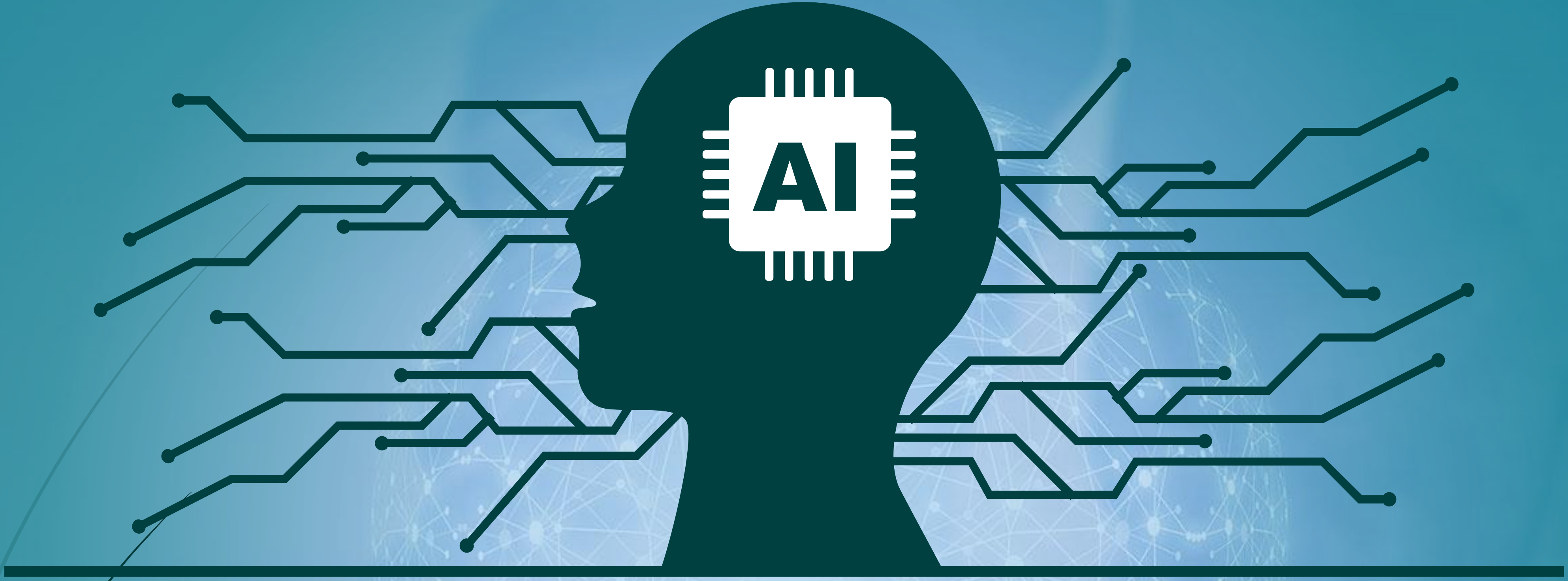


Advanced Artificial Intelligence

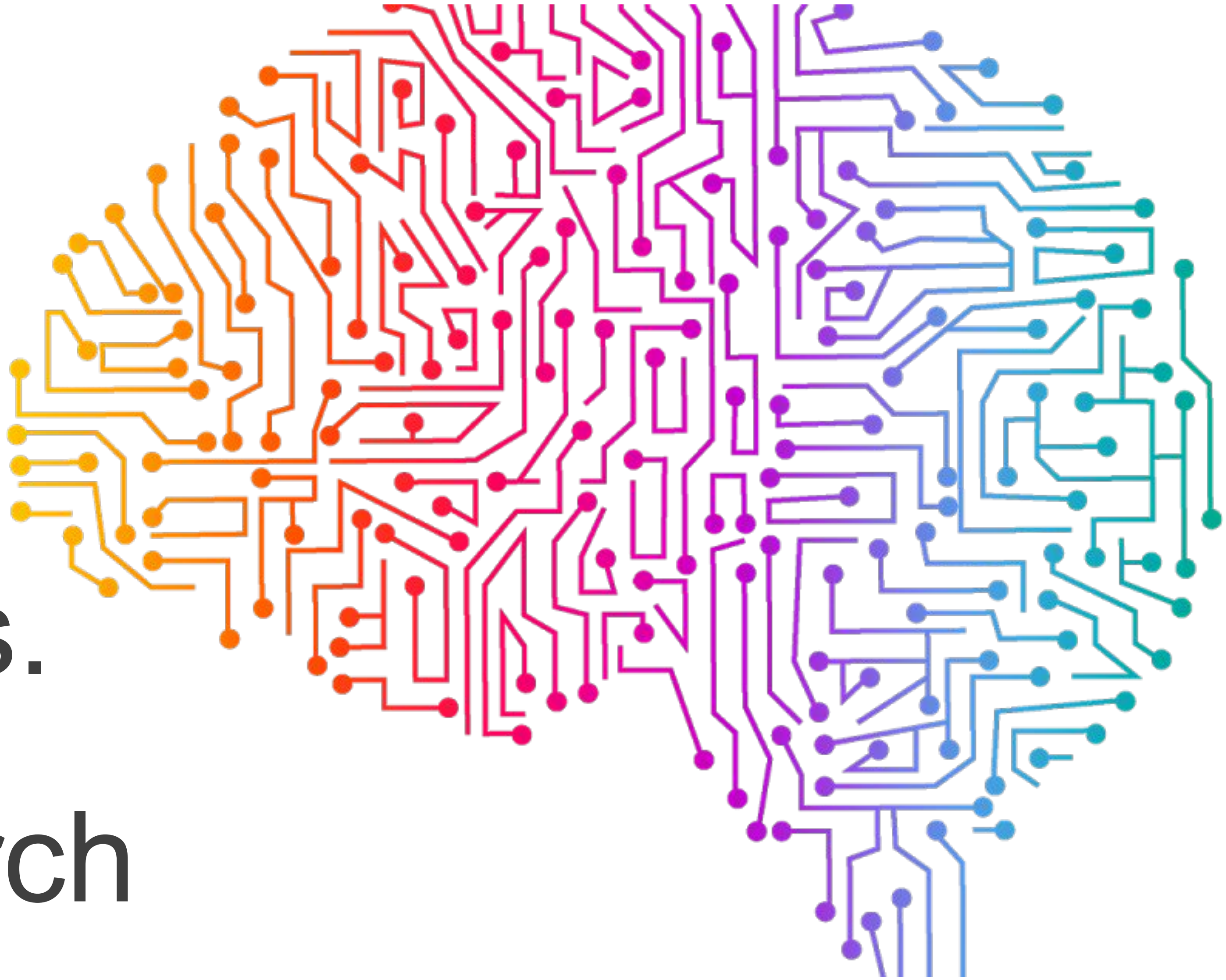
Week #2-2

Dr. Qurat Ul Ain
Assistant Professor
Dept. of AI & DS
FAST NUCES, Islamabad
Email:

Learning Objective of this Topic

- Uninformed vs. Informed Search
- Uninformed Searching Algorithms
  - Breadth First Search (BFS)
  - Depth First Search (DFS)
  - Uninformed Cost Search (UCS)
  - Difference between BFS and DFS
  - When to use BFS? Real-Life Applications
  - When to use DFS? Real-Life Applications

Uninformed vs. Informed Search



Uninformed vs. Informed Search

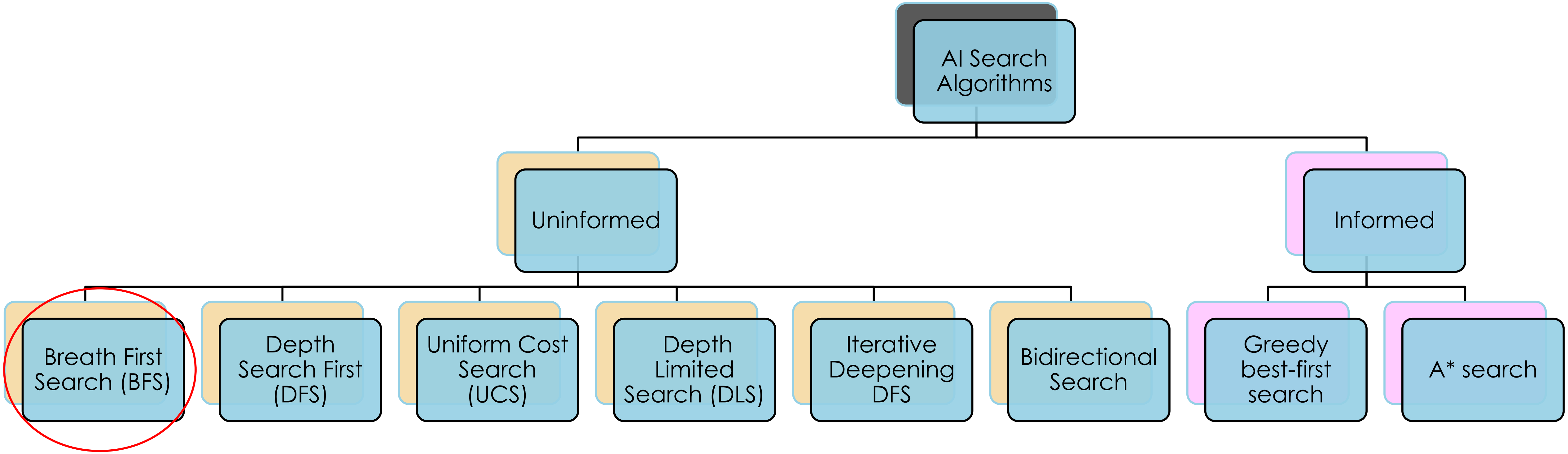
Uninformed

- Also known as Blind search.
- Do not require information to perform a search.
- May be time-consuming.
- Comparatively high in cost.
- The AI does not get any suggestions regarding the solution and where to find it.

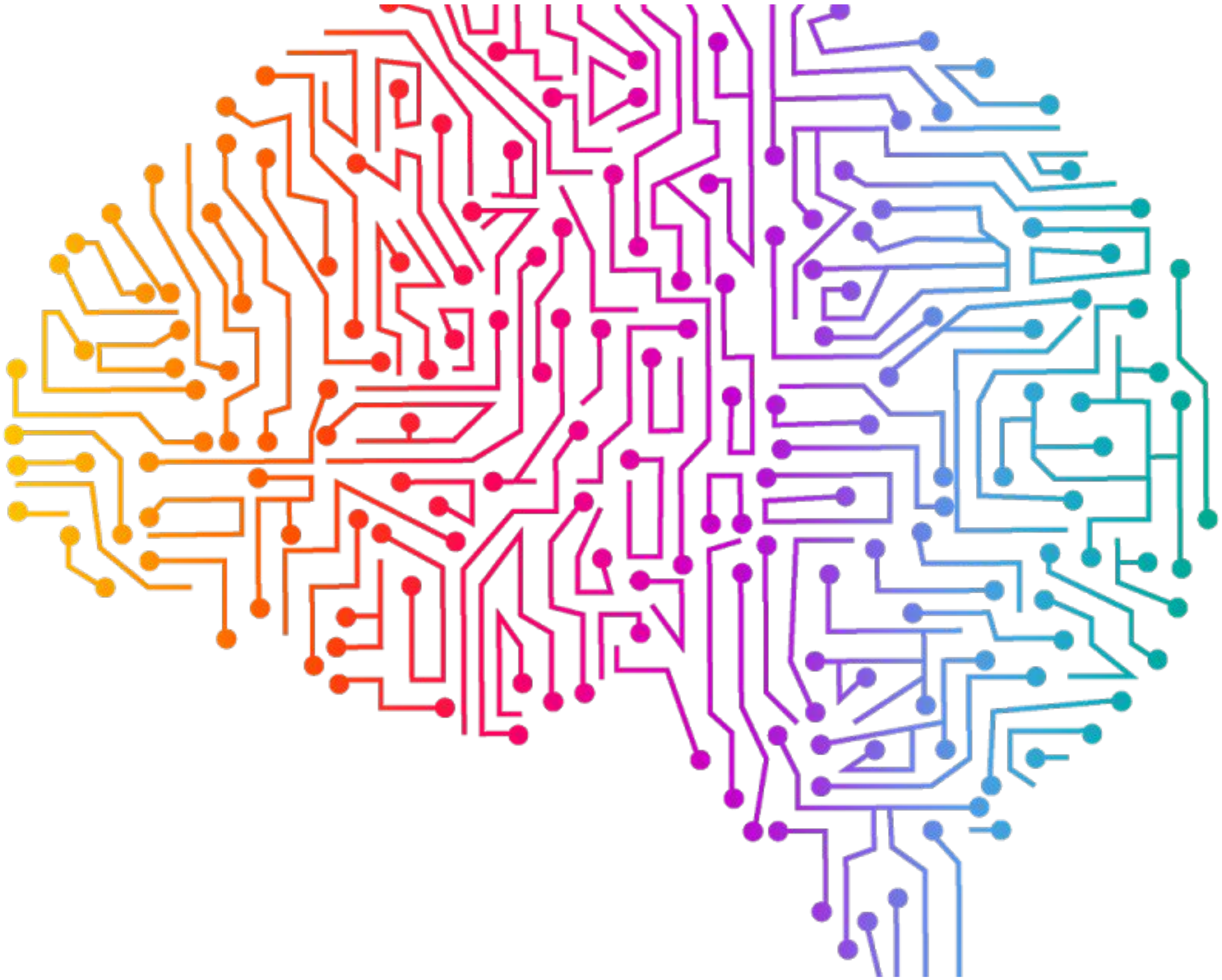
Informed

- Also known as Heuristic search.
- Requires information to perform search Quick solution to the problem.
- Cost is low.
- The AI gets suggestions regarding how and where to find a solution to any problem.

UNINFORMED SEARCH ALGORITHM

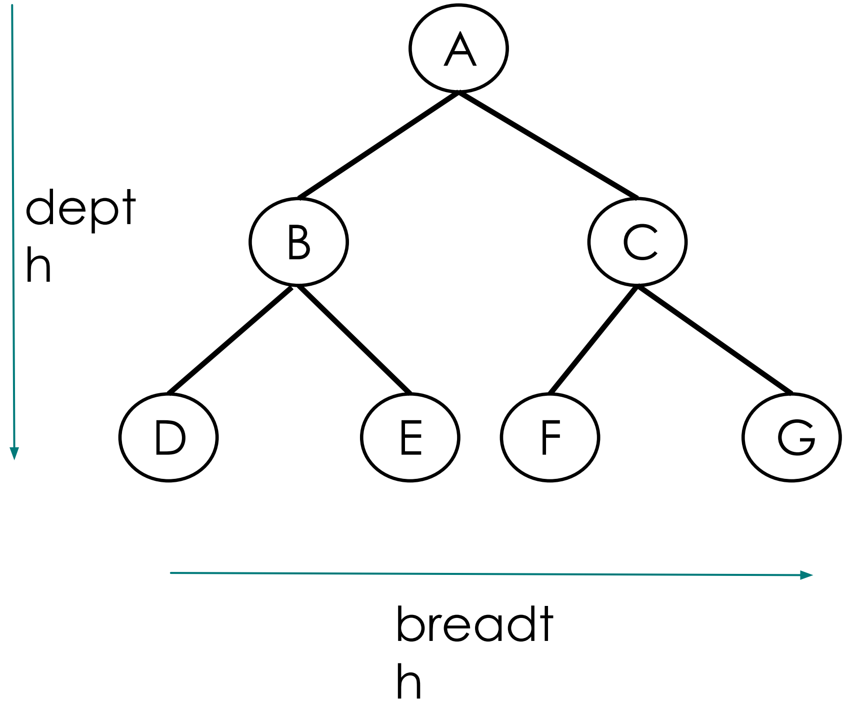

AI Search Algorithms
Uninformed
Informed
Breath First Search (BFS)
Depth Search First (DFS)
Uniform Cost Search (UCS)
Depth Limited Search (DLS)
Iterative Deepening DFS
Bidirectional Search
Greedy best-first search
A* search

Breadth First Search (BFS)



Breadth First Search

- Expand Shallowest Node First (Level First Search)
- *Queue*: nodes in the queue to be explored
- explored: Nodes that are already explored
- *Queue* is a first-in-first-out (FIFO) queue, i.e., new successors go at the end of the queue.



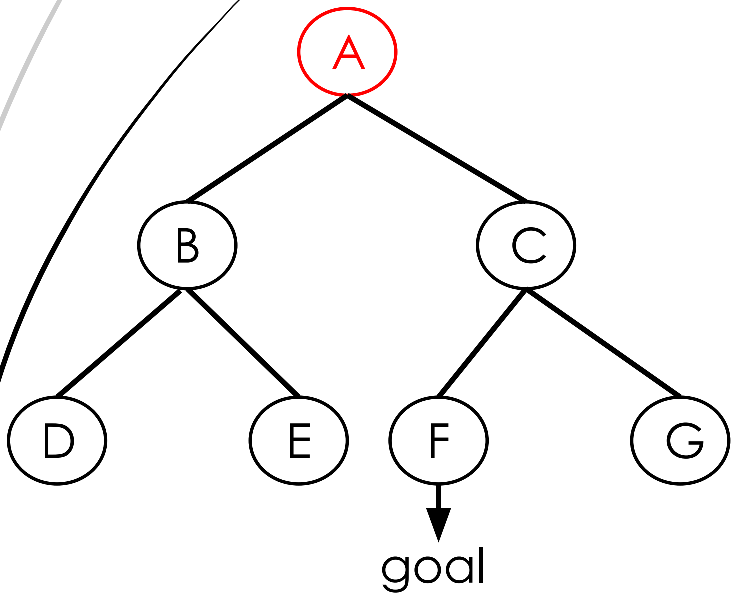
Breadth First Search

- Expand Shallowest Node First (Level First Search)
- *Queue*: nodes in the queue to be explored
- explored: Nodes that are already explored
- *Queue* is a first-in-first-out (FIFO) queue, i.e., new successors go at the end of the queue.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  Queue ← a FIFO queue with source node as the only element
  explored ← an empty set
  if source node == goal then return SOLUTION
  loop do
     if EMPTY?( Queue ) then return failure
     node ← POP( Queue ) /* chooses the shallowest node in Queue */
     add node to explored
     for each node do
     adjacent ← ADJACENT-NODE( node )
        if adjacent is not in explored or Queue then
           Queue ← INSERT( adjacent )
           if adjacent == goal then return SOLUTION
```

Breadth First Search

Queue = [A]
Explored = [ ]
Is A a goal state?

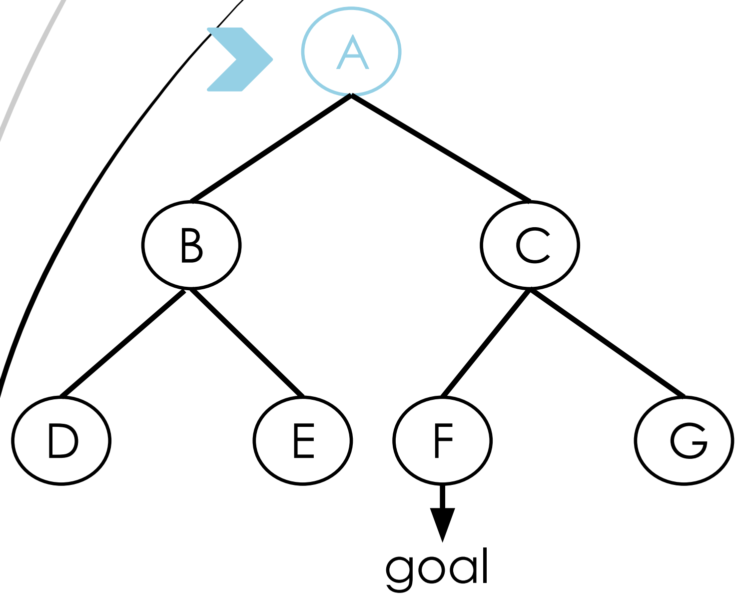



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
   Queue ← a FIFO queue with source node as the only element
   explored ← an empty set
   if source node == goal then return SOLUTION
   loop do
      if EMPTY?( Queue ) then return failure
      node ← POP( Queue ) /* chooses the shallowest node in Queue */
      add node to explored
      for each node do
      adjacent ← ADJACENT-NODE( node )
         if adjacent is not in explored or Queue then
            Queue ← INSERT( adjacent )
            if adjacent == goal then return SOLUTION
```

Breadth First Search

Queue = [A]
Explored = [ ]

Queue = [ ]
Explored = [A]



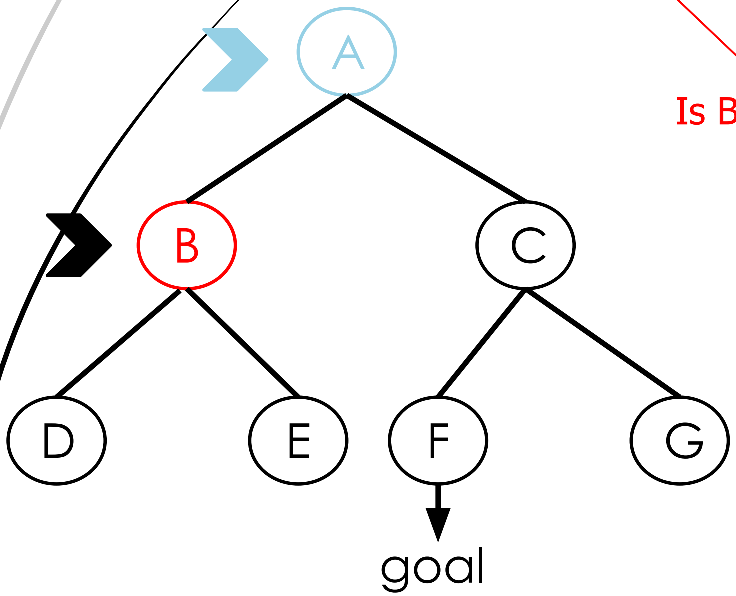
function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure
Queue ← a FIFO queue with *source node* as the only element
explored ← an empty set
if *source node* == *goal* **then return** SOLUTION
loop do
if EMPTY?(*Queue*) **then return** failure
node ← POP(*Queue*) /* chooses the shallowest node in *Queue* */
add *node* to *explored*
for each *node* **do**
adjacent ← ADJACENT-NODE(*node*)
if *adjacent* is not in *explored* or *Queue* **then**
Queue ← INSERT(*adjacent*)
if *adjacent* == *goal* **then return** SOLUTION

Node
Child

Breadth First Search

Queue = [ ]
Explored = [A]

Queue = [B]
Explored = [A]



Is B a goal state?

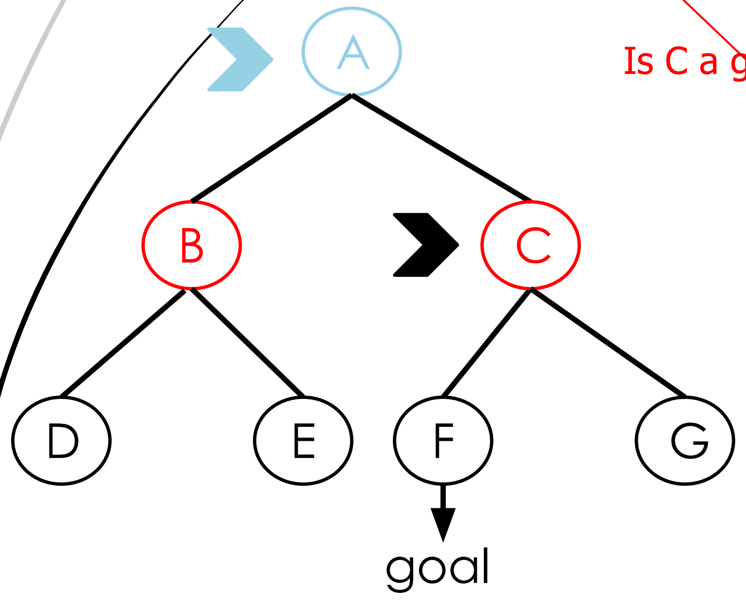
```
function BREADTH-FIRST-SEARCH( problem) returns a solution, or failure
    Queue ← a FIFO queue with source node as the only element
    explored ← an empty set
    if source node == goal then return SOLUTION
    loop do
        if EMPTY?( Queue ) then return failure
        node ← POP( Queue ) /* chooses the shallowest node in Queue */
        add node to explored
        for each node do
        adjacent ← ADJACENT-NODE( node )
            if adjacent is not in explored or Queue then
                Queue ← INSERT( adjacent )
                if adjacent == goal then return SOLUTION
```

Node
Child

Breadth First Search

Queue = [B]
Explored = [A]

Queue = [B, C]
Explored = [A]



Is C a goal state?

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

Queue ← a FIFO queue with *source node* as the only element
explored ← an empty set
if *source node* == *goal* **then return** SOLUTION
loop do
  if EMPTY?(*Queue*) **then return** failure
  node ← POP(*Queue*) /* chooses the shallowest node in *Queue* */
  add *node* to *explored*
  for each *node* **do**
  adjacent ← ADJACENT-NODE(*node*)
    if *adjacent* is not in *explored* or *Queue* **then**
      Queue ← INSERT(*adjacent*)
      if *adjacent* == *goal* **then return** SOLUTION

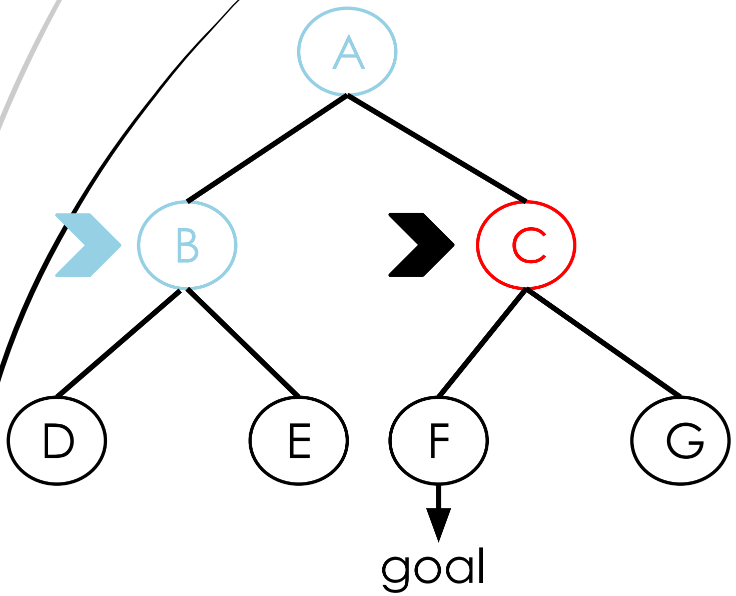
Node

Child

Breadth First Search

Queue = [B, C]
Explored = [A]

Queue = [C]
Explored = [A, B]



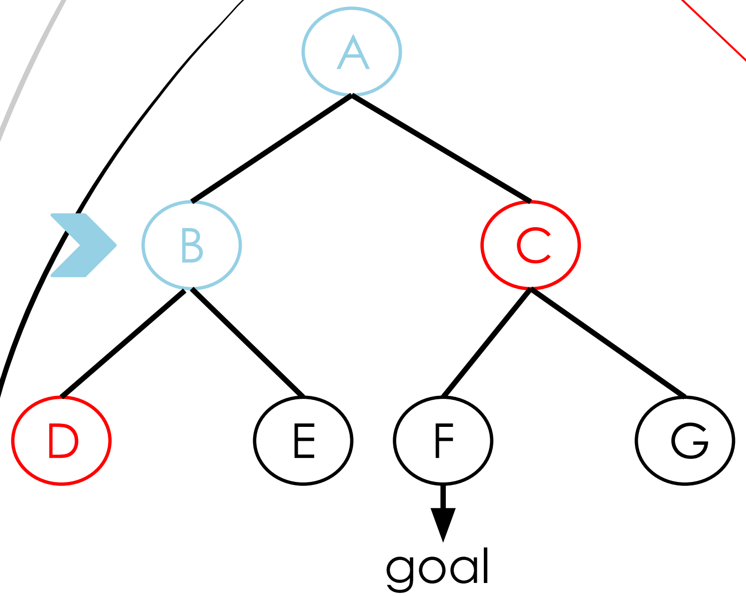

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  Queue ← a FIFO queue with source node as the only element
  explored ← an empty set
  if source node == goal then return SOLUTION
  loop do
     if EMPTY?( Queue ) then return failure
     node ← POP( Queue )  /* chooses the shallowest node in Queue */
     add node to explored
     for each node do
     adjacent ← ADJACENT-NODE( node )
        if adjacent is not in explored or Queue then
           Queue ← INSERT( adjacent )
           if adjacent == goal then return SOLUTION
```

Node
Child

Breadth First Search

Queue = [C]
Explored = [A, B]

Queue = [C, D]
Explored = [A, B]



Is D a goal state?

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

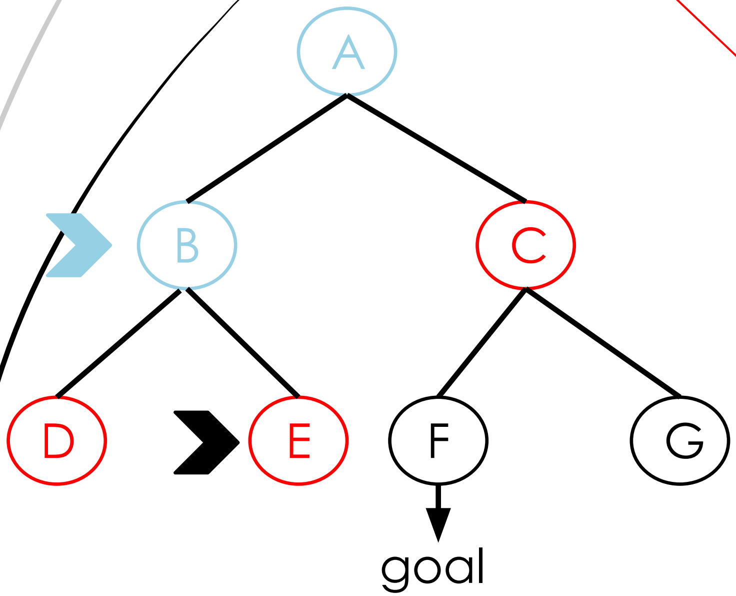
```
Queue ← a FIFO queue with source node as the only element
explored ← an empty set
if source node == goal then return SOLUTION
loop do
    if EMPTY?( Queue ) then return failure
    node ← POP( Queue ) /* chooses the shallowest node in Queue */
    add node to explored
    for each node do
    adjacent ← ADJACENT-NODE( node )
        if adjacent is not in explored or Queue then
            Queue ← INSERT( adjacent )
            if adjacent == goal then return SOLUTION
```

Node
Child

Breadth First Search

Queue = [C, D]
Explored = [A, B]

Queue = [C, D, E]
Explored = [A, B]




```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  Queue ← a FIFO queue with source node as the only element
  explored ← an empty set
  if source node == goal then return SOLUTION
  loop do
    if EMPTY?( Queue ) then return failure
    node ← POP( Queue )  /* chooses the shallowest node in Queue */
    add node to explored
    for each node do
    adjacent ← ADJACENT-NODE( node )
      if adjacent is not in explored or Queue then
        Queue ← INSERT( adjacent )
        if adjacent == goal then return SOLUTION
```

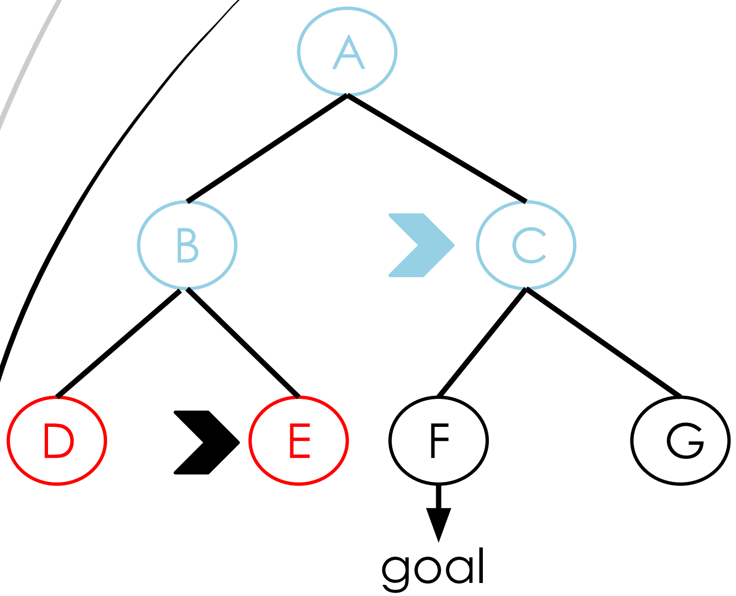
Is E a goal state?

Node
Child

Breadth First Search

Queue = [C, D, E]
Explored = [A, B]

Queue = [D, E]
Explored = [A, B, C]



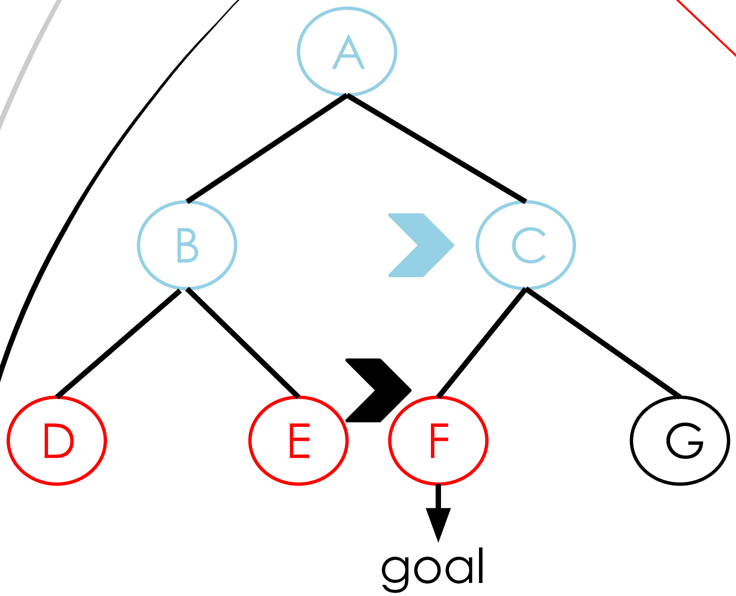
function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  Queue ← a FIFO queue with *source node* as the only element
  explored ← an empty set
  if *source node* == *goal* **then return** SOLUTION
  loop do
    if EMPTY?(*Queue*) **then return** failure
    node ← POP(*Queue*) /* chooses the shallowest node in *Queue* */
    add *node* to *explored*
    for each *node* **do**
    adjacent ← ADJACENT-NODE(*node*)
      if *adjacent* is not in *explored* or *Queue* **then**
        Queue ← INSERT(*adjacent*)
        if *adjacent* == *goal* **then return** SOLUTION

Nod
e
Chil
d

Breadth First Search

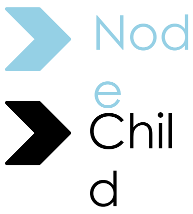
Queue = [D, E]
Explored = [A, B, C]

Queue = [D, E, F]
Explored = [A, B, C]

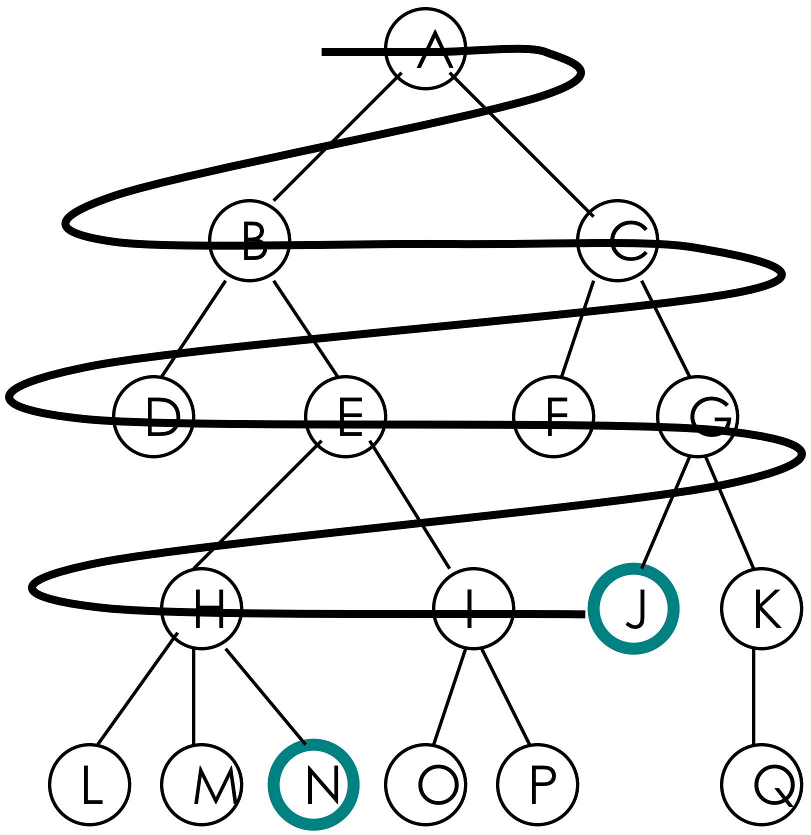


function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure
Queue ← a FIFO queue with *source node* as the only element
explored ← an empty set
if *source node* == *goal* **then return** SOLUTION
loop do
if EMPTY?(*Queue*) **then return** failure
node ← POP(*Queue*) /* chooses the shallowest node in *Queue* */
add *node* to *explored*
for each *node* **do**
adjacent ← ADJACENT-NODE(*node*)
if *adjacent* is not in *explored* or *Queue* **then**
Queue ← INSERT(*adjacent*)
if *adjacent* == *goal* **then return** SOLUTION

Is F a goal state?



BFS: Summary



BFS: Completeness

- Completeness: Yes (guaranteed to find a solution if there exists one)

- if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite) .

the branching factor is the number of children at each node, the outdegree.

BFS: Optimal?

- Not necessarily optimal
- Only optimal if every action has same cost.

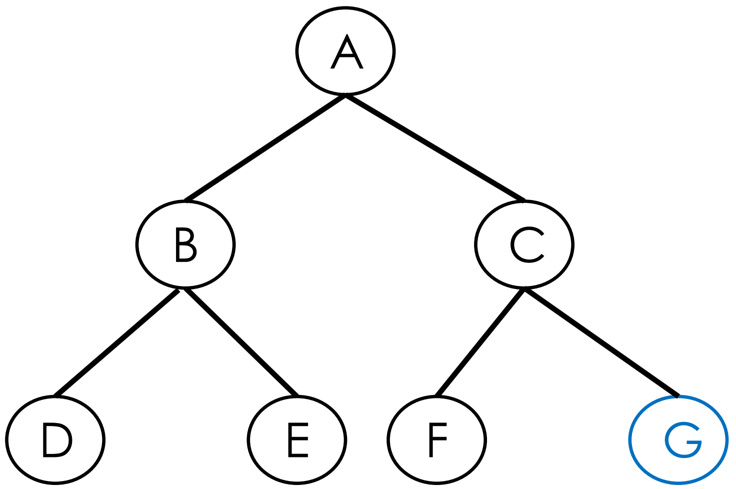
BFS: Time Complexity?

- Worst complexity is when G is a goal state.
- In this case, total number of nodes generated is

$$b+b^2+b^3 + \ldots + b^d = O(b^d)$$

(The d^{th} layer contains nodes much larger than all the nodes in previous layers combined!)

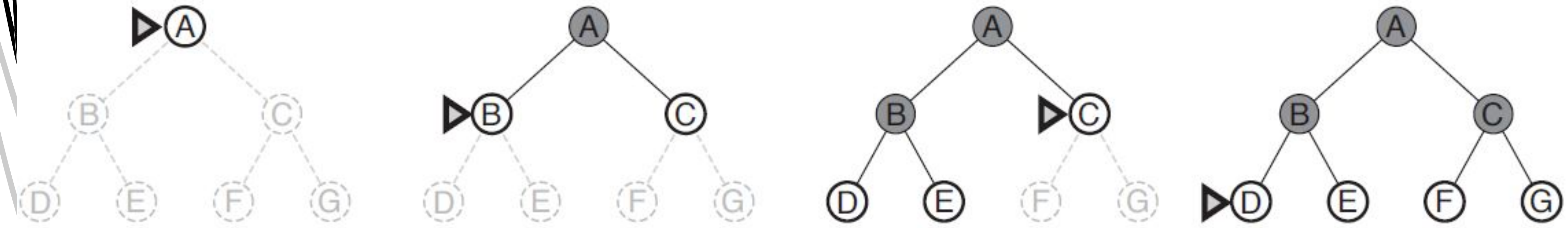
So the time complexity is $O(b^d)$

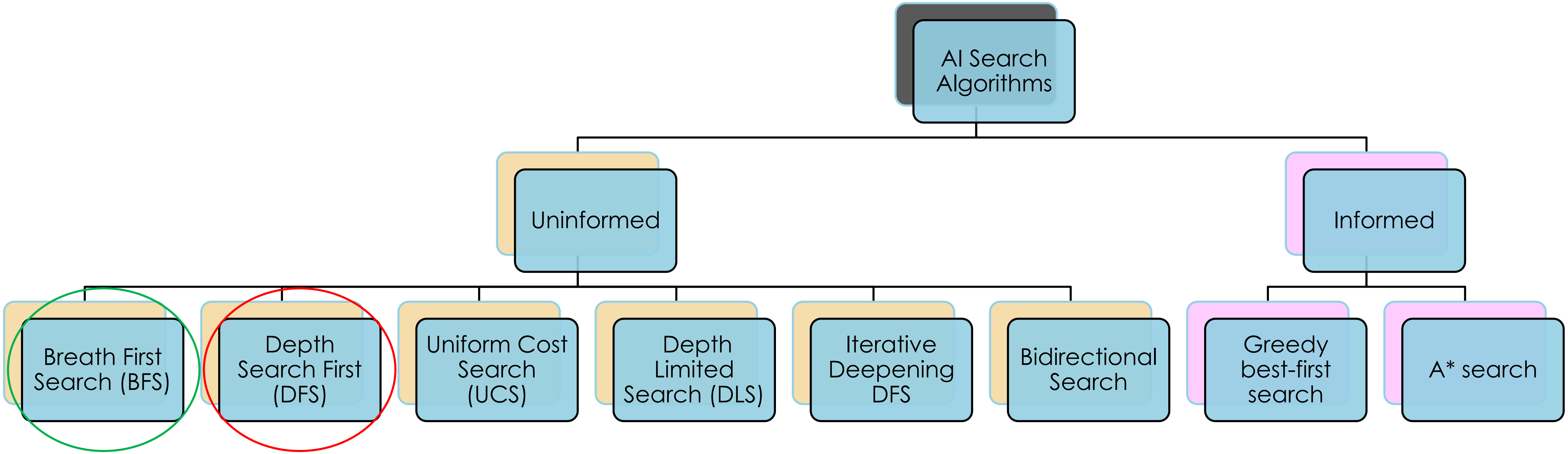


A binary tree, b=2, d=2

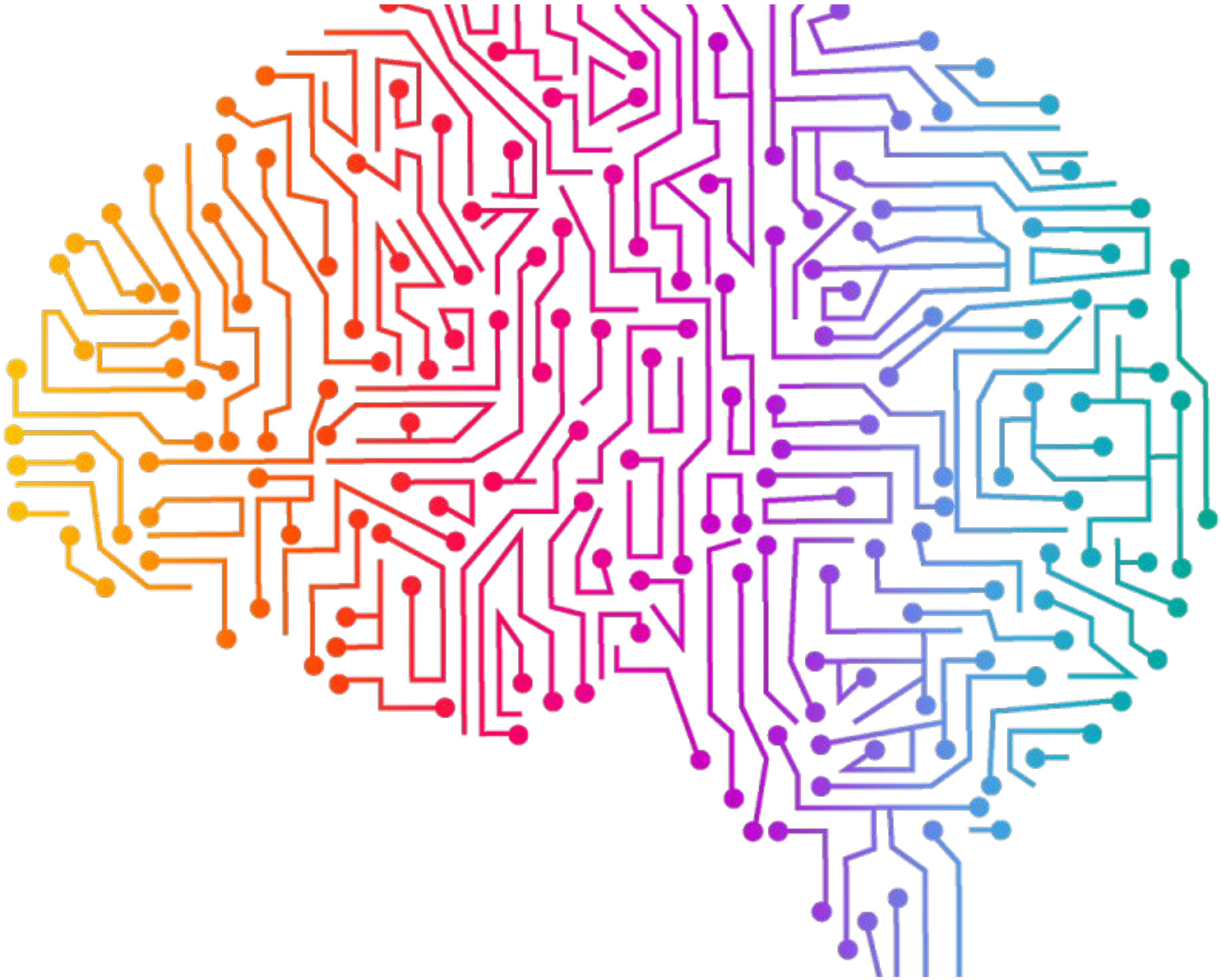
BFS: Space Complexity?

- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the queue.
- So the space complexity is $O(b^d)$, i.e., it is dominated by the size of the queue.
- Exponential time complexity can be accepted but exponential space complexity is BAD !



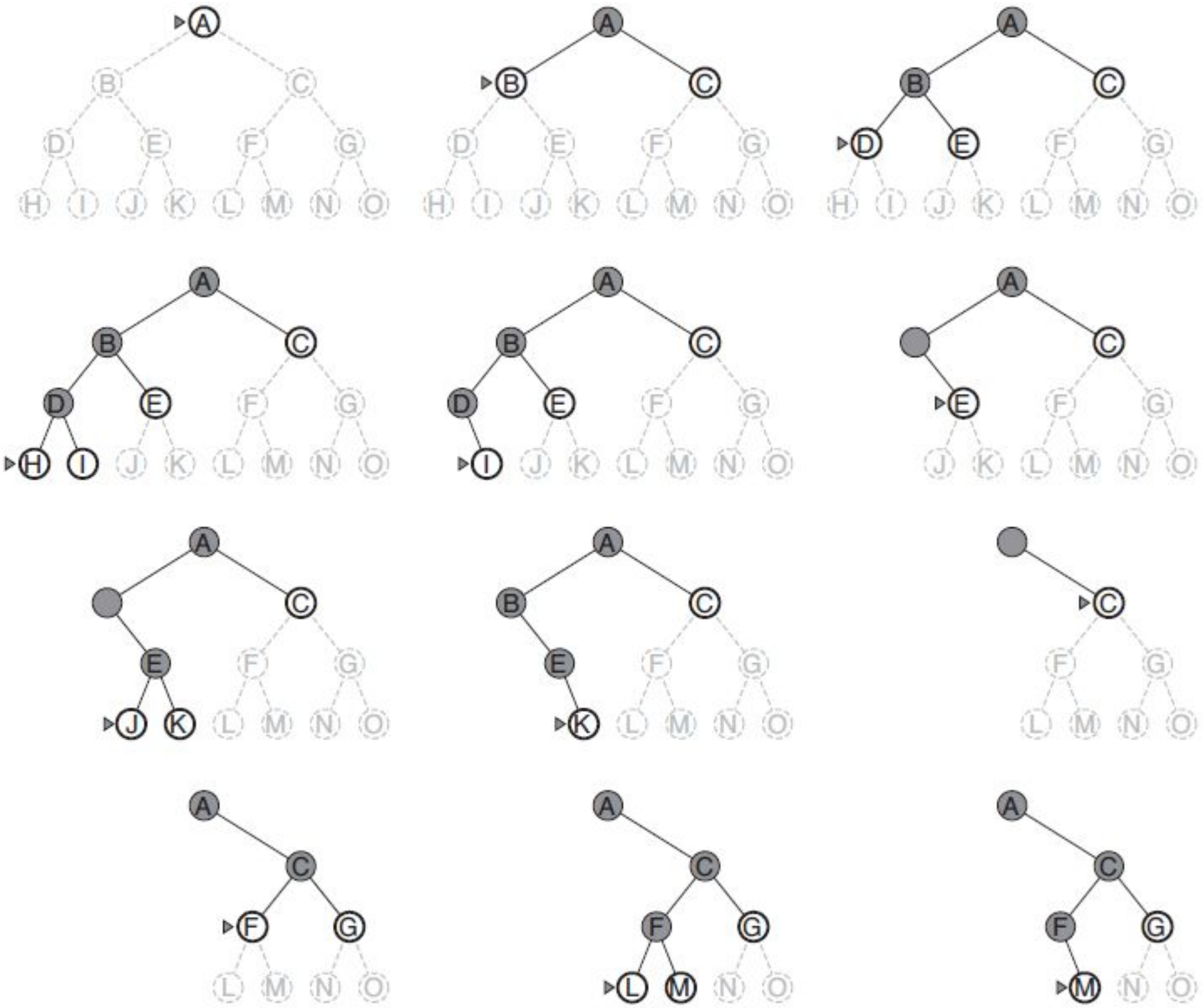

AI Search Algorithms
Uninformed
Informed
Breath First Search (BFS)
Depth Search First (DFS)
Uniform Cost Search (UCS)
Depth Limited Search (DLS)
Iterative Deepening DFS
Bidirectional Search
Greedy best-first search
A* search

Depth First Search (DFS)



Depth First Search

- It works vertically.
- It doesn't generate a solution always.
- It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
- Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backward on the same path to find nodes to traverse.



M is the goal node

Depth First Search

- Expand Depth Node First

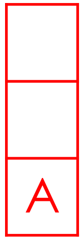
  (Depth First Search)

- *Stack*: nodes in the stack to be explored
- explored: Nodes that are already explored
- *Stack* is a Last-in-first-out (LIFO) queue.

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  Stack    ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(Stack) then return failure
     node ← POP(Stack)   /* chooses the top node in Stack */
     add node to explored
     if node == goal then return SOLUTION
     for all adjacent node (n) of node
        if n is not in explored or Stack then
           Stack   ← INSERT(n)
```

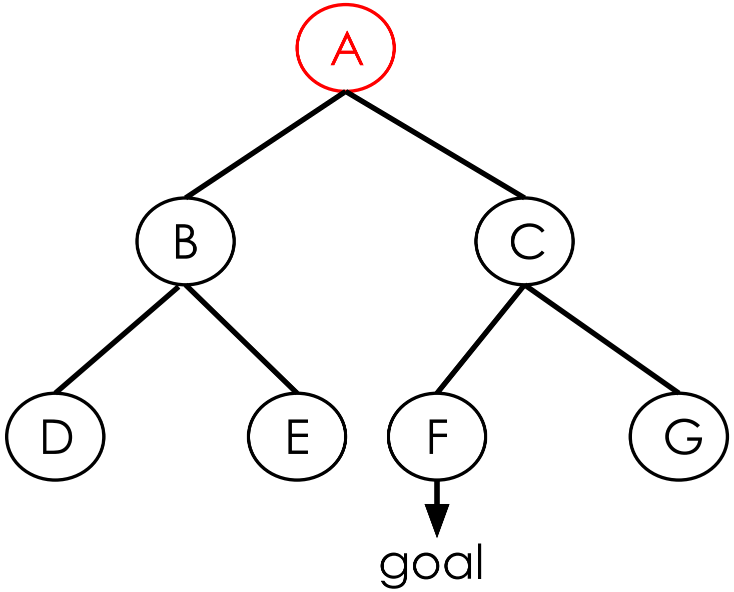
Depth First Search

Stack =



Explored = [ ]

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
    Stack    ← a LIFO with source node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(Stack) then return failure
        node ← POP(Stack)   /* chooses the top node in Stack */
        add node to explored
        if node == goal then return SOLUTION
        for all adjacent node (n) of node
            if n is not in explored or Stack then
                Stack   ← INSERT(n)
```



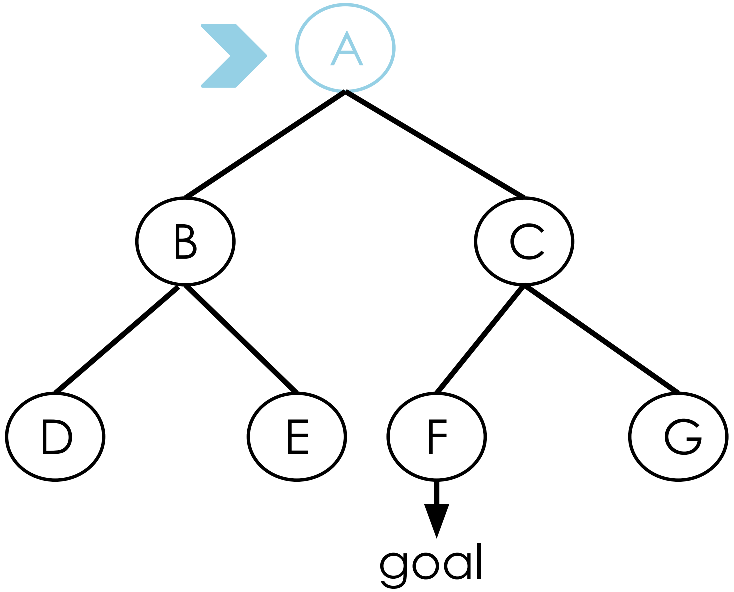
Stack = [| | A]

Explored = [ ]

Stack = [| |]

Is A a goal state?

Explored = [A]



function DEPTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

Stack ← a LIFO with *source node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*Stack*) **then return** failure
  node ← POP(*Stack*) /* chooses the top node in *Stack* */
  add *node* to *explored*
  if *node* == *goal* **then return** SOLUTION
  for all *adjacent node (n)* of *node*
    if *n* is not in *explored* or *Stack* **then**
      Stack ← INSERT(*n*)

Node

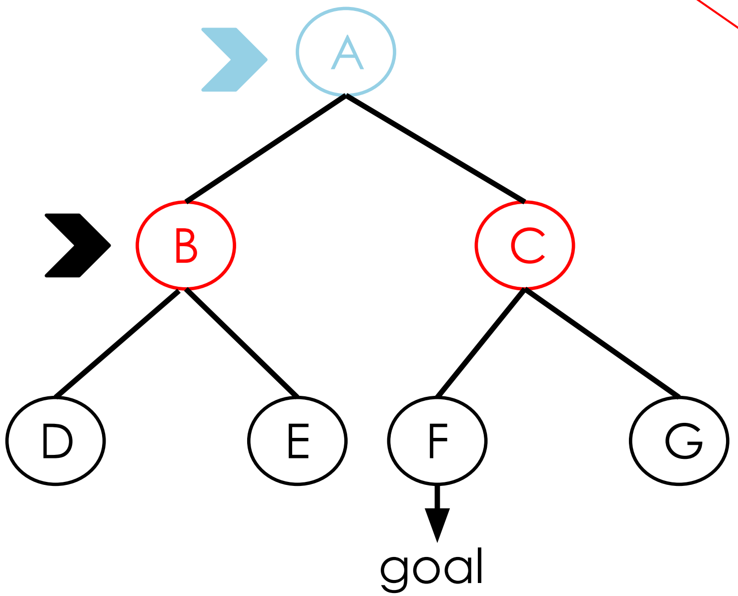
Child

Stack =

Explored = [A]

Stack = | B | C |

Explored = [A]



Node

Child

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  Stack ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(Stack) then return failure
     node ← POP(Stack)  /* chooses the top node in Stack */
     add node to explored
     if node == goal then return SOLUTION
     for all adjacent node (n) of node
        if n is not in explored or Stack then
           Stack ← INSERT(n)
```

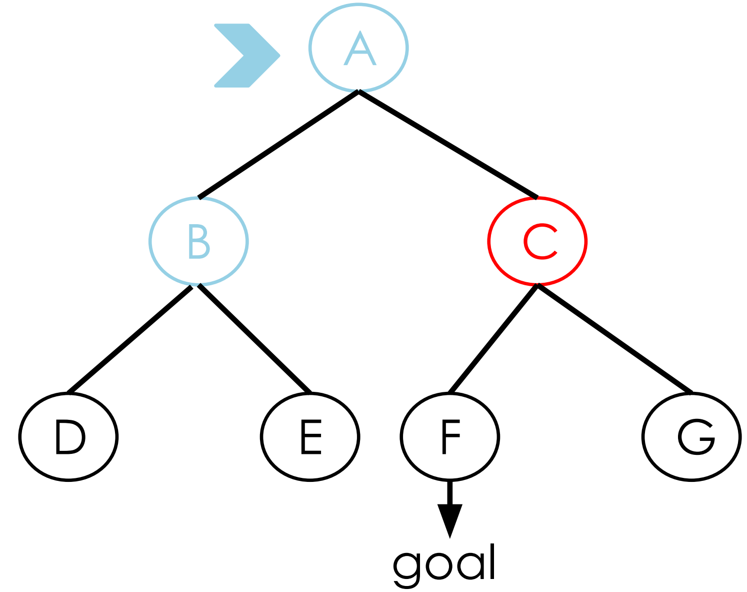
Stack = [B, C]

Explored = [A]

Stack = [C]

Is B a goal state?

Explored = [A, B]




```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  Stack ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(Stack) then return failure
     node ← POP(Stack)  /* chooses the top node in Stack */
     add node to explored
     if node == goal then return SOLUTION
     for all adjacent node (n) of node
        if n is not in explored or Stack then
           Stack ← INSERT(n)
```

Node

Child

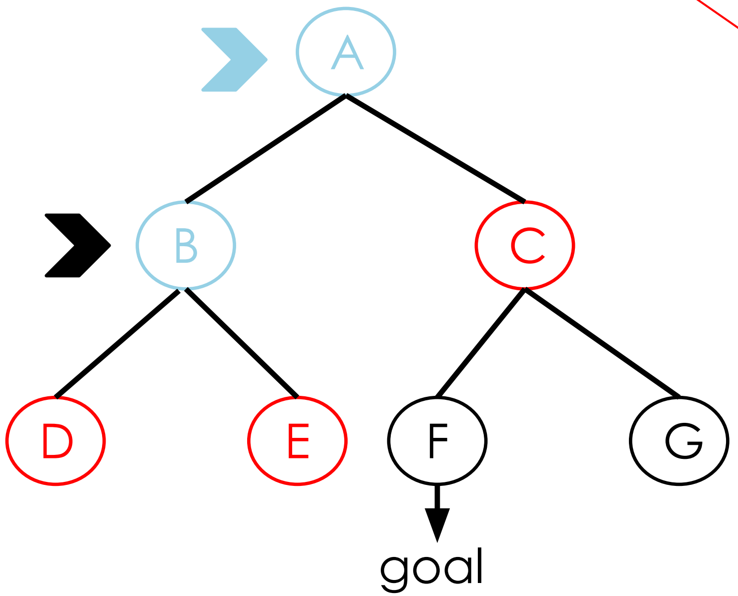
Stack = [, , C]

Explored = [A, B]

Stack = [D, E, C]

Explored = [A, B]

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  Stack ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(Stack ) then return failure
     node ← POP(Stack )  /* chooses the top node in Stack */
     add node to explored
     if node == goal then return SOLUTION
     for all adjacent node (n) of node
        if n is not in explored or Stack then
           Stack ← INSERT( n )
```



Node

Child

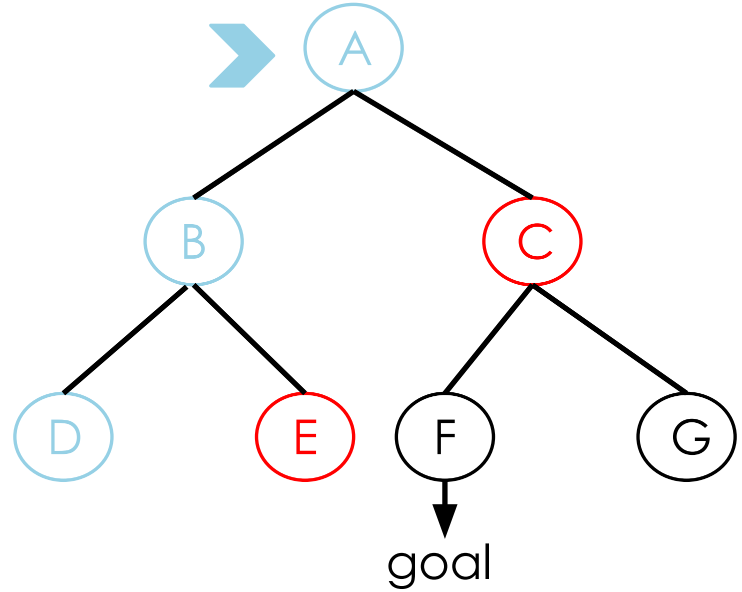
Stack = [D, E, C]

Explored = [A, B]

Stack = [, E, C]

Is D a goal state?

Explored = [A, B, D]



function DEPTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

```
Stack ← a LIFO with source node as the only element
explored ← an empty set
loop do
    if EMPTY?(Stack ) then return failure
    node ← POP(Stack )   /* chooses the top node in Stack */
    add node to explored
    if node == goal then return SOLUTION
    for all adjacent node (n) of node
        if n is not in explored or Stack then
            Stack ← INSERT( n )
```

Node

Child

Stack = [, E, C]

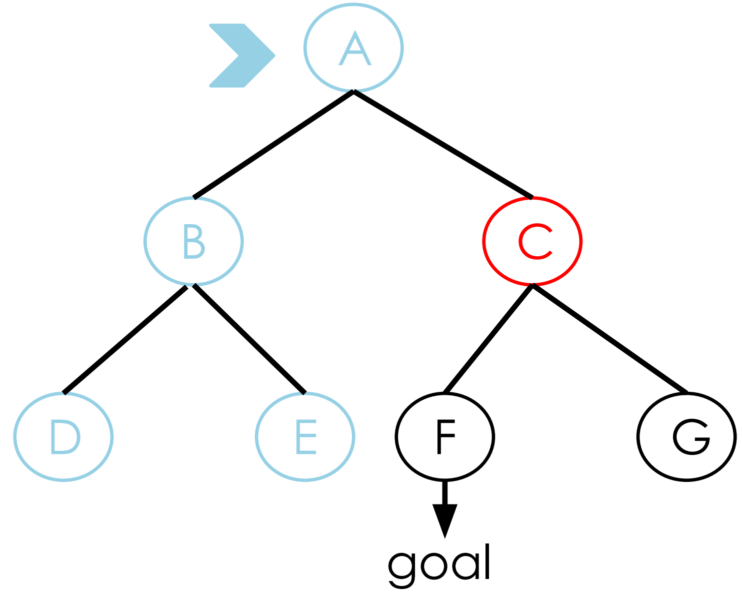
Explored = [A, B, D]

Stack = [, , C]

Is E a goal state?

Explored = [A, B, D, E]

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
   Stack   ← a LIFO with source node as the only element
   explored ← an empty set
   loop do
      if EMPTY?(Stack ) then return failure
      node ← POP(Stack )   /* chooses the top node in Stack */
      add node to explored
      if node == goal then return SOLUTION
      for all adjacent node (n) of node
         if n is not in explored or Stack then
            Stack   ← INSERT( n )
```



Node

Child

Stack =

C

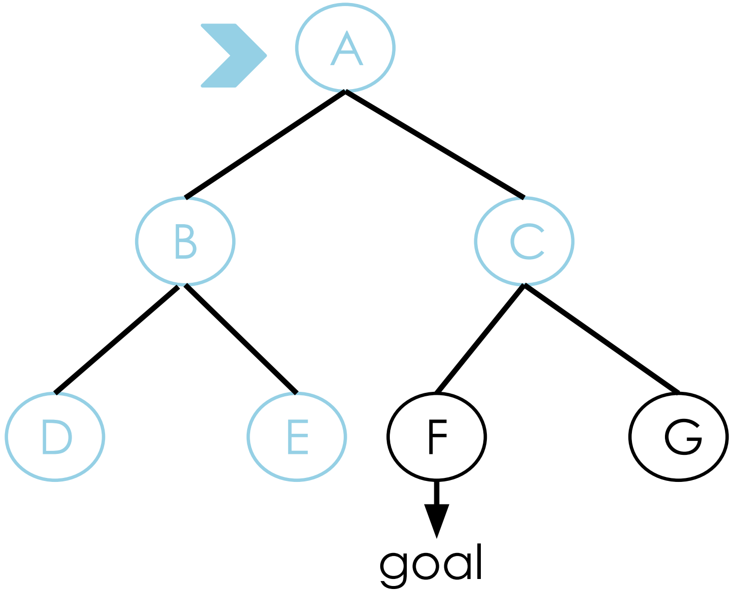
Explored = [A, B, D, E]

Stack =

Is C a goal state?

Explored = [A, B, D, E, C]

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  Stack  ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(Stack ) then return failure
     node ← POP(Stack )  /* chooses the top node in Stack */
     add node to explored
     if node == goal then return SOLUTION
     for all adjacent node (n) of node
        if n is not in explored or Stack then
           Stack  ← INSERT( n )
```



Node

Child

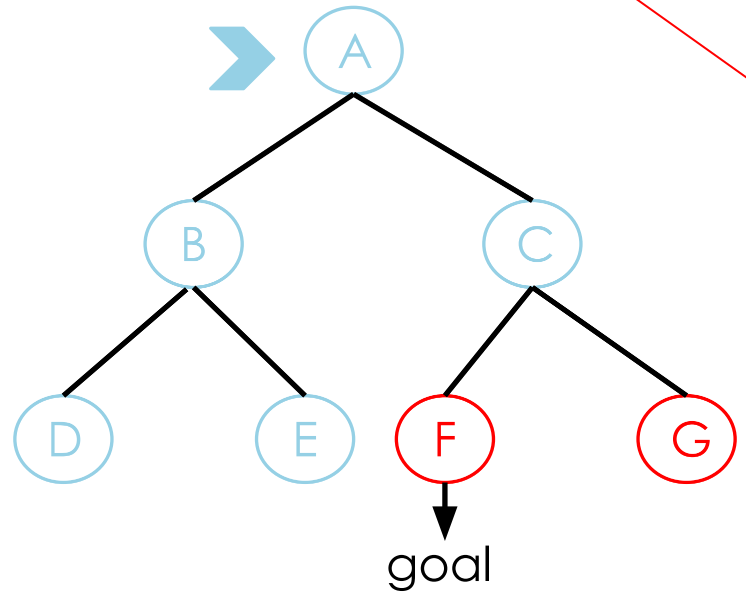
Stack = [ ] [ ] [ ]

Explored = [A, B, D, E , C]

Stack = [ ] [F] [G]

Explored = [A, B, D, E, C]

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  Stack ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(Stack) then return failure
     node ← POP(Stack)  /* chooses the top node in Stack */
     add node to explored
     if node == goal then return SOLUTION
     for all adjacent node (n) of node
        if n is not in explored or Stack then
           Stack ← INSERT( n )
```



Node

Child

Stack = F, G

Explored = [A, B, D, E , C]

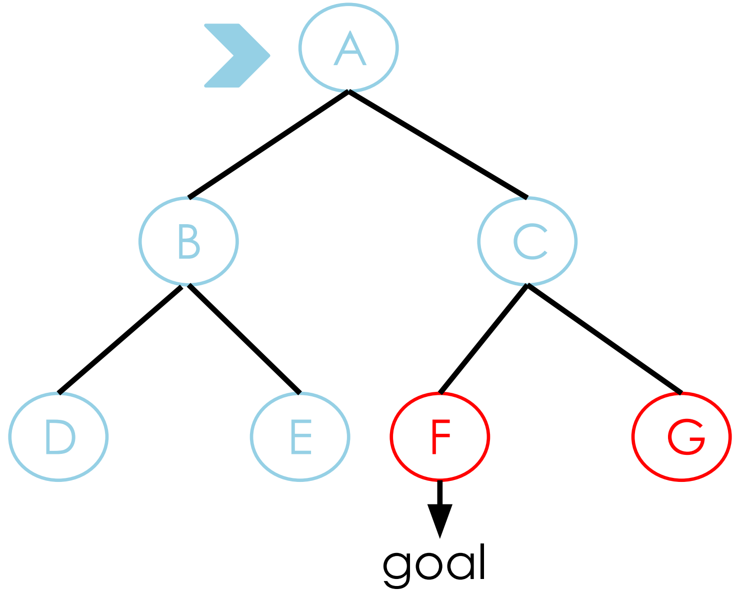
Stack = G

Is F a goal state?

Explored = [A, B, D, E, C, F]

function DEPTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

  Stack ← a LIFO with *source node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*Stack*) **then return** failure
    node ← POP(*Stack*) /* chooses the top node in *Stack* */
    add *node* to *explored*
    if *node* == *goal* **then return** SOLUTION
    for all *adjacent node (n)* of *node*
      if *n* is not in *explored* or *Stack* **then**
        Stack ← INSERT(*n*)



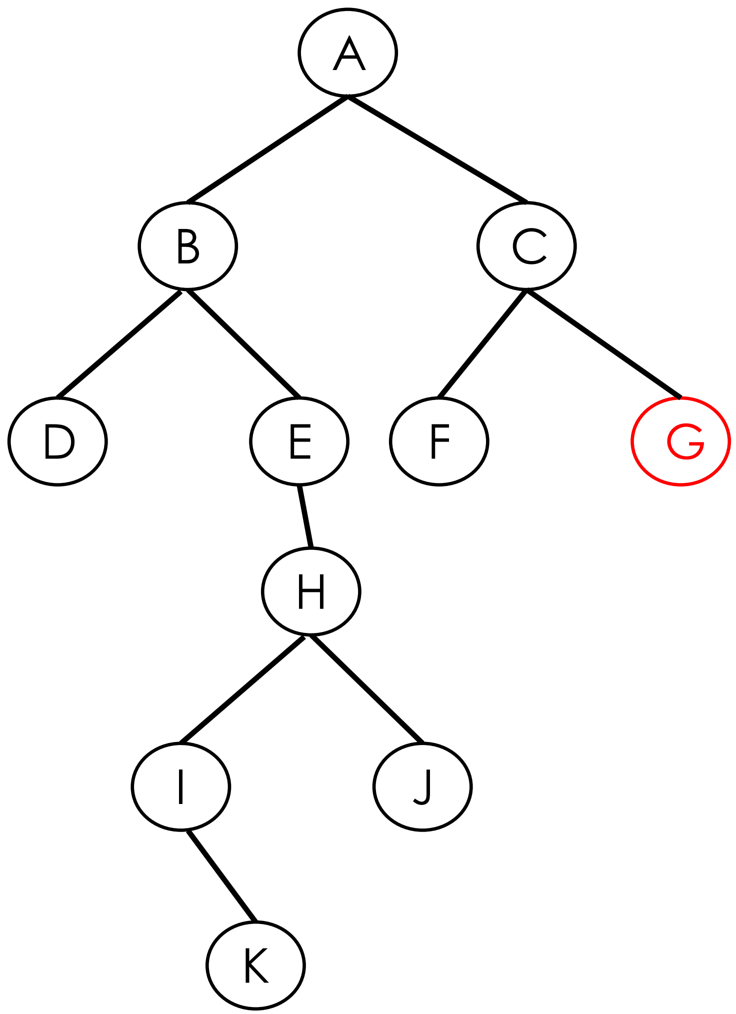
Node

Child

Completeness?

- Goal, G is at depth, d=2
- Max depth of tree, m = 5

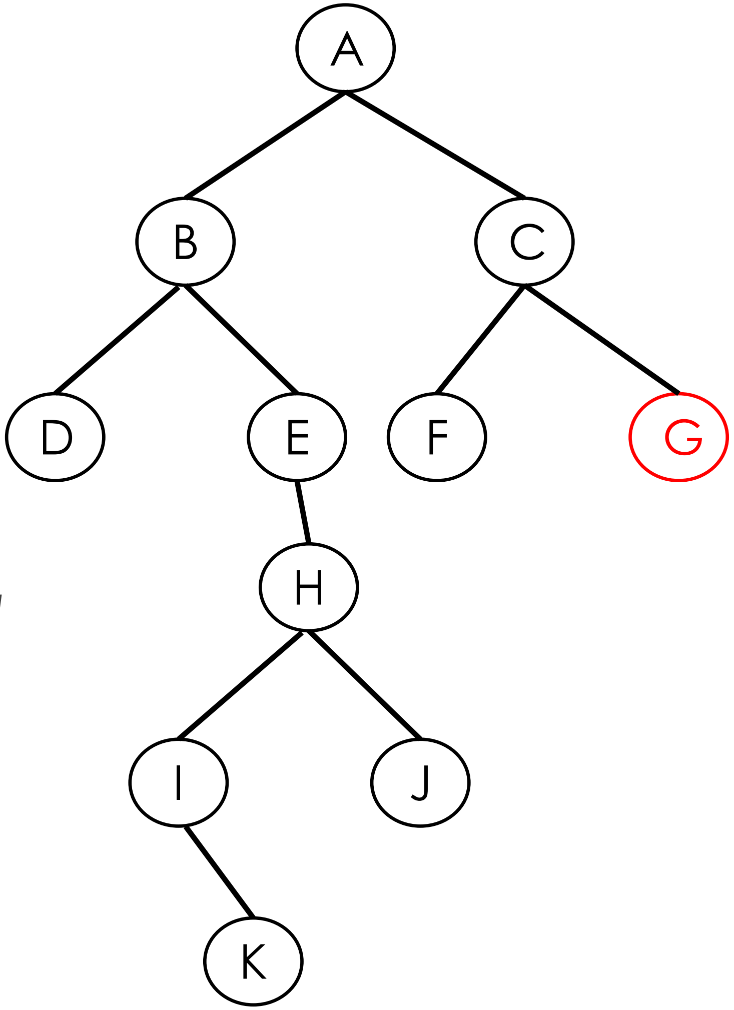
- DFS is complete ONLY if m is finite



Time Complexity

- Number of nodes visited? $O(b^m)$
- For BFS, remember it was $O(b^d)$

- DFS is terrible if *m* is much larger than *d*
  - but if solutions are dense (goal is at K instead of G), DFS is faster than BFS.

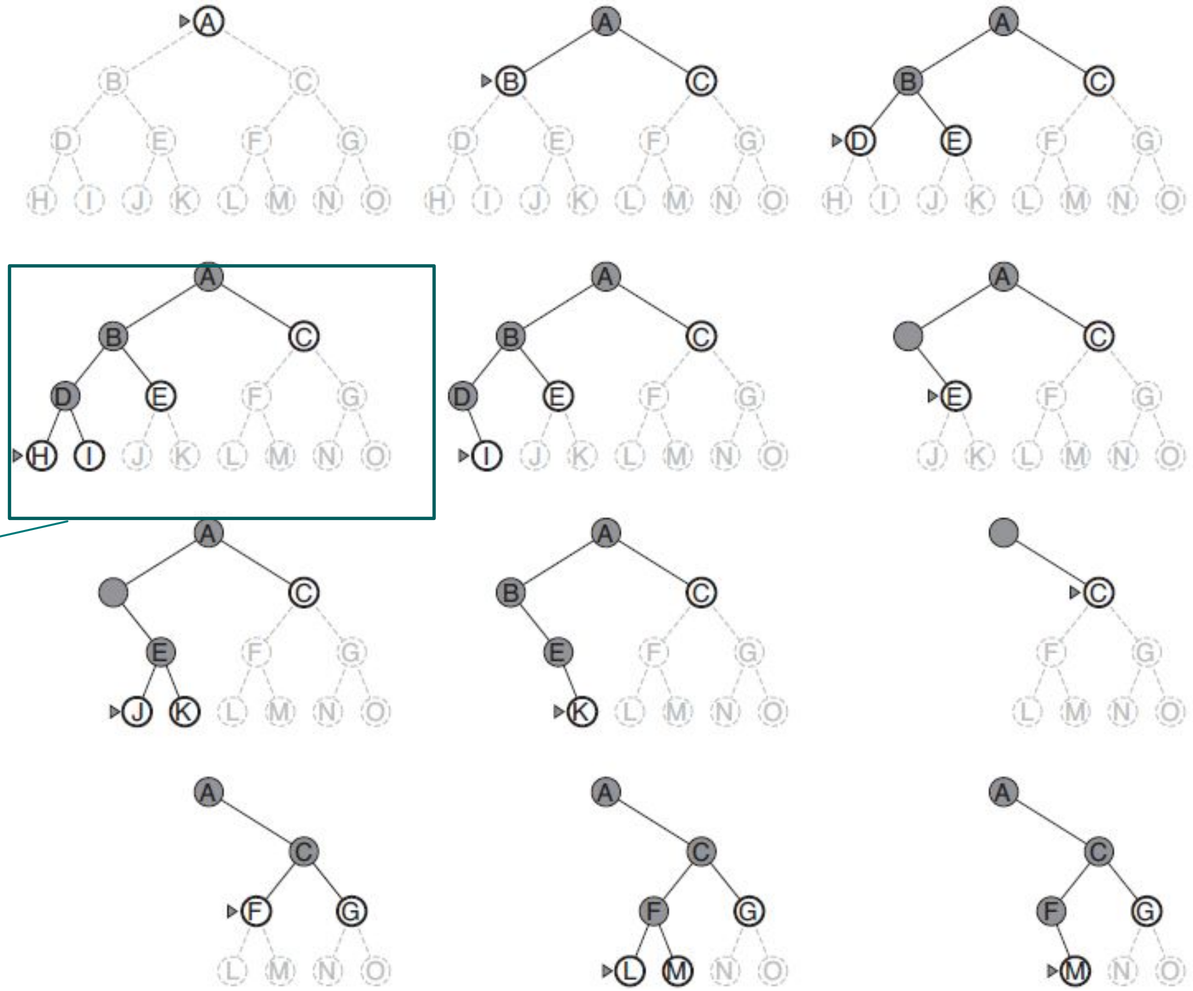


Space Complexity?

- *O(bm)*, i.e., linear space!

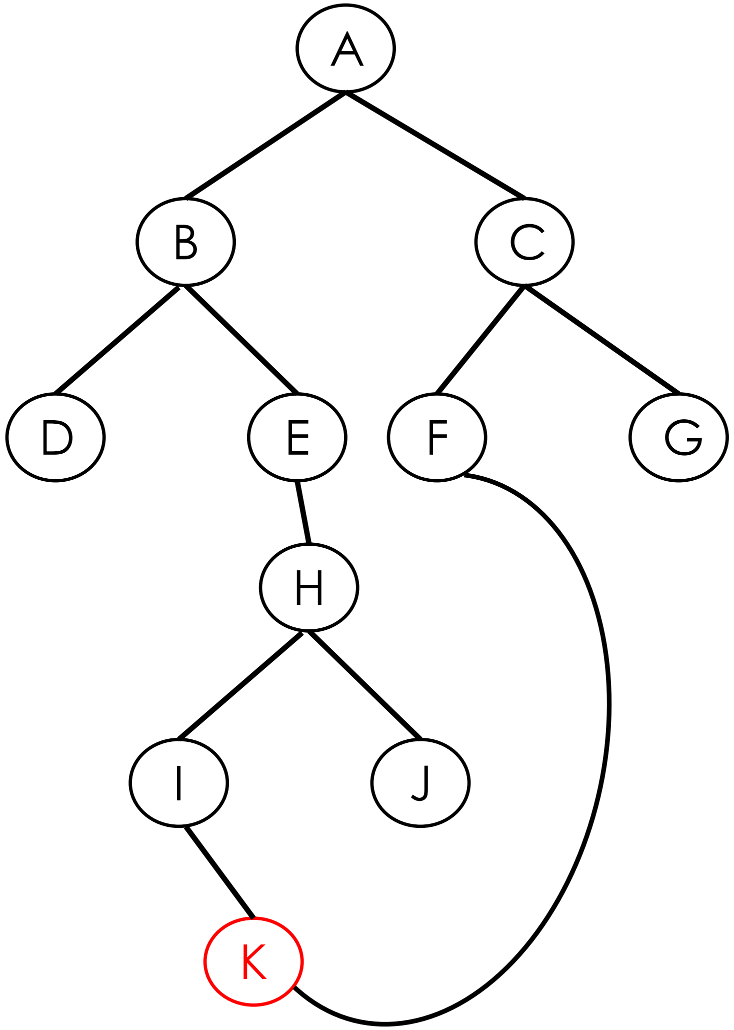
(we only need to remember a single path + expanded unexplored nodes)

- O(bm)? E.g. m=3, b=2 (we need to save max of 3*2=6 nodes in memory)



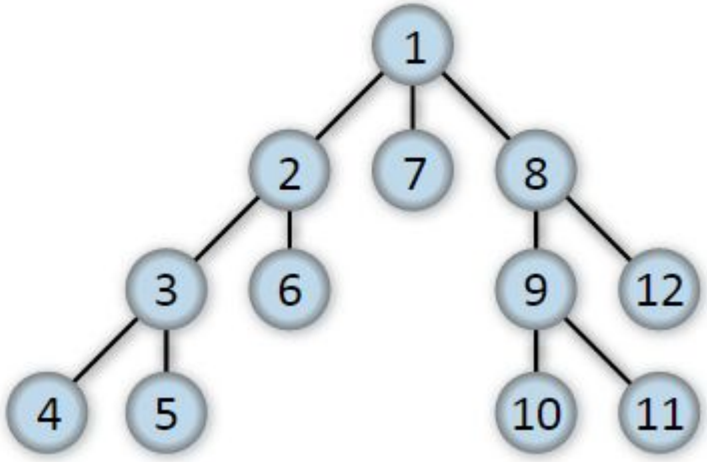
Optimal?

- Remember BFS was optimal if each step has same cost.
- Even if this is the case, DFS is not guaranteed to find the optimum path to the goal.
- For the graph on the right, BFS will return optimal path but DFS won't (assuming each step has same cost)



Class Activity

- Find the Goal Node 9 with the help of DFS using Stack.



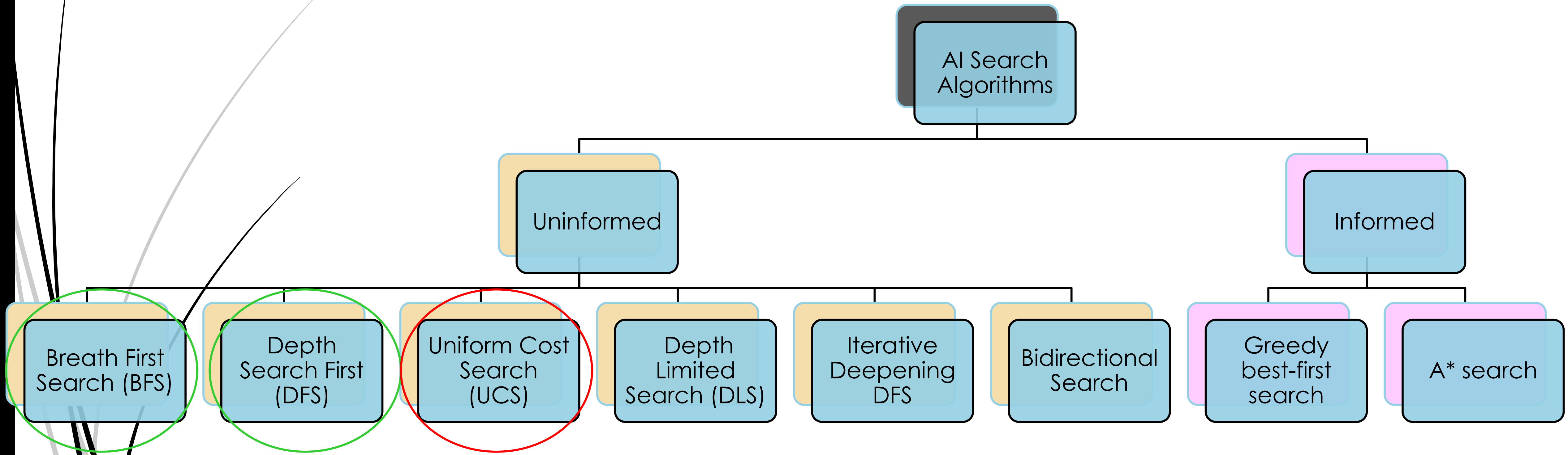
Difference between BFS and DFS

BFS

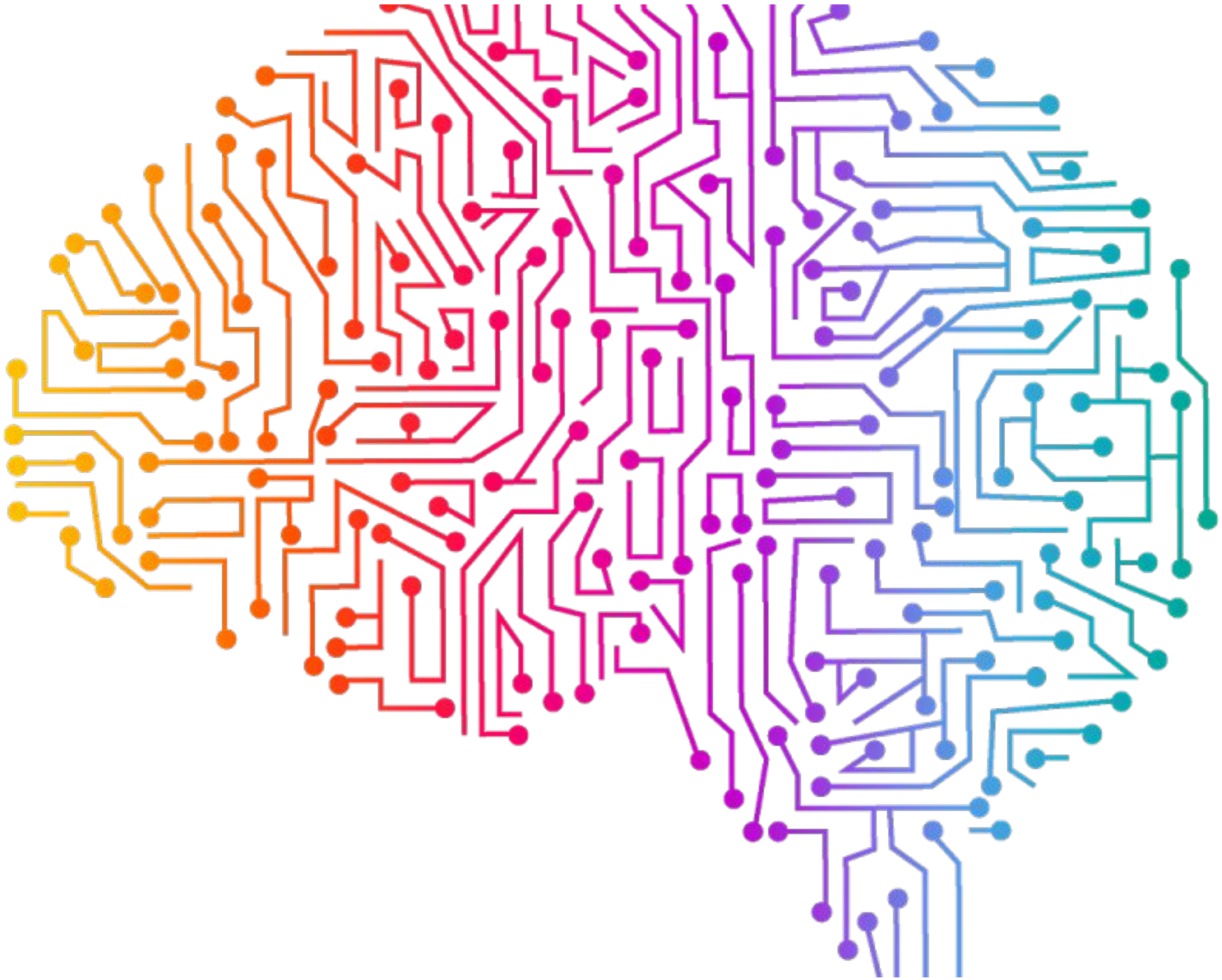
- Visit nodes level by level.
- Uses Queue data structure.
- May give an optimal solution.
- Slower, need more memory.

DFS

- Visit nodes graph depth-wise.
- Uses Stack data structure.
- Do not guarantee an optimal solution.
- Faster, need less memory.


AI Search Algorithms
Uninformed
Informed
Breath First Search (BFS)
Depth Search First (DFS)
Uniform Cost Search (UCS)
Depth Limited Search (DLS)
Iterative Deepening DFS
Bidirectional Search
Greedy best-first search
A* search

Uniform Cost Search (UCS)



Uniform Cost Search (UCS) --- How it is different from BFS and DFS

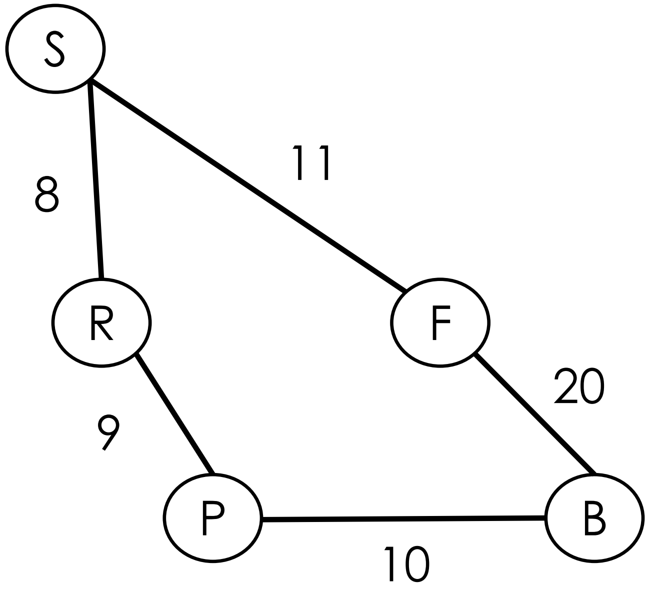
- In contrast to BFS and DFS algorithms that don't take into consideration either the cost between two nodes, the UCS algorithm uses the path's cost from the initial node to the current node as the extension criterion.
- Starting from the initial state (starting node), the UCS algorithm, in each step chooses the node that is closer to the initial node.
- When the algorithm finds the solution, returns the path from the initial state to the final state.
- The UCS algorithm is characterized as complete, as it always returns a solution if exists.
- Moreover, the UCS algorithm guarantees the optimum solution.

Uniform Cost Search (UCS)

- Uniform cost search modifies BFS such that it works with any cost function.
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost.
- This is done by storing the opened list as a priority queue ordered by cost.

Uniform Cost Search (UCS)

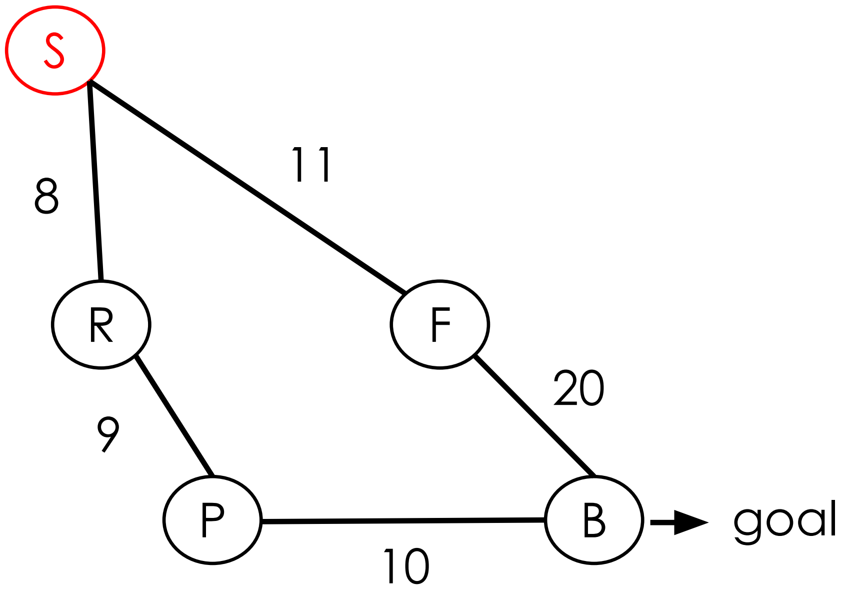
- Expand Node with lowest path cost
- *Opened*: a *priority queue* ordered by path cost
- *explored*: Nodes that are already explored



Uniform Cost Search (UCS)

Opened	Explored
{(S,0)}	{-}

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
opened ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*opened*) **then return** failure
  node ← POP(*opened*)   /* chooses the lowest-cost node in *opened* */
  if *node* == *goal* **then return** SOLUTION
  add *node* to *explored*
  for each *adjacent node (n)* of *node* **do**
    if *n* is not in *explored* or *opened* **then**
      opened ← INSERT(*n*)
    else if *n* is in *opened* with higher PATH-COST **then**
      replace that *opened* node with *n*

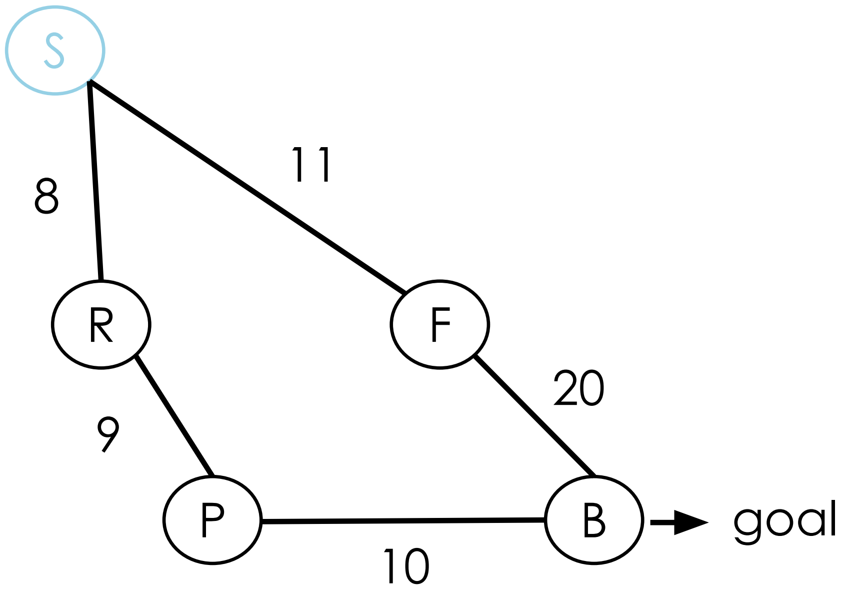


Uniform Cost Search (UCS)

Opened	Explored
{(S,0)}	{-}
	{(S,0)}

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  opened ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(opened) then return failure
     node ← POP(opened)  /* chooses the lowest-cost node in opened */
     if node == goal then return SOLUTION
     add node to explored
     for each adjacent node (n) of node  do
        if n is not in explored or opened then
           opened ← INSERT(n)
        else if n is in opened with higher PATH-COST then
           replace that opened node with n
```

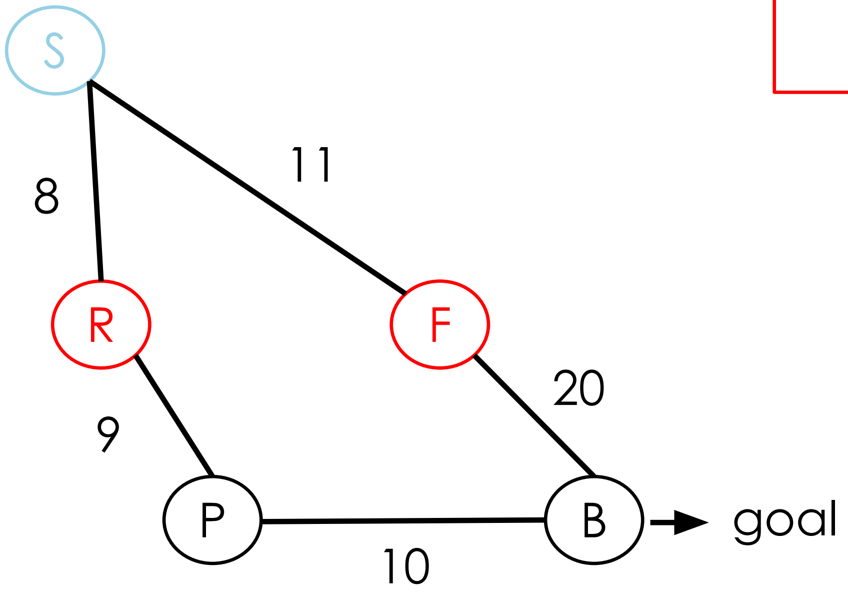
Is S a goal state?

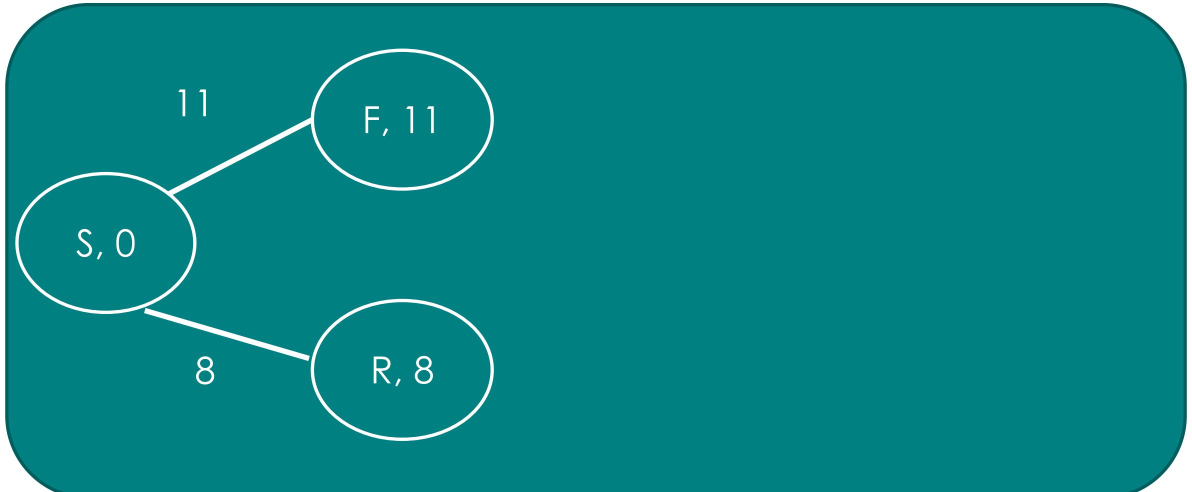


Uniform Cost Search (UCS)

Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
  opened ← a priority queue ordered by PATH-COST, with *node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*opened*) **then return** failure
    node ← POP(*opened*) /* chooses the lowest-cost node in *opened* */
    if *node* == *goal* **then return** SOLUTION
    add *node* to *explored*
    for each *adjacent node (n)* of *node* **do**
      if *n* is not in *explored* or *opened* **then**
        opened ← INSERT(*n*)
      else if *n* is in *opened* with higher PATH-COST **then**
        replace that *opened* node with *n*



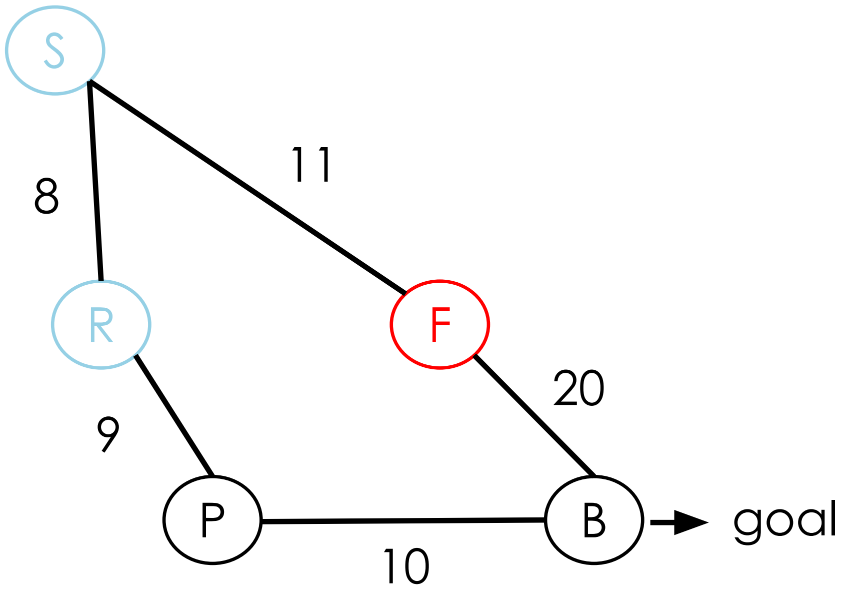


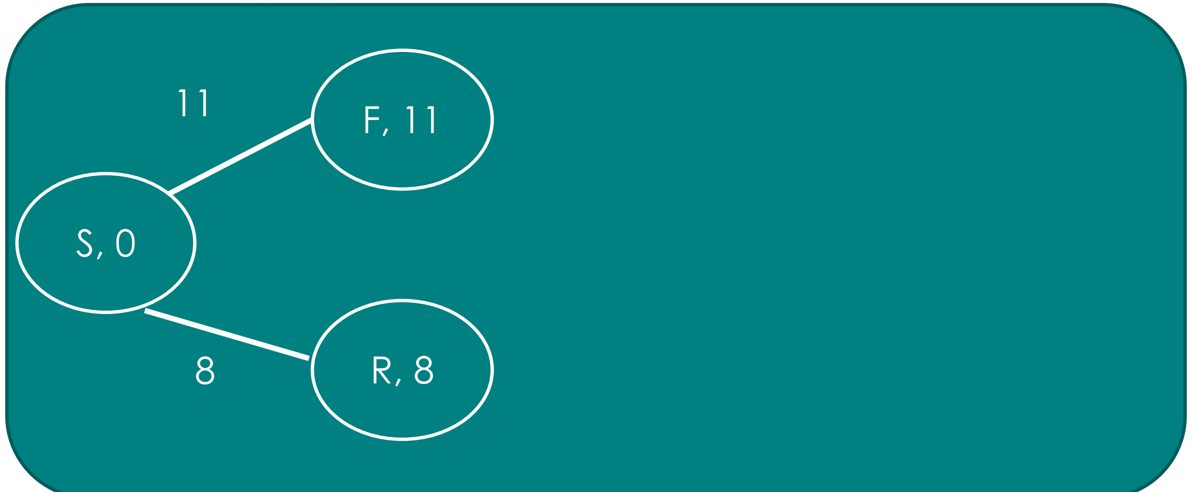
Uniform Cost Search (UCS)

Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}
{(F, 11)}	{(S,0), (R,8)}

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
  opened ← a priority queue ordered by PATH-COST, with *node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*opened*) **then return** failure
    node ← POP(*opened*) /* chooses the lowest-cost node in *opened* */
    if *node* == *goal* **then return** SOLUTION
    add *node* to *explored*
    for each *adjacent node (n)* of *node* **do**
      if *n* is not in *explored* or *opened* **then**
        opened ← INSERT(*n*)
      else if *n* is in *opened* with higher PATH-COST **then**
        replace that *opened* node with *n*

Is R a goal state?

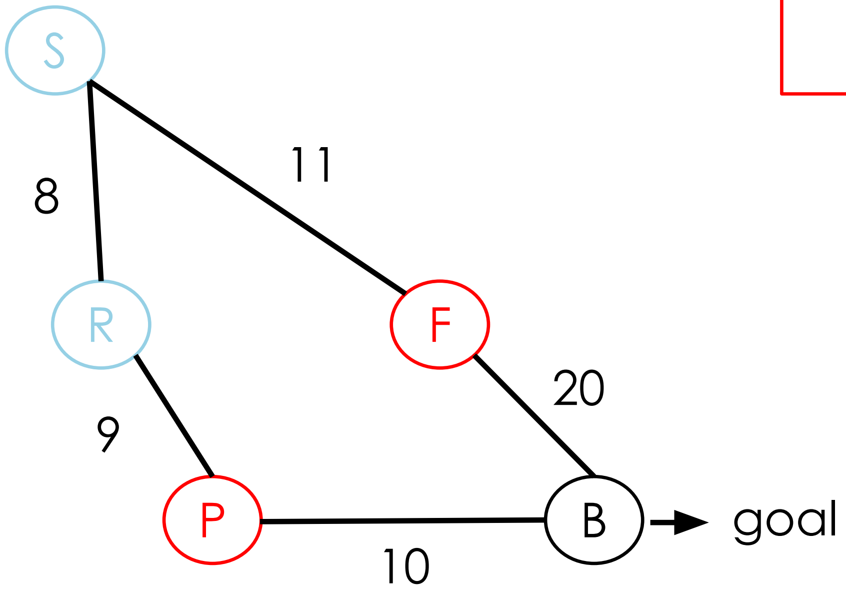


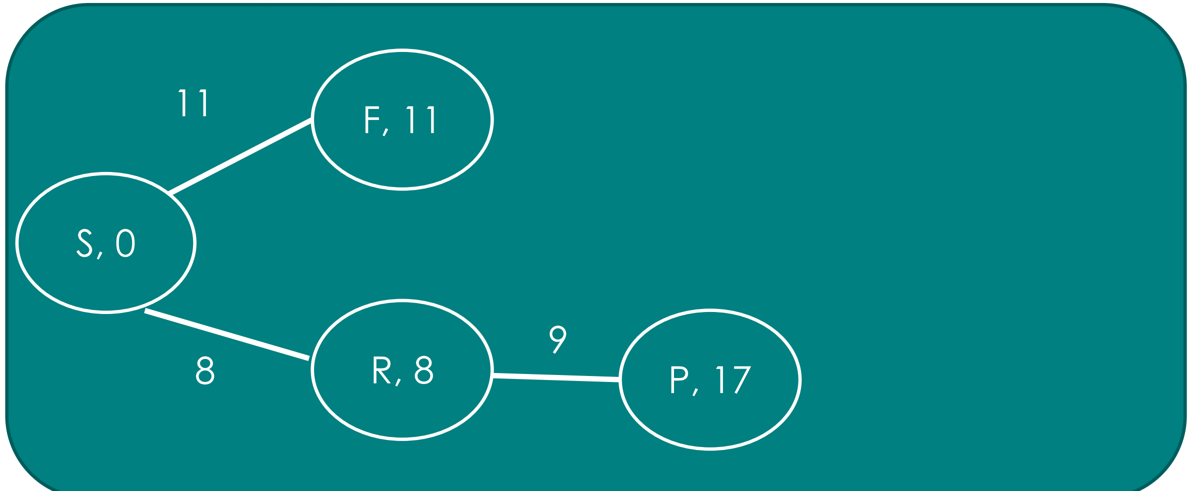


Uniform Cost Search (UCS)

Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}
{(F, 11)}	{(S,0), (R,8)}
{(F, 11), (P, 17)}	{(S,0), (R,8)}

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
  opened ← a priority queue ordered by PATH-COST, with *node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*opened*) **then return** failure
    node ← POP(*opened*) /* chooses the lowest-cost node in *opened* */
    if *node* == *goal* **then return** SOLUTION
    add *node* to *explored*
    for each *adjacent node (n)* of *node* **do**
      if *n* is not in *explored* or *opened* **then**
        opened ← INSERT(*n*)
      else if *n* is in *opened* with higher PATH-COST **then**
        replace that *opened* node with *n*



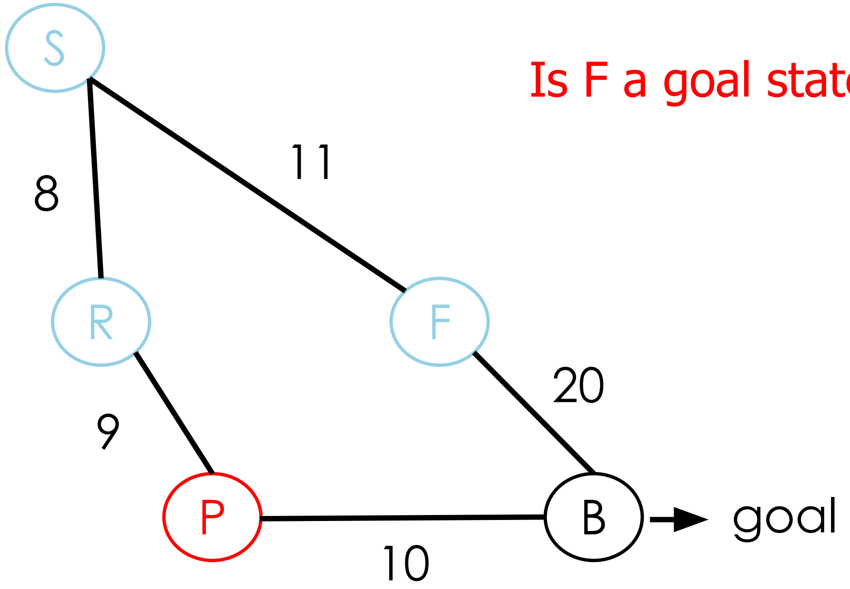


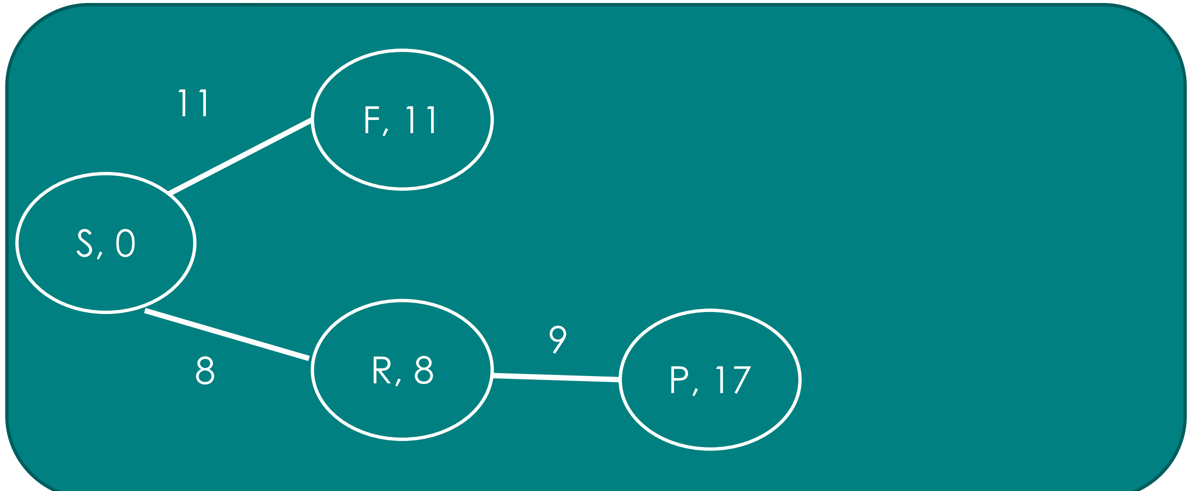
Uniform Cost Search (UCS)

Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}
{(F, 11)}	{(S,0), (R,8)}
{(F, 11), (P, 17)}	{(S,0), (R,8)}
{(P, 17)}	{(S,0), (R,8), (F, 11)}

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
   opened ← a priority queue ordered by PATH-COST, with node as the only element
   explored ← an empty set
   loop do
       if EMPTY?( opened ) then return failure
       node ← POP( opened )   /* chooses the lowest-cost node in opened */
       if node == goal then return SOLUTION
       add node to explored
       for each adjacent node (n) of node  do
           if n is not in explored or opened then
              opened ← INSERT( n )
         else if n is in opened with higher PATH-COST then
              replace that opened node with n
```

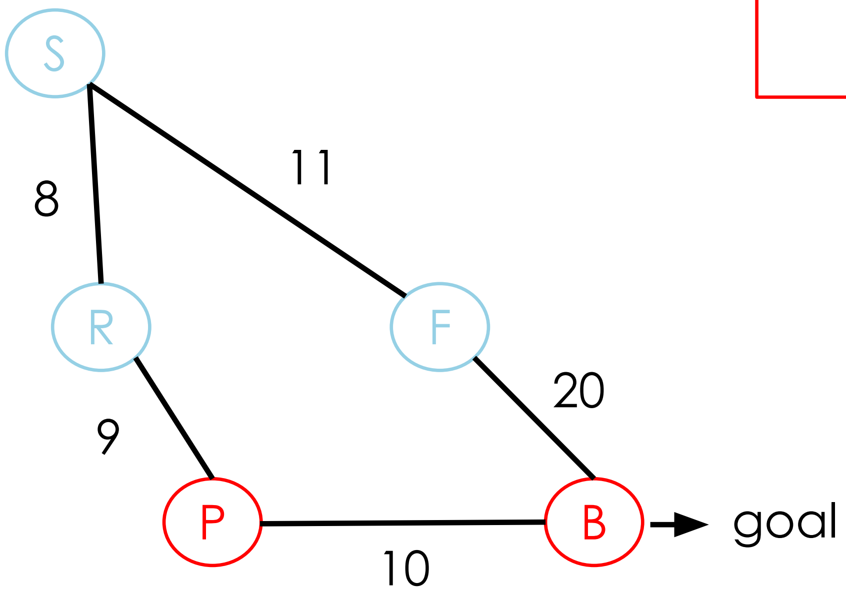
Is F a goal state?

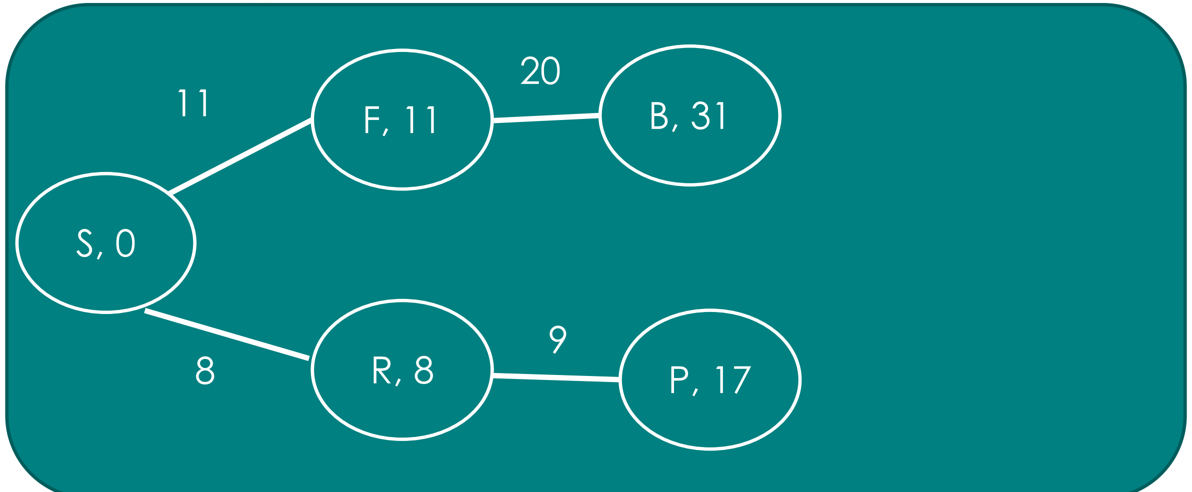




Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}
{(F, 11)}	{(S,0), (R,8)}
{(F, 11), (P, 17)}	{(S,0), (R,8)}
{(P, 17)}	{(S,0), (R,8), (F, 11)}
{(P, 17), (B, 31)}	{(S,0), (R,8), (F, 11)}

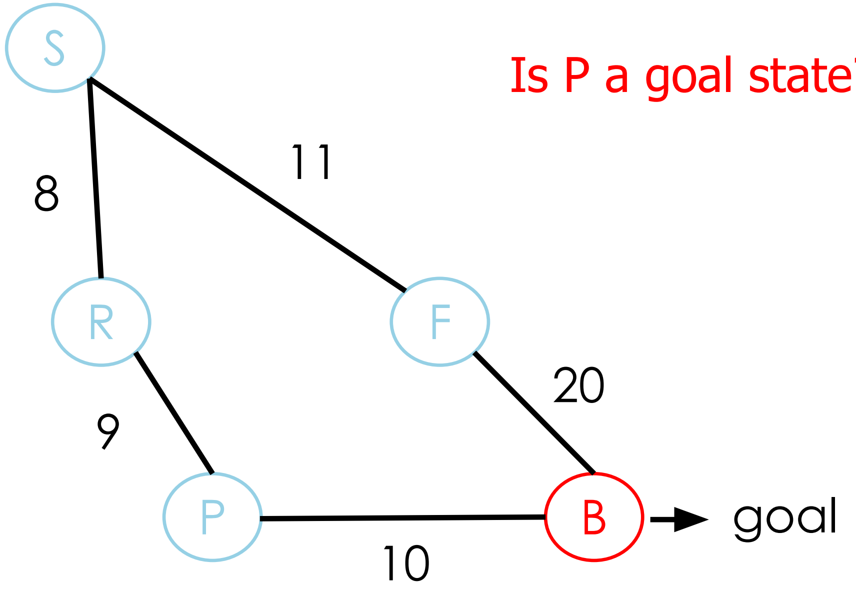
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
  opened ← a priority queue ordered by PATH-COST, with *node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*opened*) **then return** failure
    node ← POP(*opened*) /* chooses the lowest-cost node in *opened* */
    if *node* == *goal* **then return** SOLUTION
    add *node* to *explored*
    for each *adjacent node (n)* of *node* **do**
      if *n* is not in *explored* or *opened* **then**
        opened ← INSERT(*n*)
      else if *n* is in *opened* with higher PATH-COST **then**
        replace that *opened* node with *n*



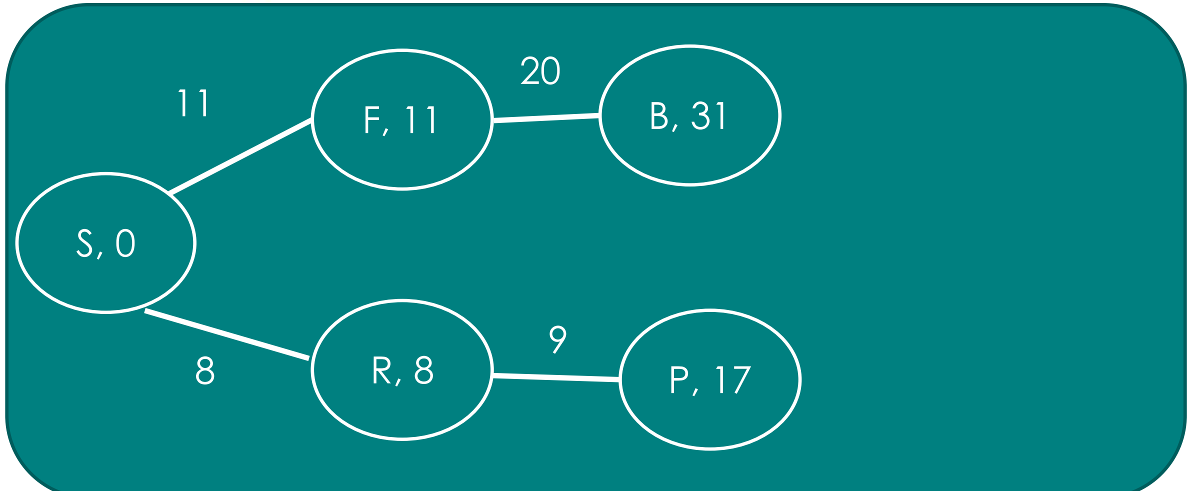


Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}
{(F, 11)}	{(S,0), (R,8)}
{(F, 11), (P, 17)}	{(S,0), (R,8)}
{(P, 17)}	{(S,0), (R,8), (F, 11)}
{(P, 17), (B, 31)}	{(S,0), (R,8), (F, 11)}
{(B, 31)}	{(S,0), (R,8), (F, 11),(P, 17)}

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
opened ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
if EMPTY?(*opened*) **then return** failure
node ← POP(*opened*) /* chooses the lowest-cost node in *opened* */
if *node* == *goal* **then return** SOLUTION
add *node* to *explored*
for each *adjacent node (n)* of *node* **do**
if *n* is not in *explored* or *opened* **then**
opened ← INSERT(*n*)
else if *n* is in *opened* with higher PATH-COST **then**
replace that *opened* node with *n*

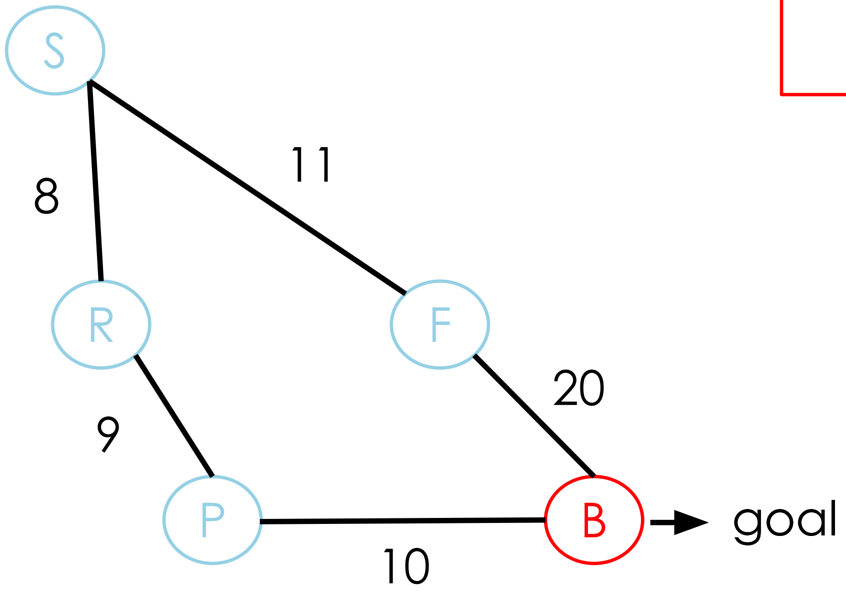


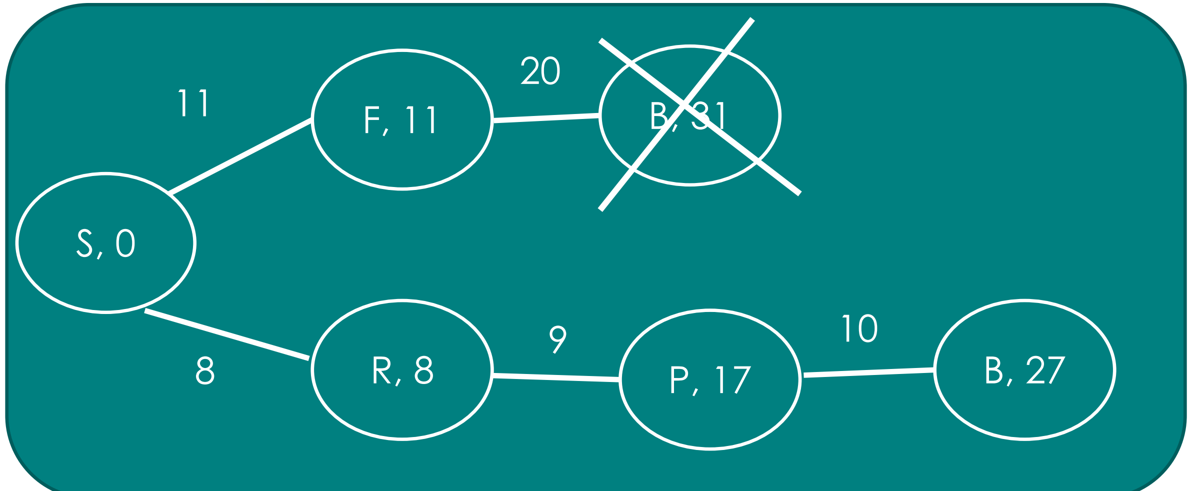
Is P a goal state?



Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}
{(F, 11)}	{(S,0), (R,8)}
{(F, 11), (P, 17)}	{(S,0), (R,8)}
{(P, 17)}	{(S,0), (R,8), (F, 11)}
{(P, 17), (B, 31)}	{(S,0), (R,8), (F, 11)}
{(B, 31)}	{(S,0), (R,8), (F, 11),(P, 17)}
{(B, 27)}	

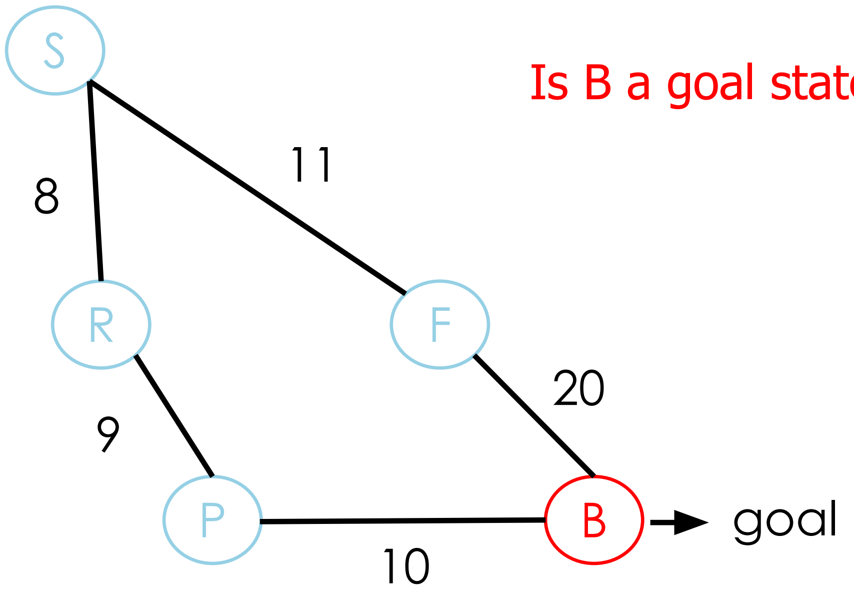
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    opened ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(opened) then return failure
        node ← POP(opened)   /* chooses the lowest-cost node in opened */
        if node == goal then return SOLUTION
        add node to explored
        for each adjacent node (n) of node do
            if n is not in explored or opened then
                opened ← INSERT(n)
            else if n is in opened with higher PATH-COST then
                replace that opened node with n
```



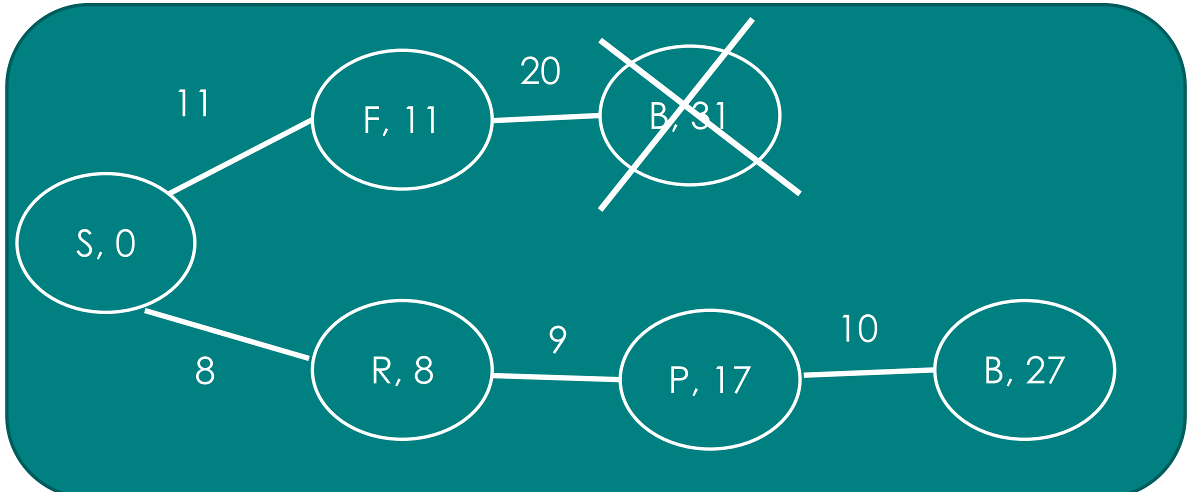


Opened	Explored
{(S,0)}	{-}
{(R, 8), (F, 11)}	{(S,0)}
{(F, 11)}	{(S,0), (R,8)}
{(F, 11), (P, 17)}	{(S,0), (R,8)}
{(P, 17)}	{(S,0), (R,8), (F, 11)}
{(P, 17), (B, 31)}	{(S,0), (R,8), (F, 11)}
{(B, 31)}	{(S,0), (R,8), (F, 11),(P, 17)}
{(B, 27)}	

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
  opened ← a priority queue ordered by PATH-COST, with *node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*opened*) **then return** failure
    node ← POP(*opened*) /* chooses the lowest-cost node in *opened* */
    if *node* == *goal* **then return** SOLUTION
    add *node* to *explored*
    for each *adjacent node (n)* of *node* **do**
      if *n* is not in *explored* or *opened* **then**
        opened ← INSERT(*n*)
    else if *n* is in *opened* with higher PATH-COST **then**
      replace that *opened* node with *n*

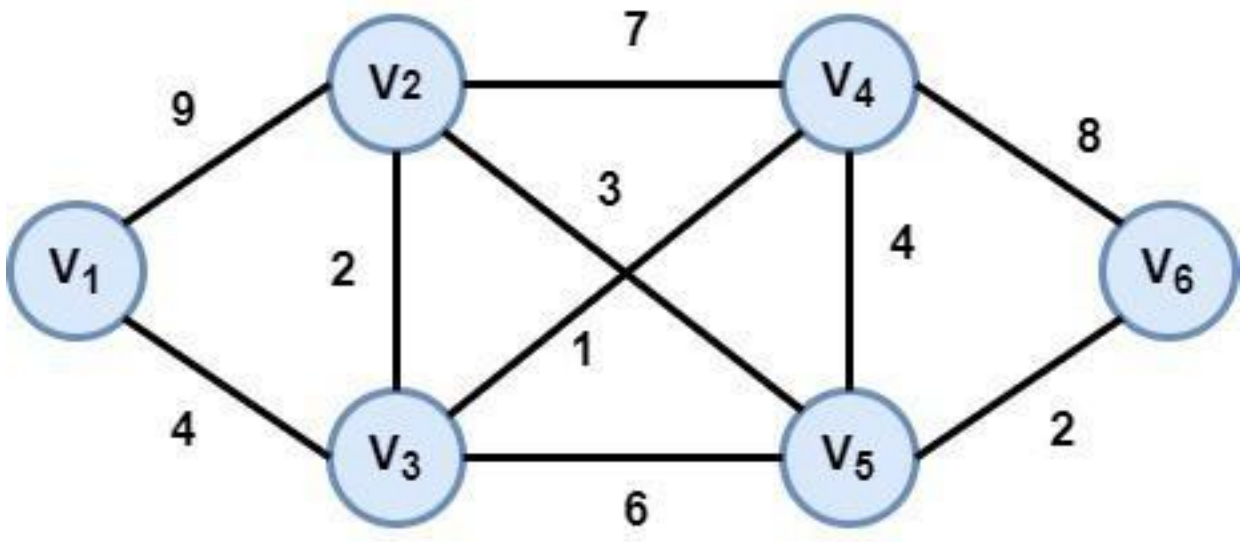


Is B a goal state?



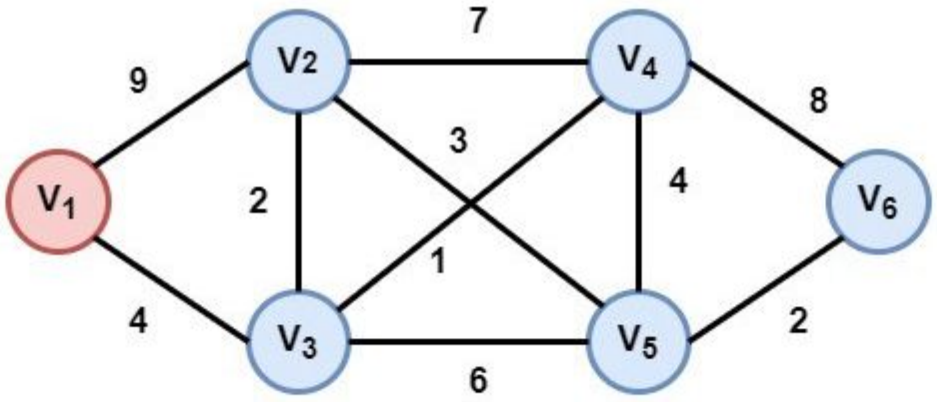
Example

Our target is to go from the city (node) V1 to V6 following the path with the smallest cost (shortest path). Let's execute the UCS algorithm:



Step 1: Initialization

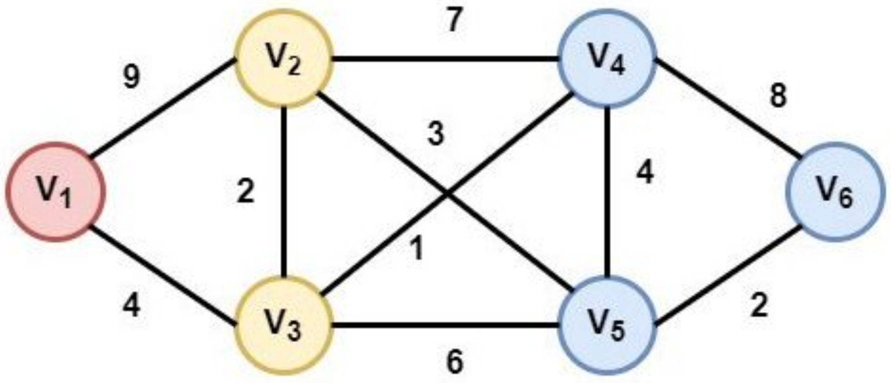
The first node V1 (initial state) of the graph is appended to the opened list. The distance of this node from itself is zero.

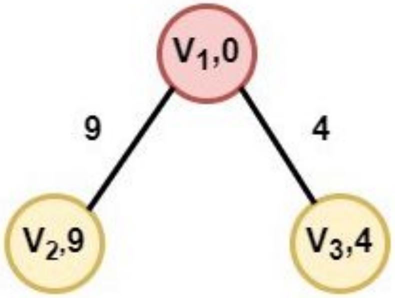


Step	Opened	Explored
0	$\{(V_1,0)\}$	{-}

Step 2: Node V1 is selected

The V1 is selected as it is the only node in the opened list. Its children V2 and V3 are appended in the opened list after the distance calculation from node V1.

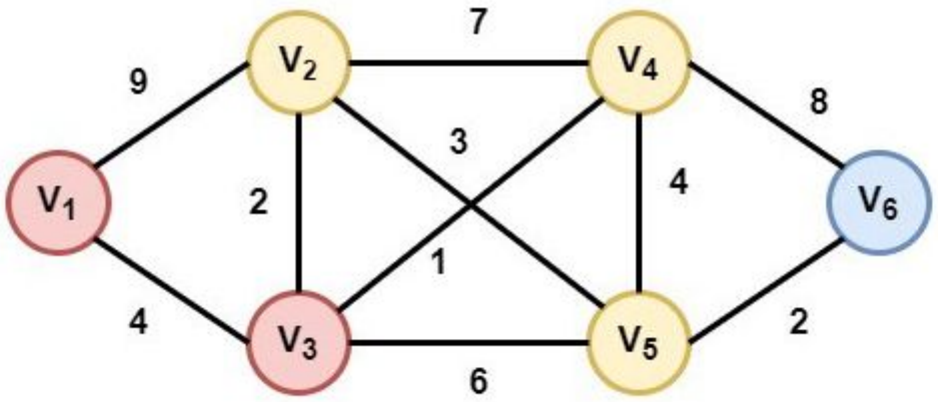


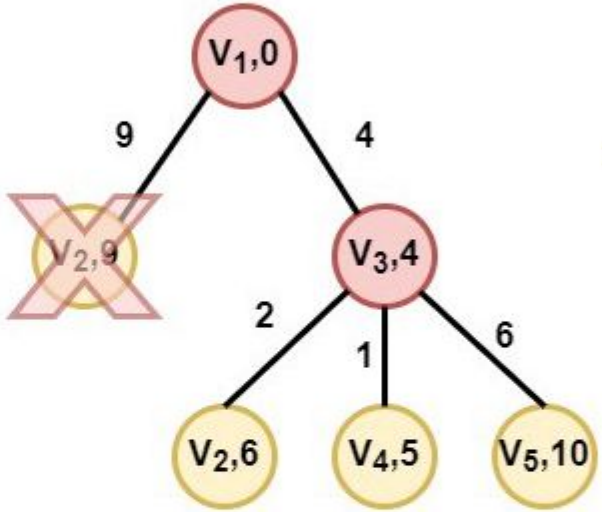


Step	Opened	Explored
0	$\{(V_1,0)\}$	{-}
1	$\{(V_2,9),(V_3,4)\}$	$\{(V_1,0)\}$

Step 3: Node V3 is selected

Node V3 is selected as it has the smallest distance value. As we can see, extending the node V3 we find the node V2 with a smaller distance value. So we replace node (V2,9) with the new node (V2,6).

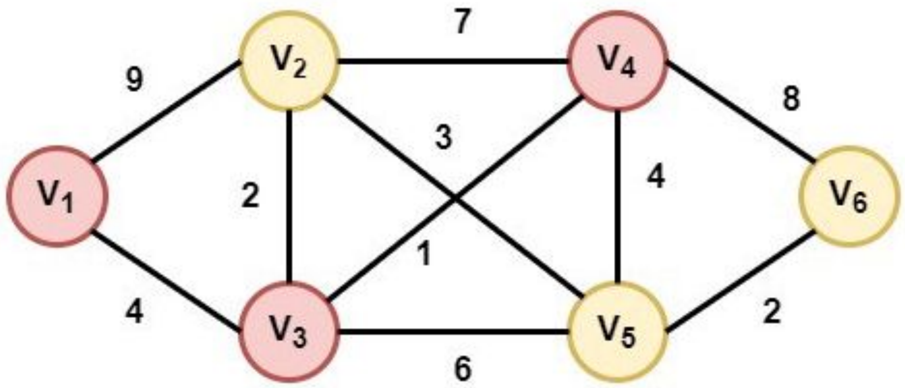




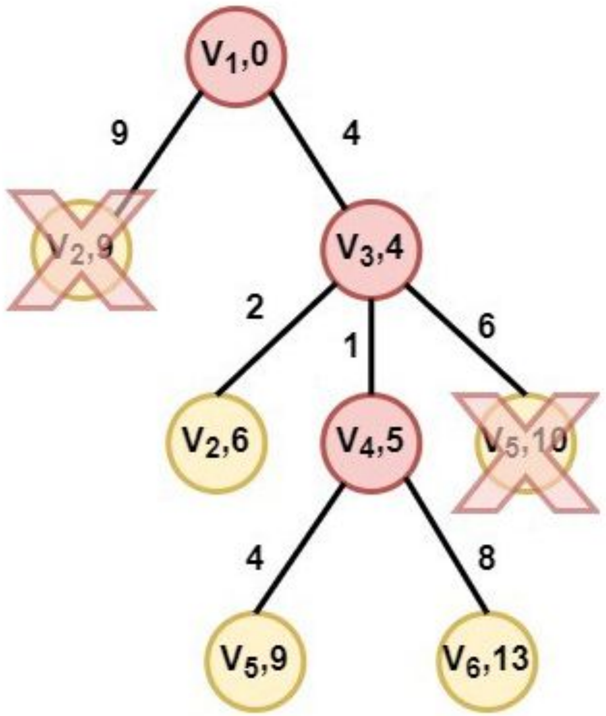
Step	Opened	Explored
0	$\{(V_1,0)\}$	{-}
1	$\{(V_2,9),(V_3,4)\}$	$\{(V_1,0)\}$
2	$\{(V_2,6),(V_4,5),(V_5,10)\}$	$\{(V_1,0),(V_3,4)\}$

Step 4: Node V4 is selected

Node V4 is selected as it has the smallest distance value. In this step, we find a better distance value for node V5, so we replace the node (V5, 10) with node (V5, 9).

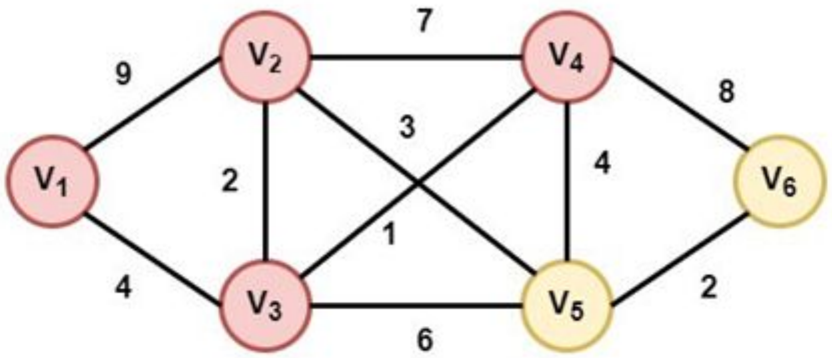


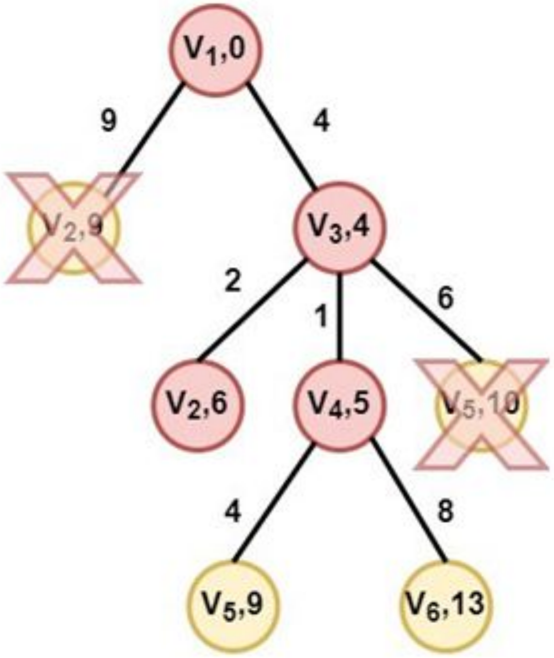
Step	Opened	Explored
0	$\{(V_1,0)\}$	{-}
1	$\{(V_2,9),(V_3,4)\}$	$\{(V_1,0)\}$
2	$\{(V_2,6),(V_4,5),(V_5,10)\}$	$\{(V_1,0),(V_3,4)\}$
3	$\{(V_2,6),(V_6,13),(V_5,9)\}$	$\{(V_1,0),(V_3,4),(V_4,5)\}$



Step 5: Node V2 is selected

Node V2 is selected as it has the smallest distance value. However, none of its children is appended in the opened list, as nodes V3 and V4 are already inserted in the explored list and the algorithm doesn't find a better distance value for node V5.

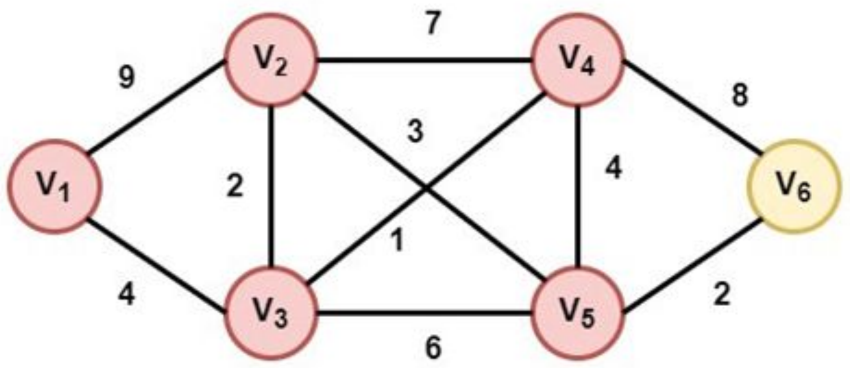




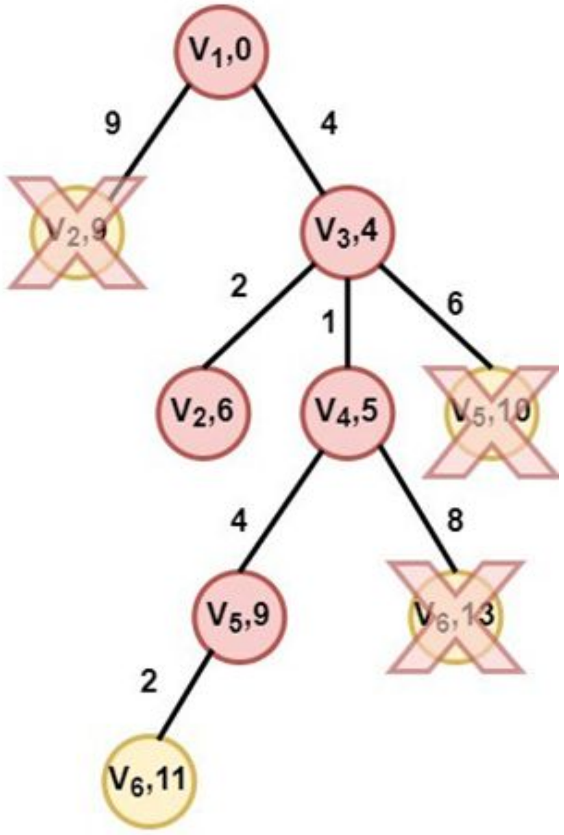
Step	Opened	Explored
0	$\{(V_1,0)\}$	{-}
1	$\{(V_2,9),(V_3,4)\}$	$\{(V_1,0)\}$
2	$\{(V_2,6),(V_4,5),(V_5,10)\}$	$\{(V_1,0),(V_3,4)\}$
3	$\{(V_2,6),(V_6,13),(V_5,9)\}$	$\{(V_1,0),(V_3,4),(V_4,5)\}$
4	$\{(V_6,13),(V_5,9)\}$	$\{(V_1,0),(V_3,4),(V_4,5),(V_2,6)\}$

Step 6: Node V5 is selected

Node V5 is selected as it has the smallest distance value. A better path to node V6 is found in this step. So, we replace the old node (V6, 13) with node (V6, 11)

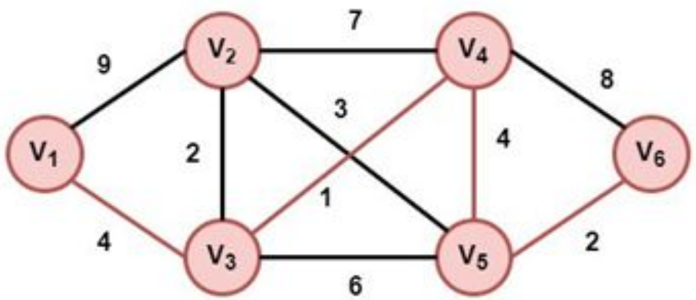


Step	Opened	Explored
0	$\{(V_1,0)\}$	{-}
1	$\{(V_2,9),(V_3,4)\}$	$\{(V_1,0)\}$
2	$\{(V_2,6),(V_4,5),(V_5,10)\}$	$\{(V_1,0),(V_3,4)\}$
3	$\{(V_2,6),(V_6,13),(V_5,9)\}$	$\{(V_1,0),(V_3,4),(V_4,5)\}$
4	$\{(V_6,13),(V_5,9)\}$	$\{(V_1,0),(V_3,4),(V_4,5),(V_2,6)\}$
5	$\{(V_6,11)\}$	$\{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)\}$



- **Step 7: Node V6 is selected**

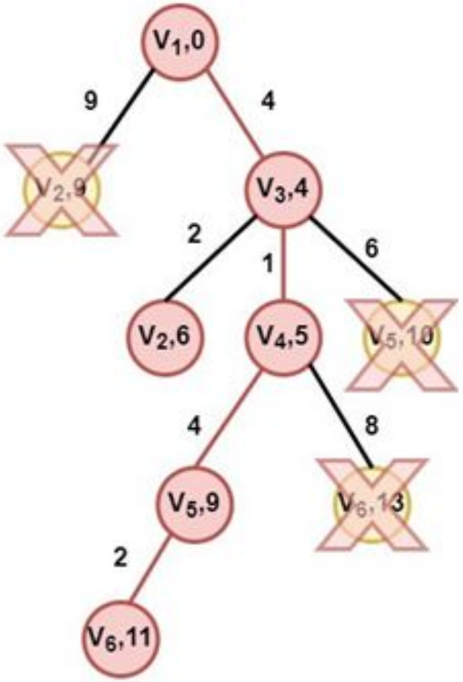
Node V6 (target node) is selected. So the algorithm returns the path from node V1 to node V6 with cost 11, which constitutes the best solution.



Step	Opened	Explored
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}
2	{(V_2,6),(V_4,5),(V_5,10)}	{(V_1,0),(V_3,4)}
3	{(V_2,6),(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5)}
4	{(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6)}
5	{(V_6,11)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)}
6	{-}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)}

Path: V_1 - V_3 - V_4 - V_5 - V_6

Total Cost: 11



When to use BFS? Real-Life Applications

- **Crawlers in Search Engines:**

BFS is used for indexing web pages. The algorithm starts traversing from the **source page and follows all the links associated with the page**. Here each web page will be considered as a node in a graph.

- **GPS Navigation systems:**

BFS is used to find **neighboring locations** by using the GPS system.

- **Broadcasting:**

Networking used to communicate **broadcasted packets** across all the nodes in a network. These packets follow a traversal method of BFS to reach various networking nodes.

- **Peer to Peer Networking:**

BFS can be used as a traversal method to find all the **neighboring nodes** in a Peer to Peer Network. For example, **BitTorrent uses Breadth-First Search for peer to peer communication**.

When to use DFS? Real-Life Applications

- **Cycle detection:**

DFS can be used to detect cycles in a graph. If a node is visited again during a DFS traversal, it indicates **that there is a cycle in the graph**.

- **Pathfinding:**

DFS can be used to find a **path between two nodes in a graph.**

- **Solving puzzles:**

DFS can be used to **solve puzzles such as mazes**, where the goal is to find a path from the start to the end.

- **Backtracking:**

DFS can be used for **backtracking in algorithms like Sudoku**.


Questions
Answers