example of Dijkstra's Algorithm

The task:

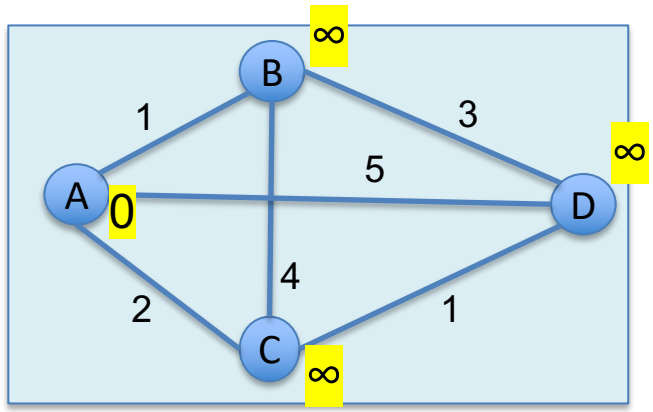
- Given a weighted graph $G=(V,E,w(E))$, and $s \in V$, and supposing that *all weights are positive*.
- Find shortest path (path with min total weight / min distance) from s to all other vertices.

Input



Output

Destination	Shortest paths	Distance
A	A→A	0
B	A→B	1
C	A→C	2
D	A→C→D	3



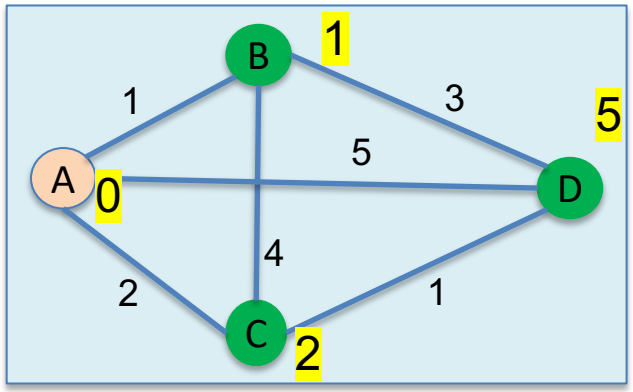
Find a shortest path:

- From A to any other node

A is the source:

- shortest path A→A has `dist[A]= 0`
- and distance-so-far `dist[]= {0, ∞, ∞, ∞}` (for {A,B,C,D})

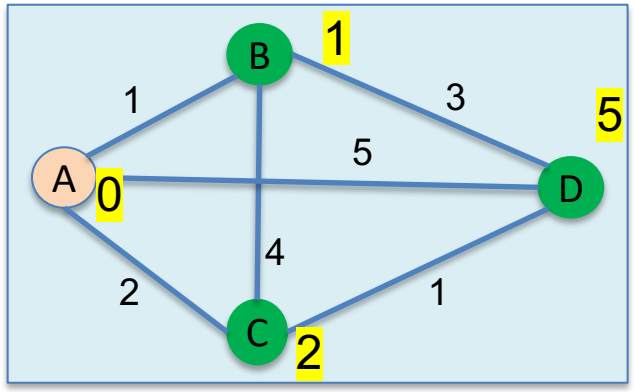
→ Clearly, we've done with A, and can remove it from the job list!
But with that, we should use the information from A to update
`dist[]= {0, 1, 2, 5}`



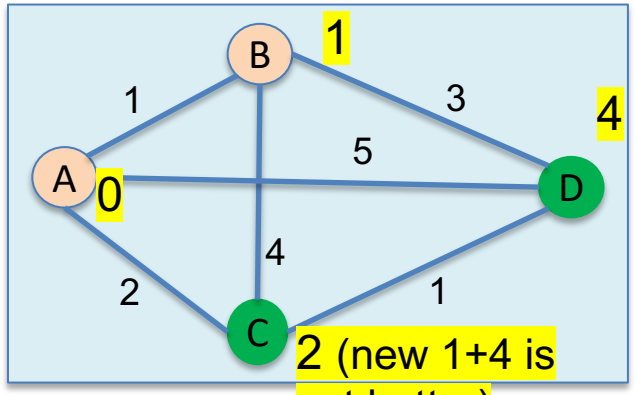
What's next?

But with that, we should use the information from A to update

`dist[]= {0, 1, 2, 5}`

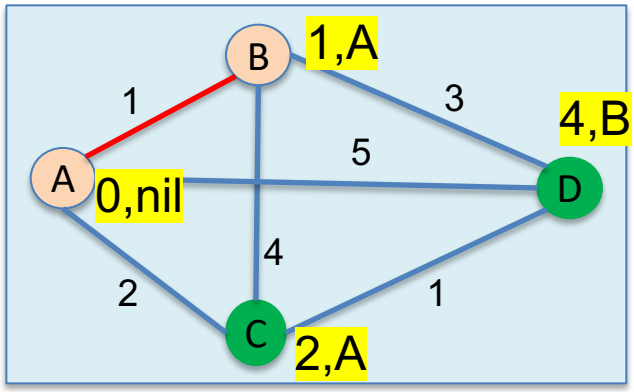


What's next?
explore C, B, or D?



So shortest path from A to B found with `dist[B]= 1`.

But how can we retrieve the detailed path A→B?
need to keep track of ???



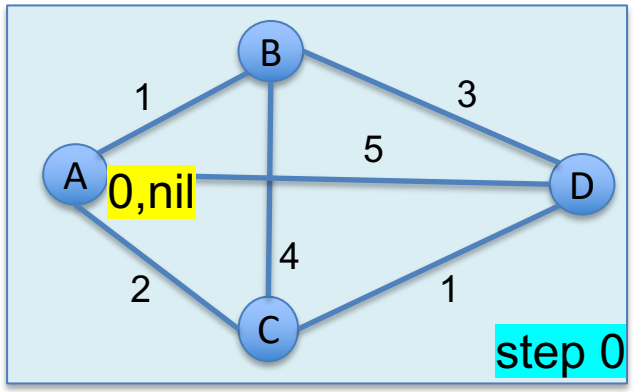
	A	B	C	D
	0, nil	∞,nil	∞,nil	∞,nil
A		**1,A**	2,A	**5,A**
B			2,A	**4,B**

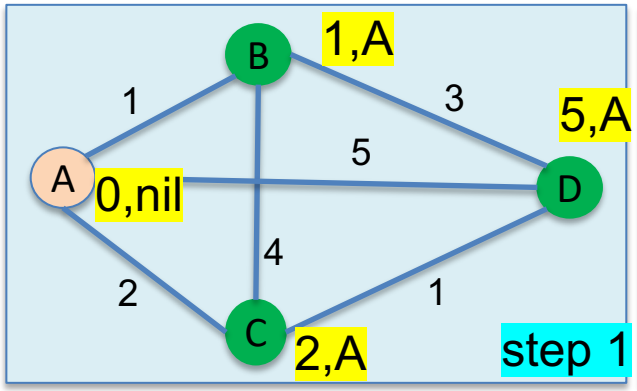
set {B,C,D} includes not-yet-done elements at this stage.
How to best keep track of not-yet-done distance?

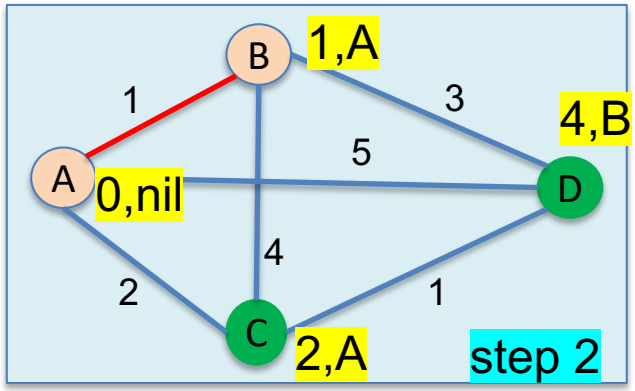
this column: nodes with found shortest path

dist[B]: shortest-so-far distance from A

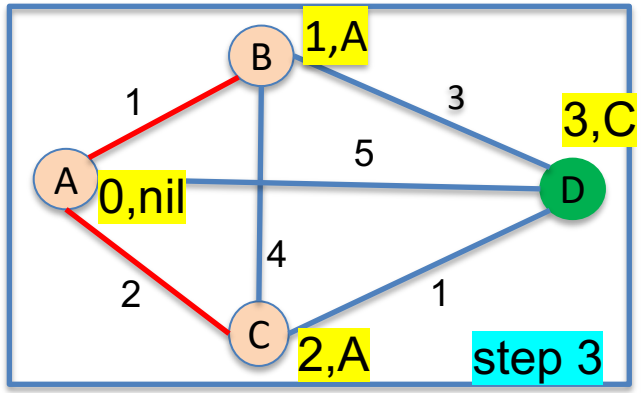
prev[D]: node that precedes D in the path A→D

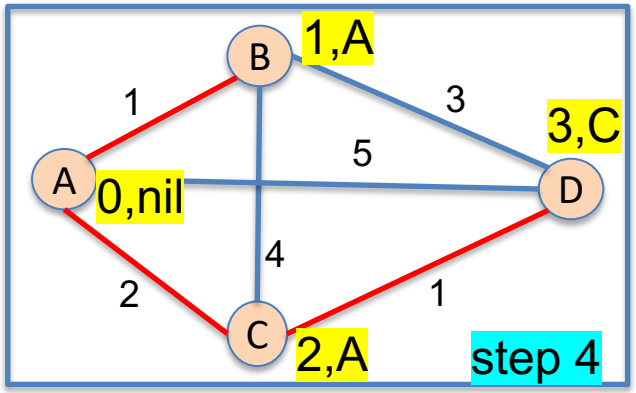






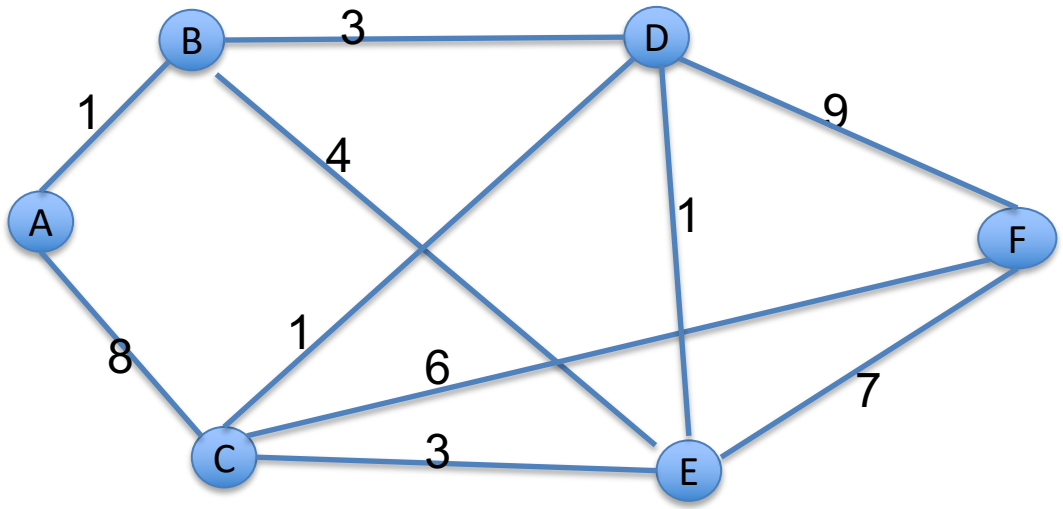
Running Dijkstra from node A, ie. finding shortest paths from A to all nodes.
The number at each **node** is the total distance from **A** to this **node** = distance of *previous node* + weight of the edge (*previous node*, **node**)





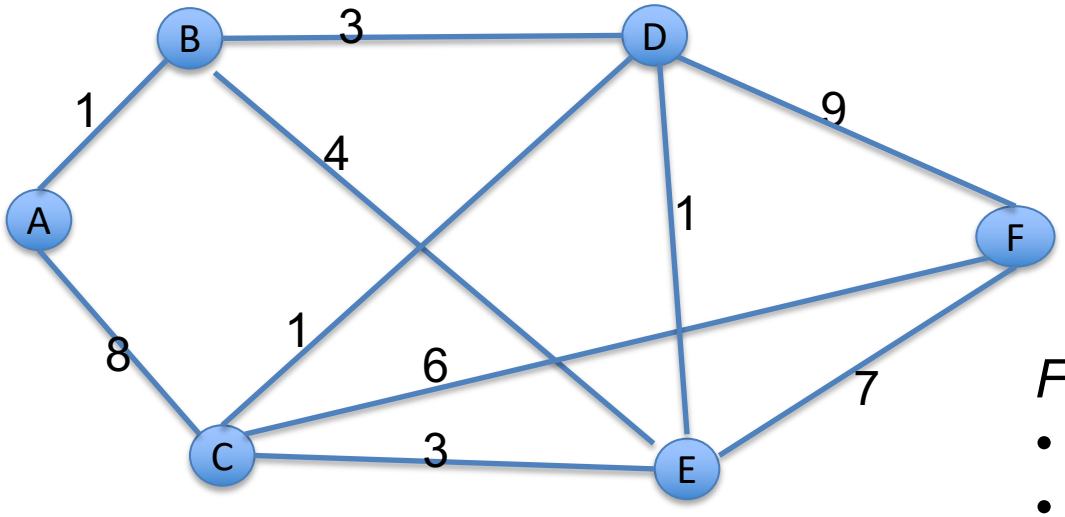
step	node with found shortest path	A	B	C	D
0		0,nil	∞,nil	∞,nil	∞,nil
1	A		1,A	2,A	5,A
2	B			2,A	4,B
3	C				3,C
4	D				

DIY: run Dijkstra's Algorithm for this graph



Find a shortest path:

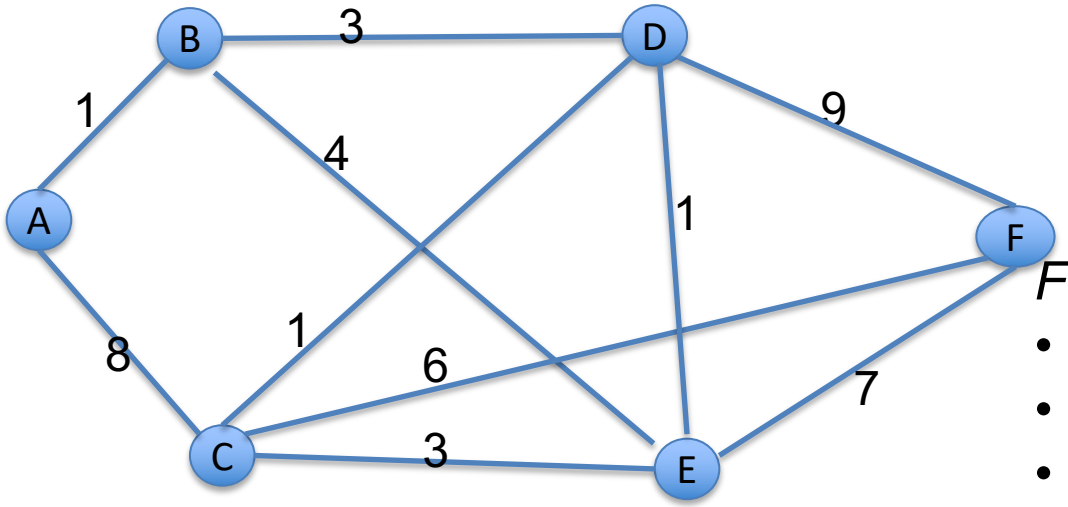
- From A to B
- From A to C
- From A to F
- From A to any other node



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						



Find a shortest path:

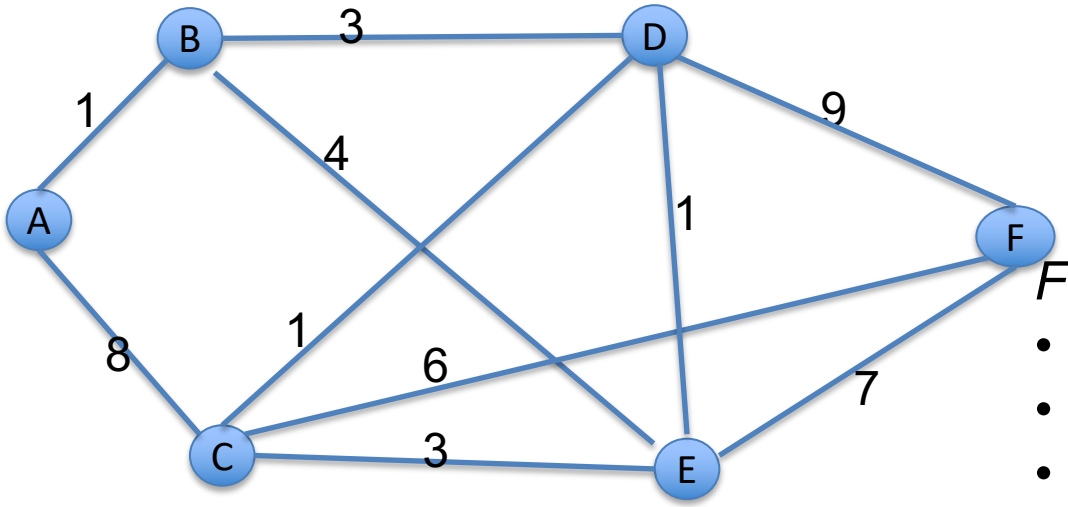
- From A to B
- From A to C
- From A to F
- SP A->F=

4 The dist at SA 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 previuosly-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!



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

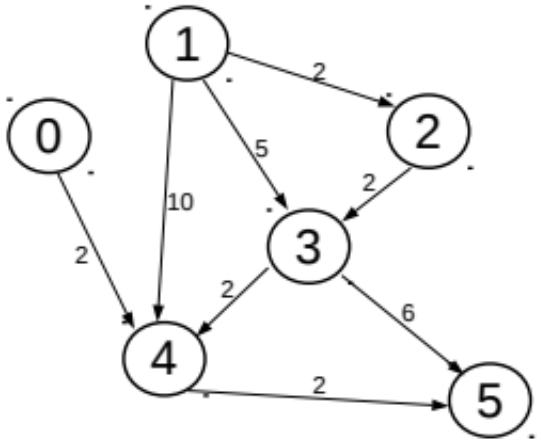
pred[D]= B:
B→D→ C→F

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

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



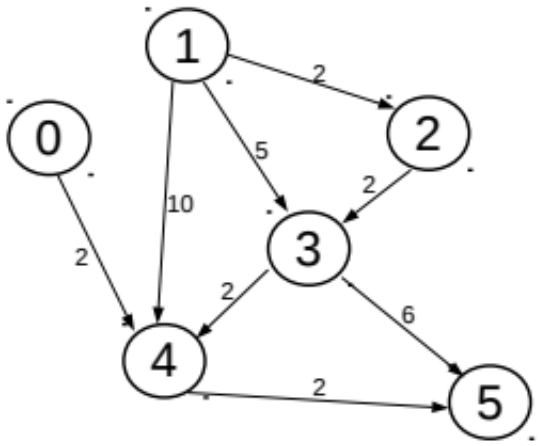
Trace the Floyd-Warshall algorithm.
Step i= 0, 1, 2, 3, 4, 5

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

Run FWA *manually*



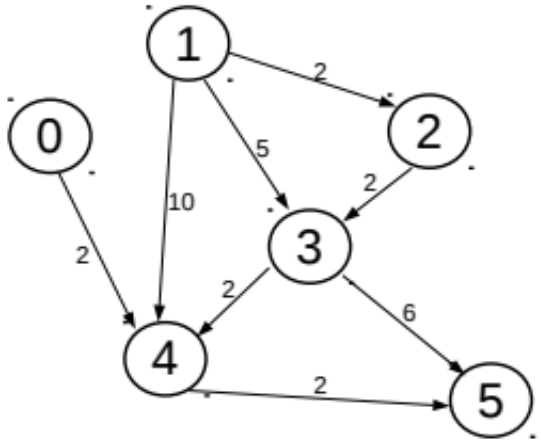
Notes:

- 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

Run FWA *manually*

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

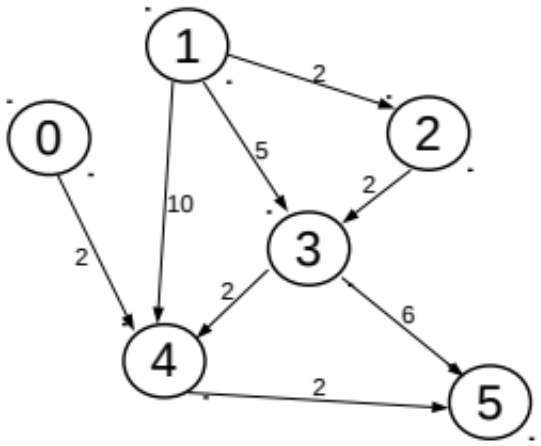


column **2** gives paths "from s to **2**"

Only this cell need to be considered. Why?

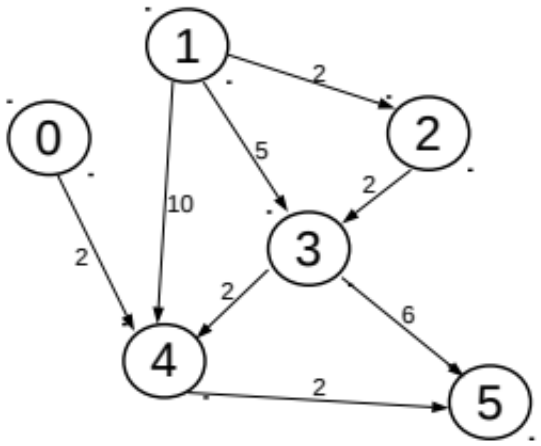
row **2** gives paths "from **2** to t"

	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



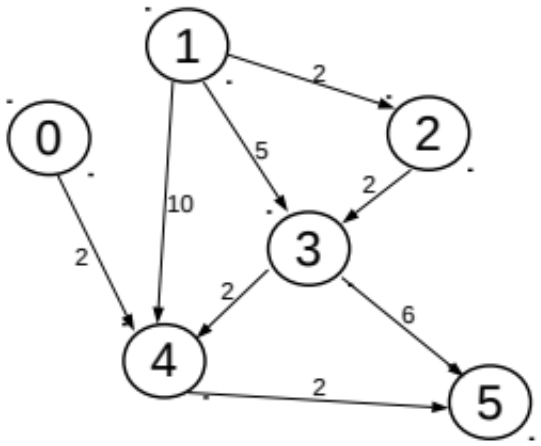
i= **2** as the stepstone

	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



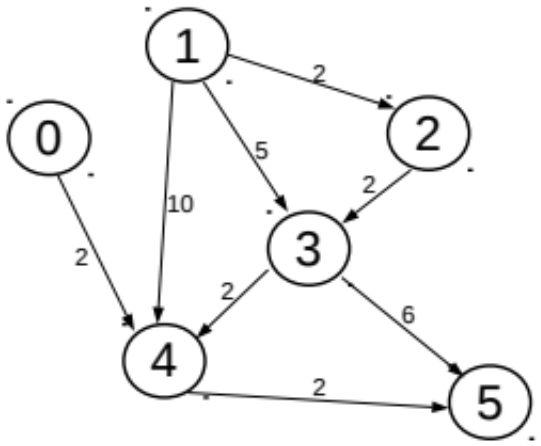
i= 3 as the stepstone

	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



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



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