MTRN4110 Robot Design
Week 4 – Planning I

Liao "Leo" Wu, Lecturer

School of Mechanical and Manufacturing Engineering

University of New South Wales, Sydney, Australia

https://sites.google.com/site/wuliaothu/

Today's agenda

- Introduction to planning
- Graph construction
- Graph search

Introduction to Planning

What is planning?

- The process that the robot determines a plan to get from A to B based on the information about the environment and the robot itself

How am I getting to the target?

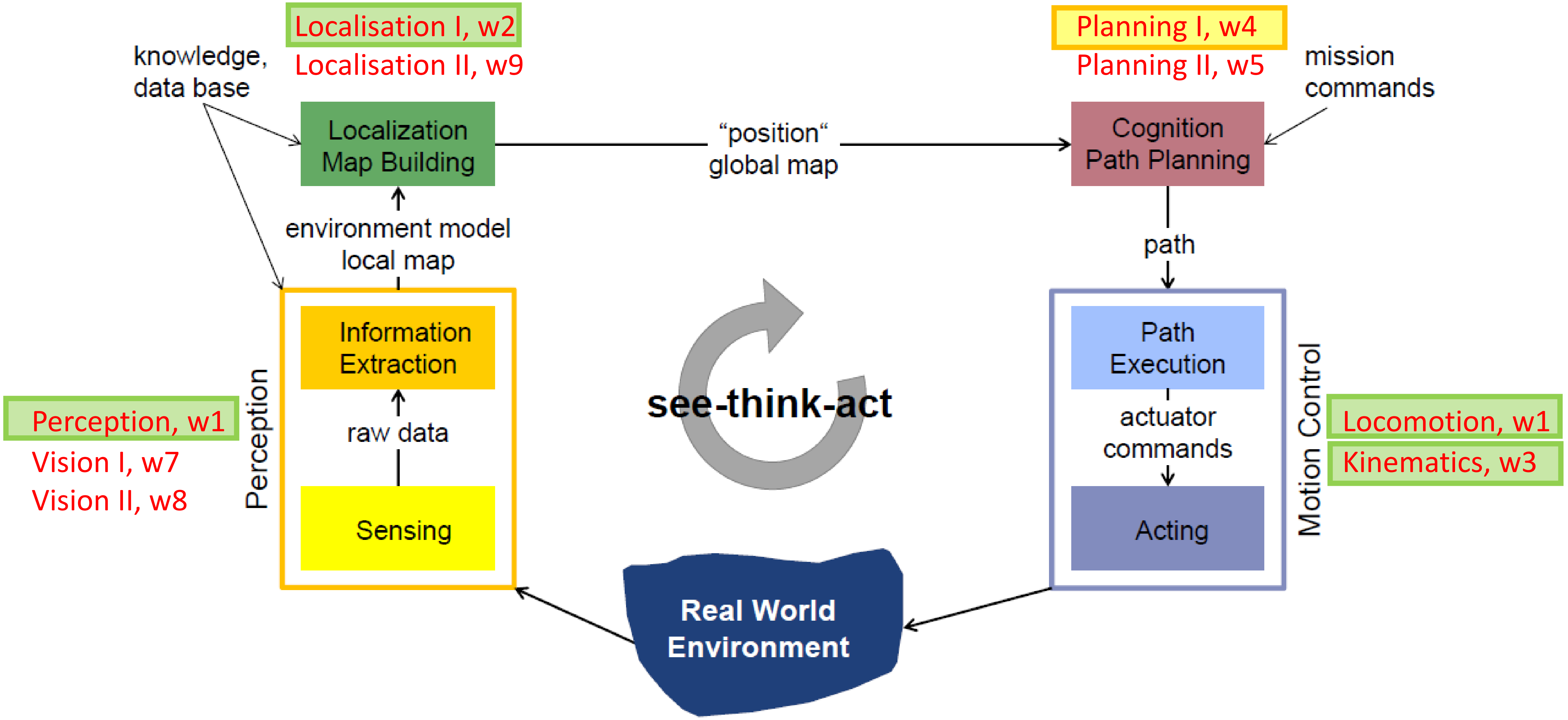
Two types of planning

Path Planning

Trajectory Planning

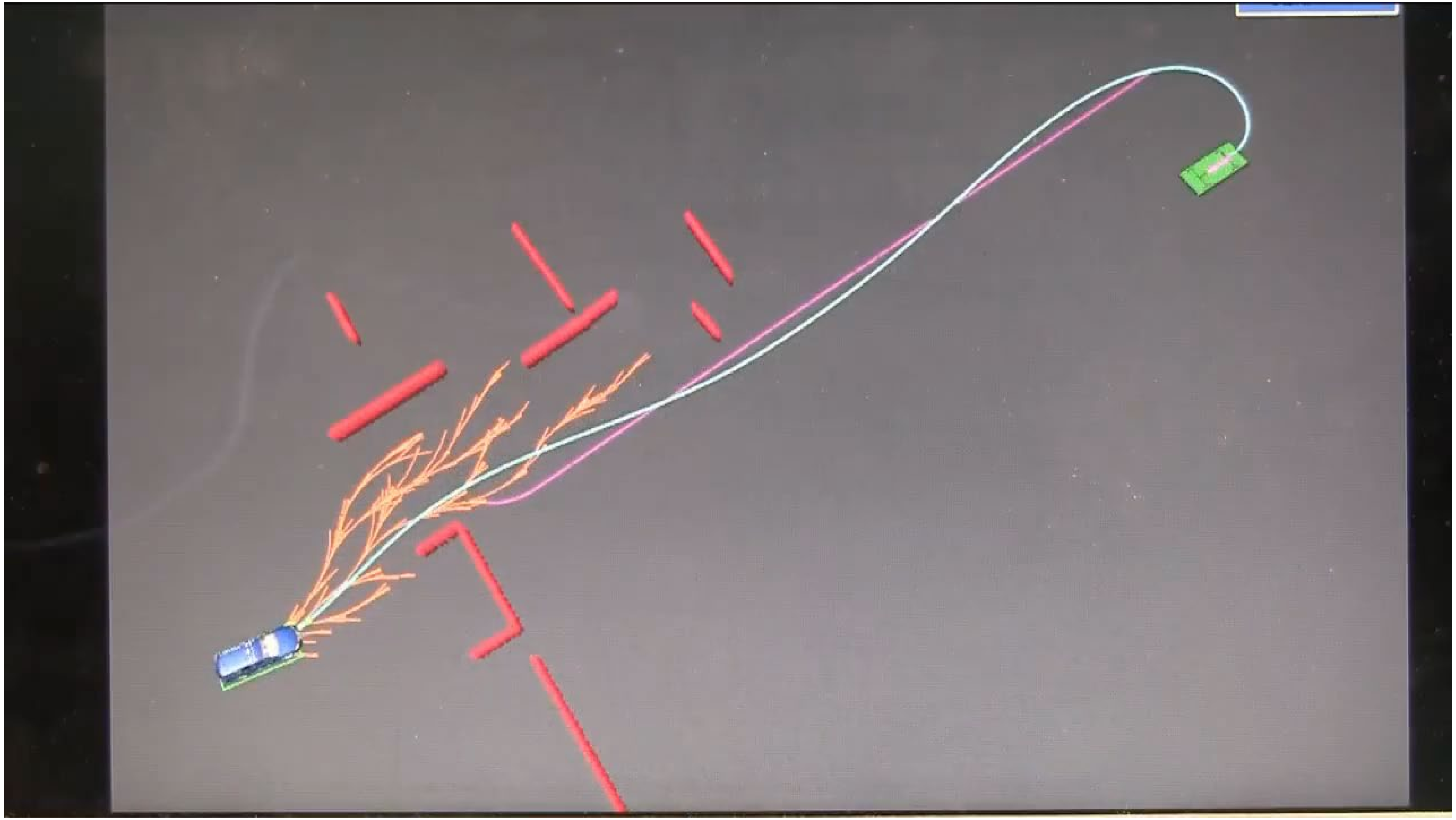
https://www.sportsessionplanner.com/s/eIxQ/Shooting-U11s.html

The See-Think-Act cycle



R. Siegwart, I. R. Nourbakhsh, D. Scaramuzza. Introduction to autonomous mobile robots. The MIT Press. Second edition. 2011.

Planning - Example



https://www.youtube.com/watch?v=zS3st_7og3A&feature=youtu.be
Dolgovet al, Practical Search Techniques in Path Planning for Autonomous Driving, 2008

Planning in Micromouse

Search Run

- Exploration
- Mapping

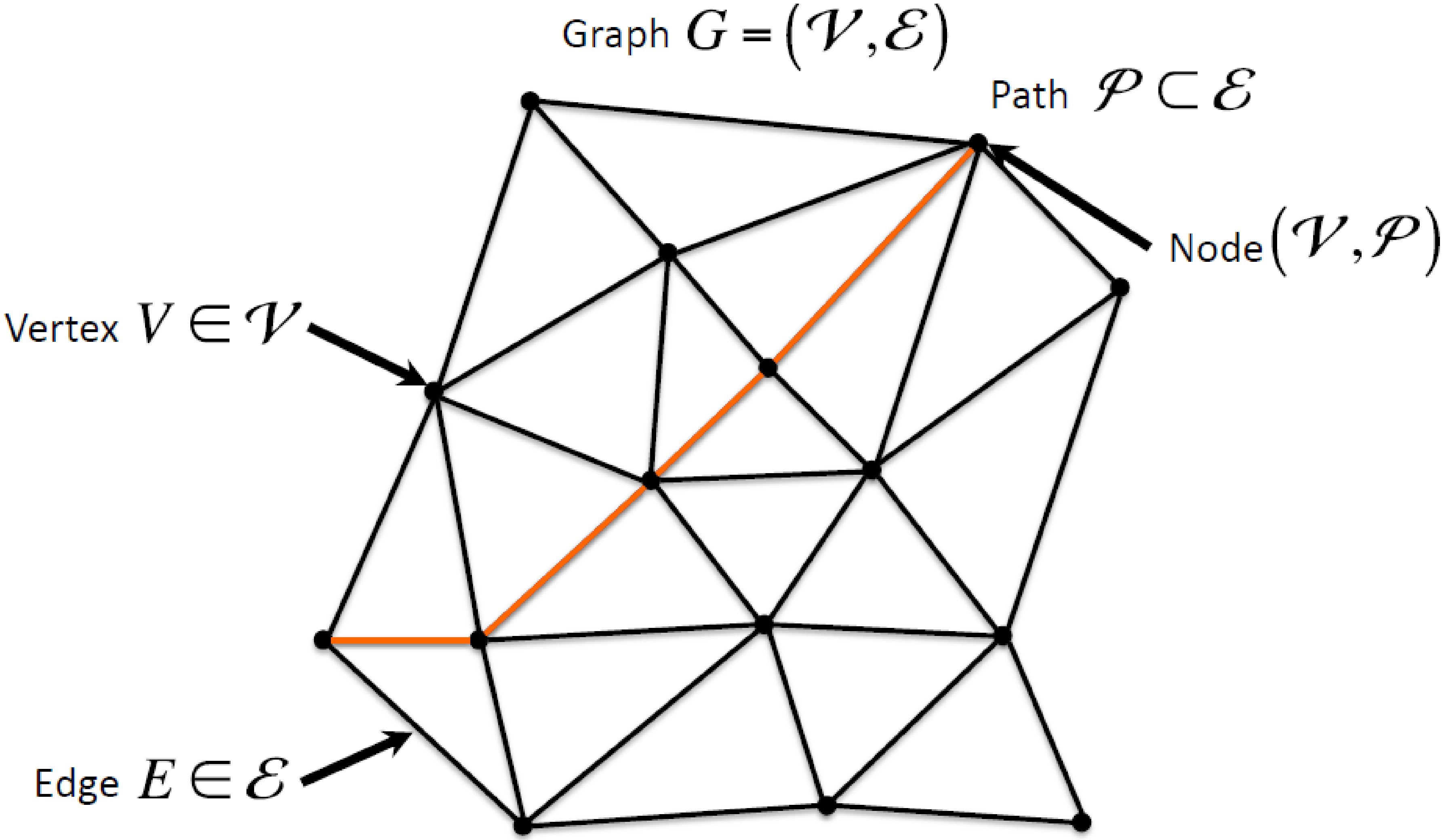
Speed Run

- Planning
- Execution

https://youtu.be/aXMcEDy-ly8

Graph Construction

What is a graph?



Graph construction – Construct graphs that can be searched

- Continuous map
  - Visibility graph
  - Voronoi diagram

- Cell decomposition
  - Exact cell decomposition
  - Fixed cell decomposition
  - Adaptive cell decomposition

- Topological representation

Graph construction – Construct graphs that can be searched

- **Continuous map**
  - **Visibility graph**
  - **Voronoi diagram**

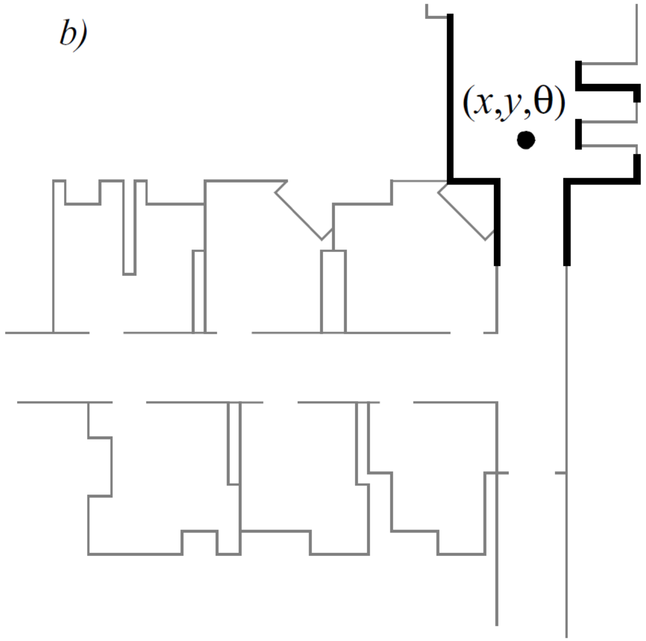
- Cell decomposition
  - Exact cell decomposition
  - Fixed cell decomposition
  - Adaptive cell decomposition

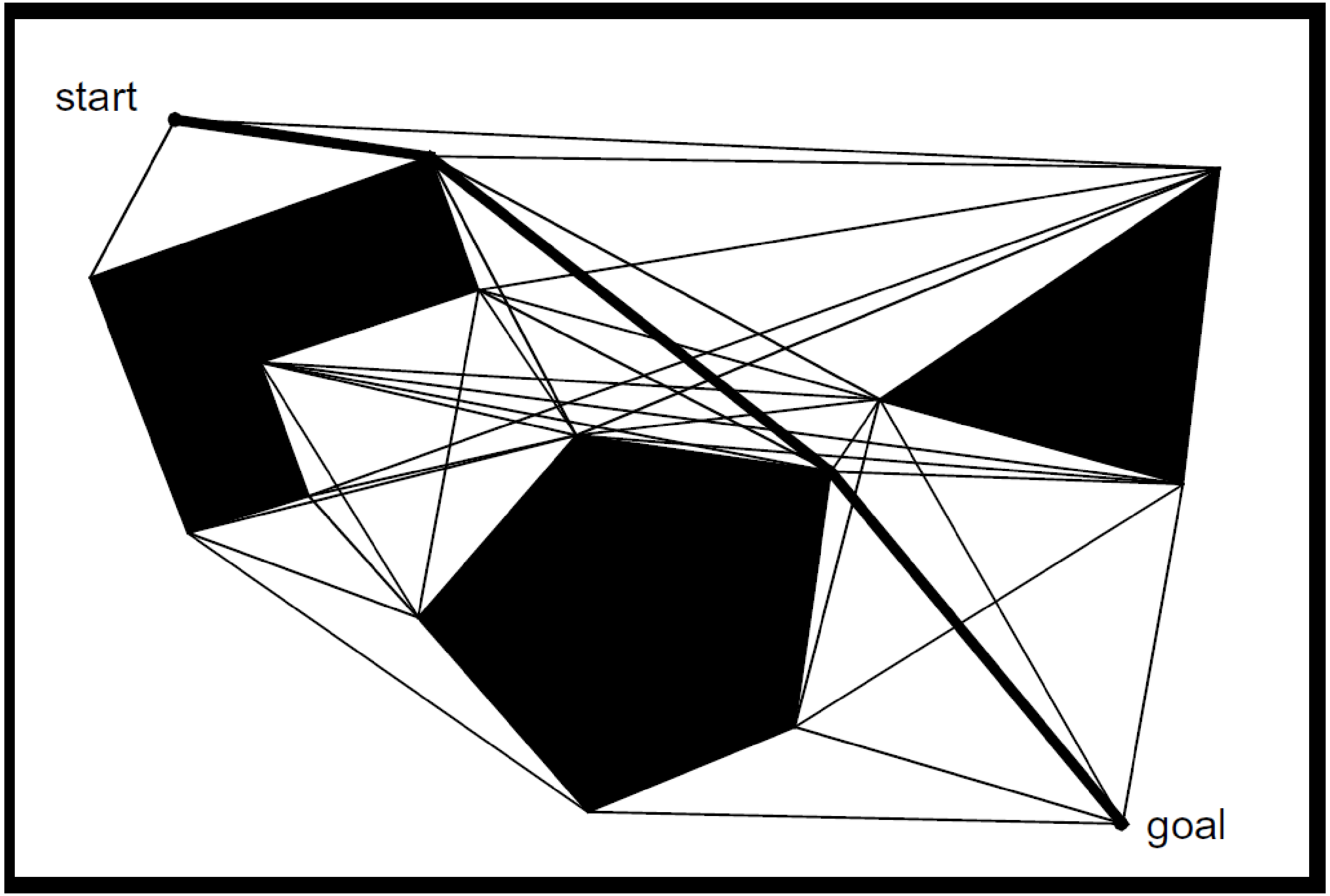
- Topological representation

Visibility graph – Connect all the vertices visible to each other

Short but not safe

Continuous Map



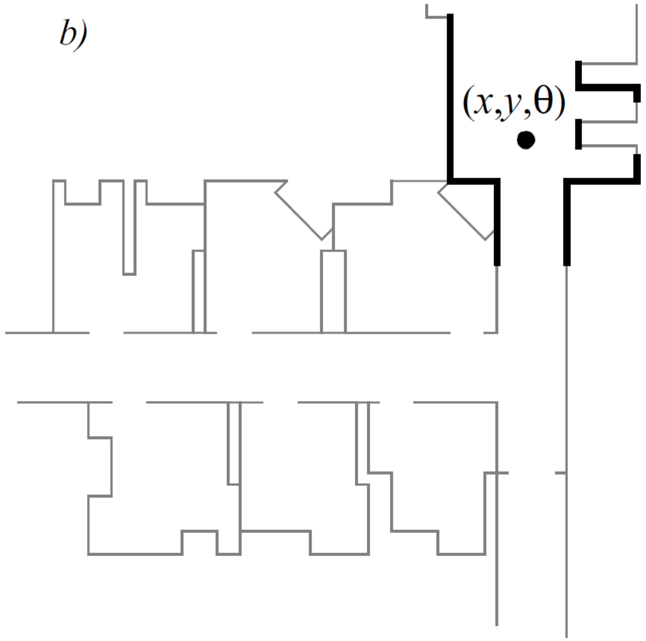


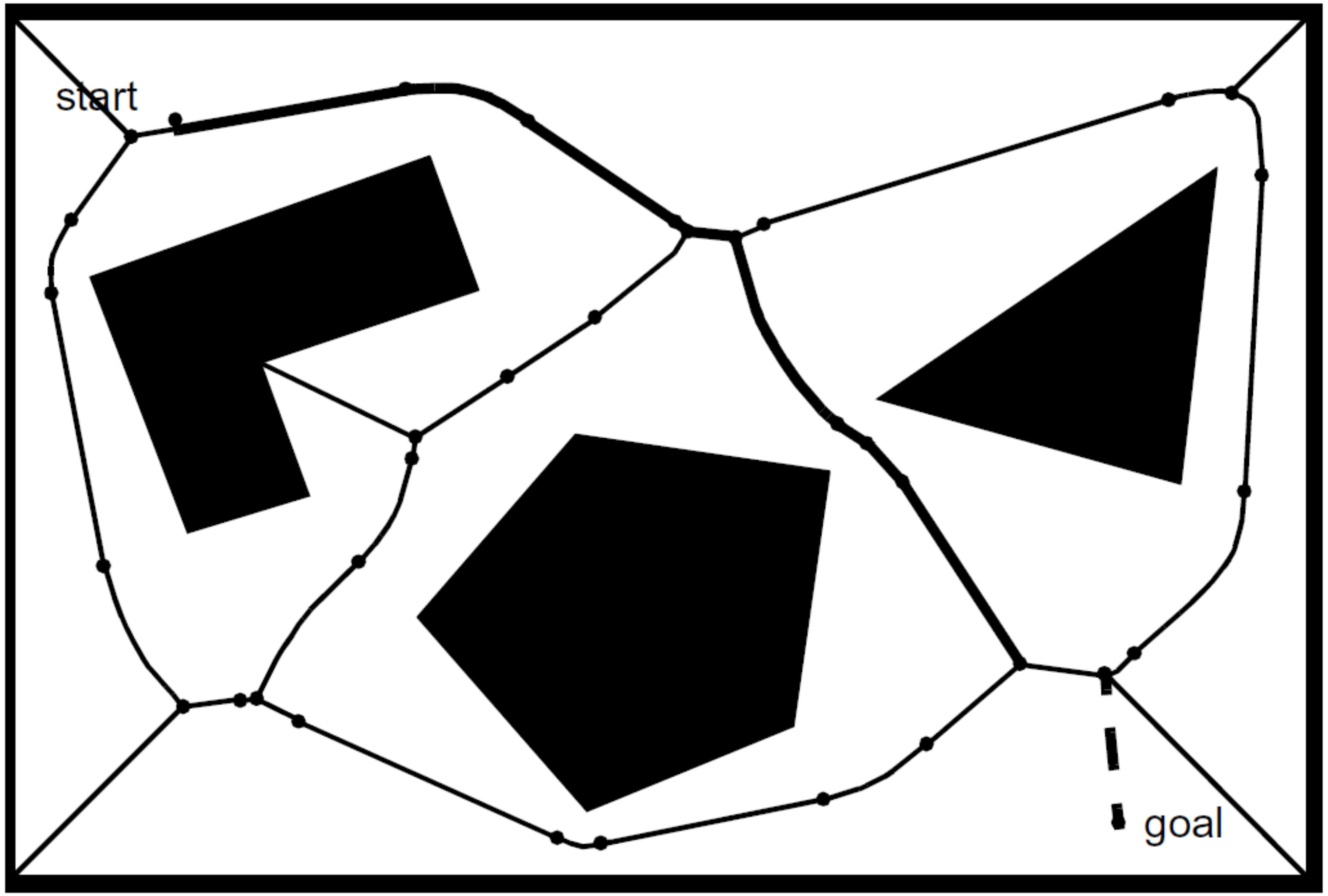
Latombe, J.-C., *Robot Motion Planning*. Norwood, MA, Kluwer Academic Publishers, 1991.
R. Siegwart, I. R. Nourbakhsh, D. Scaramuzza. Introduction to autonomous mobile robots. The MIT Press. Second edition. 2011.

Voronoi diagram – Points with equal distance to nearest edges

Safe but not short

Continuous Map





Georgy Voronoy
1868-1908

Graph construction – Construct graphs that can be searched

- Continuous map
  - Visibility graph
  - Voronoi diagram

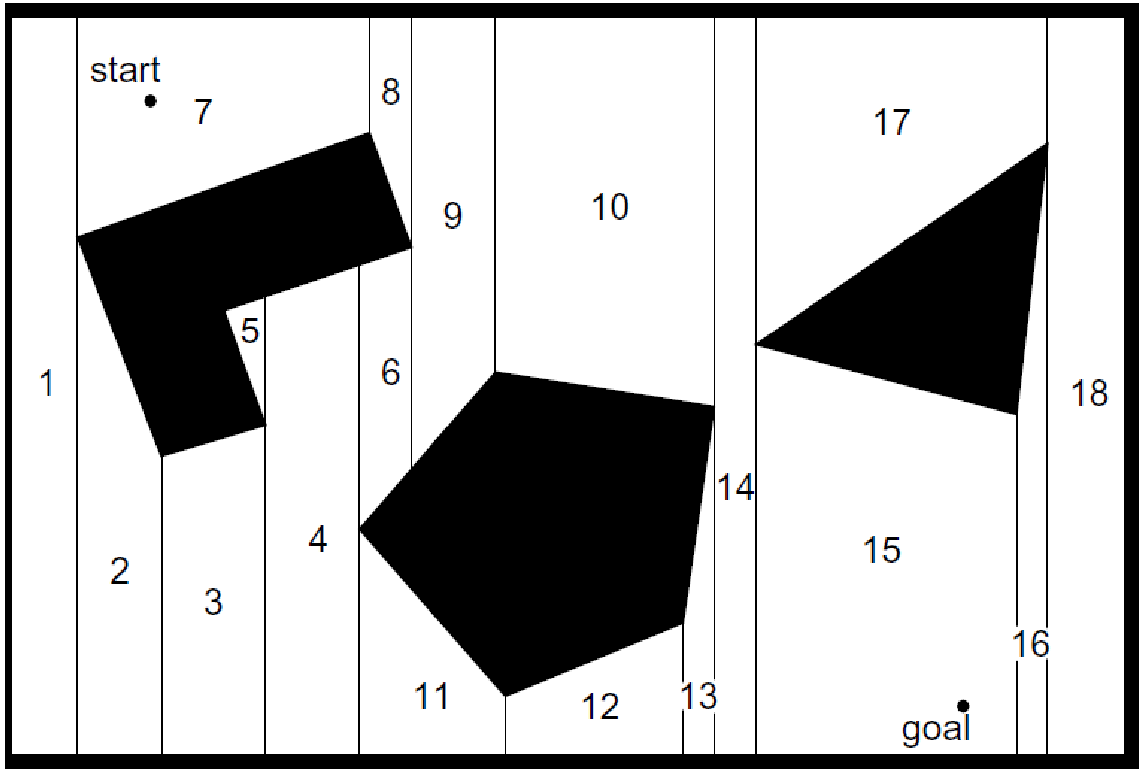
- **Cell decomposition**
  - **Exact cell decomposition**
  - **Fixed cell decomposition**
  - **Adaptive cell decomposition**

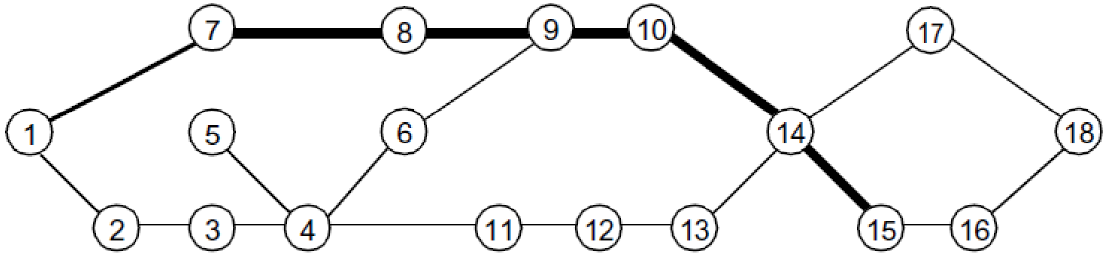
- Topological representation

Exact cell decomposition

Efficient for large, sparse environment

Complex in implementation



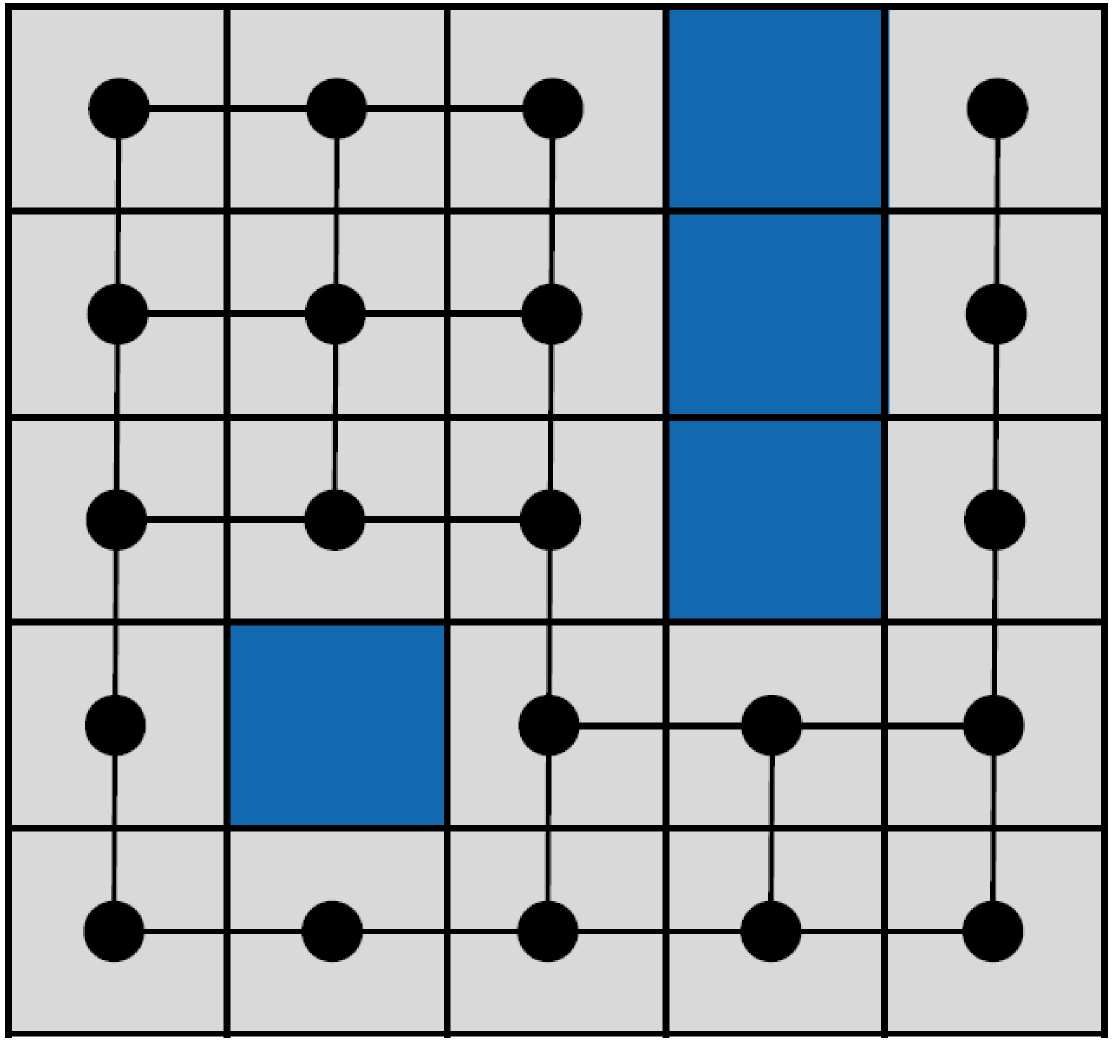


Fixed cell decomposition – Occupancy grid

Very popular

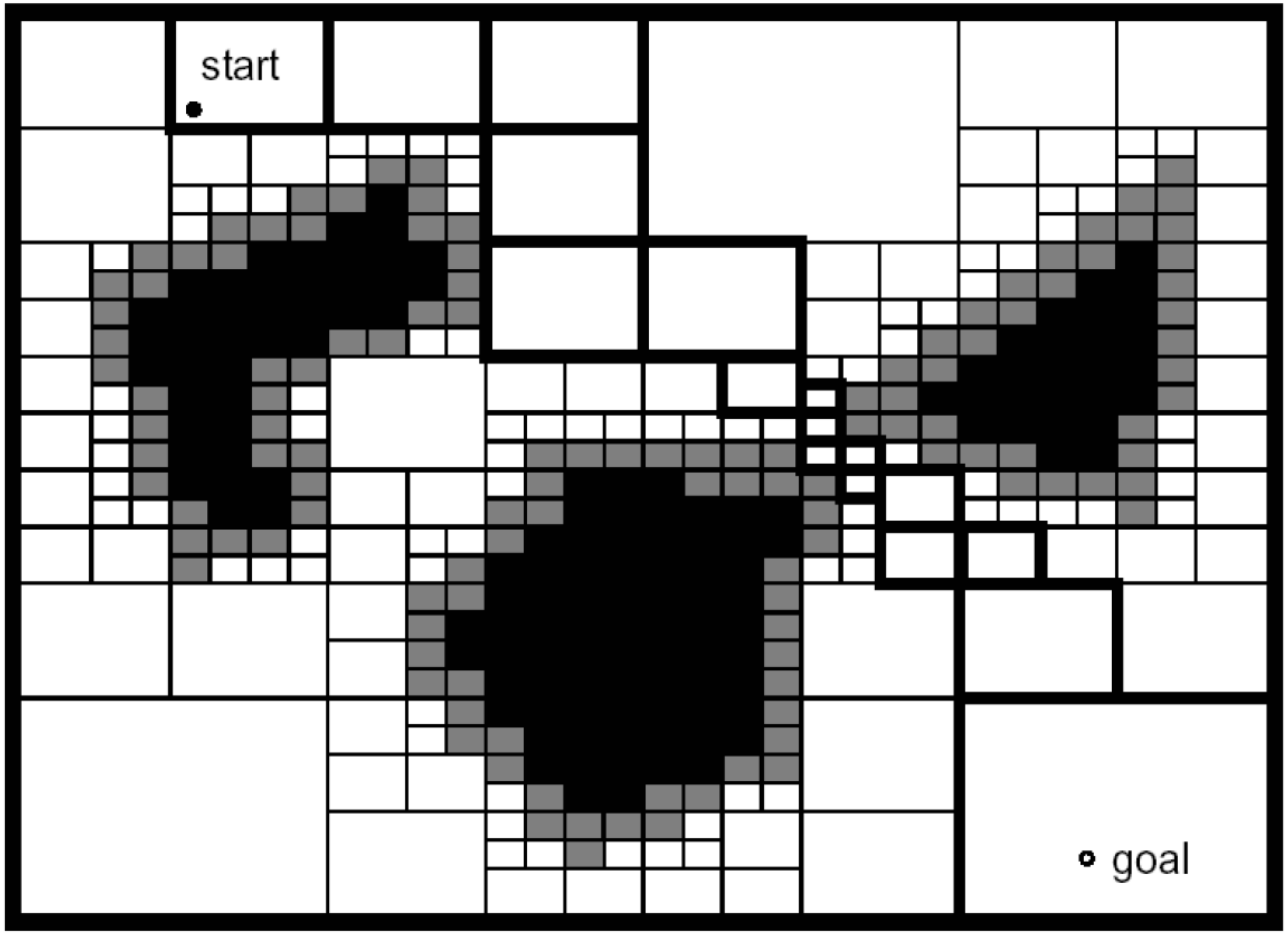
Easy in implementation

High need in memory



Adaptive cell decomposition

Similar to occupancy grid
Fewer nodes for large areas
Complex in implementation



Graph construction – Construct graphs that can be searched

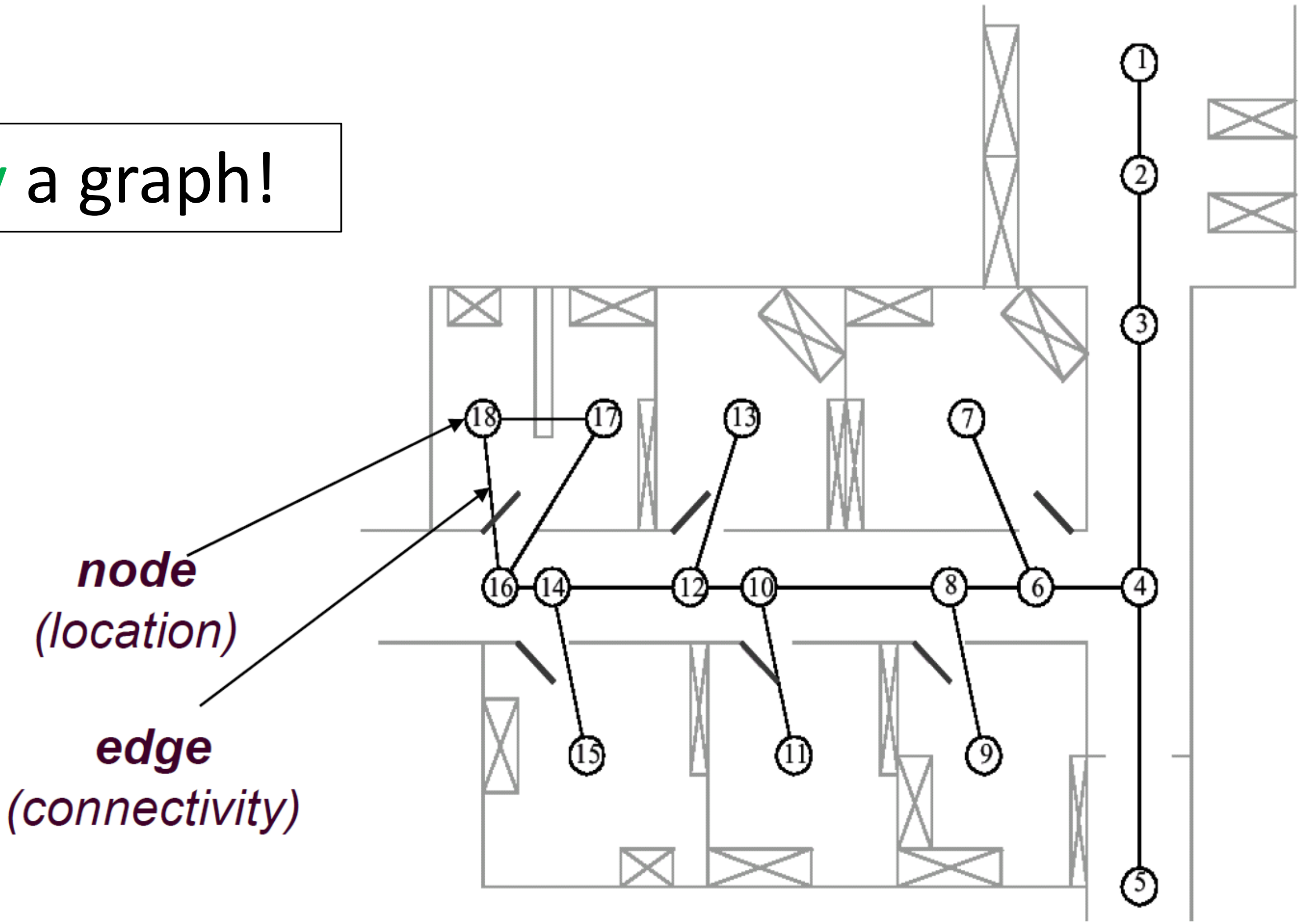
- Continuous map
  - Visibility graph
  - Voronoi diagram

- Cell decomposition
  - Exact cell decomposition
  - Fixed cell decomposition
  - Adaptive cell decomposition

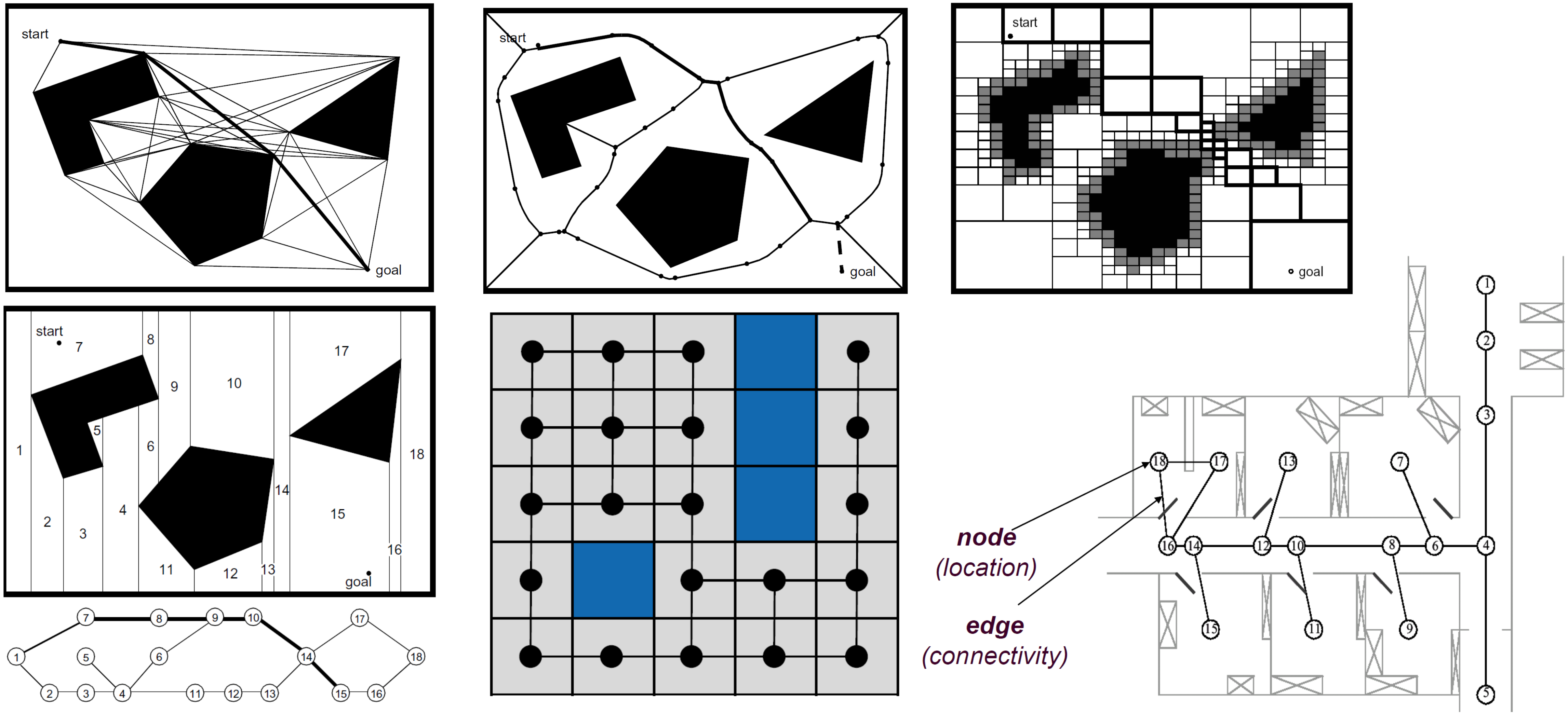
- **Topological representation**

Topological representation

It's already a graph!



From Map to Graph



Why constructing graphs for path planning?

- Because graph search has been well studied!

Graph Search

Graph search

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Dijkstra’s algorithm
- A* algorithm
- Bellman-Ford algorithm
  - Flood Fill algorithm

Graph search - Terminology

- Current state

$$x \in X$$

- Next state

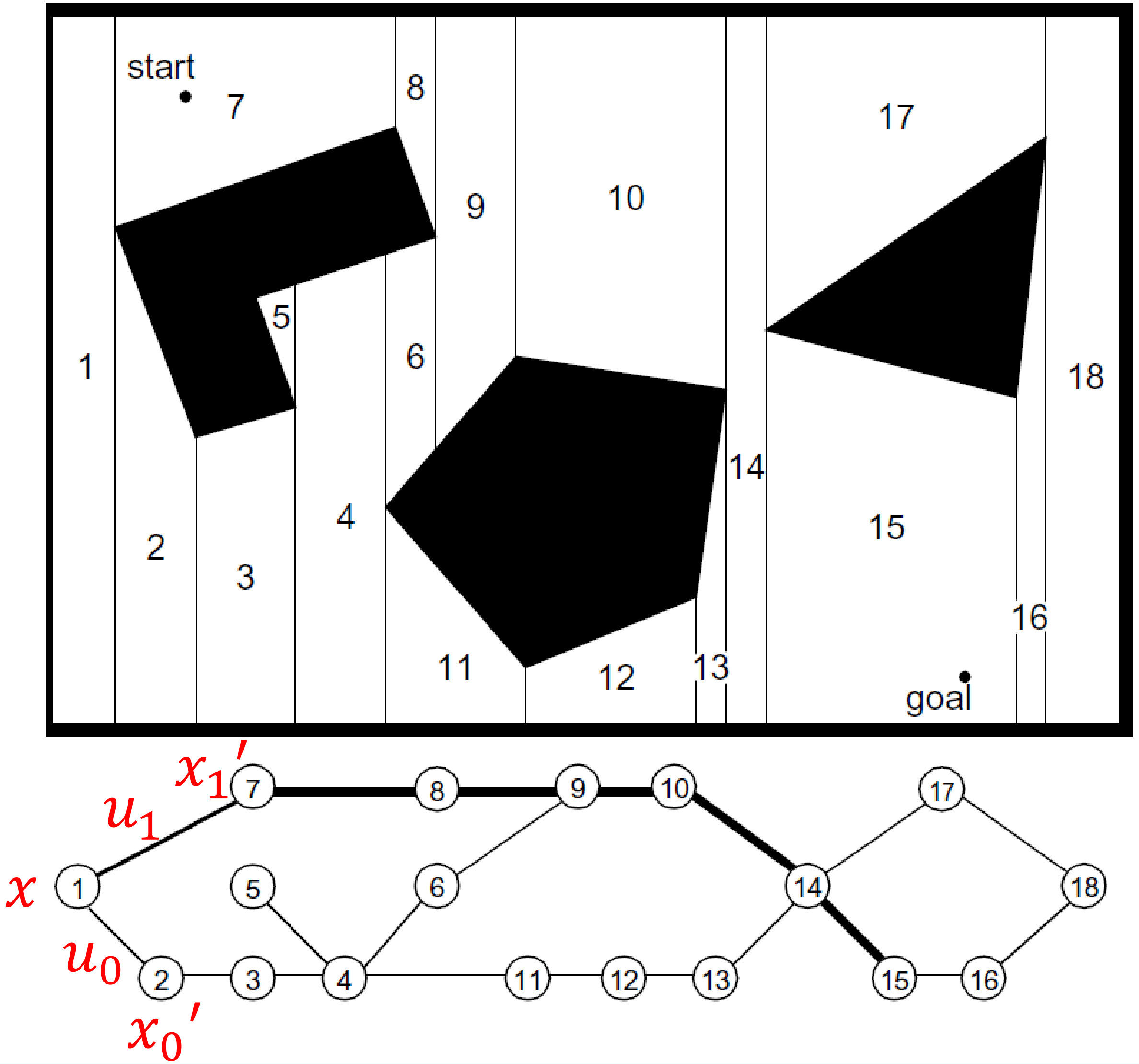
$$x' \in X$$

- Actions
  - Feasible transitions between states

$$U(x) = \{u_0, u_1, u_2, \cdots, u_n\}$$

- Transition function

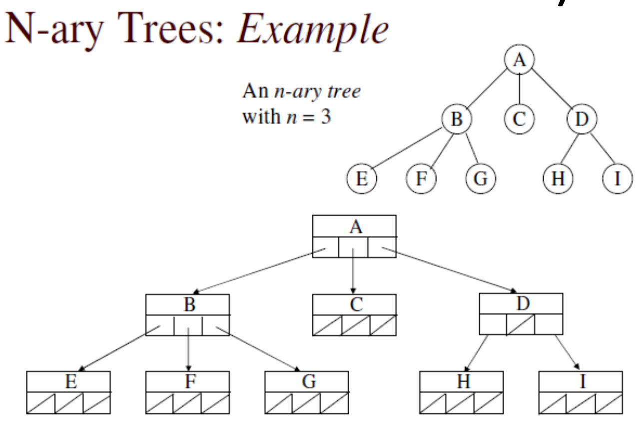
$$f(x, u) = x'$$



R. Siegwart, I. R. Nourbakhsh, D. Scaramuzza. Introduction to autonomous mobile robots. The MIT Press. Second edition. 2011.

Graph search - Terminology

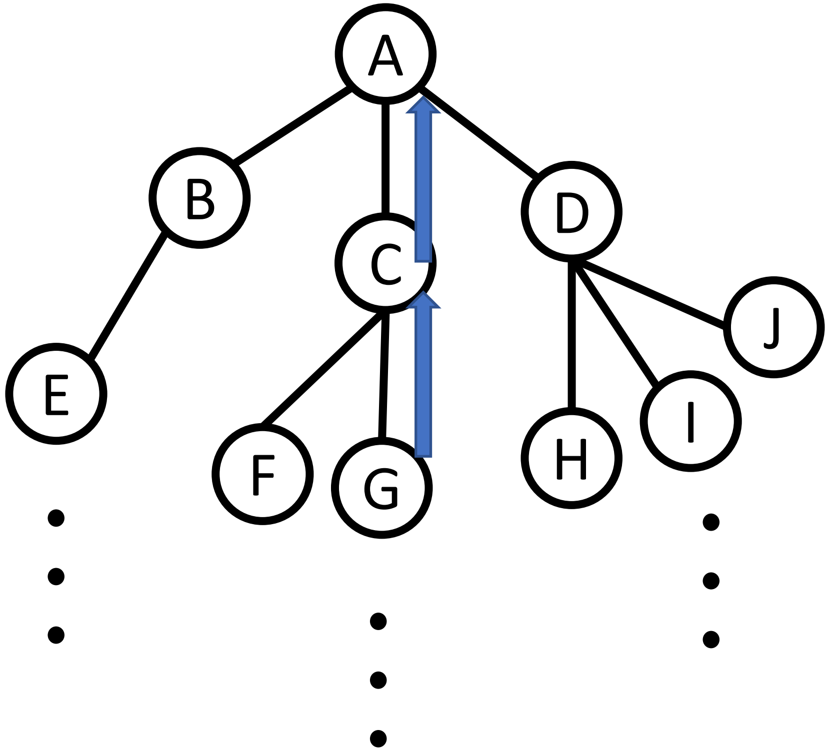
- Tree:
  - Start state at the root node
  - Children correspond to successors
  - Different implementations
    (refer to the references)



Pointer-based implementation of the *n-ary tree*

- A node corresponds to a unique plan from start to that state (<u>follow up tree from node</u>)

We can store information in a tree and easily retrieve the path from start to a node



Graph search - A general template

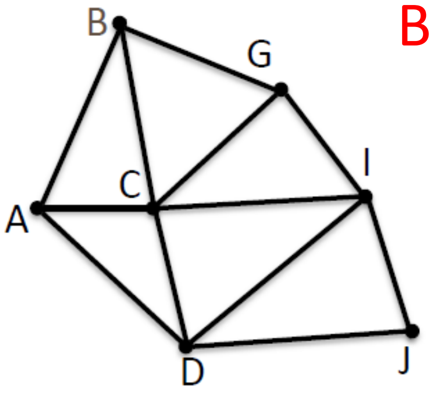
FORWARD_SEARCH

$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if x' not visited
            Mark x' as visited
            $Q.Insert(x')$
        else
            Resolve duplicate x'
return FAILURE

- Breadth-first search
- Depth-first search
- Dijkstra’s algorithm
- A* algorithm

LaValle, Steven M. *Planning algorithms*. Cambridge university press, 2006, p. 33

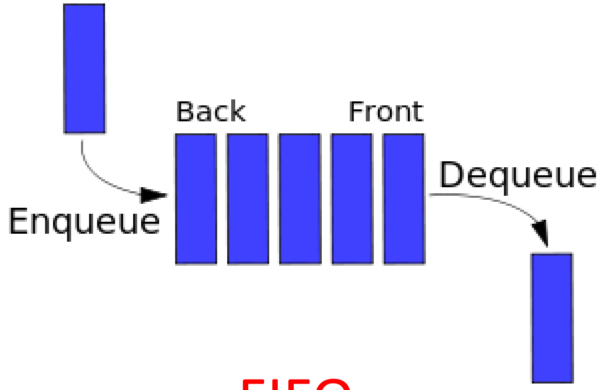
Breadth-first search (BFS)



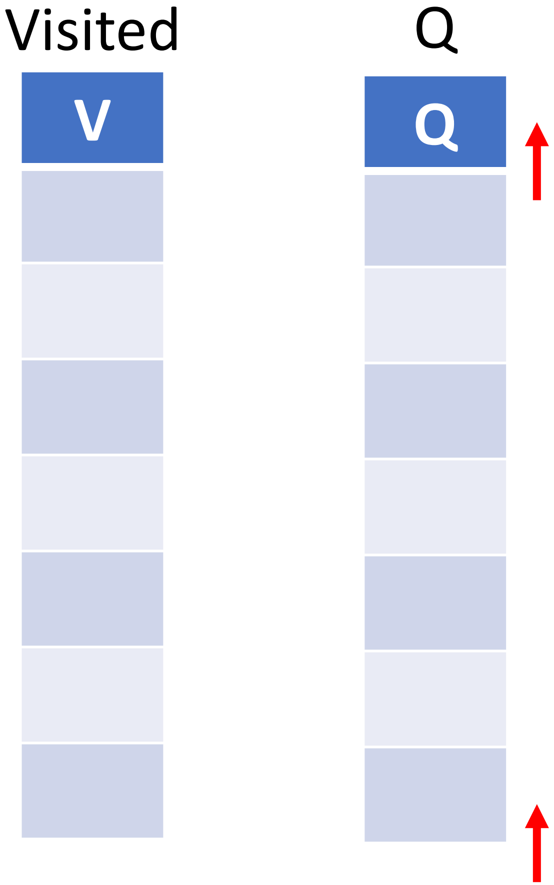
B -> J ?

FORWARD_SEARCH

$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if x' not visited
            Mark x' as visited
            $Q.Insert(x')$
        else
            Resolve duplicate x'
return FAILURE



FIFO



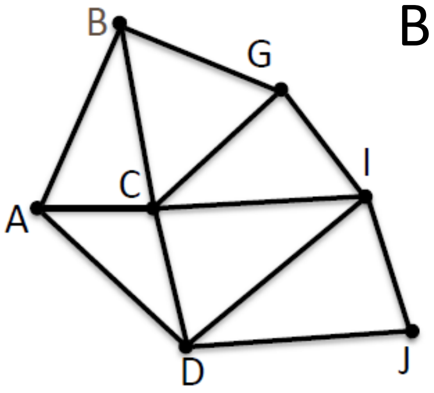
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

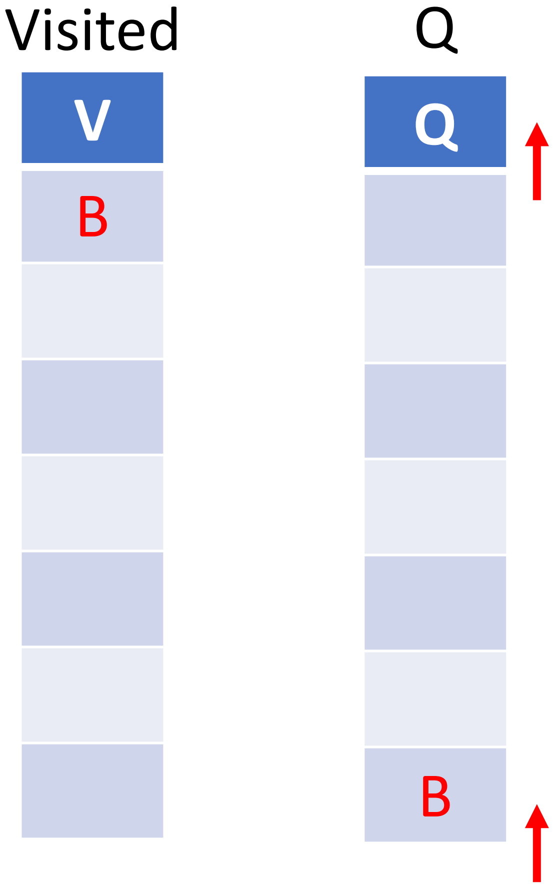
https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)

Breadth-first search (BFS)

Green – Open node



B -> J ?



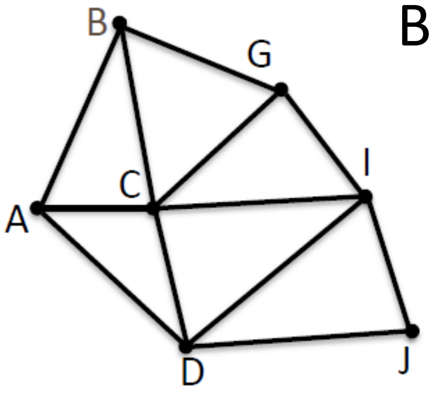
FORWARD_SEARCH

$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$
    return SUCCESS
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if x' not visited
      Mark x' as visited
      $Q.Insert(x')$
    else
      Resolve duplicate x'
return FAILURE

For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)

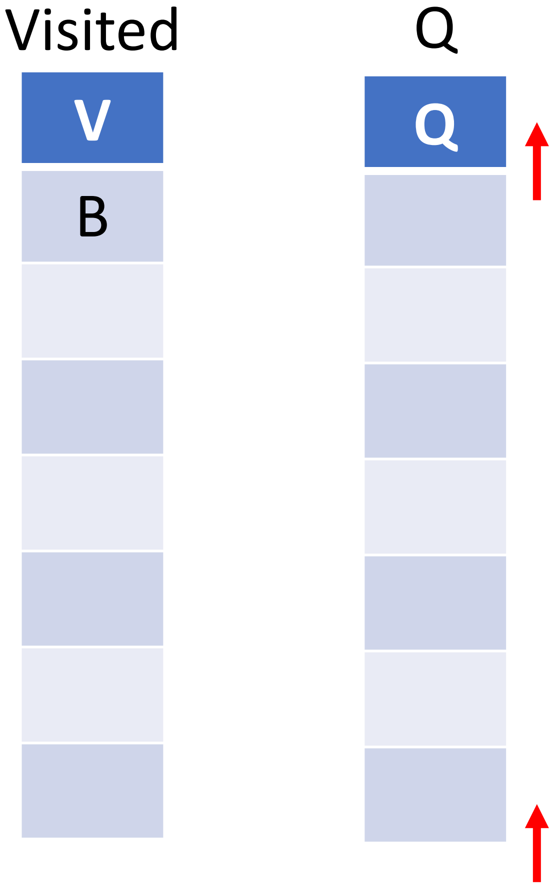


B -> J ?

B

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

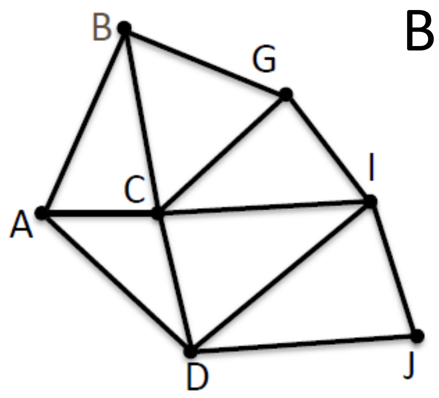
x = B



For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?

B

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

x = B

X_G = J

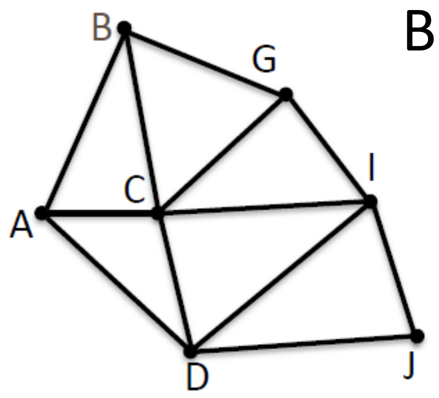
Visited	Q
V	Q
B	

For BFS, Q is FIFO (First In First Out)

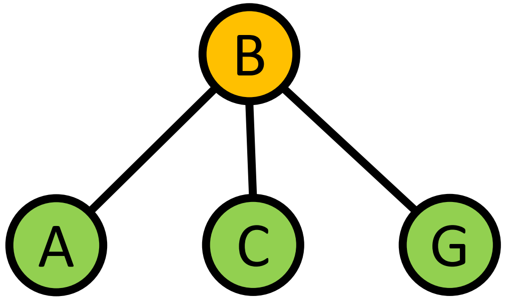
- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)

Yellow – Being explored



B -> J ?



FORWARD_SEARCH

```
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x′ ← f(x, u)
        if x′ not visited
            Mark x′ as visited
            Q.Insert(x′)
        else
            Resolve duplicate x′
return FAILURE
```

x = B

X_G = J

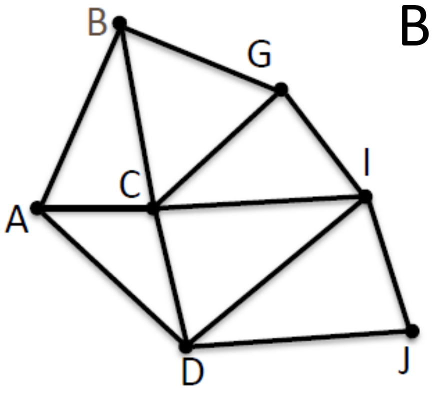
Visited	Q
V	**Q**
B	
A	
C	
G	
	A
	C
	G

For BFS, Q is FIFO (First In First Out)

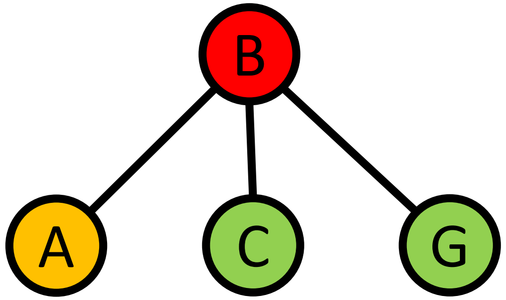
- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)

Red – Closed node



B -> J ?



Visited	Q
V	Q
B	
A	
C	
G	
	C
	G

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

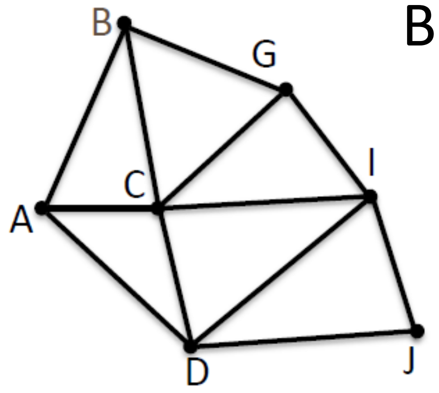
x = A

X_G = J

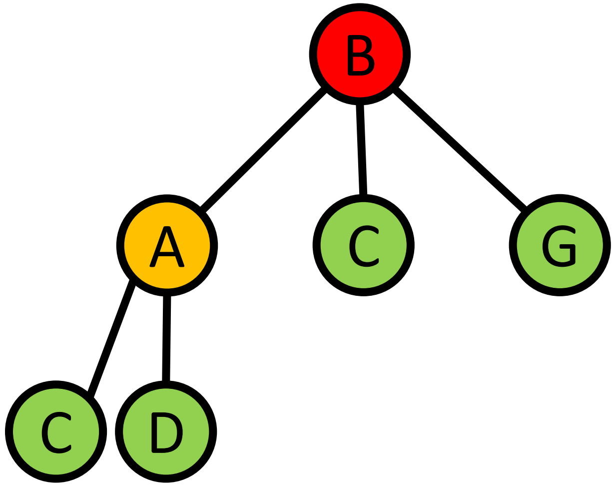
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



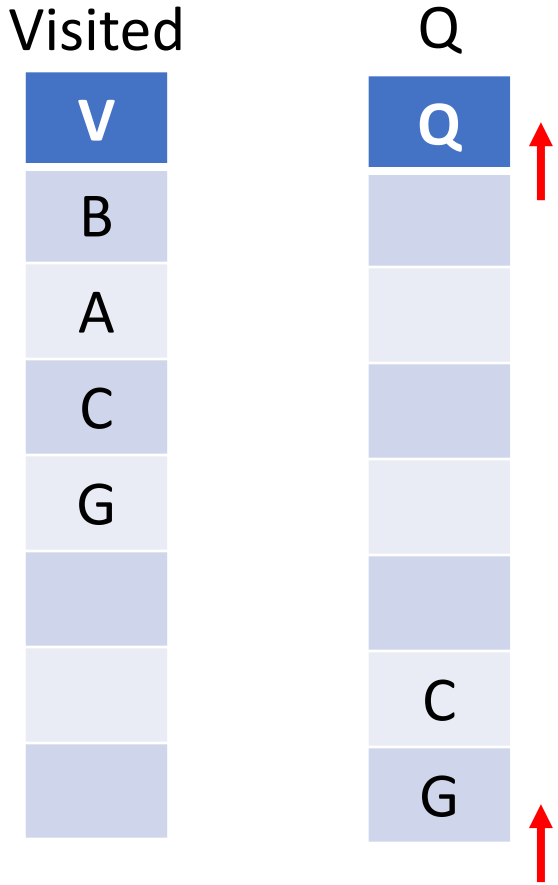
B -> J ?




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

x = A

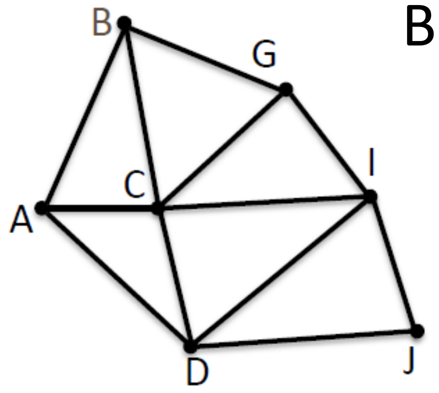
X_G = J



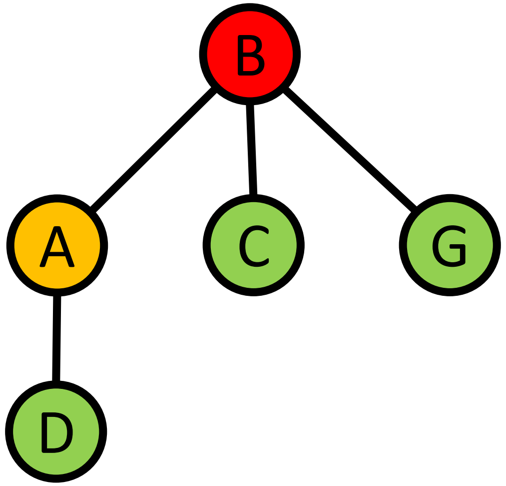
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?



Visited	Q
V	**Q**
B	
A	
C	
G	
D	C
	G
	D

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

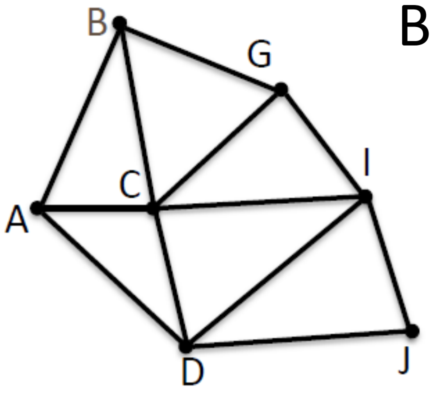
x = A

X_G = J

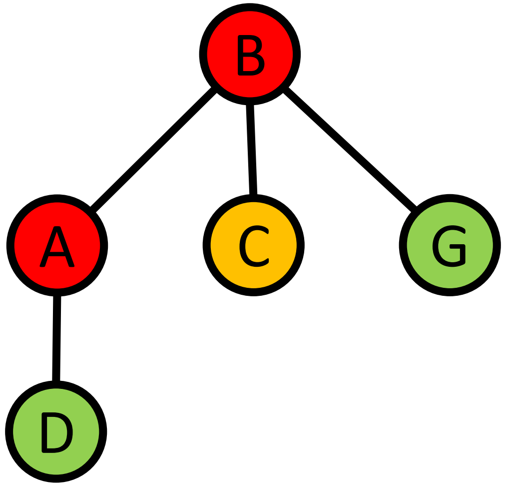
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?



Visited

V

B

A

C

G

D

Q

Q

G

D

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

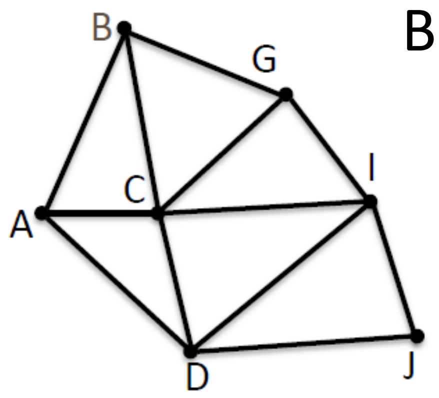
x = C

X_G = J

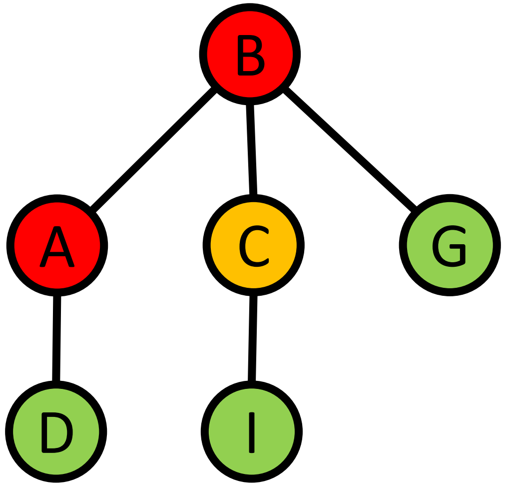
For BFS, Q is FIFO (First In First Out)

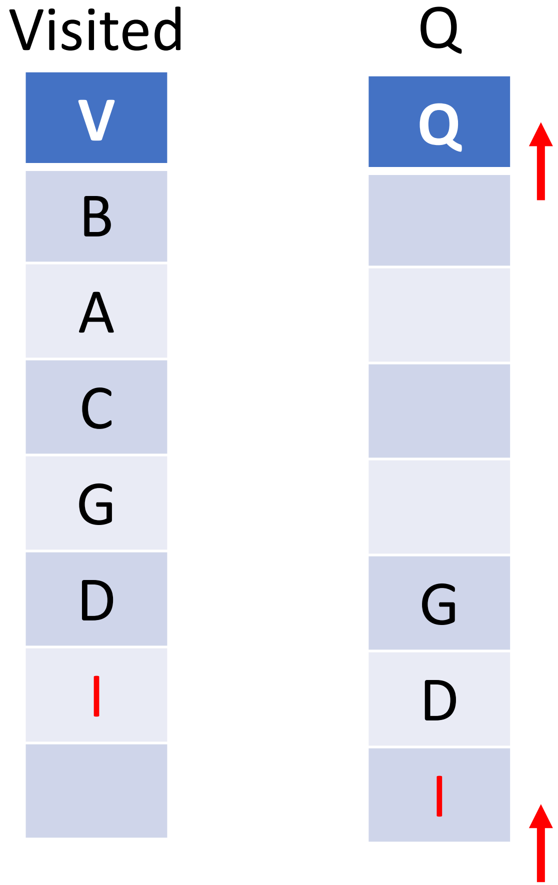
- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?






```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

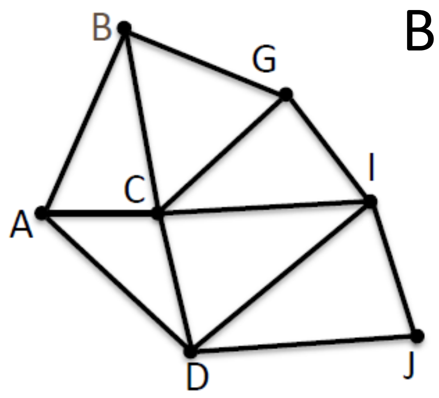
x = C

X_G = J

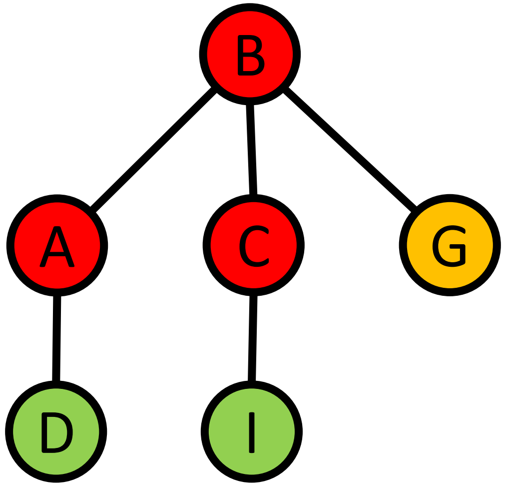
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



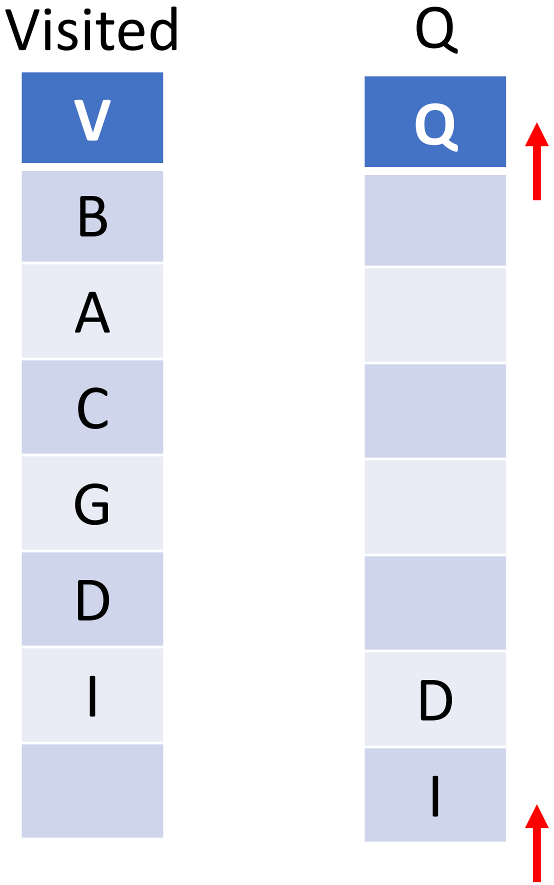
B -> J ?




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

x = G

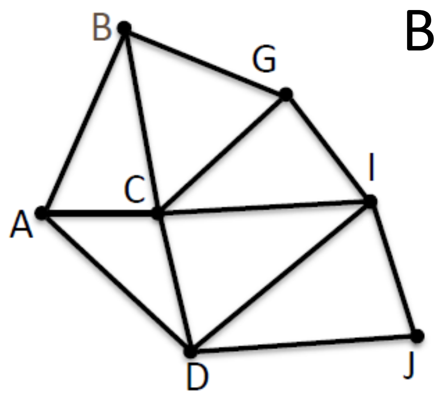
X_G = J



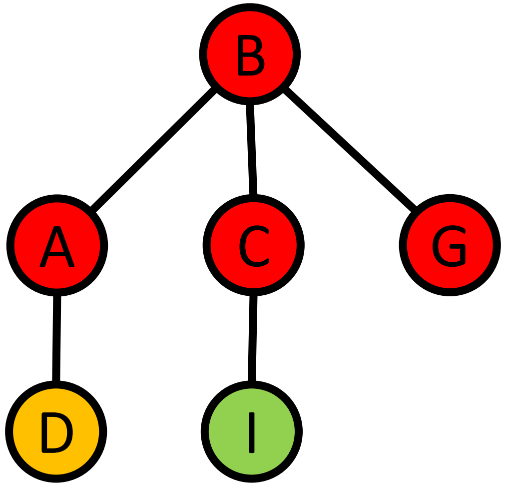
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?



FORWARD_SEARCH
```
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

x = D

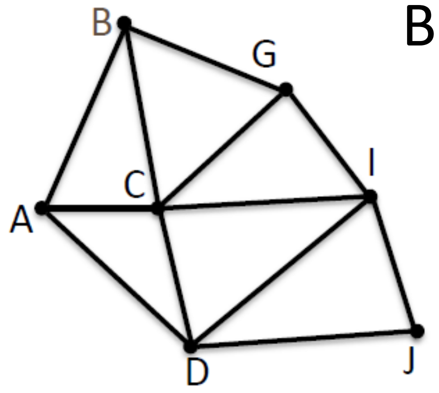
X_G = J

Visited	Q
V	**Q**
B	
A	
C	
G	
D	
I	
	I

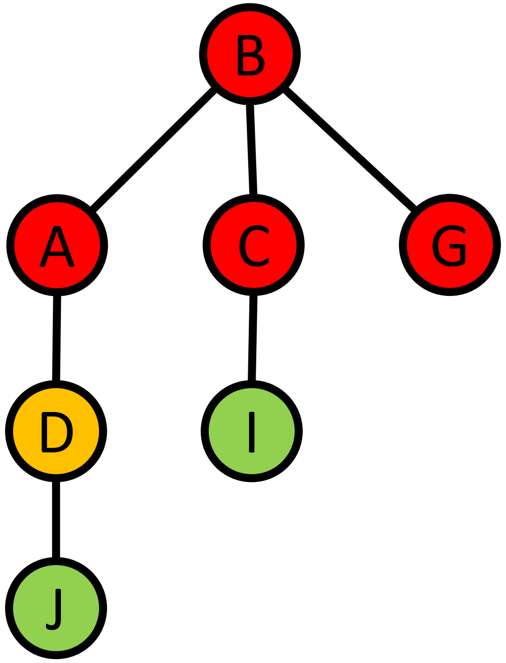
For BFS, Q is FIFO (First In First Out)
- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

UNSW SYDNEY

Breadth-first search (BFS)



B -> J ?




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

x = D

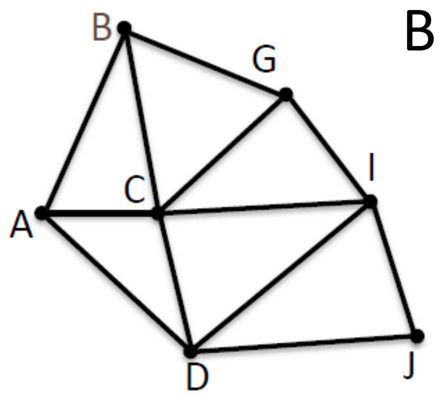
X_G = J

Visited	Q
V	Q
B	
A	
C	
G	
D	
I	I
J	J

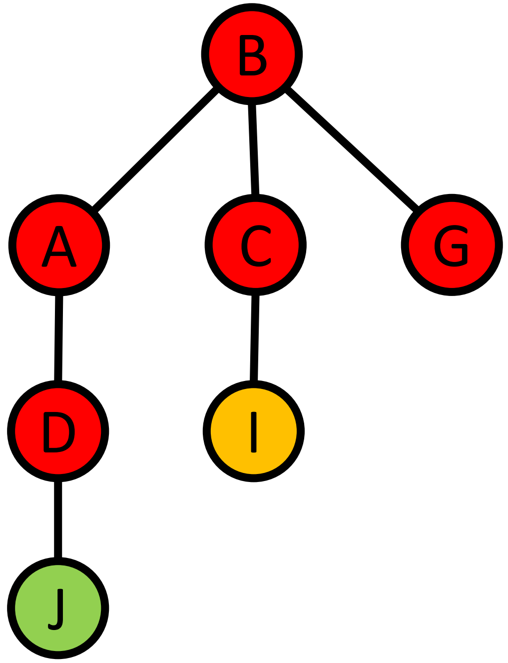
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?



x = I

X_G = J

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

Visited

V
B
A
C
G
D
I
J

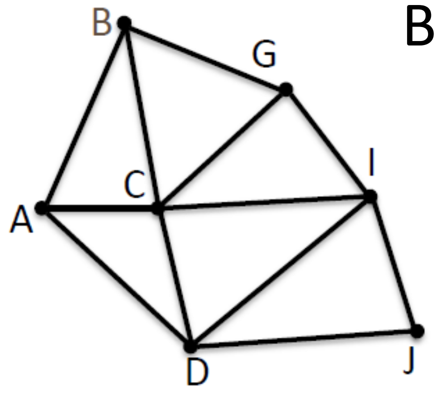
Q

Q
J

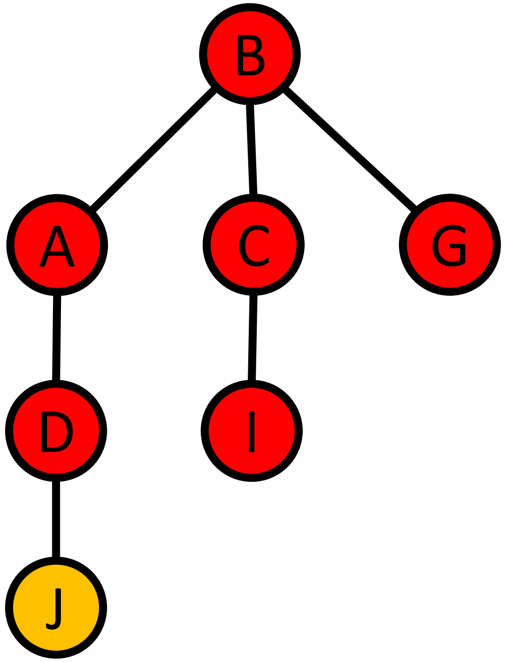
For BFS, Q is FIFO (First In First Out)

- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?



x = J

X_G = J

FORWARD_SEARCH

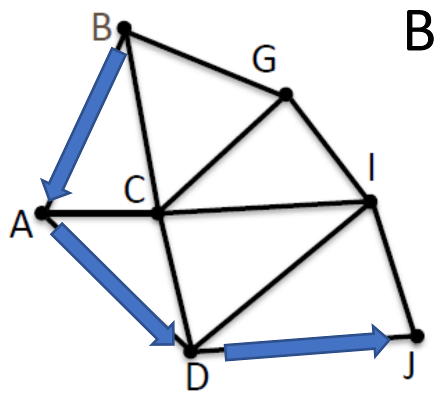
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if x' not visited
            Mark x' as visited
            $Q.Insert(x')$
        else
            Resolve duplicate x'
return FAILURE

Visited	Q
V	**Q**
B	
A	
C	
G	
D	
I	
J	

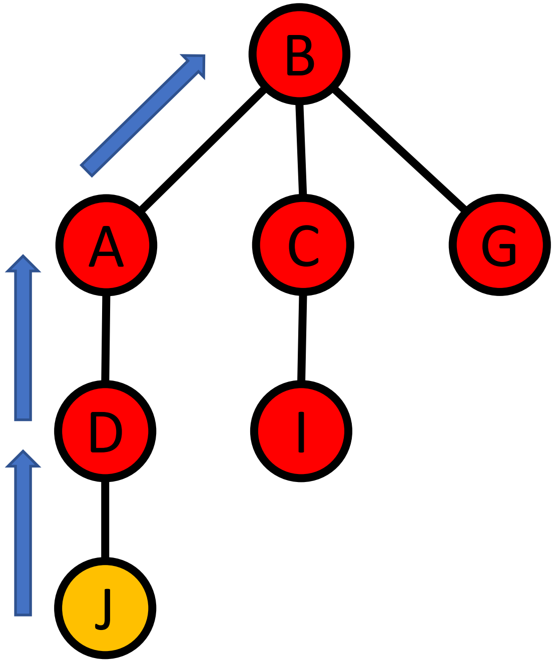
For BFS, Q is FIFO (First In First Out)

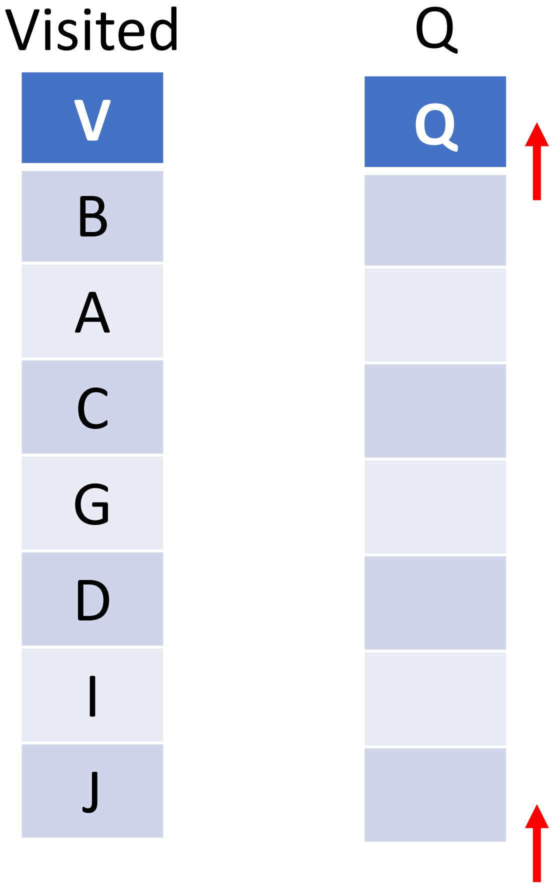
- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

Breadth-first search (BFS)



B -> J ?






```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

For BFS, Q is FIFO (First In First Out)

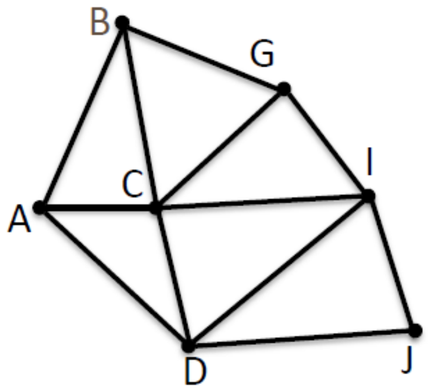
- push (Q.Insert) onto the back
- pop (Q.GetFirst) from the front

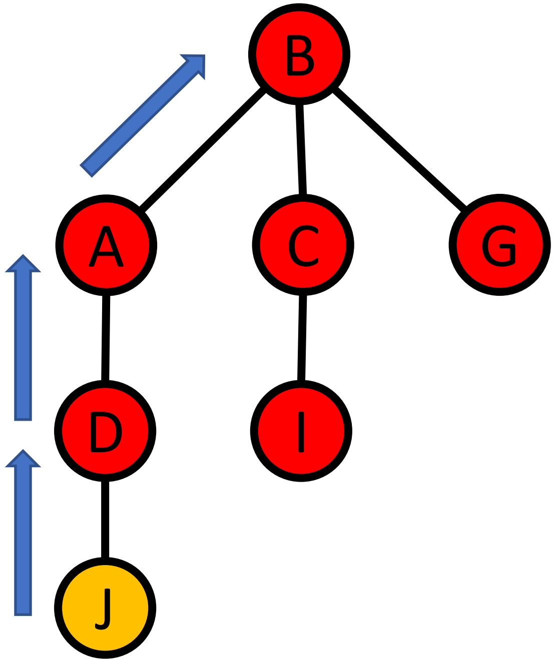
Two important concepts for graph search algorithms

- Complete: Always find a valid path if there exists one
- Optimal: When a path is found, it is always the shortest

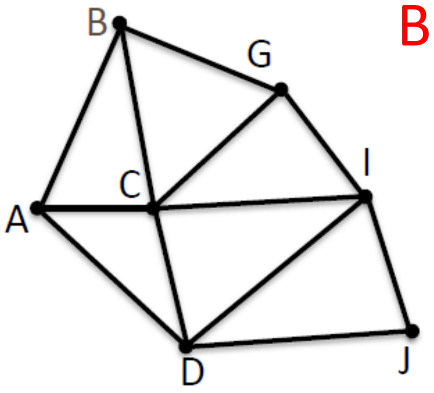
Breadth-first search (BFS)

- Complete (will find a valid path if it exists)
- Optimal (guarantee the path found to be the shortest)
  - First solution found is the optimal path



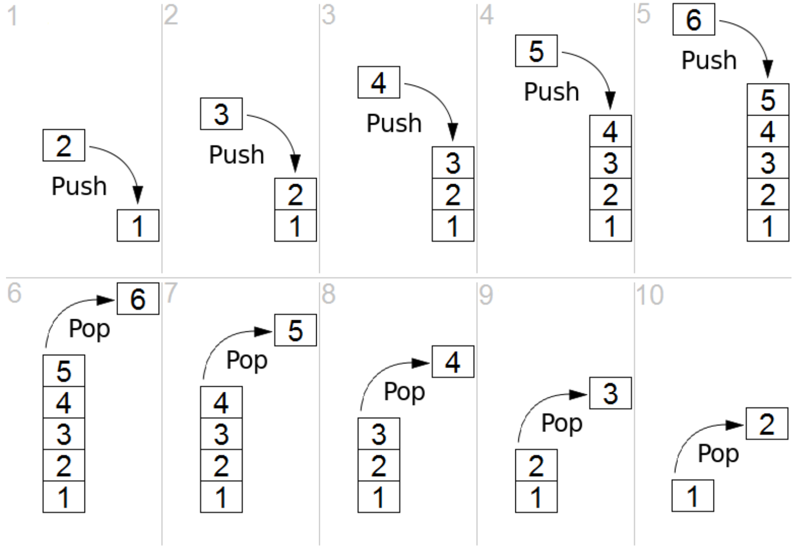


Depth-first search (DFS)

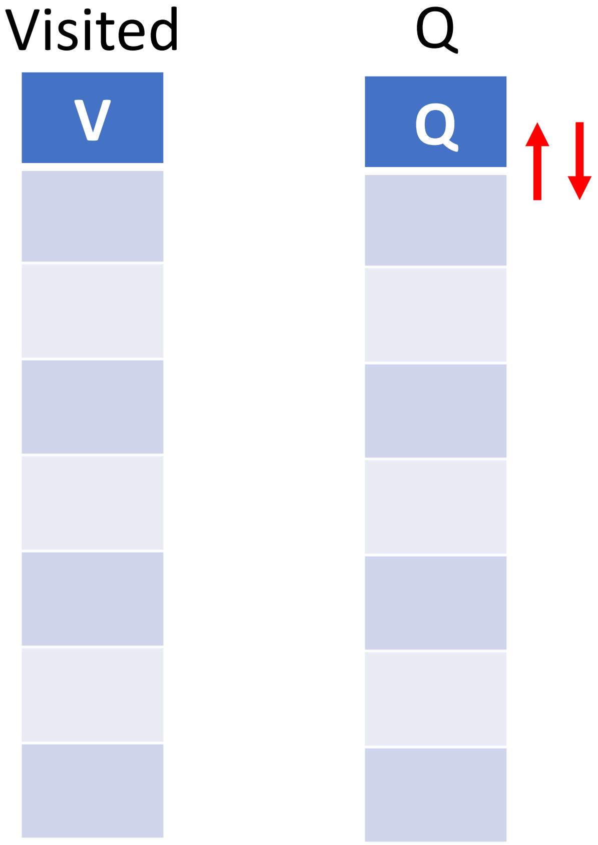


B -> J ?

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```



LIFO

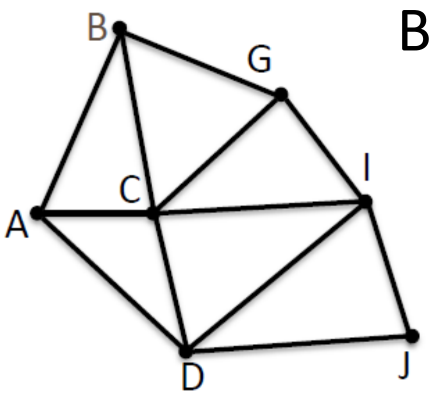


For DFS, Q is LIFO (Last In First Out)

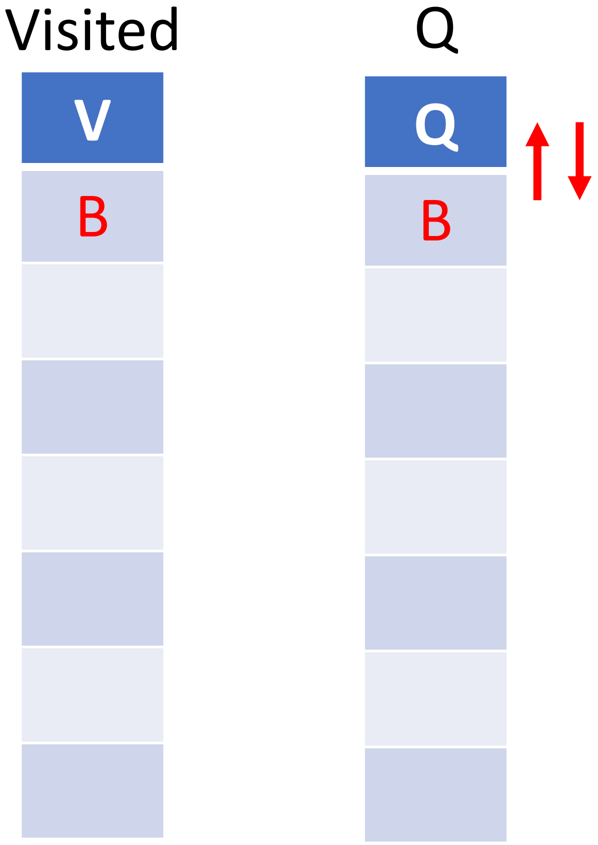
- push (Q.Insert) onto the front
- pop (Q.GetFirst) from the front

https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Depth-first search (DFS)



B -> J ?



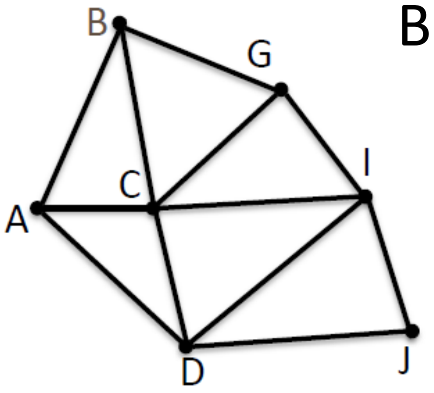

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

For DFS, Q is LIFO (Last In First Out)

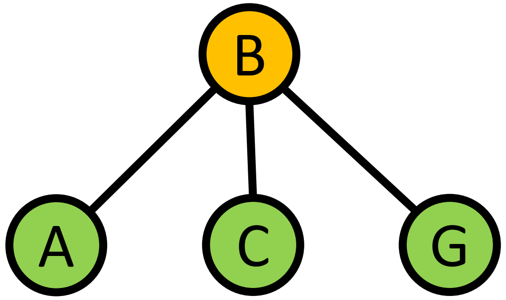
- push (Q.Insert) onto the front
- pop (Q.GetFirst) from the front

https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Depth-first search (DFS)



B -> J ?



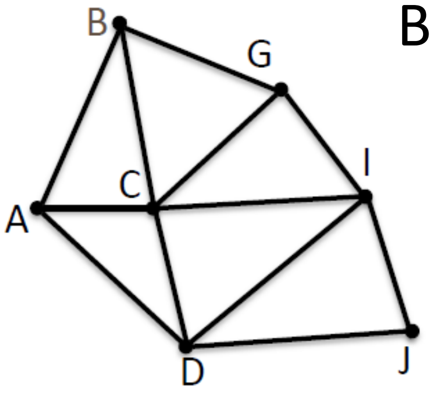

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

Visited	Q
V	Q
B	A
A	C
C	G
G	

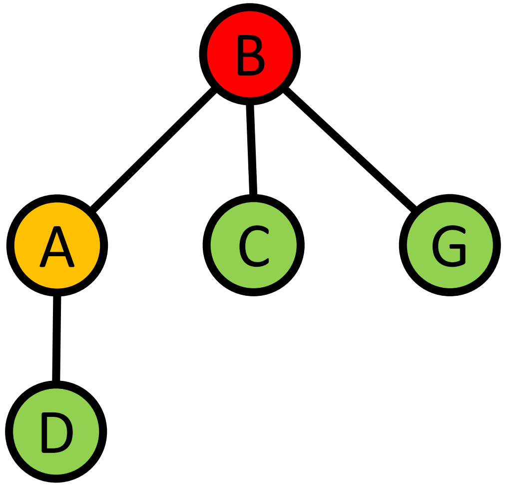
For DFS, Q is LIFO (Last In First Out)

- push (Q.Insert) onto the front
- pop (Q.GetFirst) from the front

Depth-first search (DFS)



B -> J ?




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

Visited

V
B
A
C
G
D

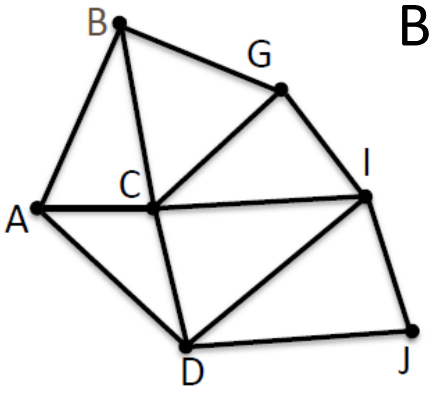
Q

Q
D
C
G

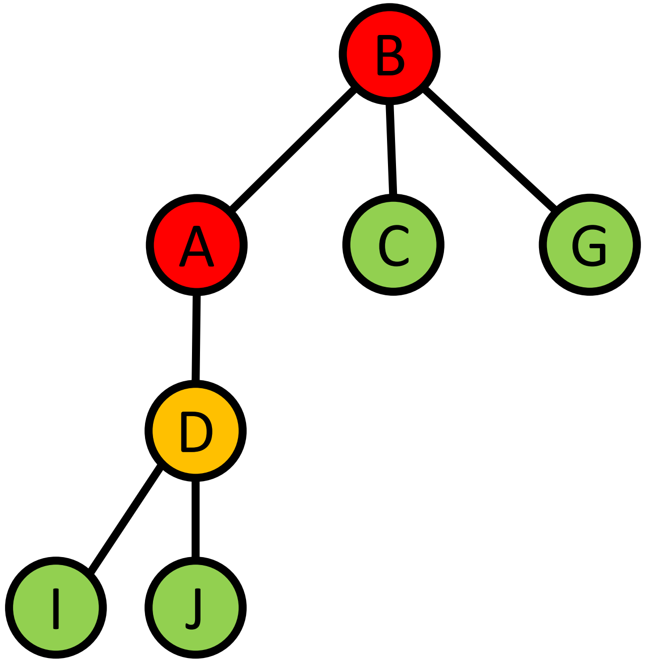
For DFS, Q is LIFO (Last In First Out)

- push (Q.Insert) onto the front
- pop (Q.GetFirst) from the front

Depth-first search (DFS)



B -> J ?



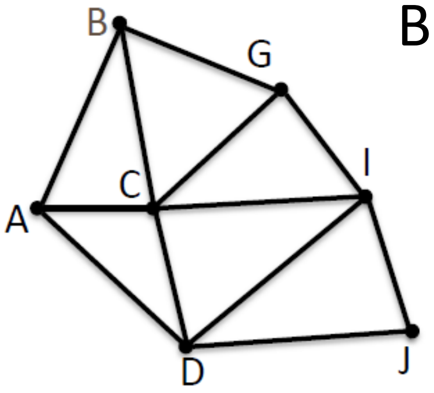

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

Visited	Q
V	Q
B	I
A	J
C	C
G	G
D	
I	
J	

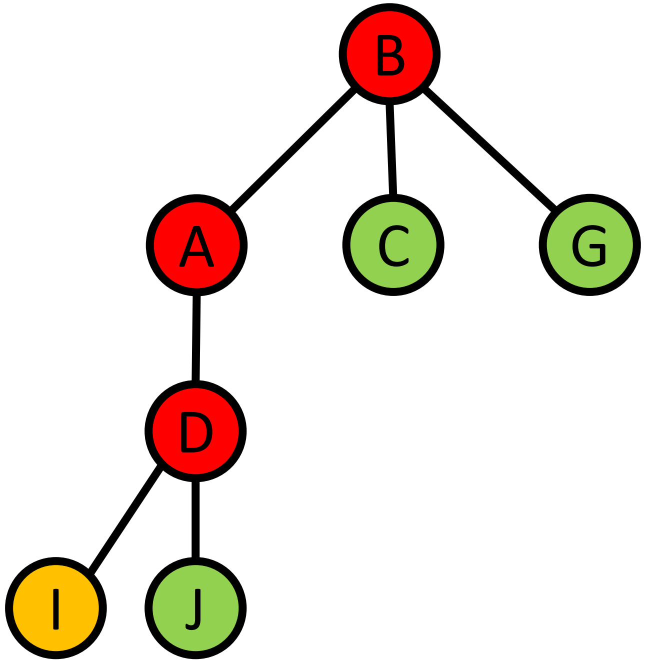
For DFS, Q is LIFO (Last In First Out)

- push (Q.Insert) onto the front
- pop (Q.GetFirst) from the front

Depth-first search (DFS)



B -> J ?



Visited

V
B
A
C
G
D
I
J

Q

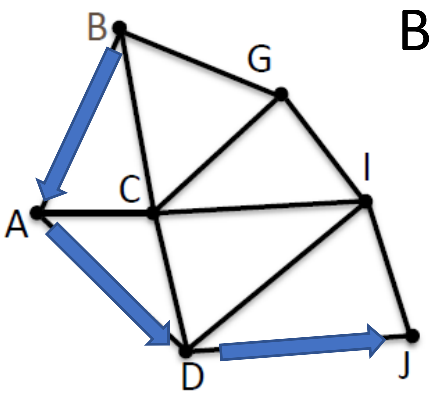
Q
J
C
G

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

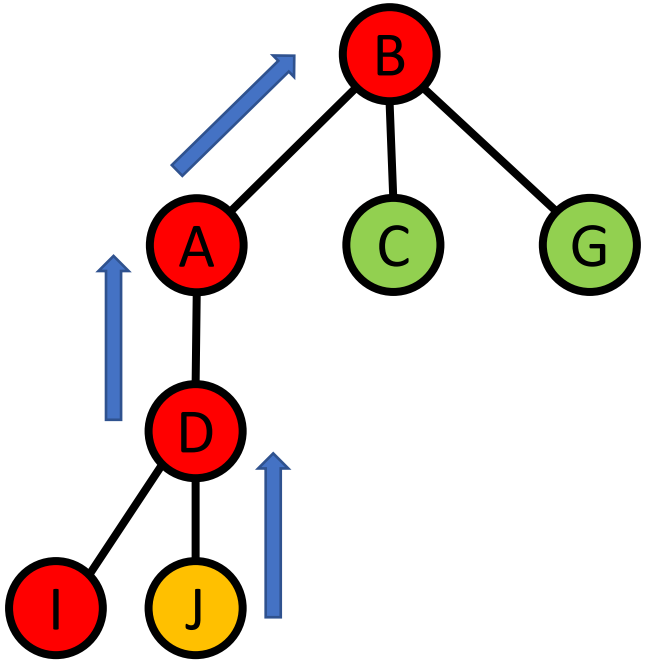
For DFS, Q is LIFO (Last In First Out)

- push (Q.Insert) onto the front
- pop (Q.GetFirst) from the front

Depth-first search (DFS)



B -> J ?



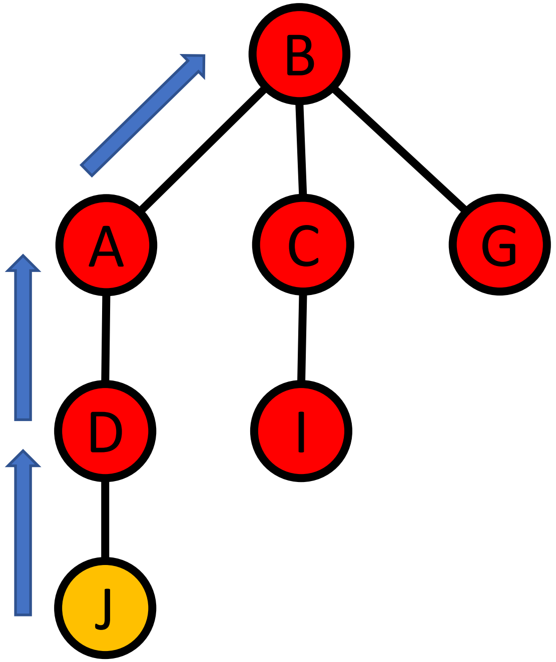
Visited	Q
V	Q
B	
A	
C	
G	
D	
I	
J	

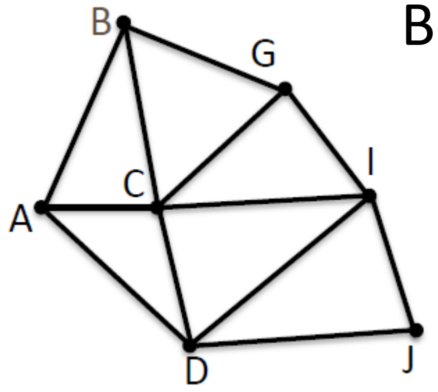
FORWARD_SEARCH
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$
    return SUCCESS
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if x' not visited
      Mark x' as visited
      $Q.Insert(x')$
    else
      Resolve duplicate x'
return FAILURE

For DFS, Q is LIFO (Last In First Out)

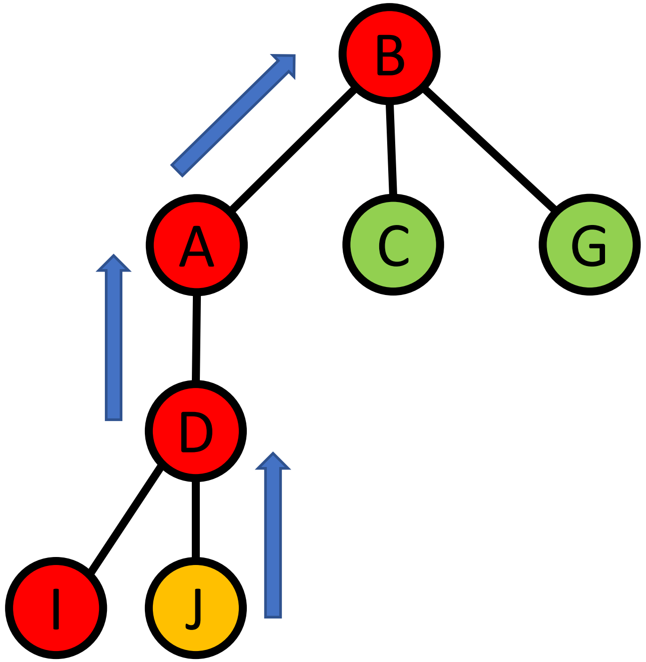
- push (Q.Insert) onto the front
- pop (Q.GetFirst) from the front

BFS vs DFS

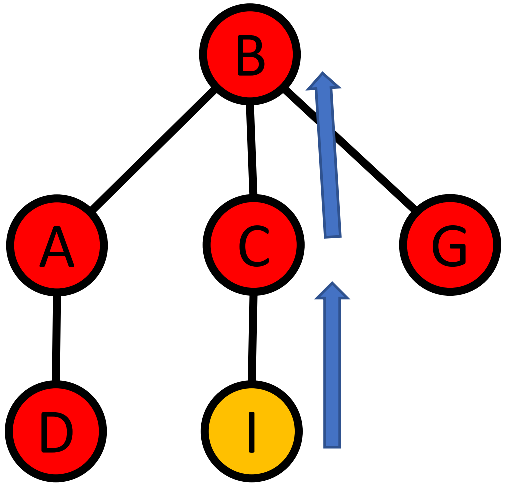


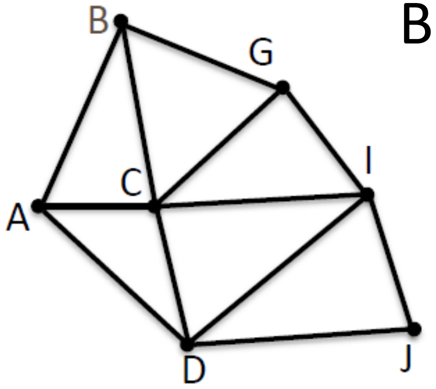


B -> J ?

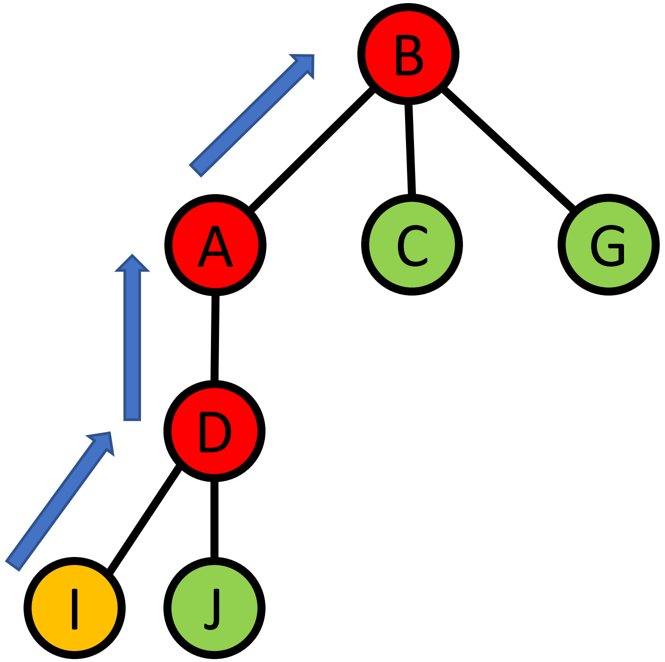


BFS vs DFS





B -> I ?



Depth-first search (DFS)

- Not complete for infinite trees (may explore an incorrect branch infinitely deep, never come back up)

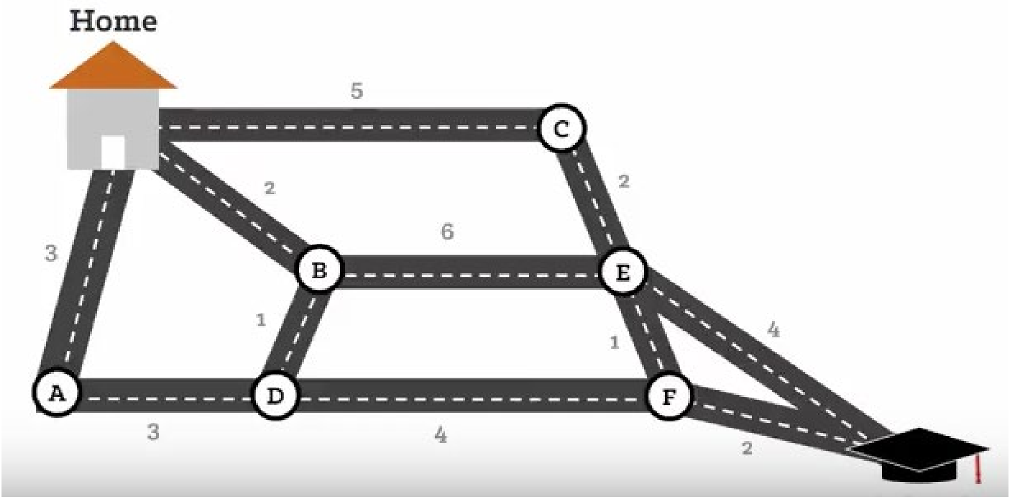
- Not optimal (cannot guarantee path found is the shortest)

- Lower memory footprint than BFS with high-branching

- Both BFS and DFS are simple to implement, but might be inefficient. More complex algorithms are faster, but generally more difficult to implement

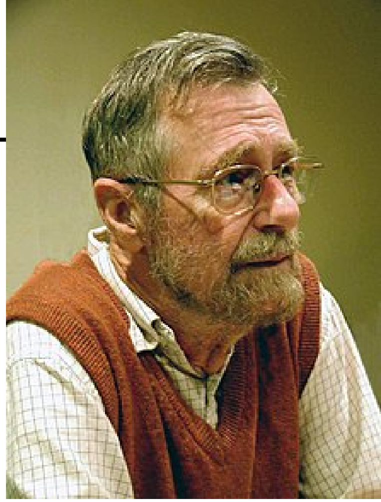
Dijkstra's Algorithm

- What if edges have non-uniform weights (costs)?



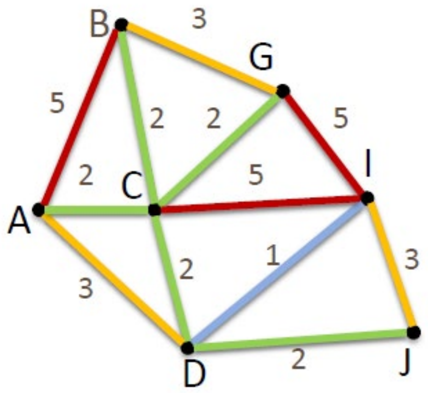
- Essentially, BFS considering edge costs

- One of the most commonly used routing algorithms in graph traversal problems



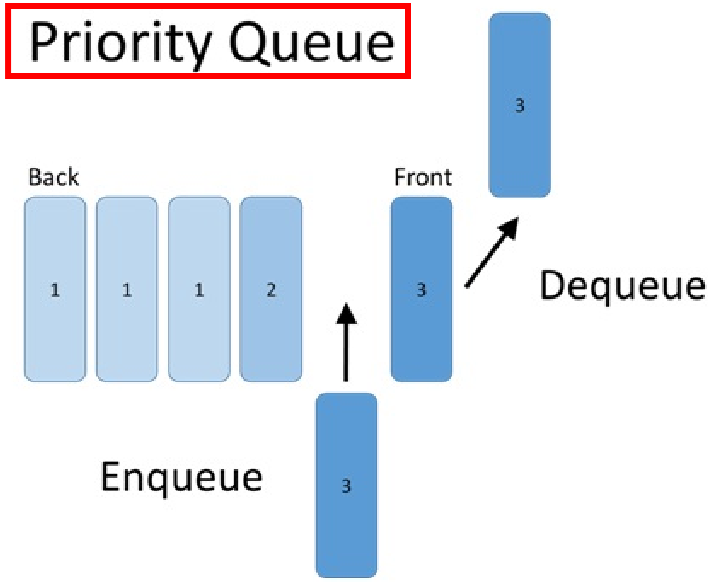
Edsger W Dijkstra
1930-2002

Dijkstra's Algorithm




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

$\boxed{\text{Priority Queue}}$



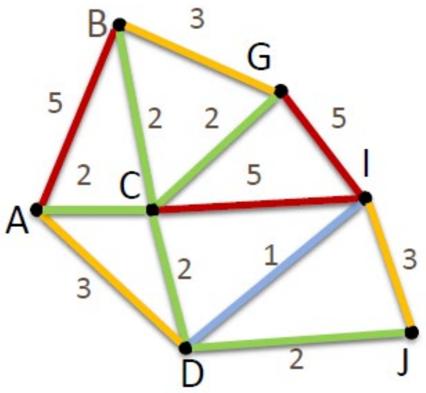
Visited	Q
V	Q

For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

https://www.includehelp.com/ds/implementation-of-priority-queue-using-linked-list.aspx

Dijkstra's Algorithm




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

B(0)

Visited

V
B(0)

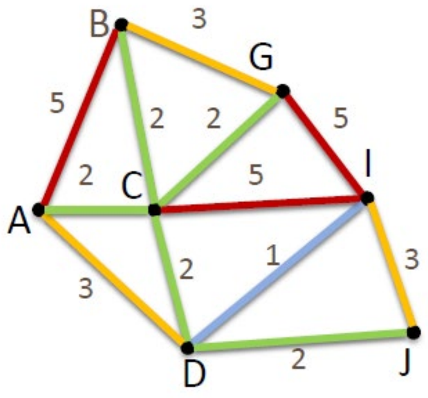
Q

Q
B(0)

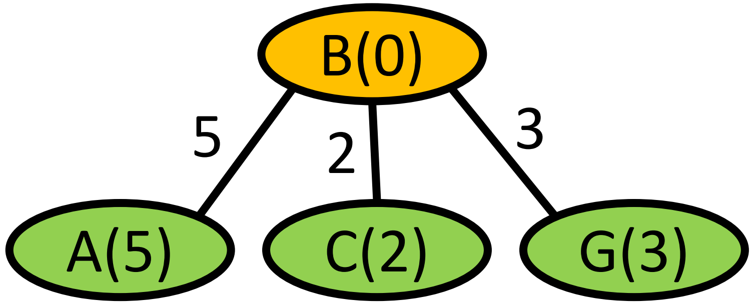
For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm



FORWARD_SEARCH
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if x' not visited
            Mark x' as visited
            $Q.Insert(x')$
        else
            Resolve duplicate x'
return FAILURE



Visited

V
B(0)
C(2)
G(3)
A(5)

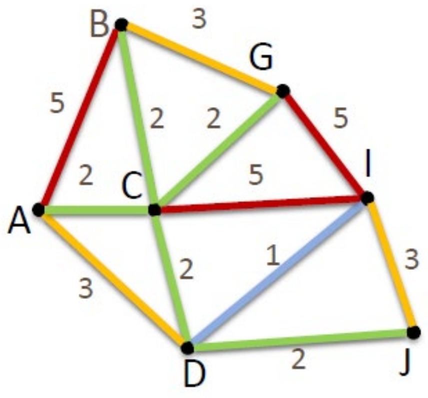
Q

Q
C(2)
G(3)
A(5)

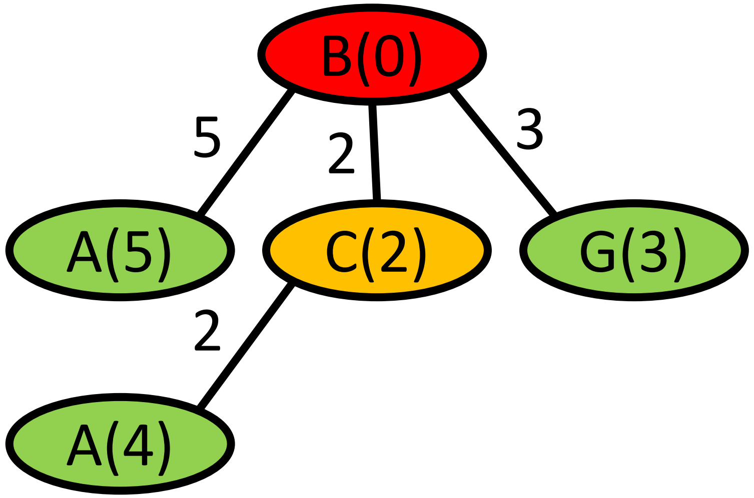
For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm



FORWARD_SEARCH
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$
    return SUCCESS
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if x' not visited
      Mark x' as visited
      $Q.Insert(x')$
    else
      Resolve duplicate x'
return FAILURE



Visited

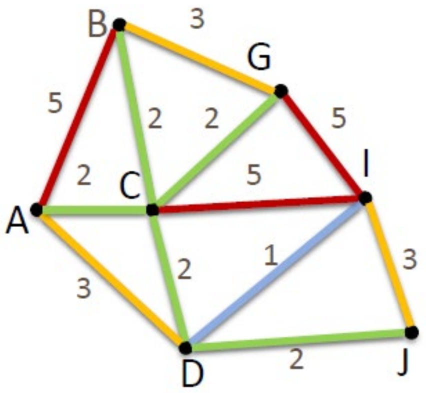
V
B(0)
C(2)
G(3)
A(4)

Q

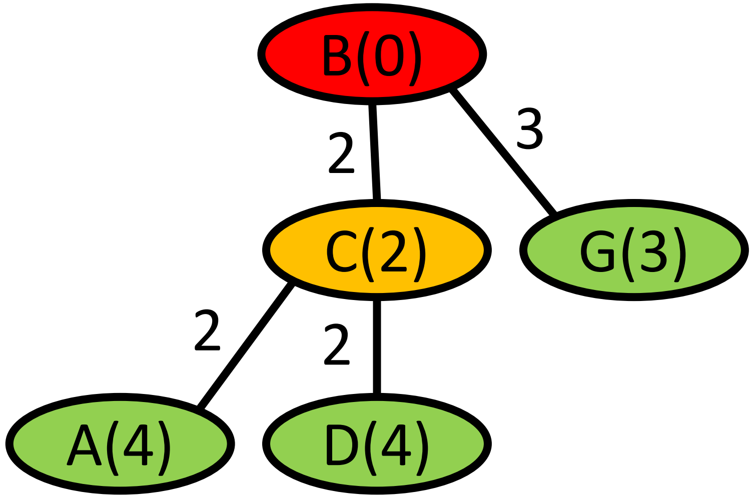
Q
G(3)
A(4)

For DA, Q is Priority Queue
- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

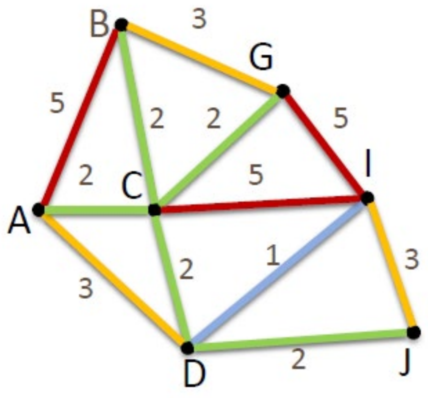


Visited	Q
V	Q
B(0)	
C(2)	
G(3)	
A(4)	
D(4)	G(3)
	A(4)
	D(4)

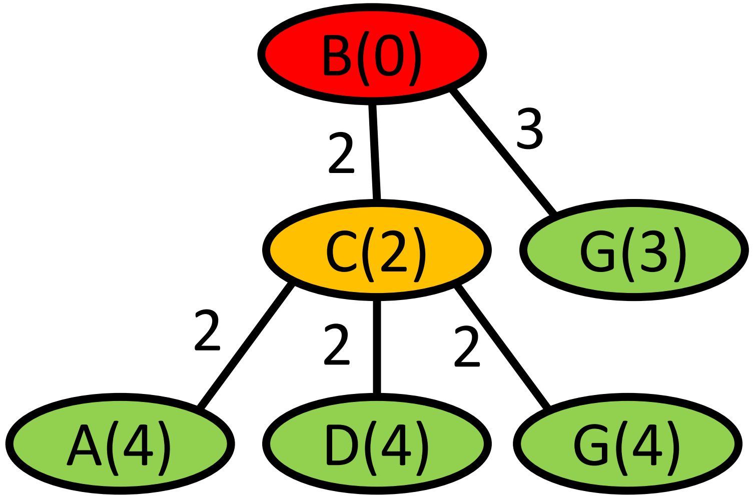
For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```



Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)

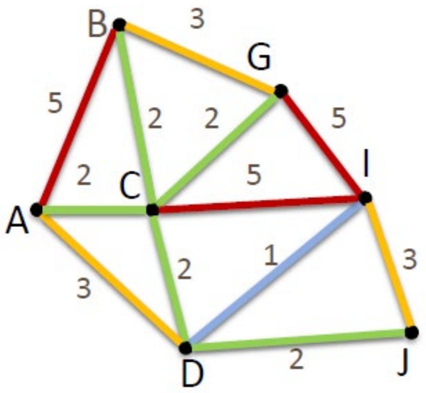
Q

Q
G(3)
A(4)
D(4)

For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm



FORWARD_SEARCH

$Q.Insert(x_I)$ and mark x_I as visited

while Q not empty **do**

  $x \leftarrow Q.GetFirst()$

  if $x \in X_G$

    return SUCCESS

  forall $u \in U(x)$

    $x' \leftarrow f(x, u)$

    if x' not visited

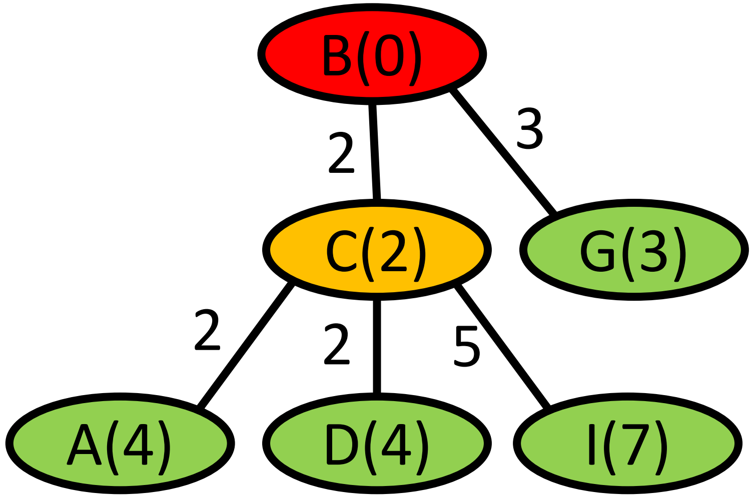
      Mark x' as visited

      $Q.Insert(x')$

    else

      Resolve duplicate x'

return FAILURE

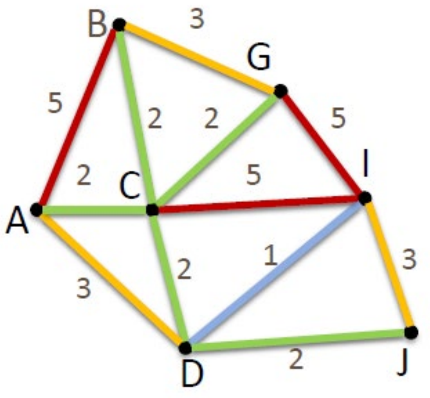


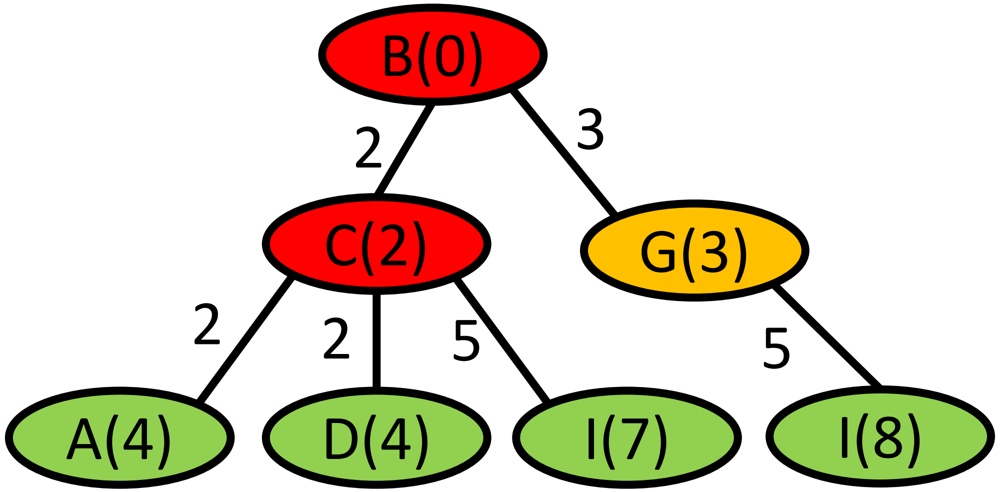
Visited	Q
V	**Q**
B(0)	
C(2)	
G(3)	
A(4)	G(3)
D(4)	A(4)
I(7)	D(4)
	I(7)

For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm





Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)
I(7)

Q

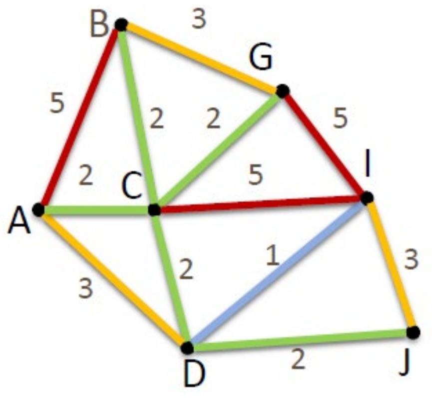
Q
A(4)
D(4)
I(7)

```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

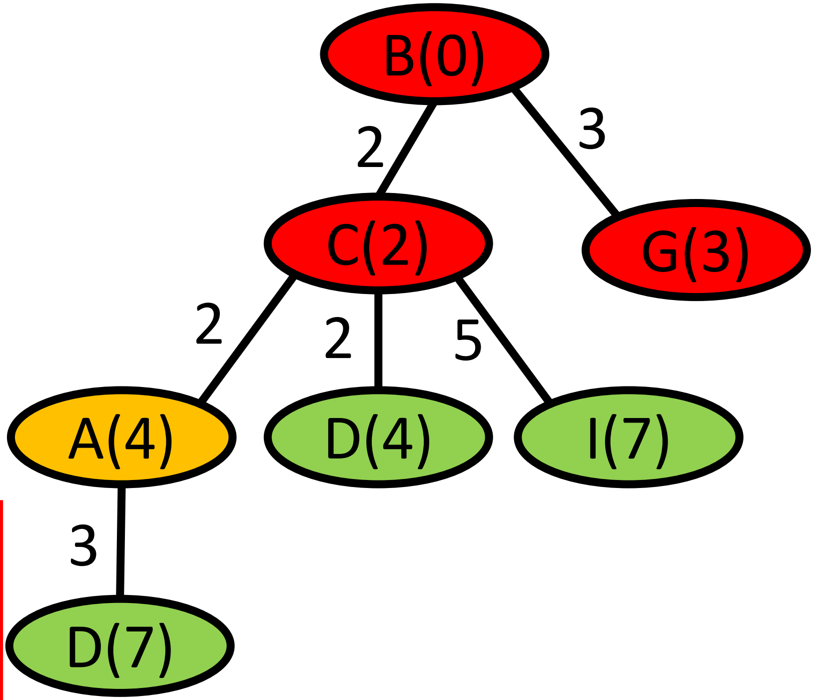
For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```



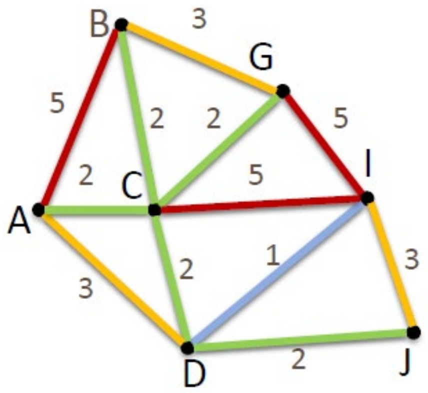
Visited	Q
V	**Q**
B(0)	
C(2)	
G(3)	
A(4)	
D(4)	
I(7)	D(4)
	I(7)

For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

UNSW SYDNEY

Dijkstra's Algorithm



FORWARD_SEARCH

$Q.Insert(x_I)$ and mark x_I as visited

while Q not empty **do**

  $x \leftarrow Q.GetFirst()$

  if $x \in X_G$

    return SUCCESS

  forall $u \in U(x)$

    $x' \leftarrow f(x,u)$

    if x' not visited

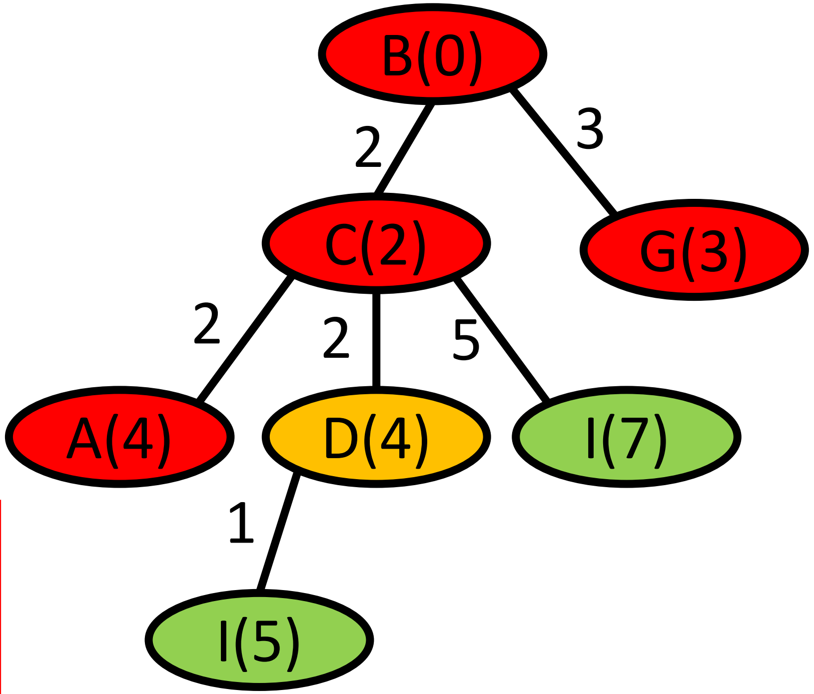
      Mark x' as visited

      $Q.Insert(x')$

    else

      Resolve duplicate x'

return FAILURE

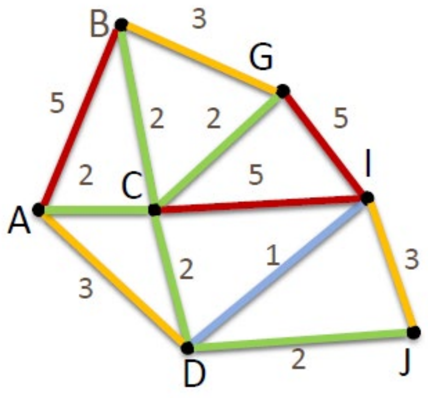


Visited	Q
V	Q
B(0)	
C(2)	
G(3)	
A(4)	
D(4)	
I(5)	
	I(5)

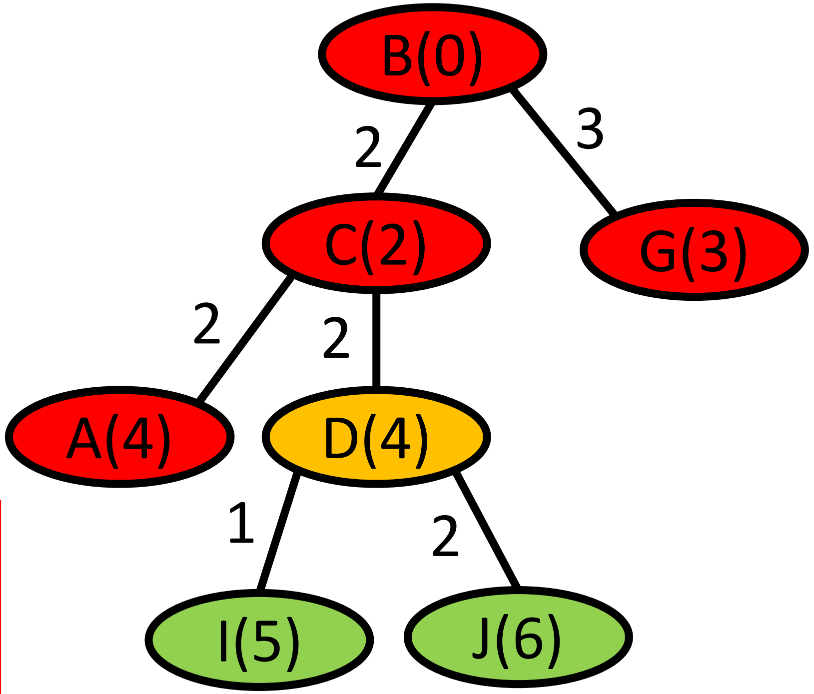
For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm



FORWARD_SEARCH
$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$
    return SUCCESS
  forall $u \in U(x)$
    $x' \leftarrow f(x, u)$
    if x' not visited
      Mark x' as visited
      $Q.Insert(x')$
    else
      Resolve duplicate x'
return FAILURE

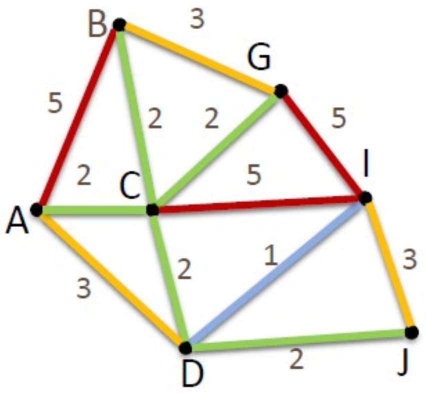


Visited	Q
V	**Q**
B(0)	
C(2)	
G(3)	
A(4)	
D(4)	
I(5)	I(5)
J(6)	J(6)

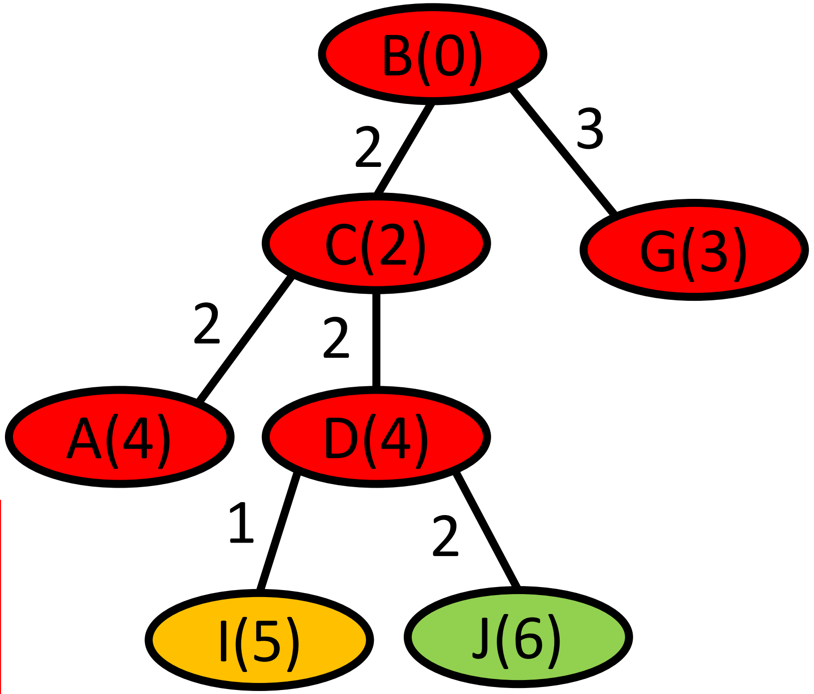
For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm




```
FORWARD_SEARCH
Q.Insert(x_I) and mark x_I as visited
while Q not empty do
    x ← Q.GetFirst()
    if x ∈ X_G
        return SUCCESS
    forall u ∈ U(x)
        x' ← f(x, u)
        if x' not visited
            Mark x' as visited
            Q.Insert(x')
        else
            Resolve duplicate x'
return FAILURE
```

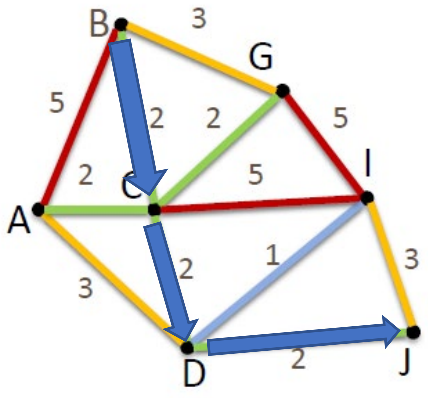


Visited	Q
V	**Q**
B(0)	
C(2)	
G(3)	
A(4)	
D(4)	
I(5)	
J(6)	J(6)

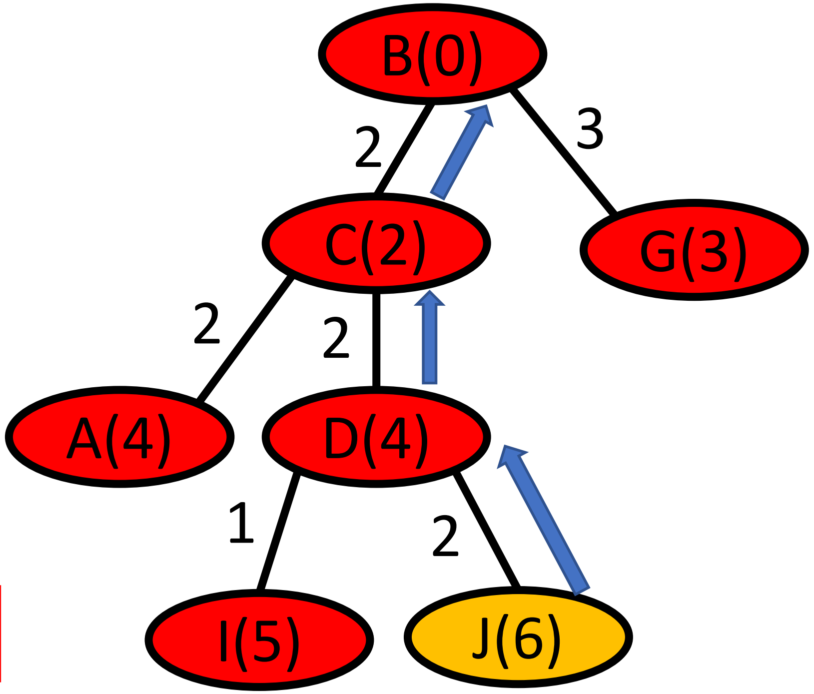
For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

Dijkstra's Algorithm




```
FORWARD_SEARCH
  Q.Insert(x_I) and mark x_I as visited
  while Q not empty do
      x ← Q.GetFirst()
      if x ∈ X_G
          return SUCCESS
      forall u ∈ U(x)
          x' ← f(x, u)
          if x' not visited
              Mark x' as visited
              Q.Insert(x')
          else
              Resolve duplicate x'
  return FAILURE
```



Visited	Q
V	**Q**
B(0)	
C(2)	
G(3)	
A(4)	
D(4)	
I(5)	
J(6)	

For DA, Q is Priority Queue

- push (Q.Insert) by arrive cost
- pop (Q.GetFirst) from the front

UNSW SYDNEY

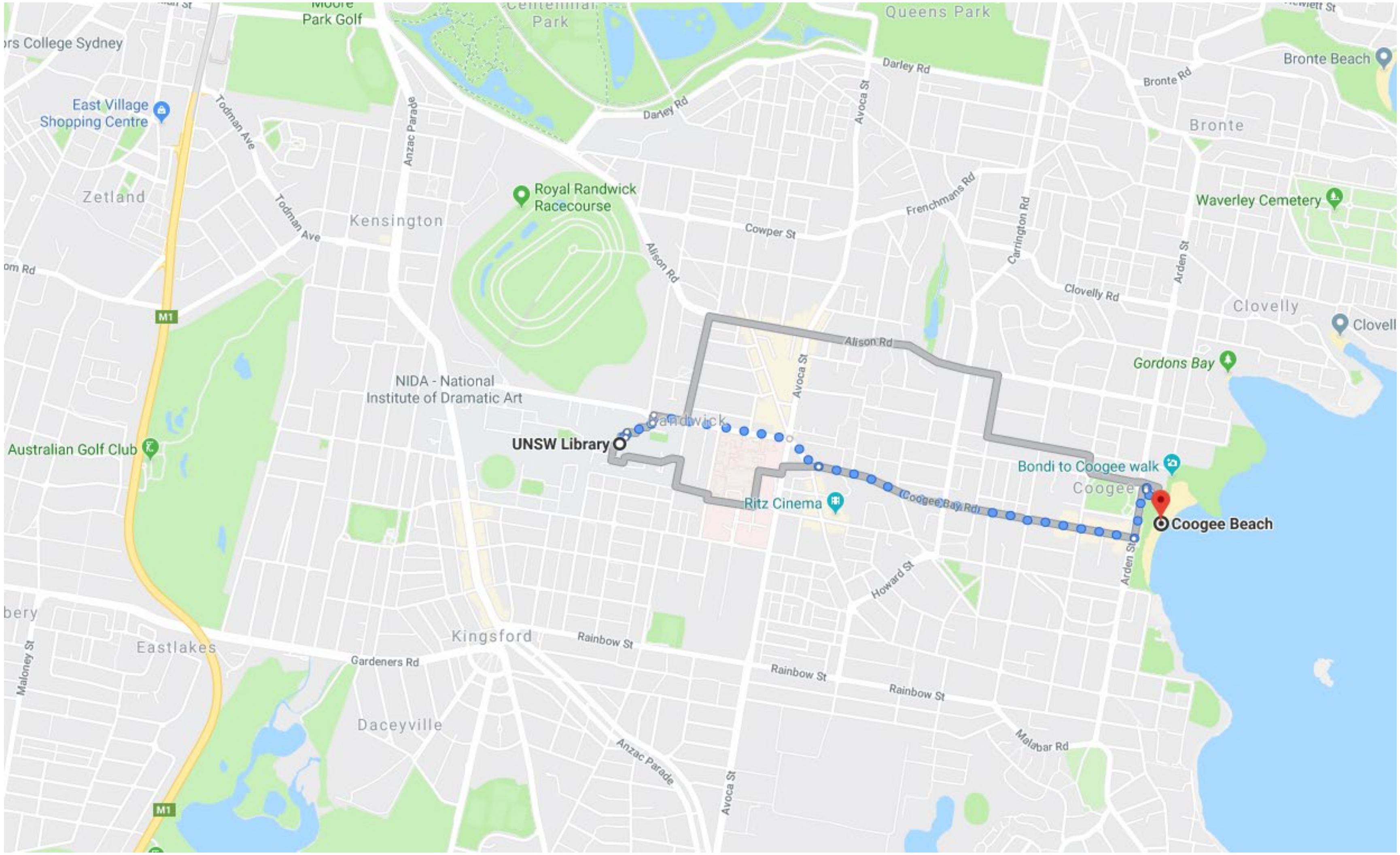
Dijkstra’s Algorithm

- At the end, we can recover the lowest-cost route from the start to any node (or any node with cost < goal if we terminate at a goal)

- Not difficult to implement, but requires a little bit of careful management with the priority queue

- Doesn’t really know the goal exists until it reaches it
  - Can we guide the search to expand nodes that are closer to the goal earlier?
  - Can we do it without breaking the condition that a node is only accepted with its lowest cost of arrival?

Dijkstra's Algorithm -> A* Algorithm

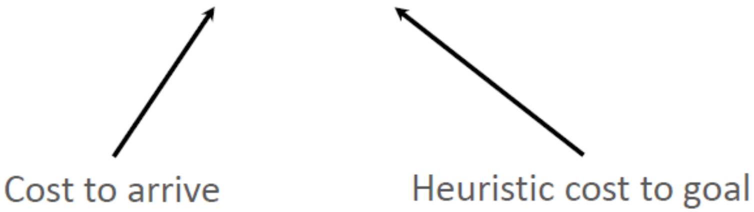


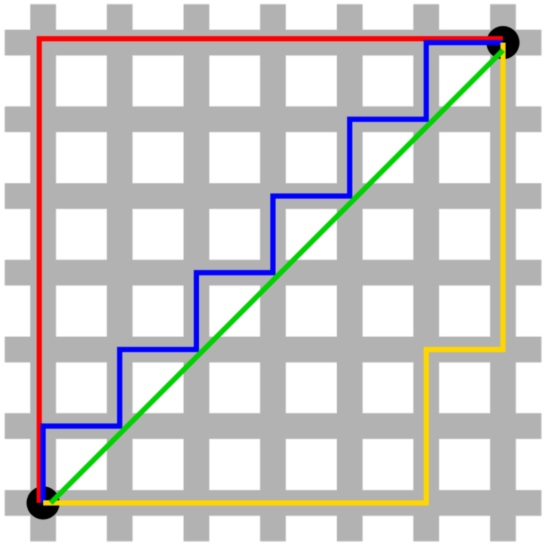
A* Heuristic Search

- A* is an extension of Dijkstra's algorithm, and achieves faster performance by using heuristics

- Heuristics:
  - Any *optimistic estimate* of how close a state is to a goal
  - Designed for a particular search problem
  - Example: Euclidean distance, Manhattan distance, etc.

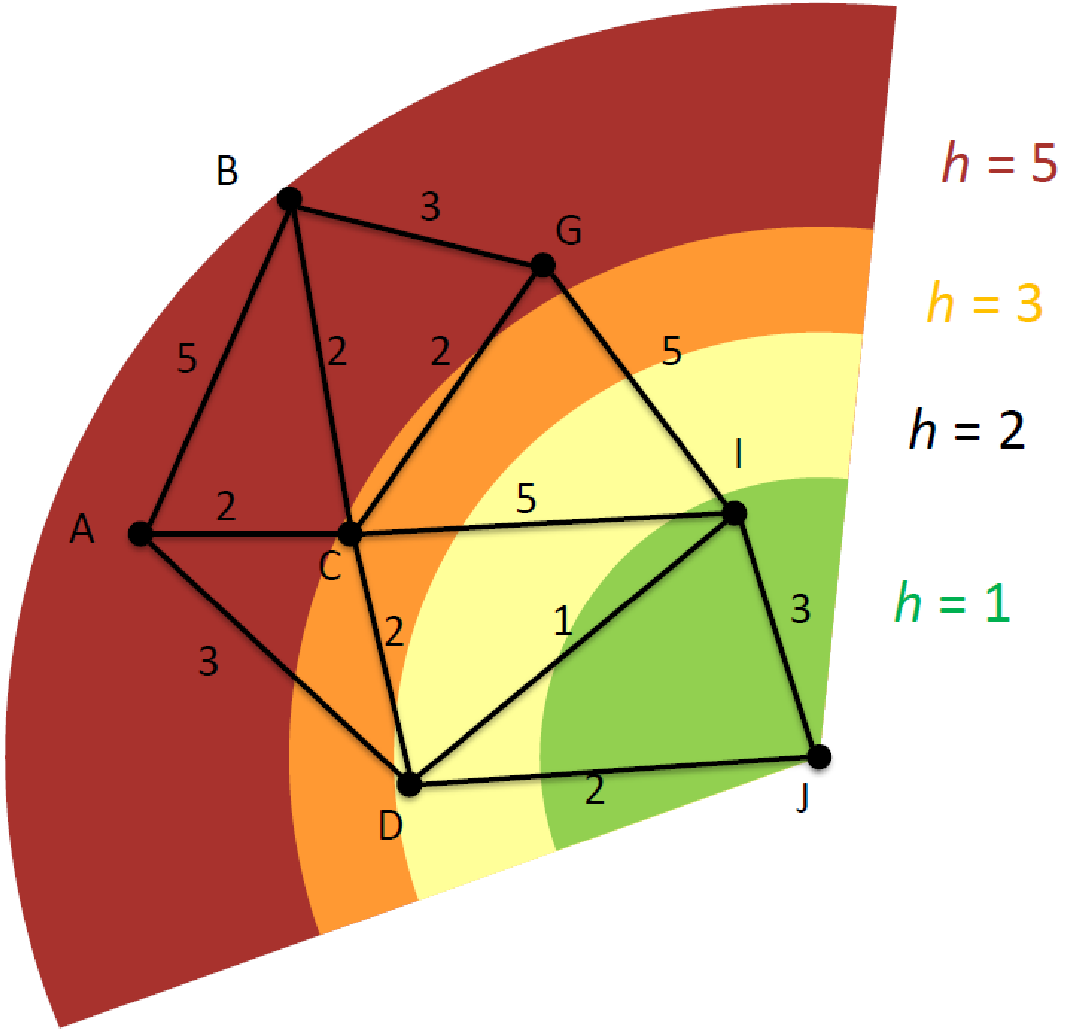
- A* Priority: $f(n) = g(n) + h(n)$



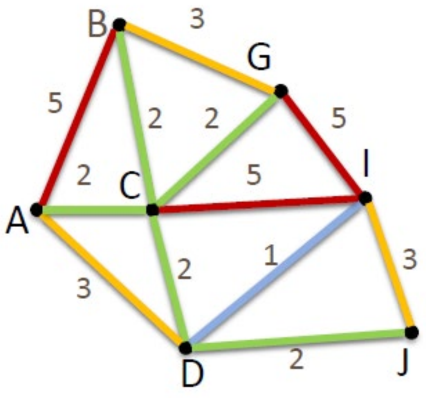


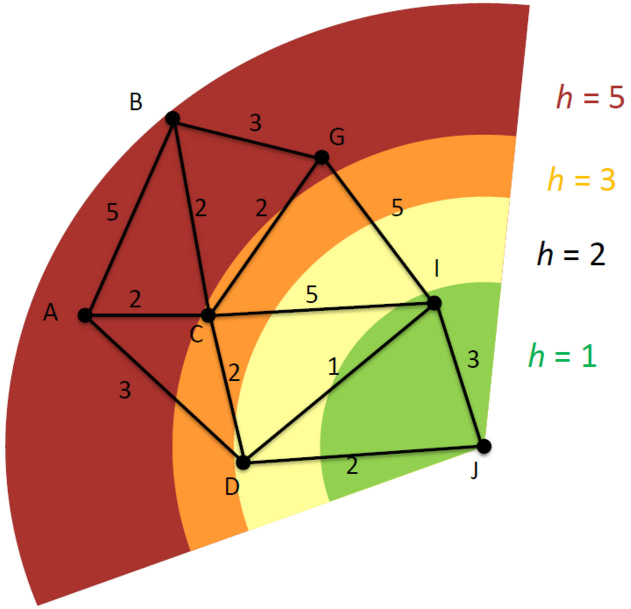
Green: Euclidean distance
Red/Yellow/Blue: Manhattan distance

A* Heuristic - Example



A* Algorithm





B(0)

Visited

V
B(0)

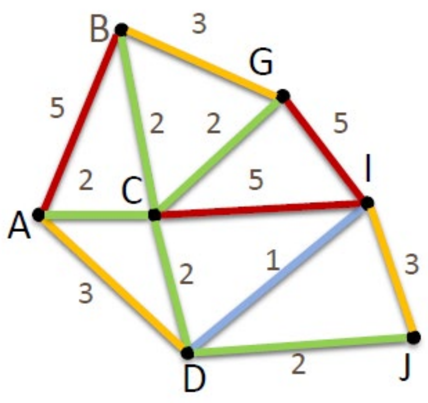
Q

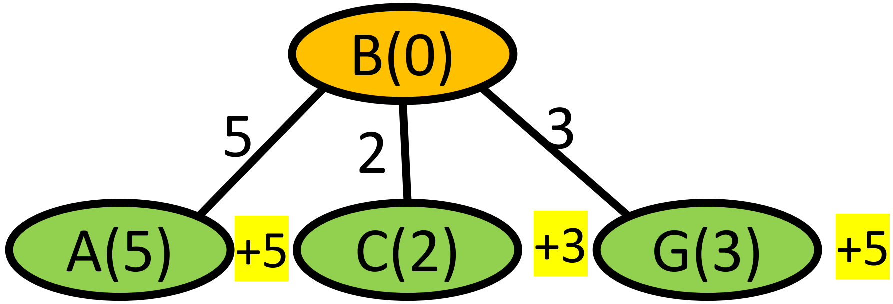
Q
B(0)

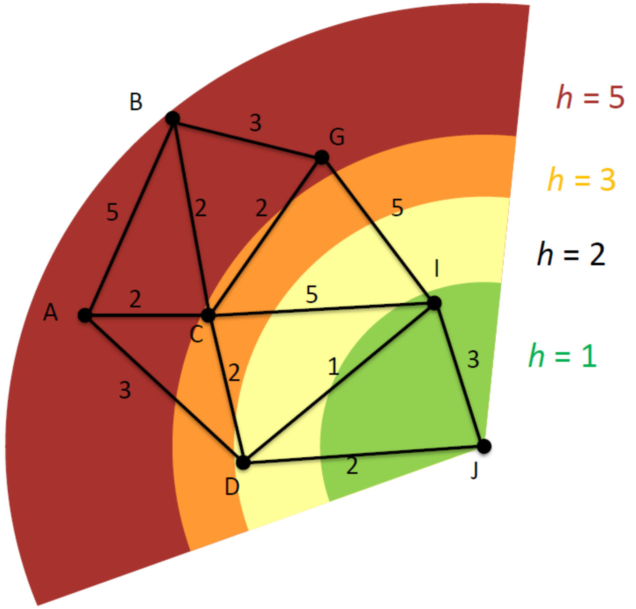
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm





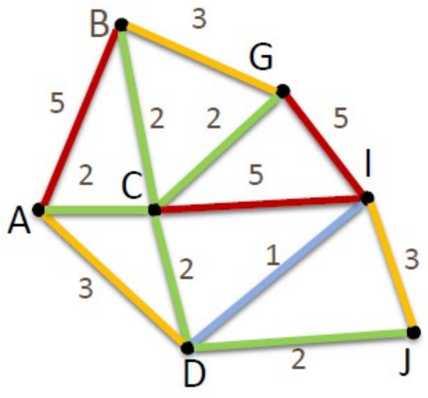


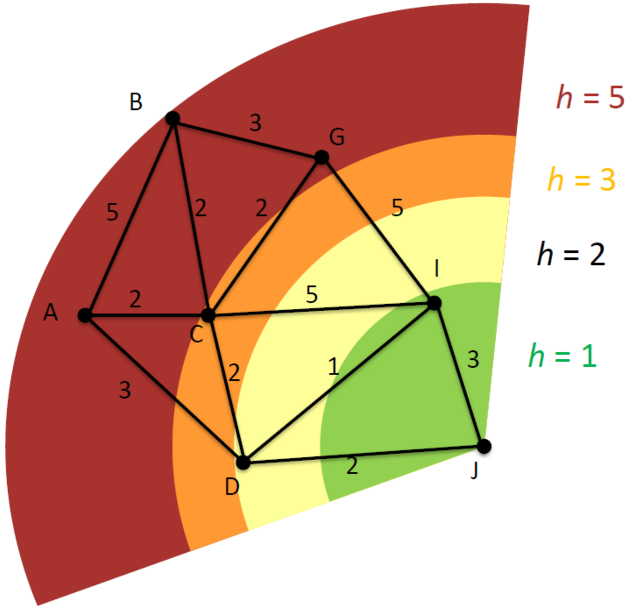
Visited	Q
V	**Q**
B(0)	
C(2)	
G(3)	
A(5)	
	C(2+3)
	G(3+5)
	A(5+5)

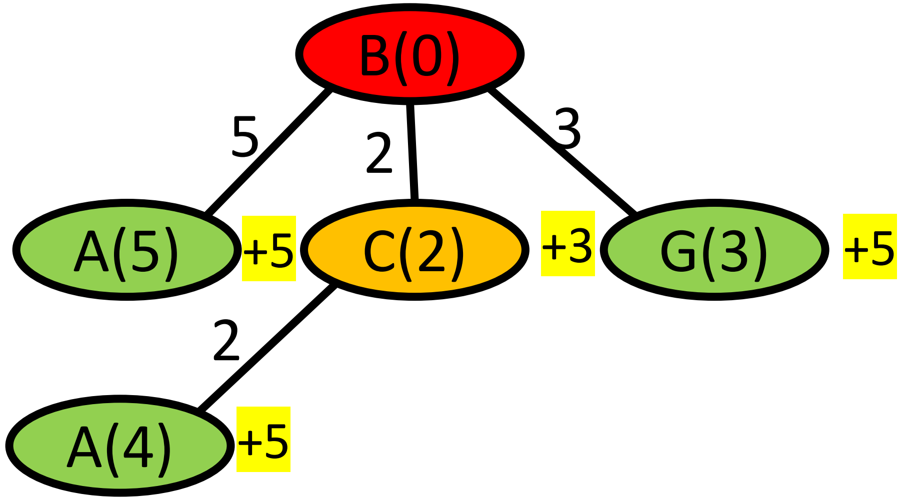
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)

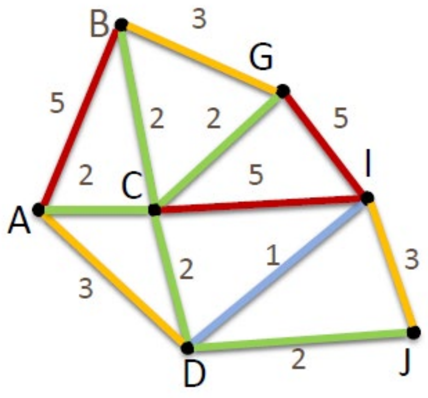
Q

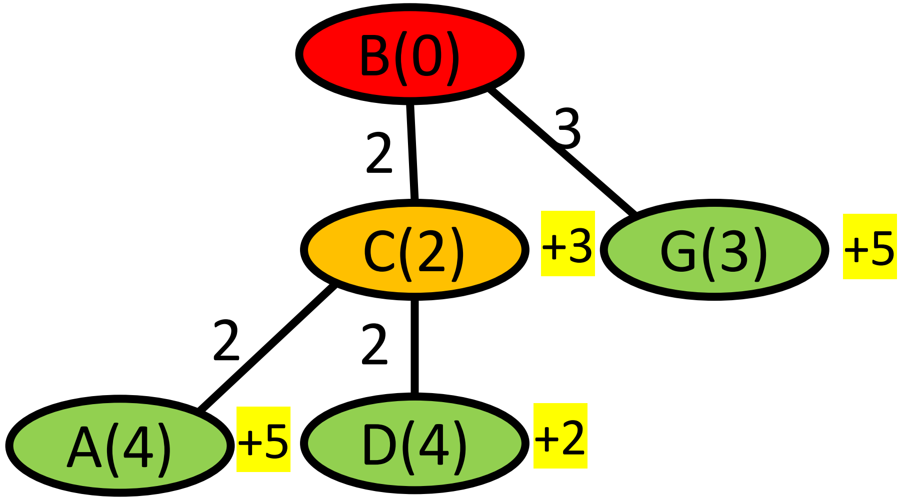
Q
G(3+5)
A(4+5)

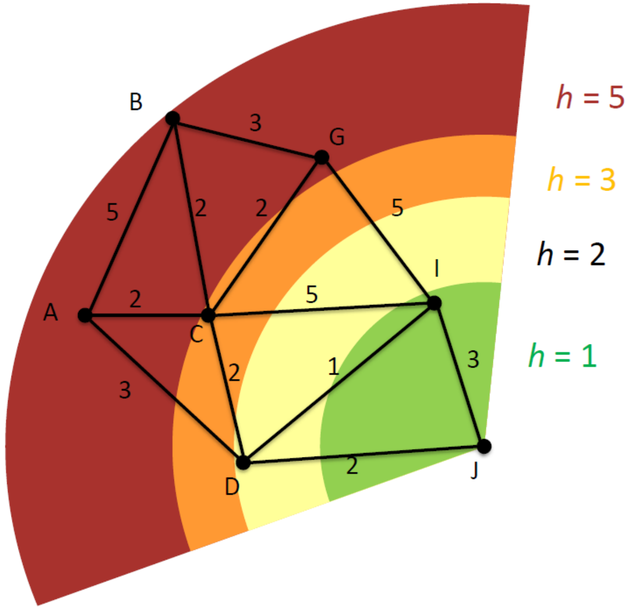
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm





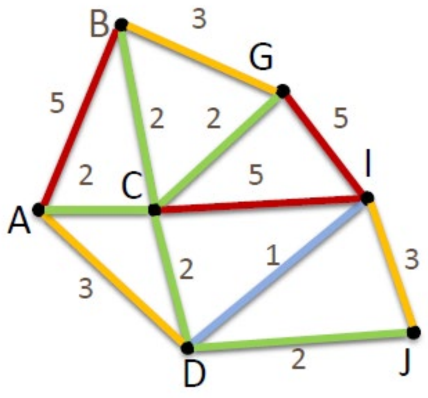


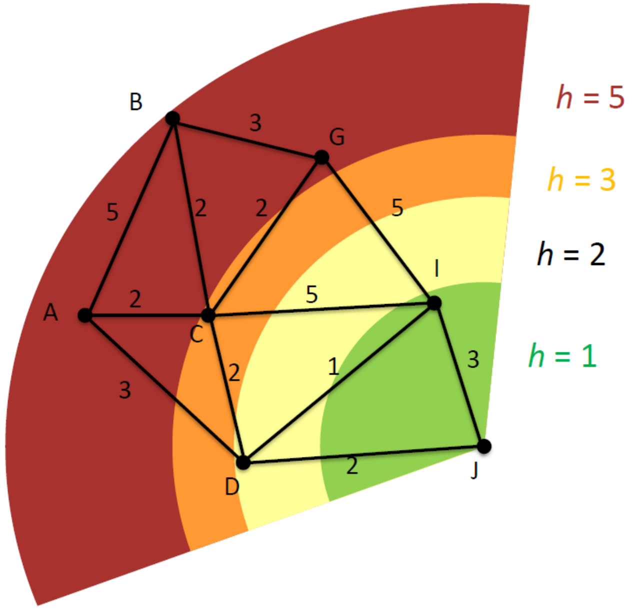
Visited: V	Q: Q
B(0)	
C(2)	
G(3)	
A(4)	
D(4)	D(4+2)
	G(3+5)
	A(4+5)

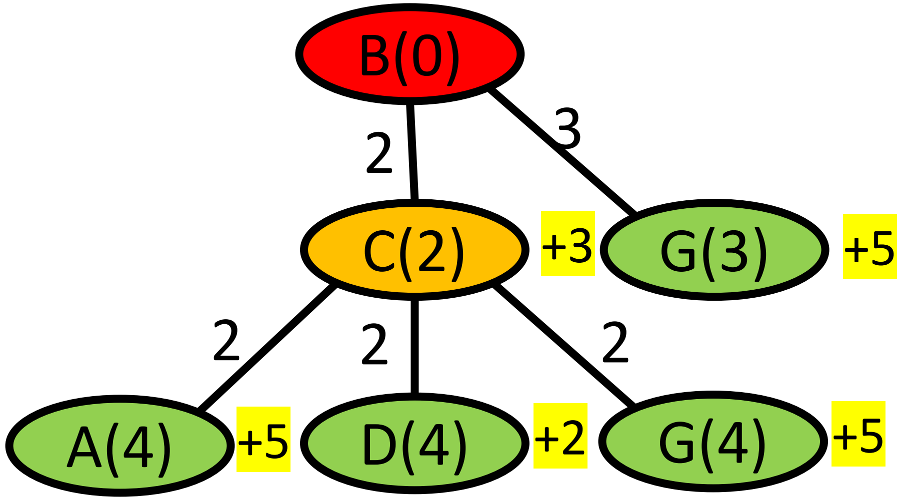
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)

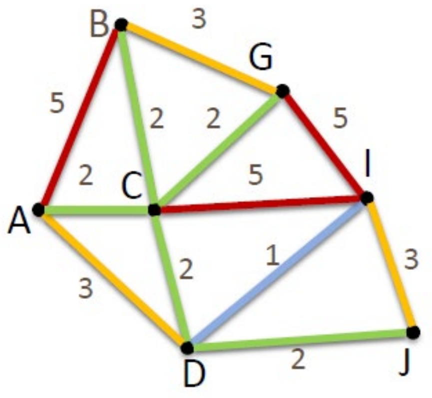
Q

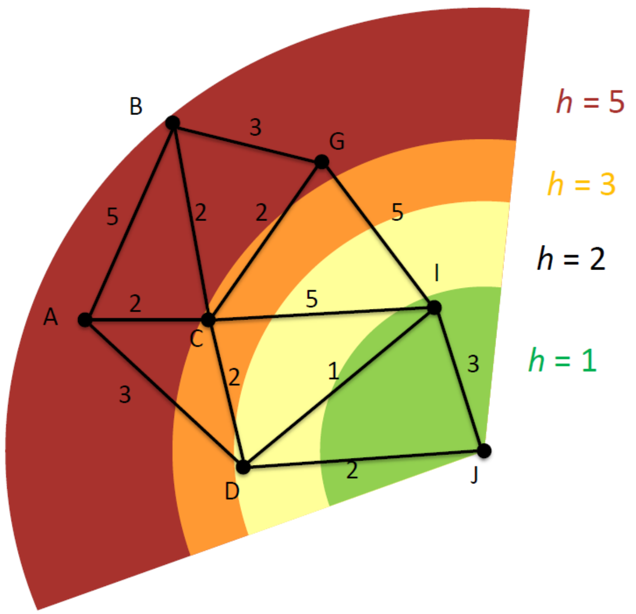
Q
D(4+2)
G(3+5)
A(4+5)

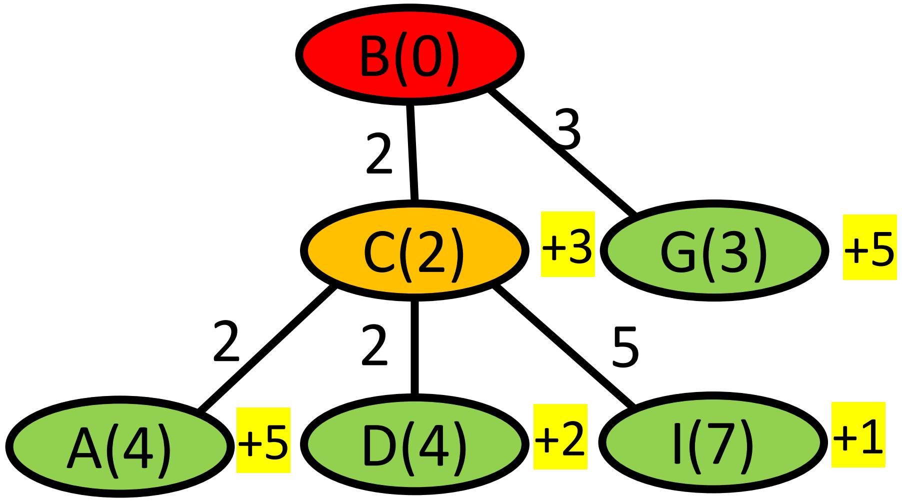
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)
I(7)

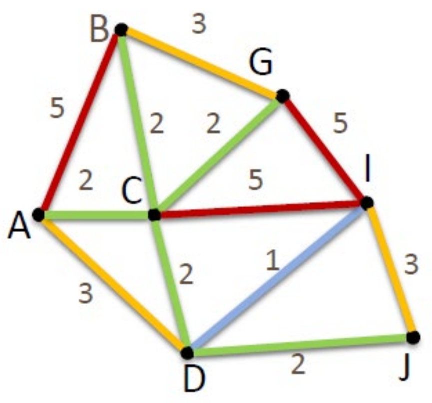
Q

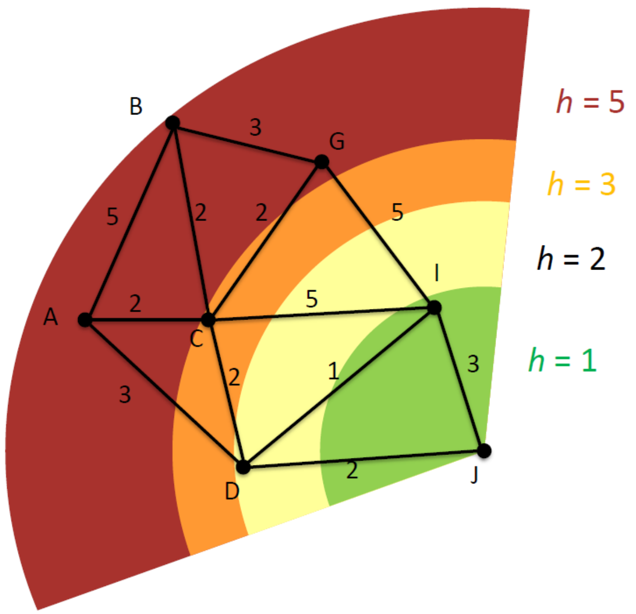
Q
D(4+2)
G(3+5)
I(7+1)
A(4+5)

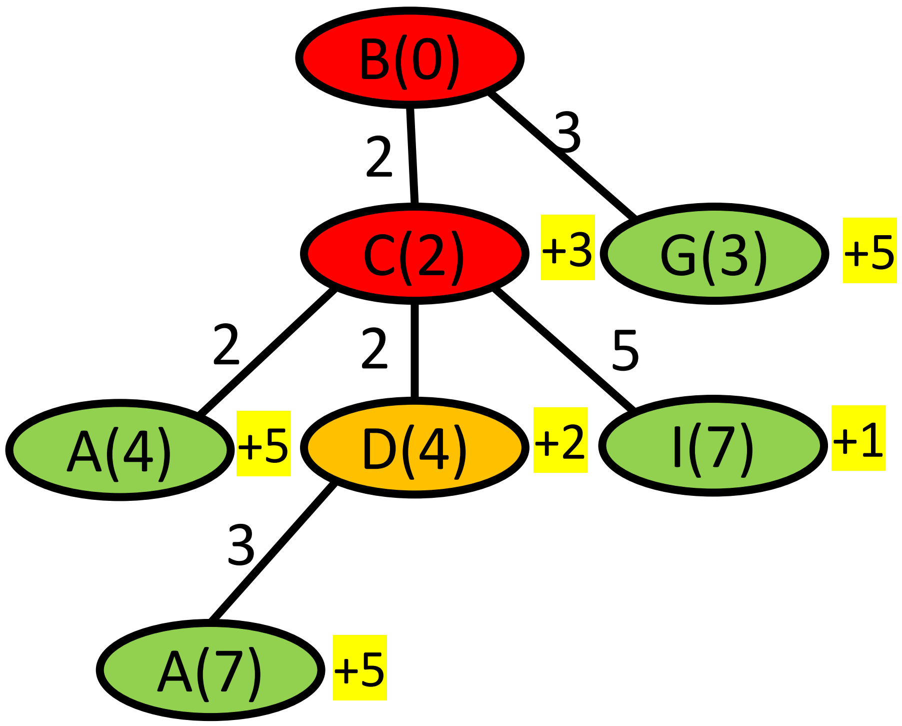
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)
I(7)

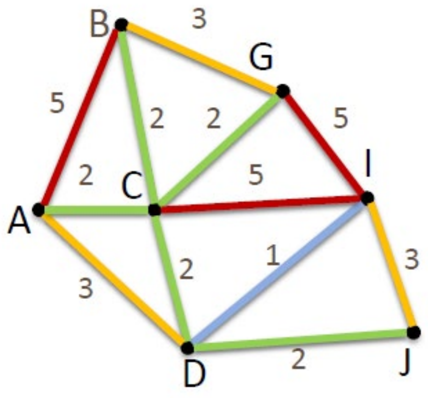
Q

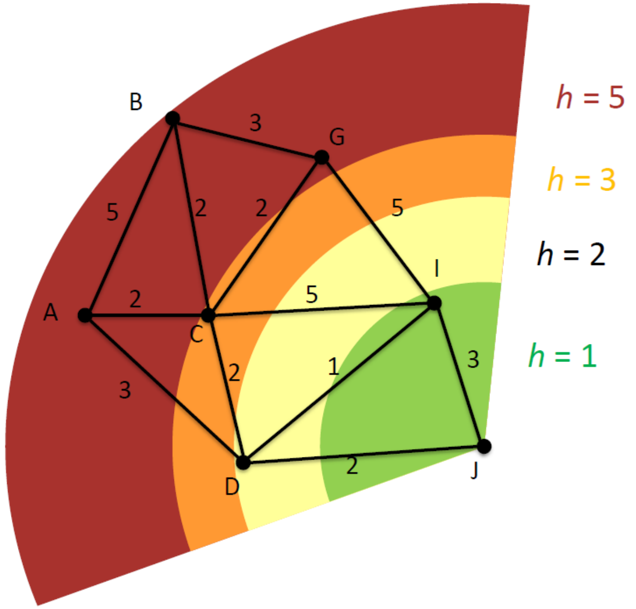
Q
G(3+5)
I(7+1)
A(4+5)

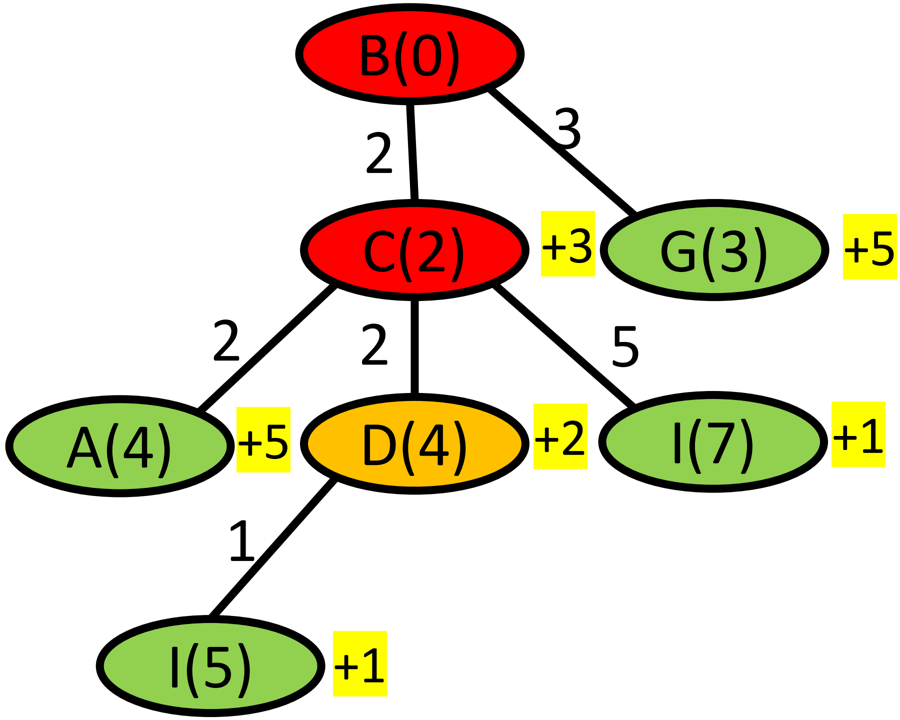
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)
I(5)

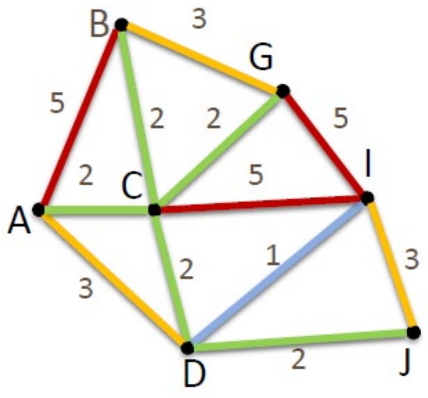
Q

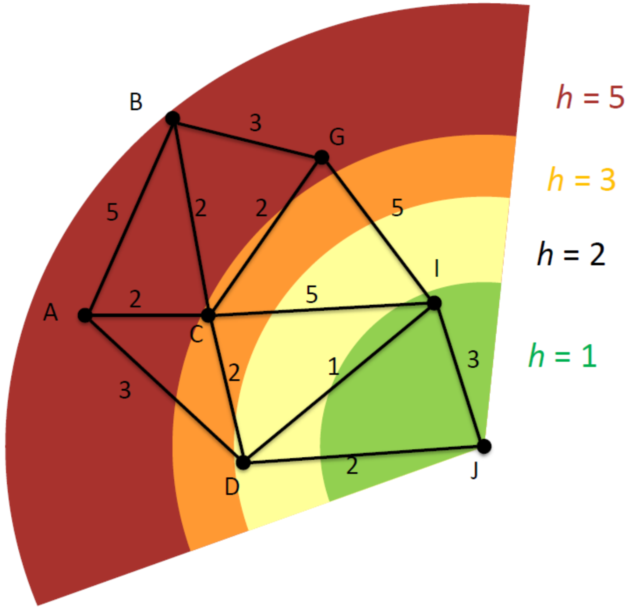
Q
I(5+1)
G(3+5)
A(4+5)

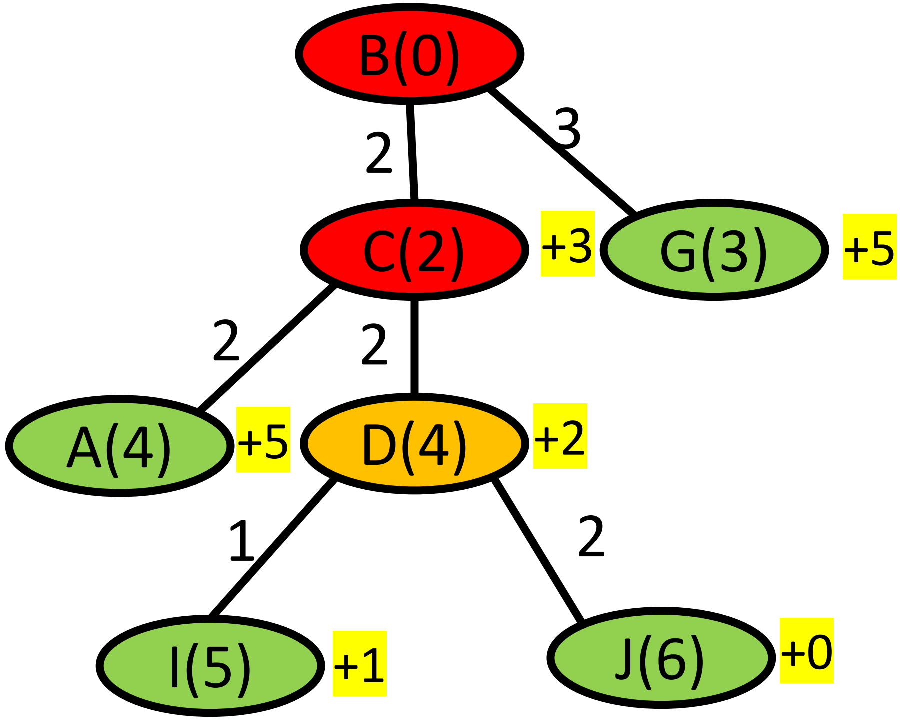
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)
I(5)
J(6)

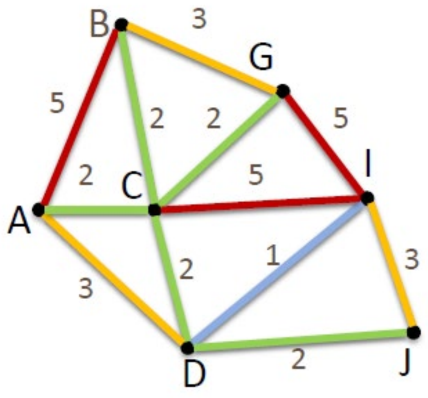
Q

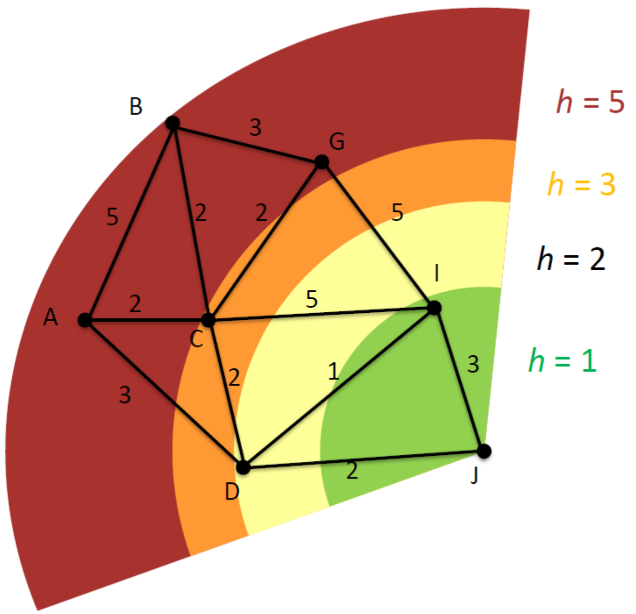
Q
I(5+1)
J(6+0)
G(3+5)
A(4+5)

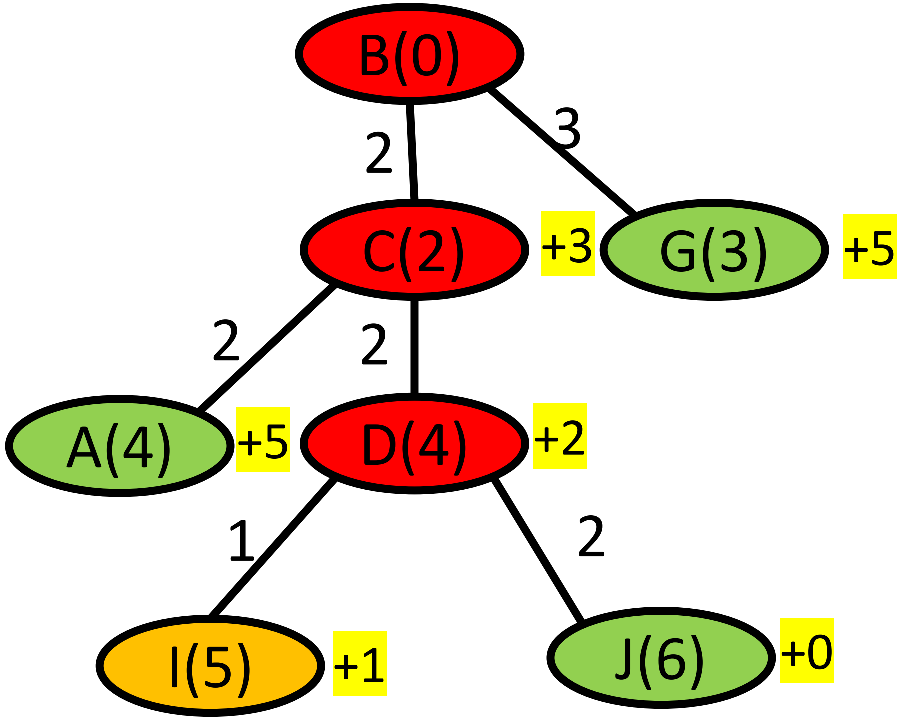
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)
I(5)
J(6)

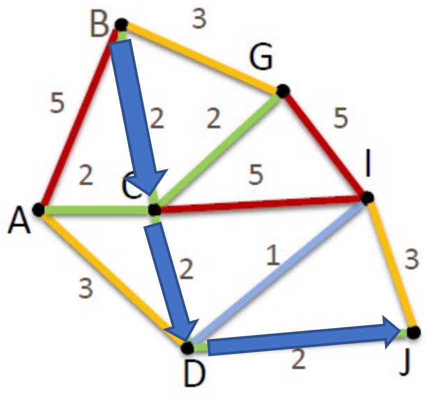
Q

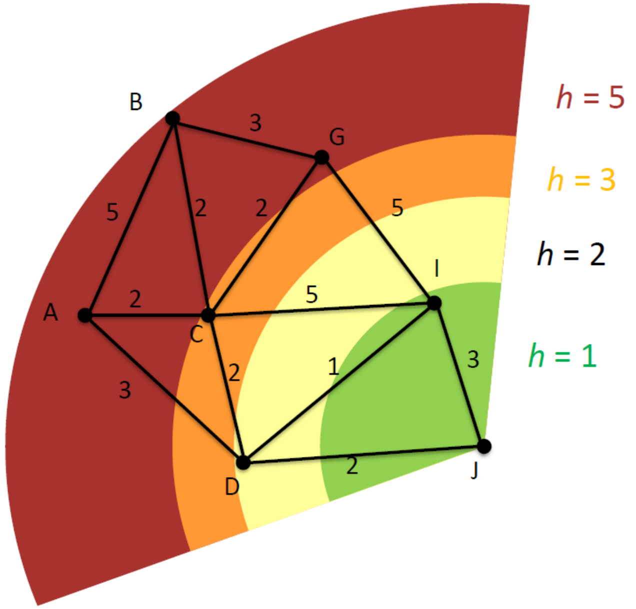
Q
J(6+0)
G(3+5)
A(4+5)

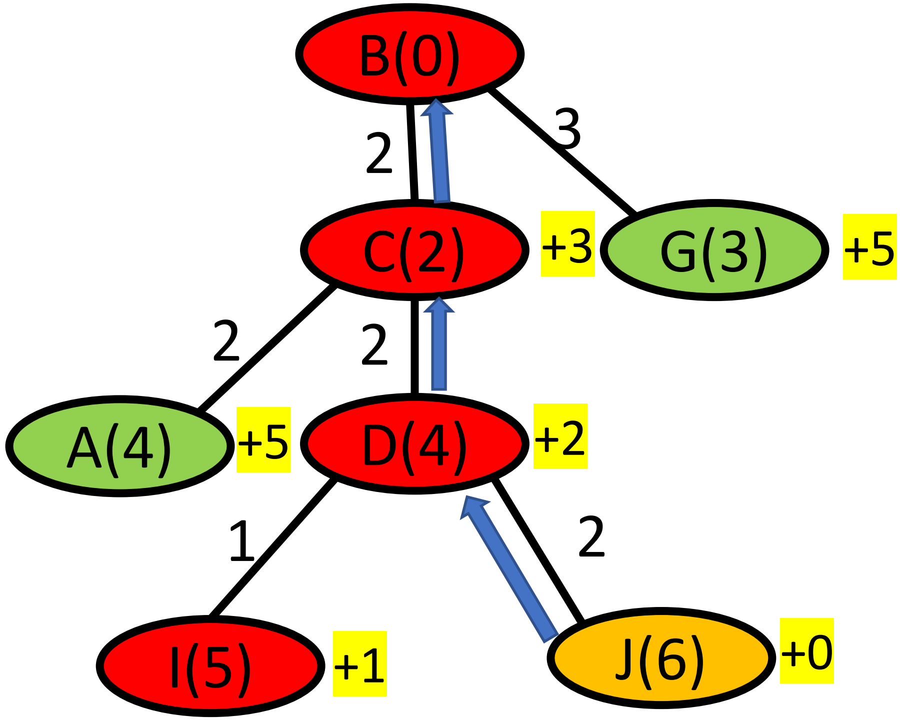
For A*, Q is Priority Queue

- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

A* Algorithm







Visited

V
B(0)
C(2)
G(3)
A(4)
D(4)
I(5)
J(6)

Q

Q
G(3+5)
A(4+5)

For A*, Q is Priority Queue

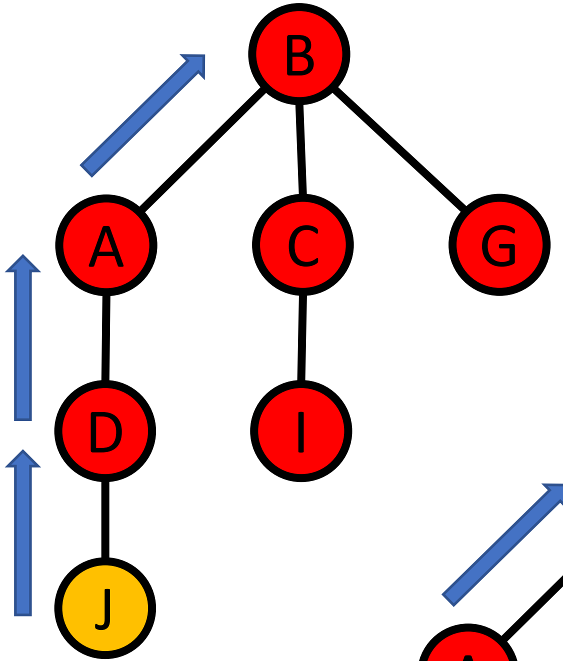
- push (Q.Insert) by f(n), arrive + heuristic cost
- pop (Q.GetFirst) from the front

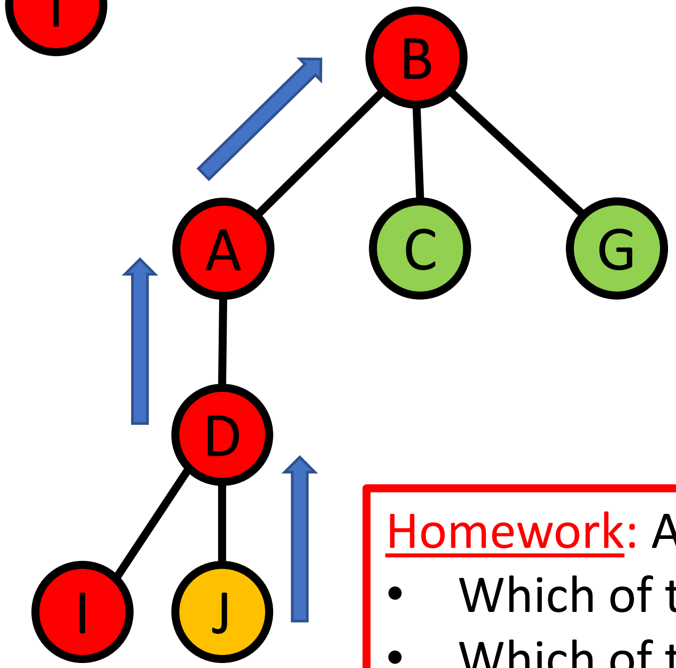
A* Algorithm

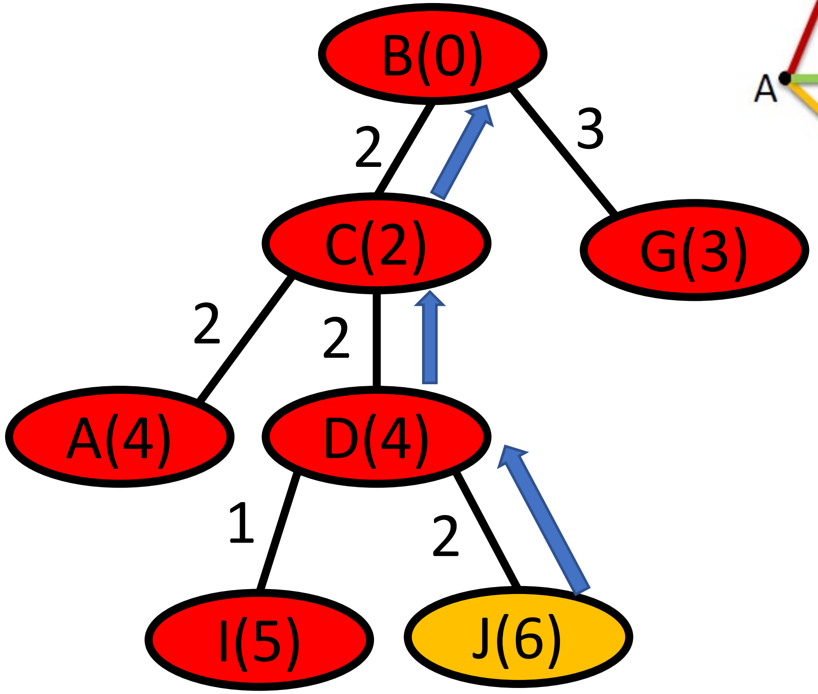
- A* is very commonly used in robot planning, especially for low-dimensional state spaces

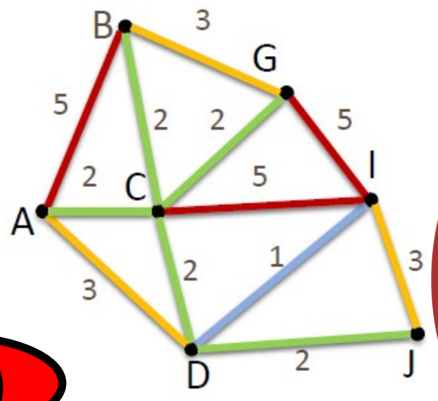
- Limitations:
  - Sometimes an admissible heuristic function is difficult to find (as hard as the problem)

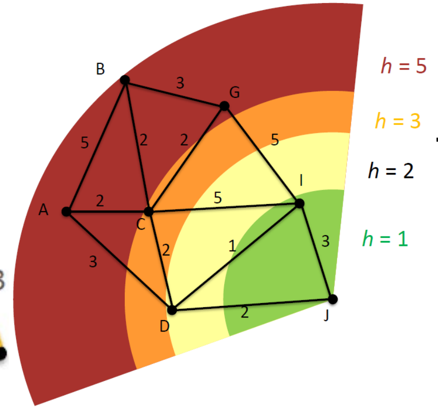
Four Planning Algorithms

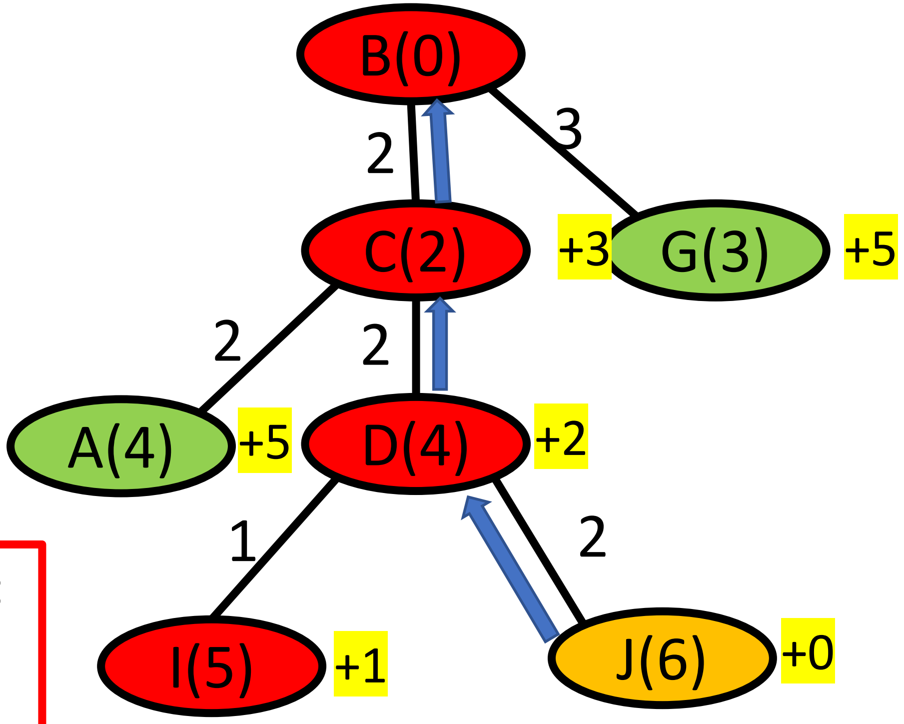












Homework: Answer the following questions:

- Which of the four methods is complete?
- Which of the four methods is optimal?

Complete: Always find a valid path if there exists one

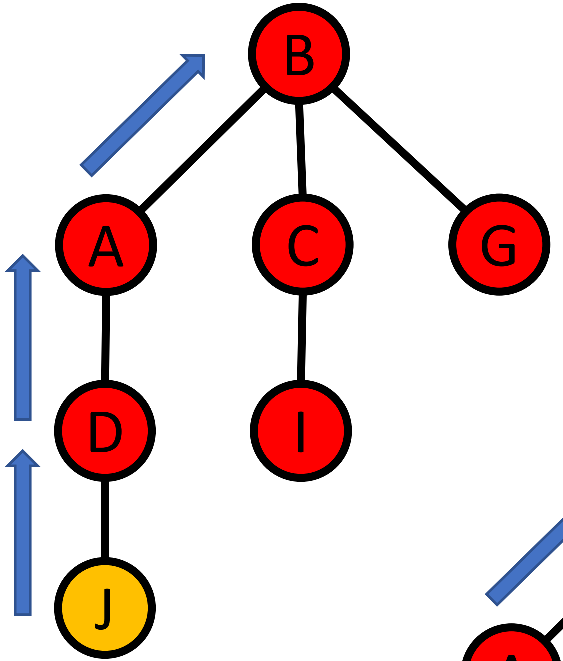
Optimal: When a path is found, it is always the shortest slido

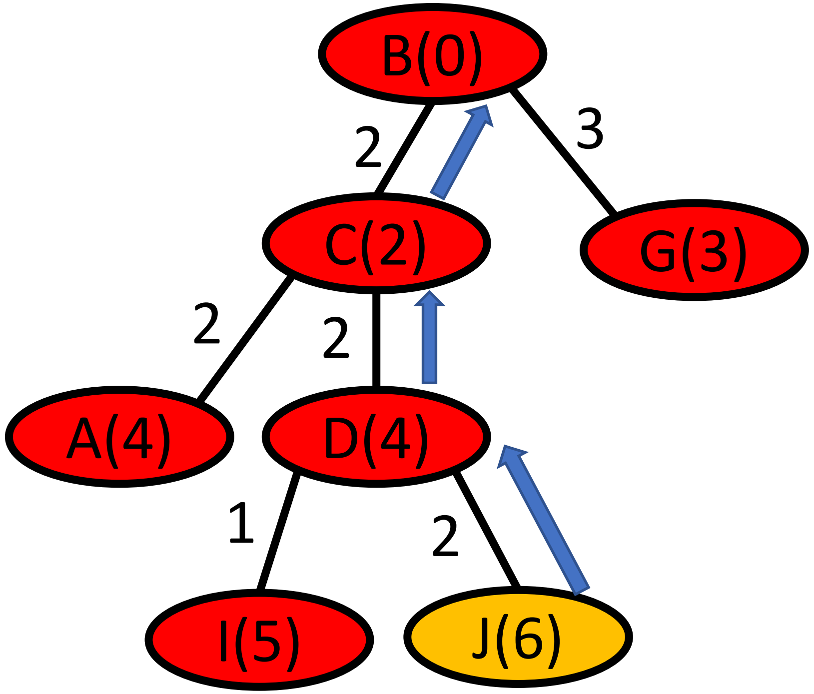
What planning algorithm would you use for the Micromouse competition?

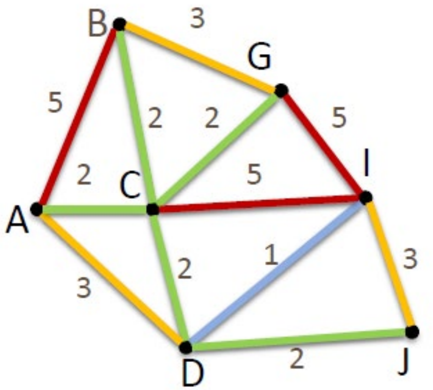
ⓘ Start presenting to display the poll results on this slide.

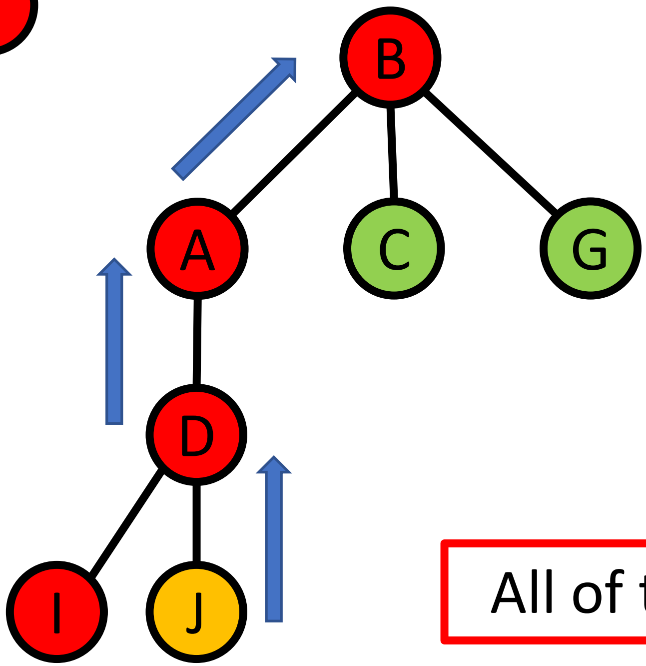
Bellman-Ford Algorithm

Four Planning Algorithms

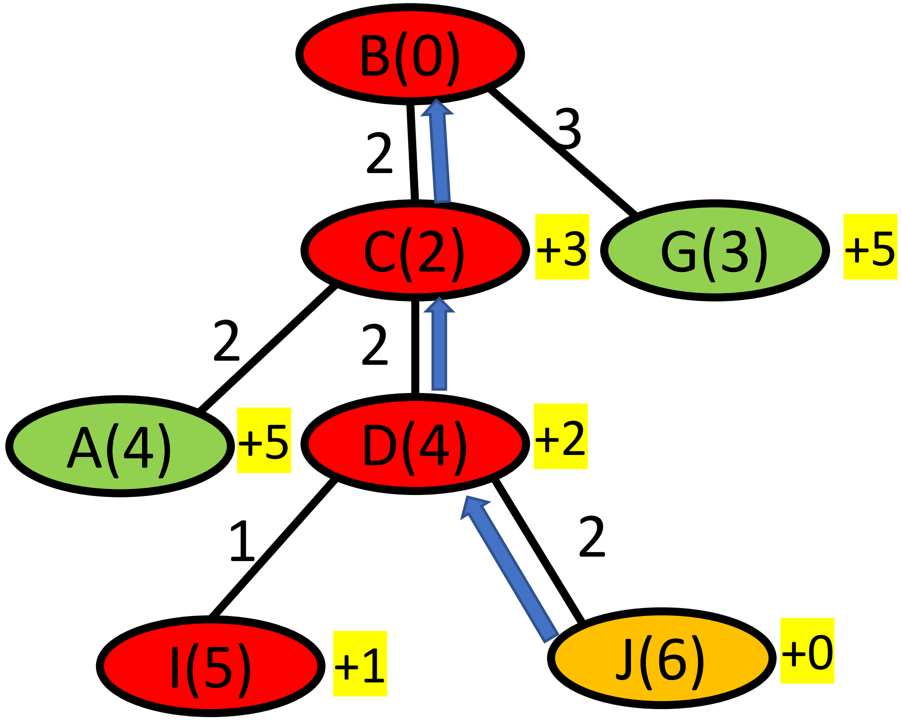








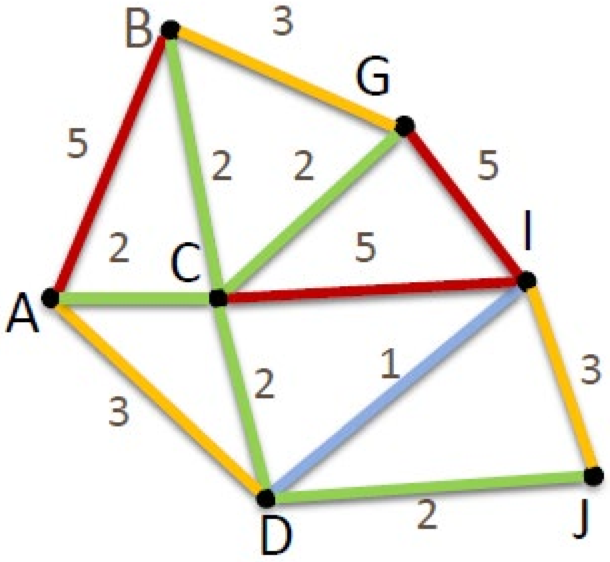
All of them focus on nodes



Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance



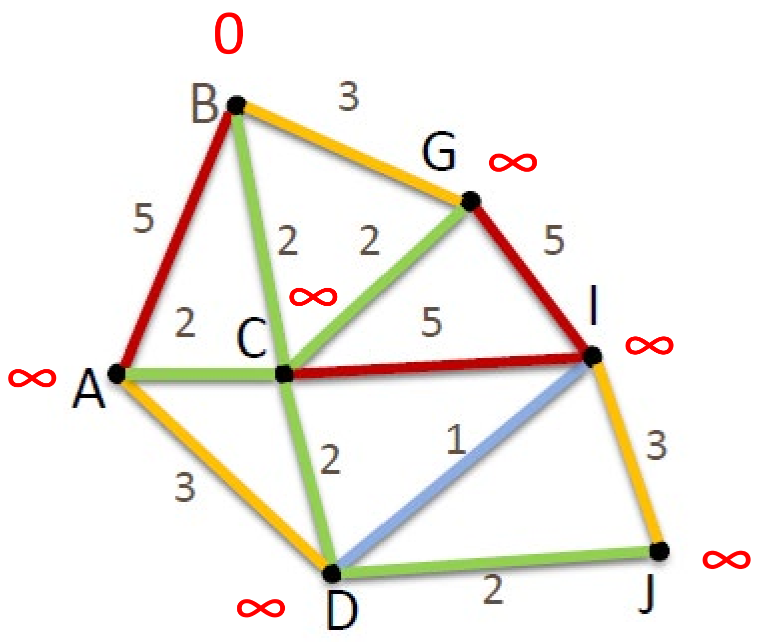
Richard E. Bellman
1920-1984

Lester R. Ford Jr.
1927-2017

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance

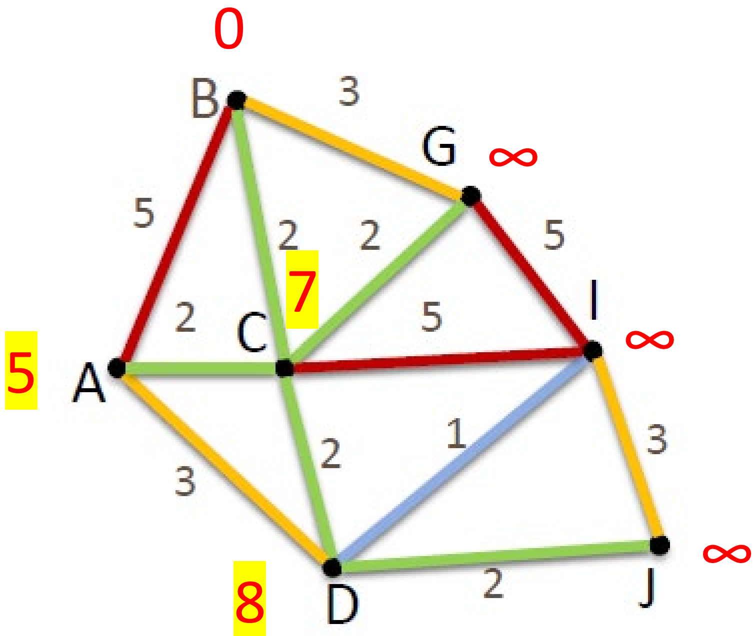


Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance

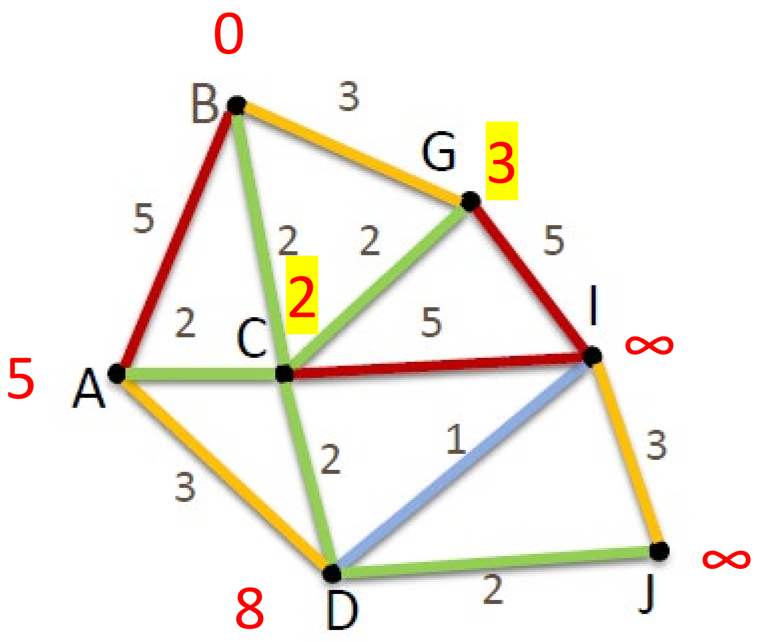


Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance



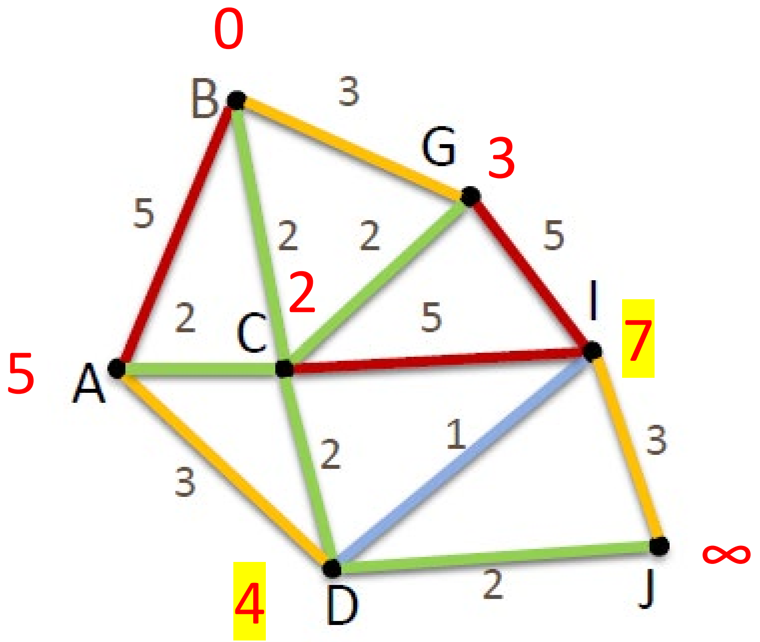
Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance

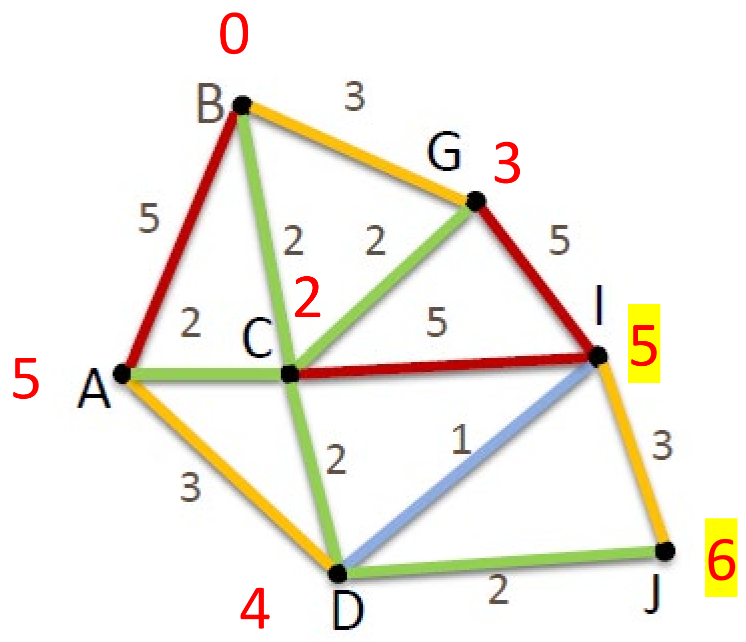


Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance

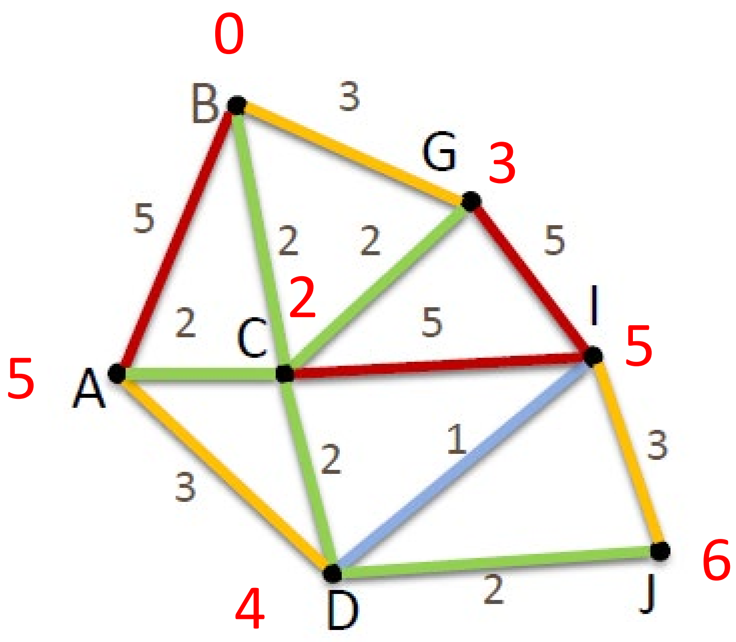


Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance

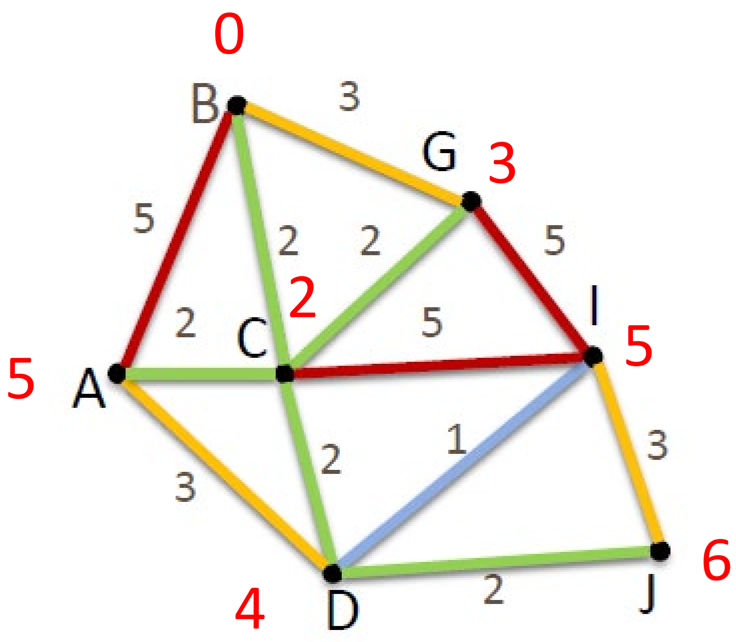


Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance



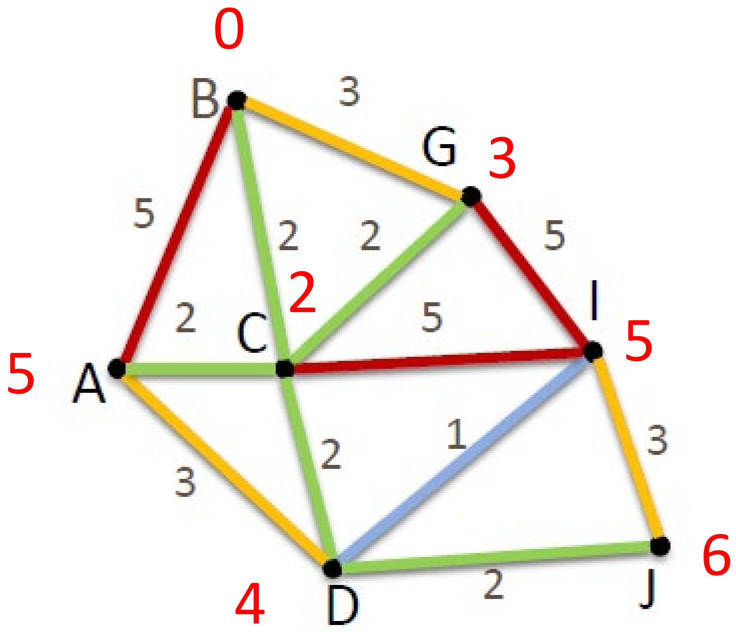
Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance

Scan all edges once again, we will luckily find no changes to the distance for each node.

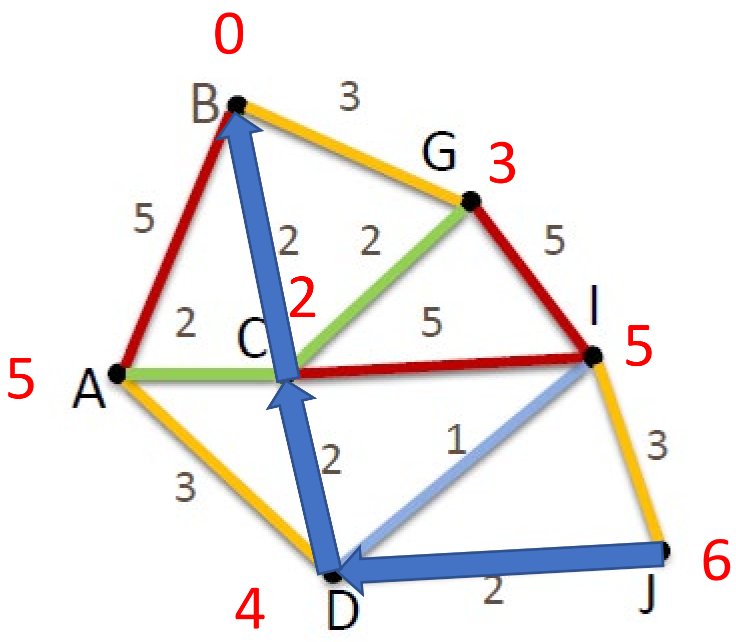


Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

Instead of examining nodes, scanning edges

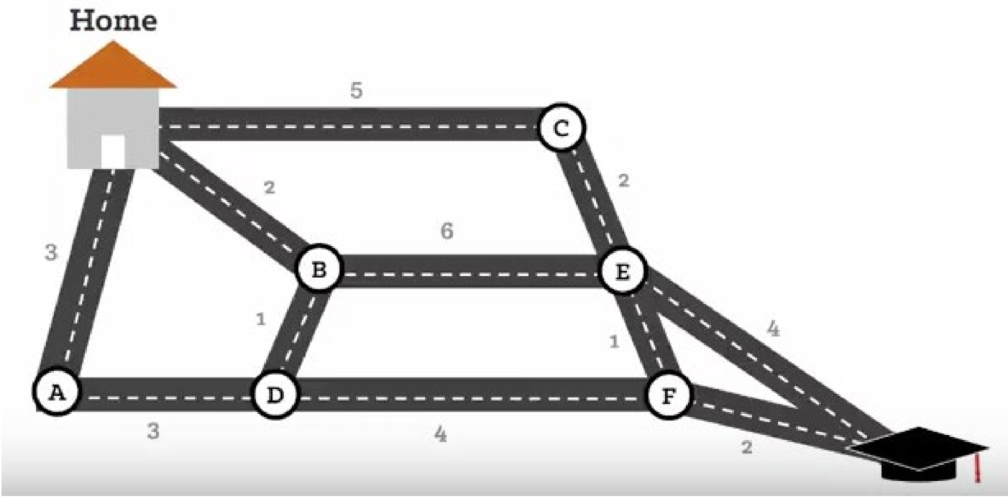
1. Initialise
   i. Set distance of start node as 0 and all the other nodes infinity
2. While (true)
   i. For all the edges
      a) If the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value
   ii. If no changes made, break while
3. Connect nodes with shortest distance



Edges:
(A,B),(A,C),(A,D),
(B,C),(B,G),
(C,D),(C,G),(C,I),
(D,I),(D,J),
(G,I),
(I,J)

Bellman-Ford Algorithm

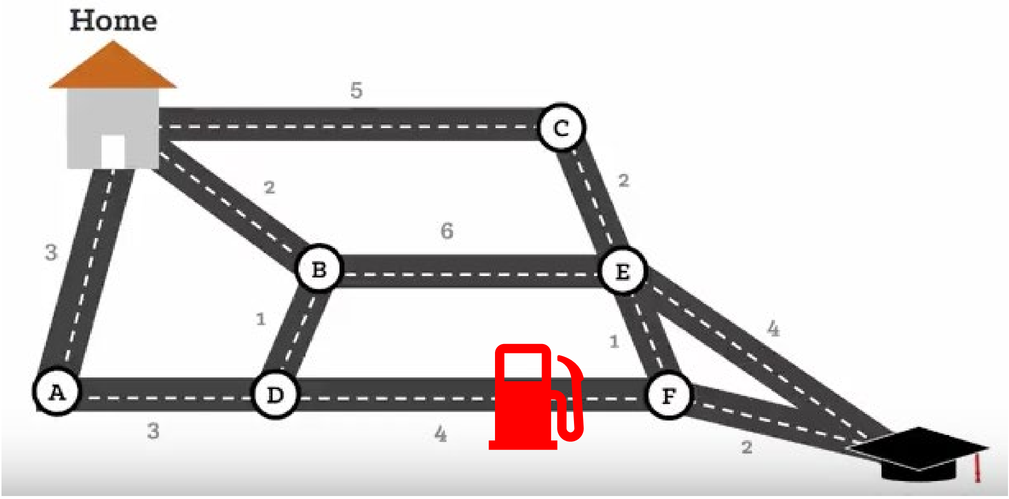
- Easy to implement!
- Work with negatively weighted edges



Cost: Distance

Bellman-Ford Algorithm

- Easy to implement!
- Work with negatively weighted edges
- Does not scale well (time complexity worse than Dijkstra's algorithm)

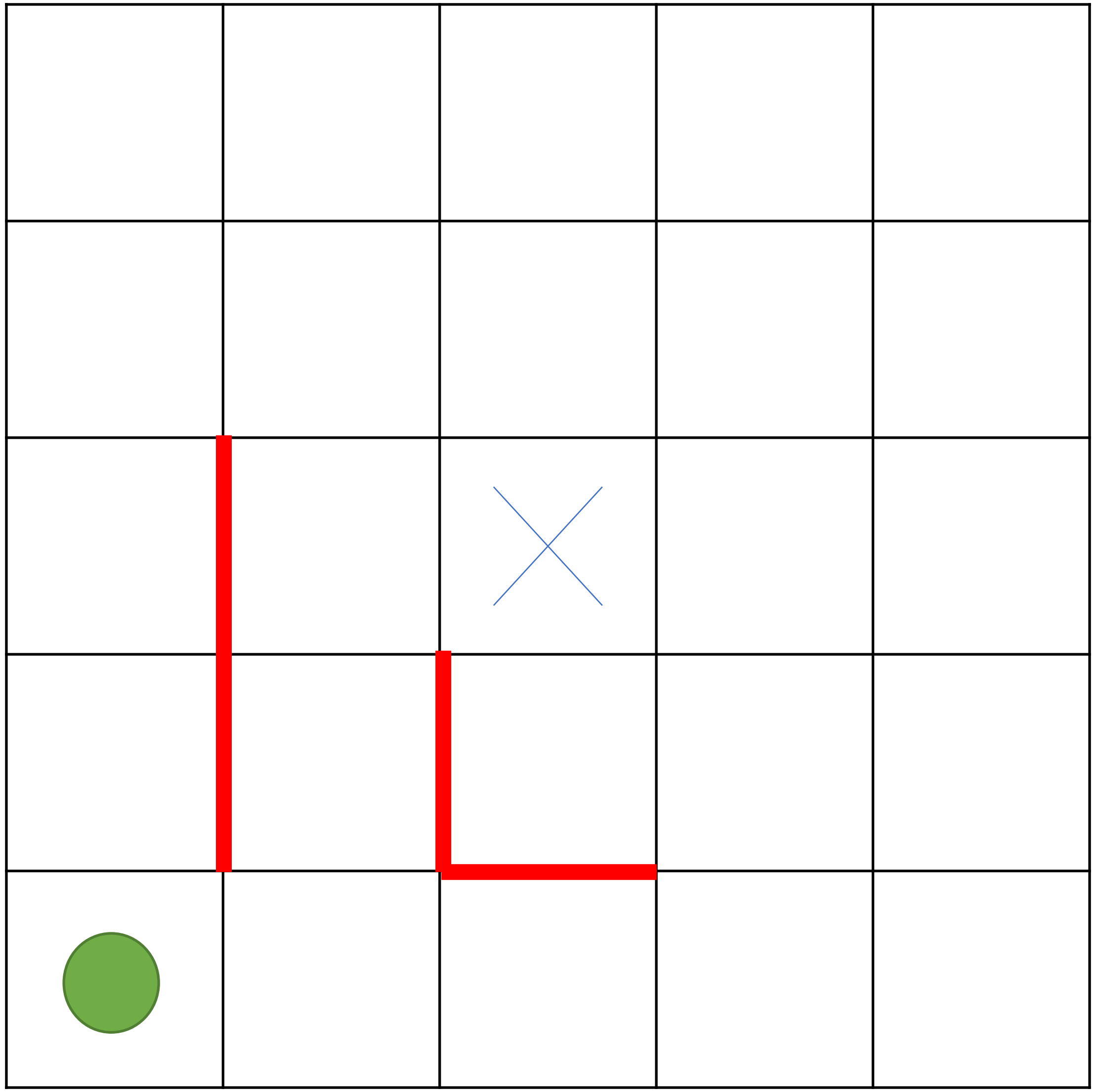


~~Cost: Distance~~

Cost: Petrol reduction

Flood Fill Algorithm

Flood-Fill Algorithm – A type of Bellman-Ford Algorithm



Flood-Fill Algorithm – A type of Bellman-Ford Algorithm

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	~~0~~	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

Flood-Fill Algorithm – A type of Bellman-Ford Algorithm

∞	∞	∞	∞	∞
∞	∞	1	∞	∞
∞	1	~~0~~	1	∞
∞	∞	1	∞	∞
∞	∞	∞	∞	∞

Flood-Fill Algorithm – A type of Bellman-Ford Algorithm

∞	∞	2	∞	∞
∞	2	1	2	∞
∞	1	~~0~~	1	2
∞	2	1	2	∞
∞	∞	∞	∞	∞

Flood-Fill Algorithm – A type of Bellman-Ford Algorithm

∞	3	2	3	∞
3	2	1	2	3
∞	1	~~0~~	1	2
∞	2	1	2	3
∞	3	∞	3	∞

Flood-Fill Algorithm – A type of Bellman-Ford Algorithm

4	3	2	3	4
3	2	1	2	3
4	1	~~0~~	1	2
∞	2	1	2	3
4	3	4	3	4

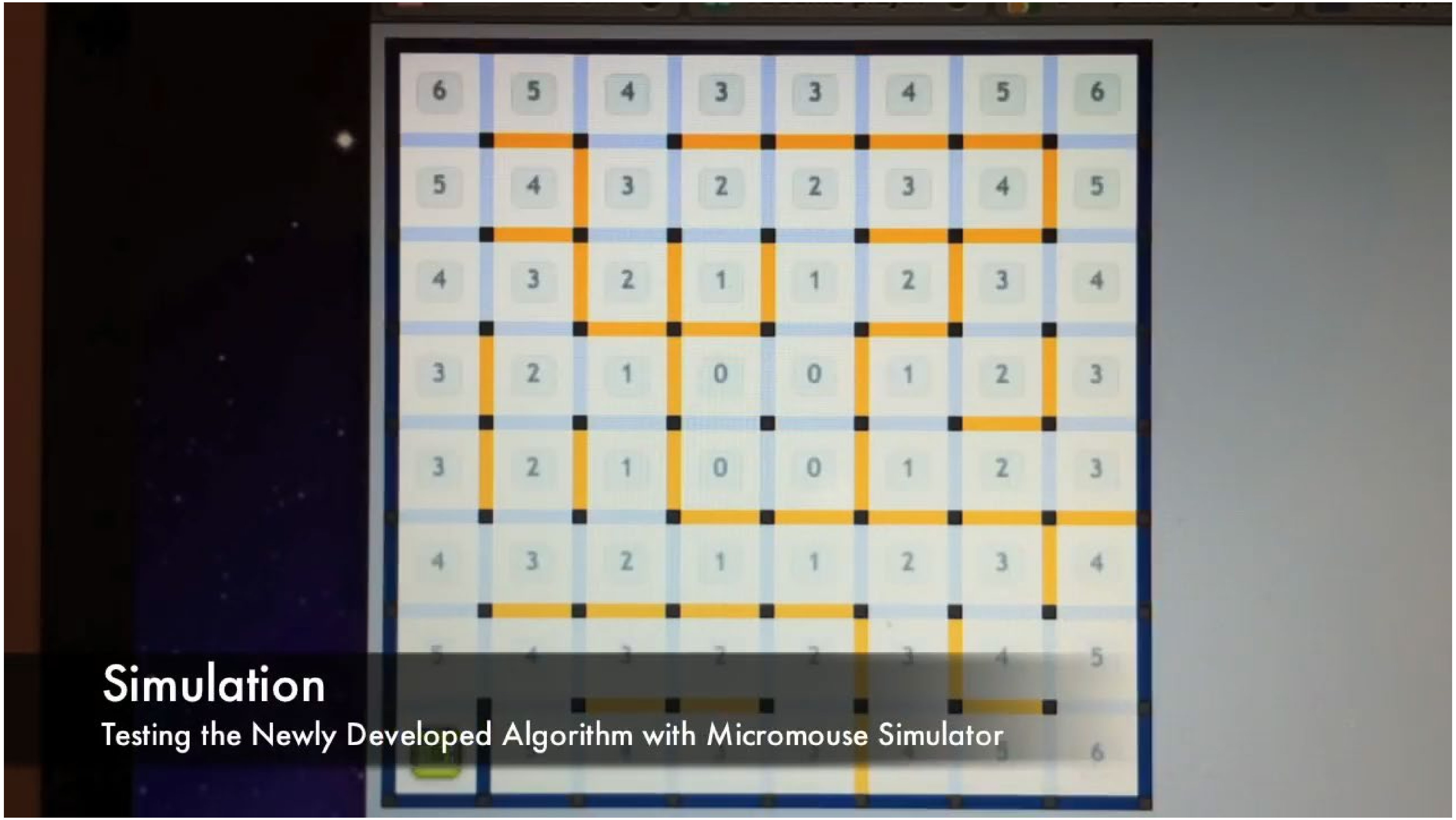
Flood-Fill Algorithm – A type of Bellman-Ford Algorithm

4	3	2	3	4
3	2	1	2	3
4	1	~~0~~	1	2
5	2	1	2	3
4	3	4	3	4

Flood-Fill Algorithm – A type of Bellman-Ford Algorithm

4	3	2	3	4
3	2	1	2	3
4	1	~~0~~	1	2
5	2	1	2	3
4	3	4	3	4

Flood-Fill Algorithm – For exploration



https://www.youtube.com/watch?v=tVNs9gdXBDc

Flood-Fill Algorithm – Pseudocode (More details in tutorial)

FloodFill

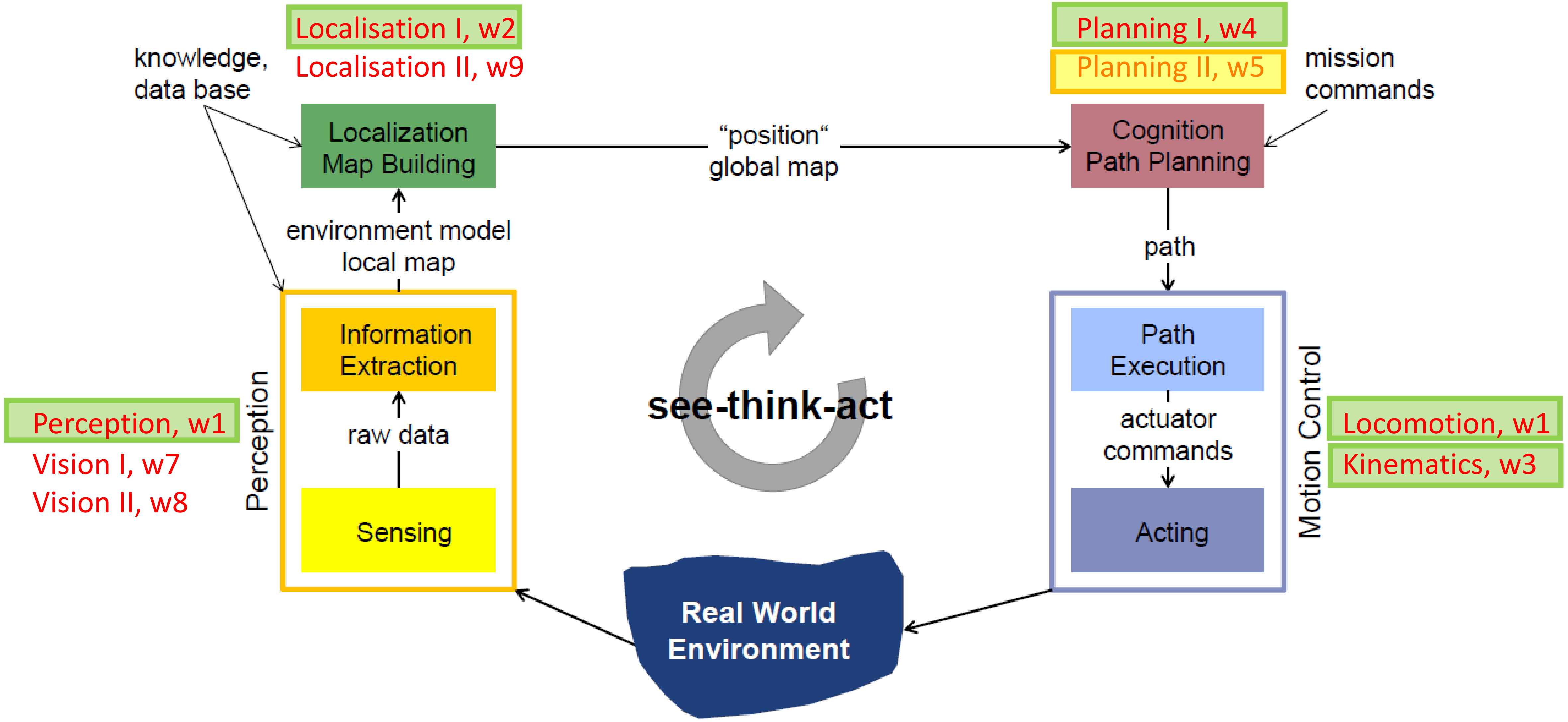
```
initialize
    all CellValues ← N (N=A Big Number, e.g. N=Rows x Columns)
    GoalCellValue ← 0
    CurrentExploredValue ← 0
    MazeValueChanged ← 1
while MazeValueChanged ≠0
    MazeValueChanged ← 0
    forall Rows
        forall Columns
            if CurrentCellValue==CurrentExploredValue
                forall Directions (North, East, South, West)
                    if NeighbouringWall does not exist
                        if NeighbouringCellValue==N
                            NeighbouringCellValue ← CurrentCellValue+1
                            MazeValueChanged ← 1
    CurrentExploredValue = CurrentExploredValue +1
return
```

What we have learnt today

- Path planning vs trajectory planning
- Constructing graph from map representation
- Graph search algorithms can be used to find an optimal path
  - Breath first search
  - Depth first search
  - Dijkstra's algorithm
  - A* algorithm
  - Bellman-Ford algorithm
    - **Flood fill algorithm (tutorial)**

UNSW SYDNEY

Next week: Planning II



R. Siegwart, I. R. Nourbakhsh, D. Scaramuzza. Introduction to autonomous mobile robots. The MIT Press. Second edition. 2011.

Acknowledgment

- Many of the slides are adapted from Nick Lawrance
  - https://www.ethz.ch/content/dam/ethz/special-interest/mavt/robotics-n-intelligent-systems/asl-dam/documents/lectures/autonomous_mobile_robots/spring-2019/Planning_I_2019.pdf