G51FAI
Fundamentals of AI

Instructor: Siang Yew Chong

Blind Searches


Outlines

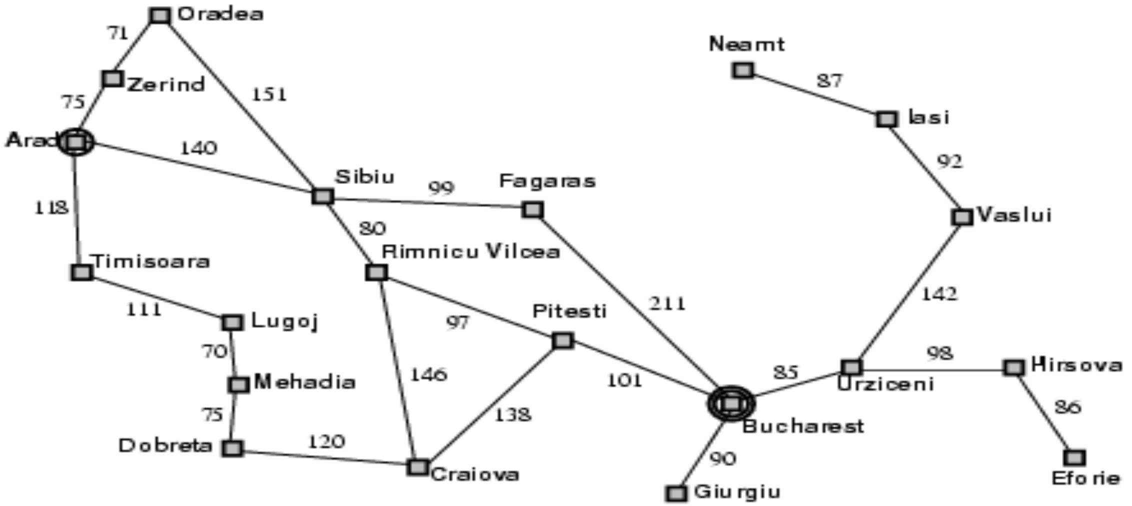
- State space vs. search tree
- Avoiding repeated states
  - Tree search
  - Graph search
- Performance evaluation of blind search strategies
  - Breath first
  - Depth first
  - Depth limited
  - Iterative deepening
  - Uniform cost

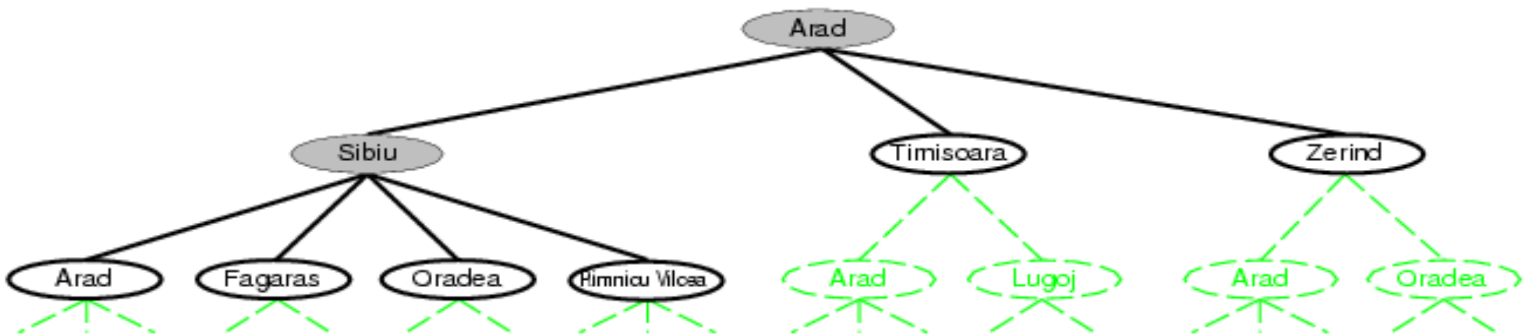
Searching in a State Space

- **Obtaining** the whole ***large state space*** is **impractical**. Instead, ***states*** are **generated** (during search)
- The ***search space*** is the **implicit tree defined** by the **initial state** and the **operators**
- A ***solution*** is a **sequence of actions** associated with a **path** in a **state space** from a **start to** a **goal state**
- Search works by considering various possible action sequences
- The ***search tree*** is the **explicit tree generated** by the **search strategy** (that defines the order of state expansion)
- The ***cost*** of a **solution** is the **sum** of the **arc costs** on the **solution path**
  - if all arcs have the same (unit) cost, then the solution cost is just the length of the solution (number of steps / state transitions)
- Search tree may be *infinite* because of *loopy* or *redundant* paths even if state space is small

Tree-Search

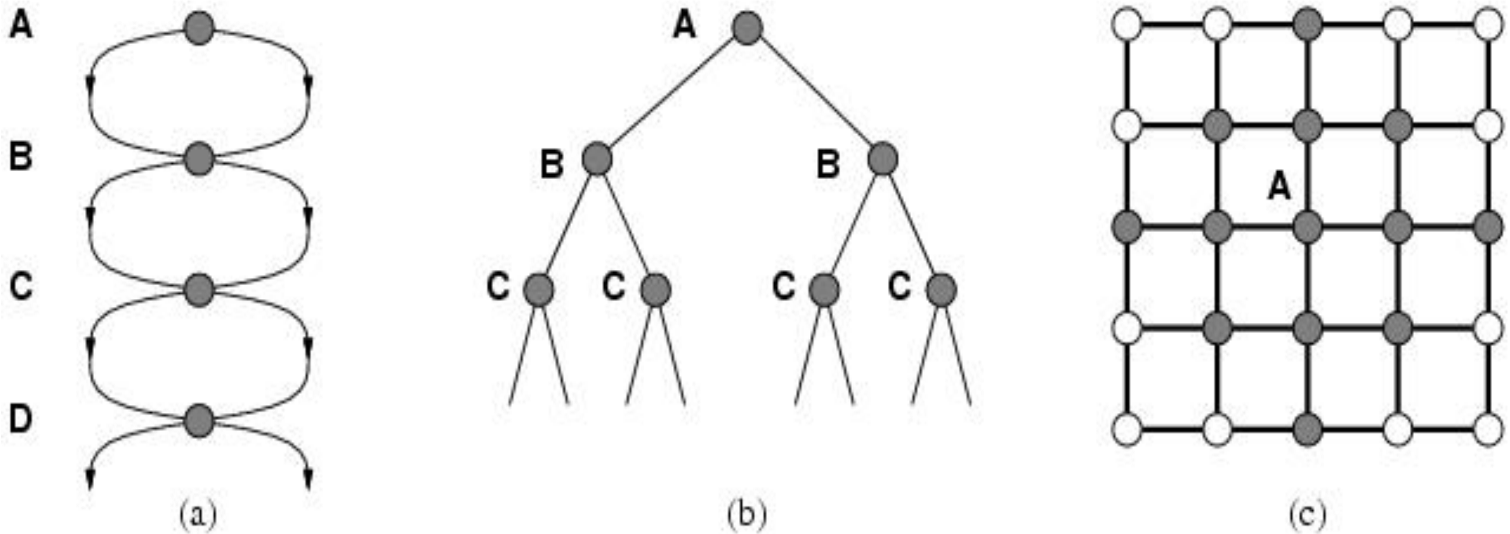
function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

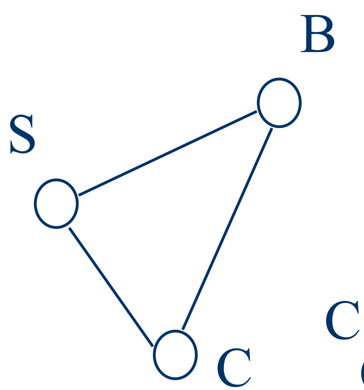




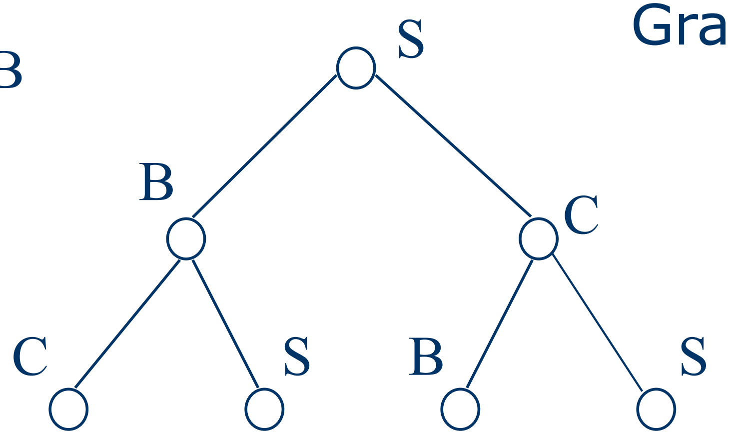
Avoiding Repeated States

- Failure to detect repeated states can turn a solvable problems into unsolvable ones.





partial state space



search subtree

Graph-search
- never generate a state generated before
- check for cycles
- must keep track of all possible states/**memory** (space complexity of $O(b^d)$)

Search Strategies

Two Categories of Search

- ❑ Uninformed/Blind

  No additional information about states beyond that provided in the problem definition

- ❑ Informed/Heuristic

  Uses strategies that know whether one state is more promising than the others in reaching the goal

Characteristics of Blind Searches

- Can only generate successors and distinguish between a goal state from a non-goal state
- **No preference** as to which **state** (node) will be **more promising**, **expansion** done **systematically** according to a specific order
- Search process constructs a search tree, where
  - ***root*** is the ***initial state***; and
  - ***leaf nodes (fringe)*** are nodes discovered but not yet expanded OR without successors
- The order of ***fringe*** **processing characterises** the **different categories** of search
  - this can have a dramatic effect on how well the search performs when measured against the four criteria defined earlier

Search Implementation

Two types of data structure are needed:

- **Fringe** are set of **nodes** that
  - have been discovered
  - but not “**processed**” (**tested** for **goal state** and **discover their children**)
  - also known as *open nodes*, *frontier*, *agenda*
- **Explored nodes** are set of nodes that
  - have been discovered
  - have been “processed”
  - Also known as *close nodes*

- Processed implies the completion of the following:
  - ✓ *tested whether they are a goal*
  - ✓ *all children have been discovered*

Search Implementation

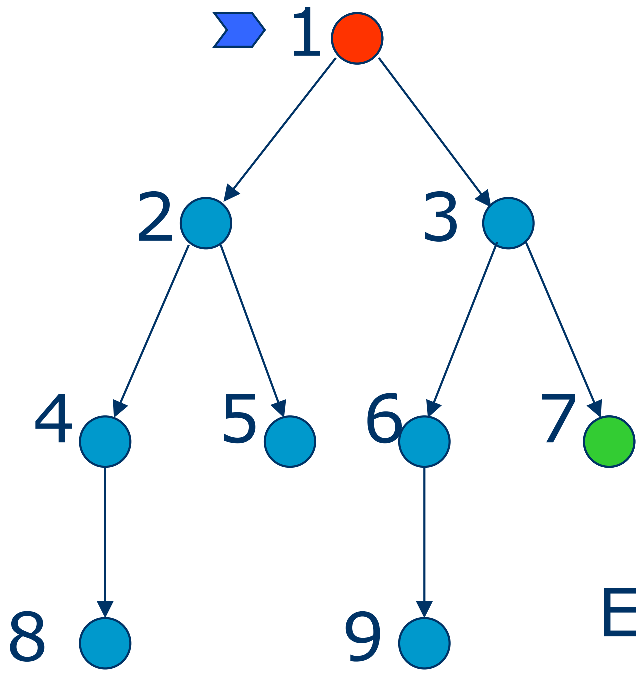
- Fundamental algorithm:
  - move nodes
    - ✓ into the “fringe” when they are discovered
    - ✓ pick a node from the fringe to be processed in a predetermined order
    - ✓ into the “explored” after they have been processed
  - processing method to compare current node with goal state and “expand a node” to discover its children when goal state not reached, as dictated by the goal state and operators of a problem

Blind Search

- Breadth-first Search
- Depth-first Search
- Depth-limited Search
- Iterative Deepening Search
- Uniform-cost Search

Breadth-First Strategy

- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue

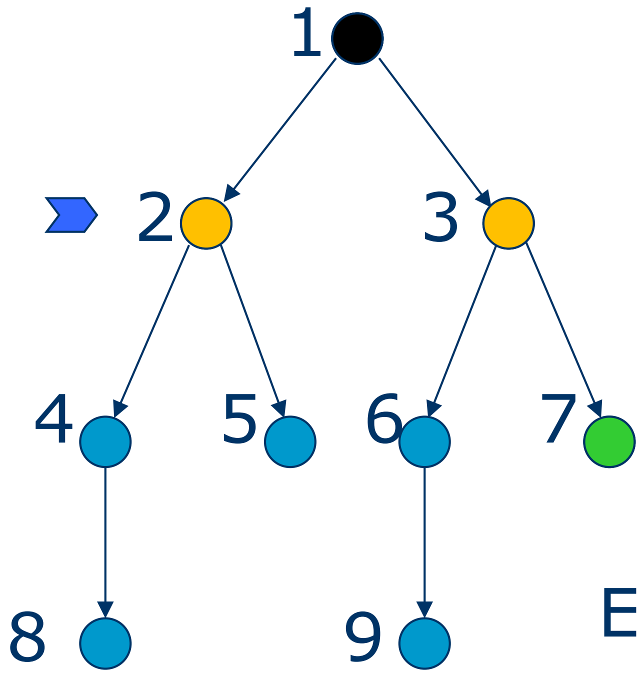


FRINGE = (1)

EXPLORED =

Breadth-First Strategy

- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue

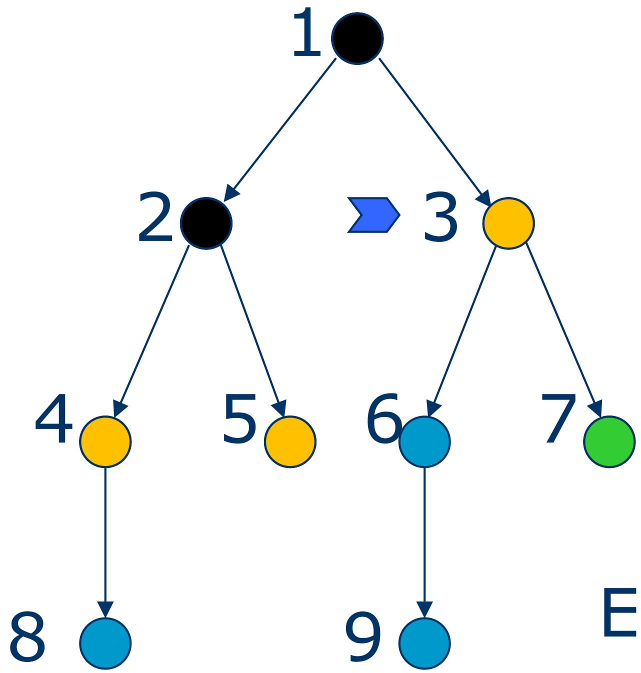


FRINGE = (2, 3)

EXPLORED = (1)

Breadth-First Strategy

- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue

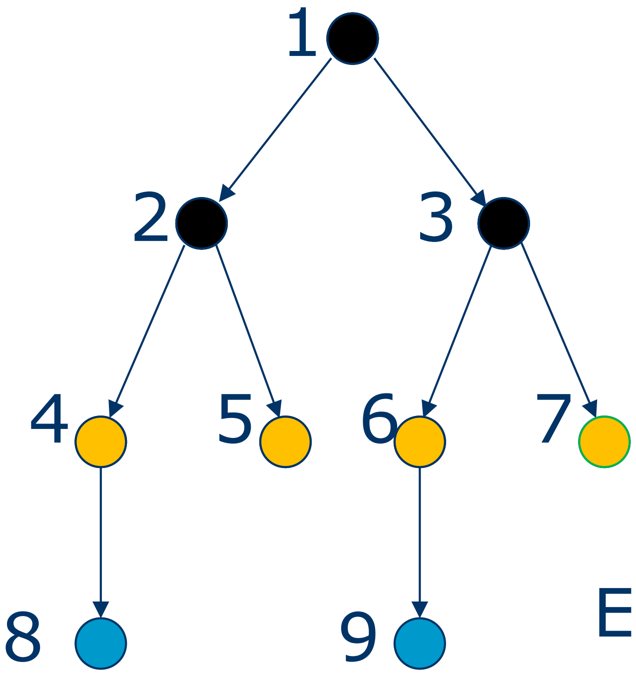


FRINGE = (3, 4, 5)

EXPLORED = (1, 2)

Breadth-First Strategy

- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue

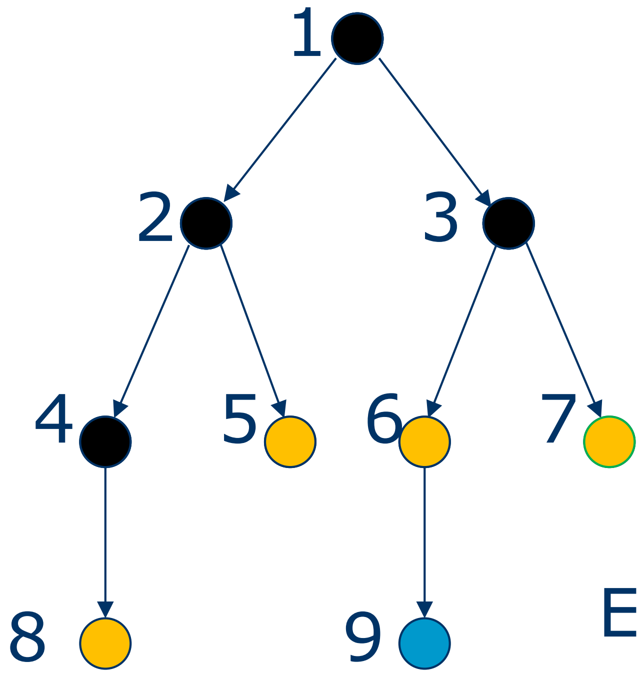


FRINGE = (4, 5, 6, 7)

EXPLORED = (1, 2, 3)

Breadth-First Strategy

- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue

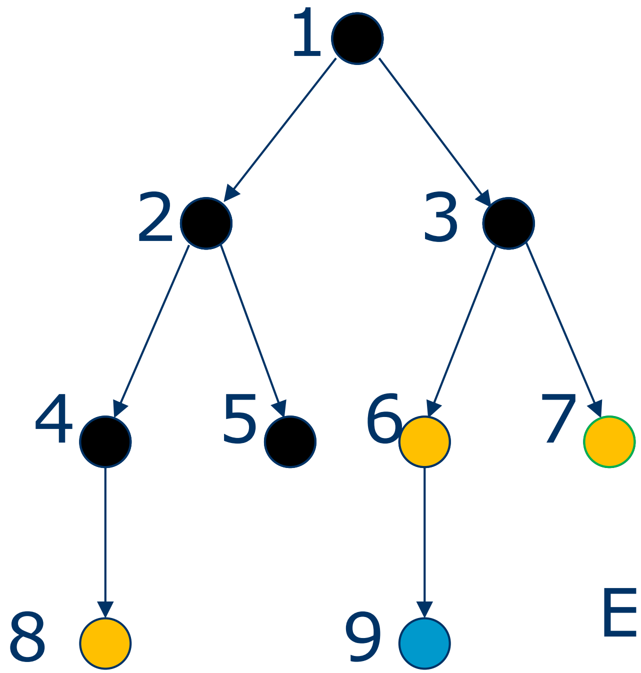


FRINGE = (5, 6, 7, 8)

EXPLORED = (1, 2, 3, 4)

Breadth-First Strategy

- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue

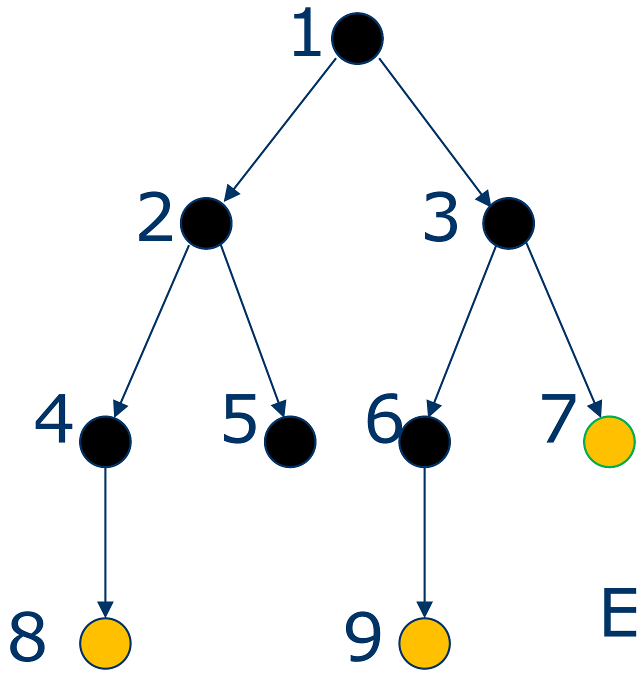


FRINGE = (6, 7, 8)

EXPLORED = (1, 2, 3, 4, 5)

Breadth-First Strategy

- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue

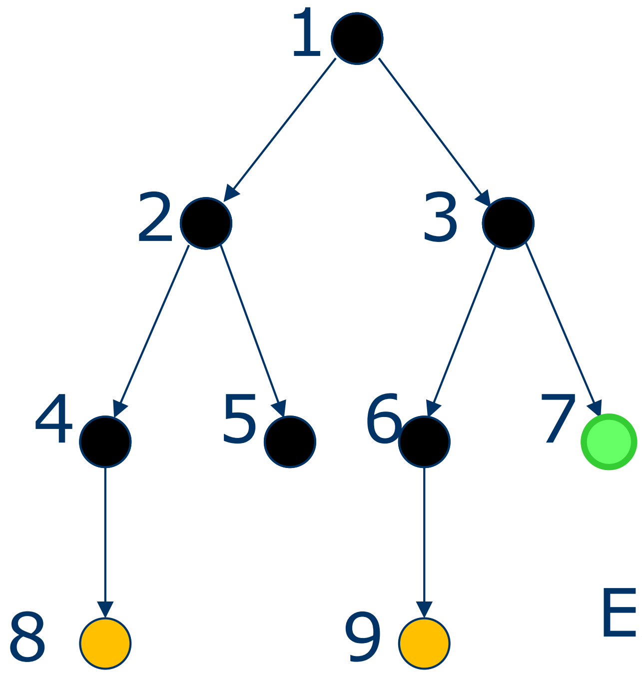


FRINGE = (7, 8, 9)

EXPLORED = (1, 2, 3, 4, 5, 6)

Breadth-First Strategy

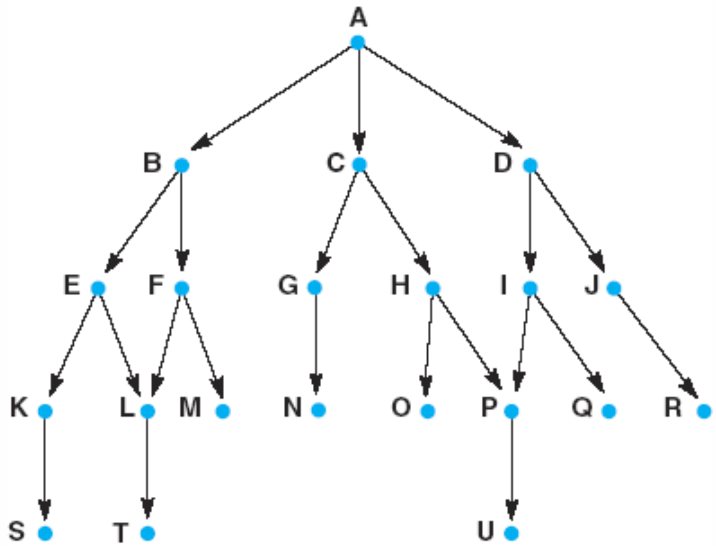
- explores nodes nearest the root before exploring nodes further away
- implementation: *fringe* is a FIFO queue
- new nodes are inserted at the end of the queue



FRINGE = (8, 9)

EXPLORED = (1, 2, 3, 4, 5, 6, 7)

Another BFS Example (Exercise)



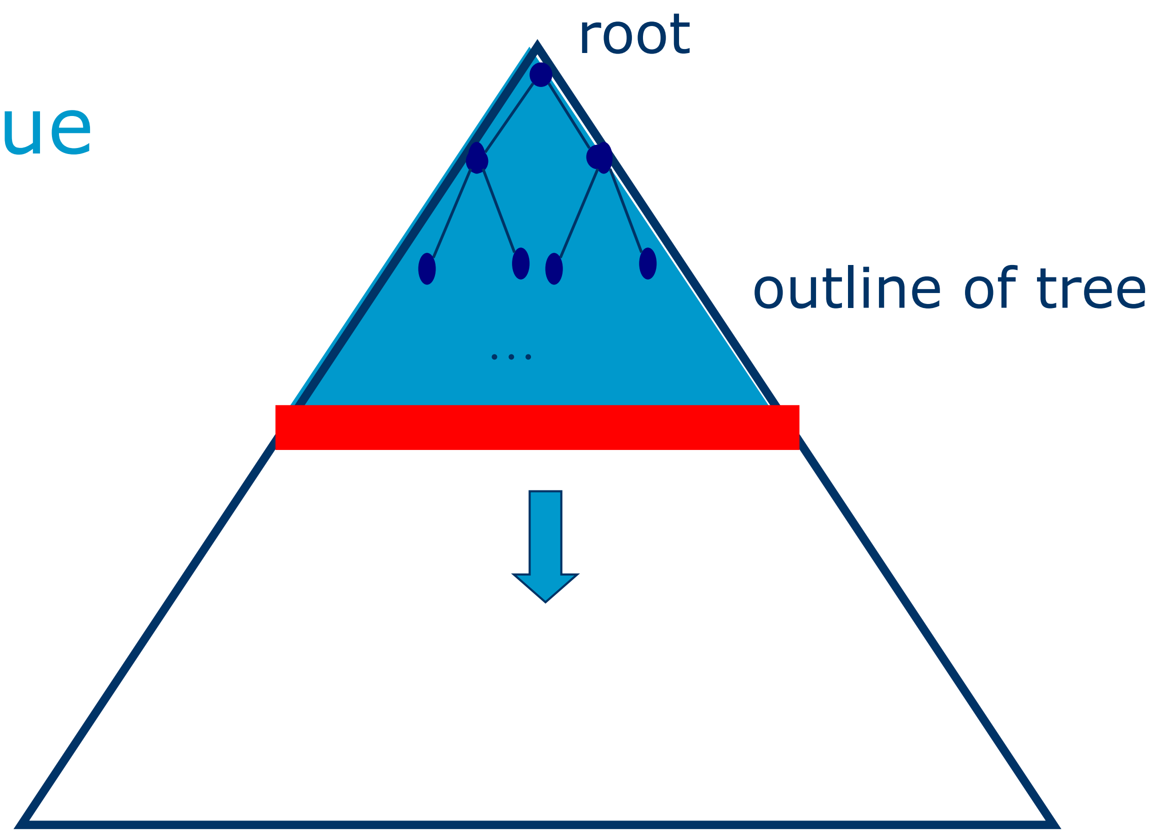
1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [C,D,E,F]; closed = [B,A]**
4. **open = [D,E,F,G,H]; closed = [C,B,A]**
5. **open = [E,F,G,H,I,J]; closed = [D,C,B,A]**
6. **open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]**
7. **open = [G,H,I,J,K,L,M]** (as L is already on open); **closed = [F,E,D,C,B,A]**
8. **open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]**
9. and so on until either U is found or **open** = [ ]

Breadth First Search Observation

- If there is a **solution** breadth first search is **guaranteed** to find it
- **If** there are **several solutions** then breadth first search will **always find** the **shallowest goal state** first and if the cost of a solution is a non-decreasing function of the depth then it will always find the cheapest solution
- Just before starting to explore level *n*, the queue holds *all* the nodes at level *n-1*
- When this method succeeds, it doesn’t give the path to the goal

Breadth-First Search

- Search Pattern: spread before dive
- Fringe in red
- Explored in blue
- Size of fringe
  - $O(b^d)$
  - exponential



Evaluating Breadth First Search

- Evaluating against four criteria
  - Complete? : Yes
  - Optimal? : Yes
  - Time Complexity:
    - assume branching factor is b, so root has *b* successors, each node at the next level has again b successors (total b^2), … and solution is at depth *d*
    - worst case; expand all but the last node at depth *d*
    - total number of nodes generated is:

      $1 + b + b^2 + \ldots + b^d + b(b^d-1) = O(b^{d+1})$
  - Space Complexity: $O(b^{d+1})$
  - Note : The space/time complexity could be less as the solution could be found anywhere on the d^{th} level.

Evaluating Breadth First Search

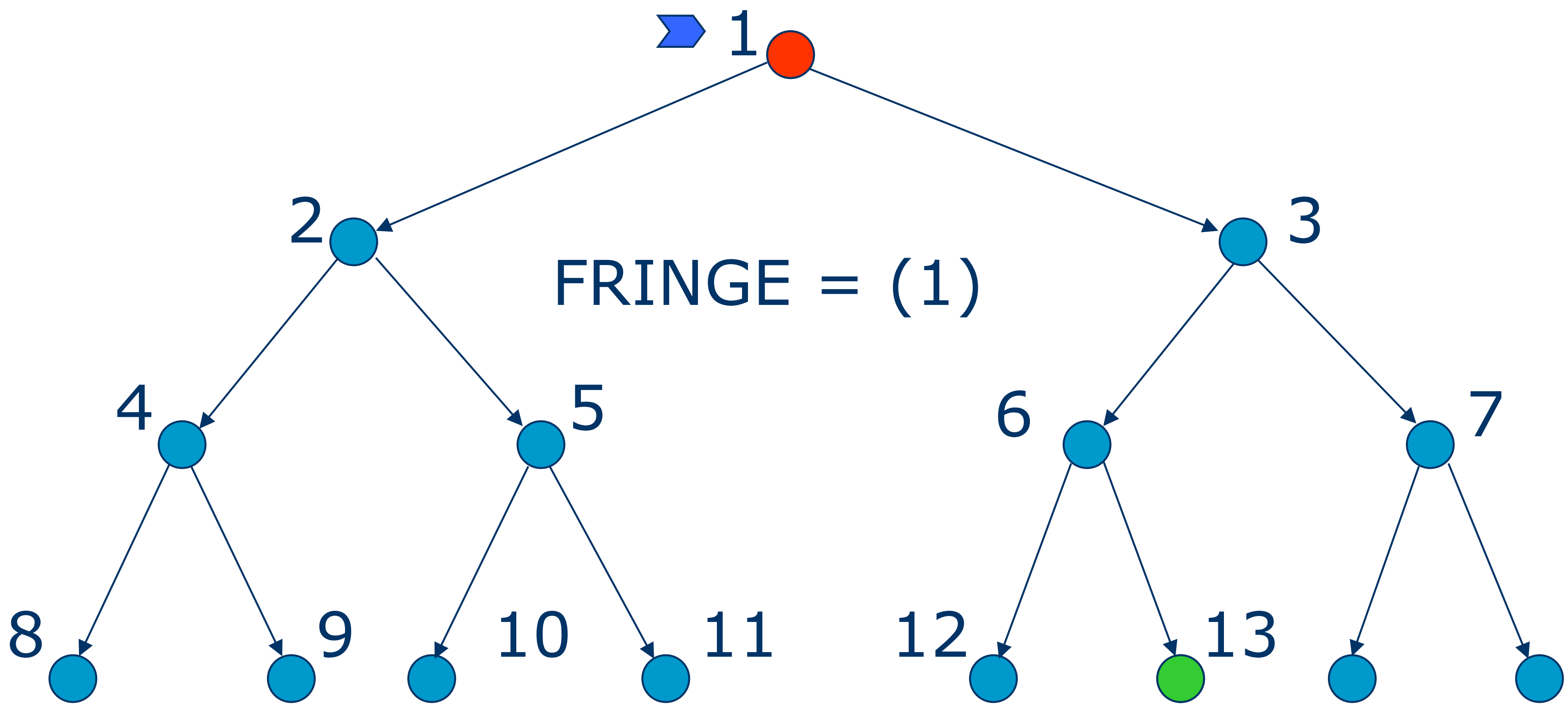
- Notes:
  - memory requirements are a bigger problem than execution time
  - exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

Assumptions: b = 10; 10,000 nodes/sec; 1000 bytes/node

Depth-First Strategy

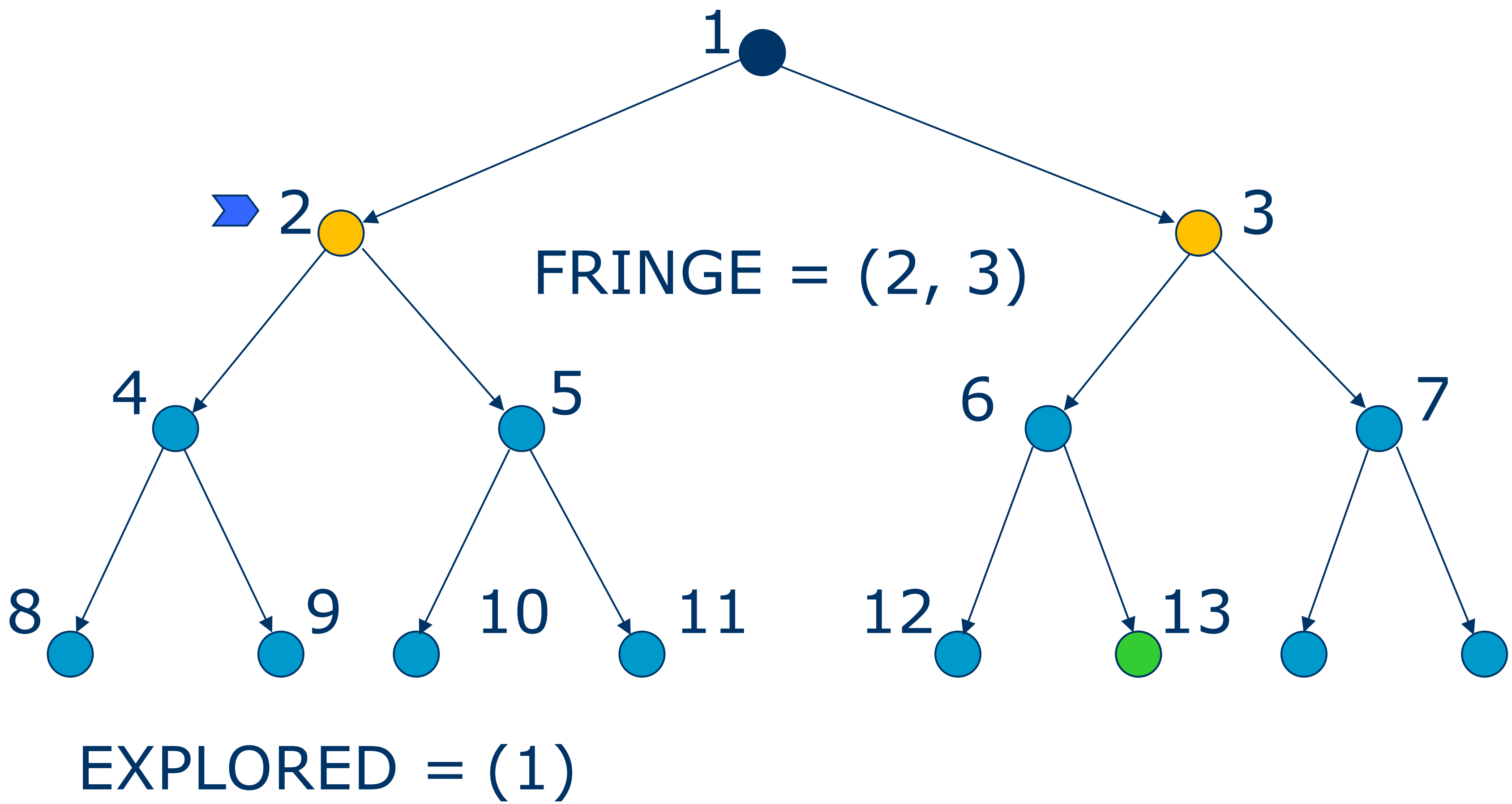
- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)
- new nodes are inserted at the front of the queue



EXPLORED =

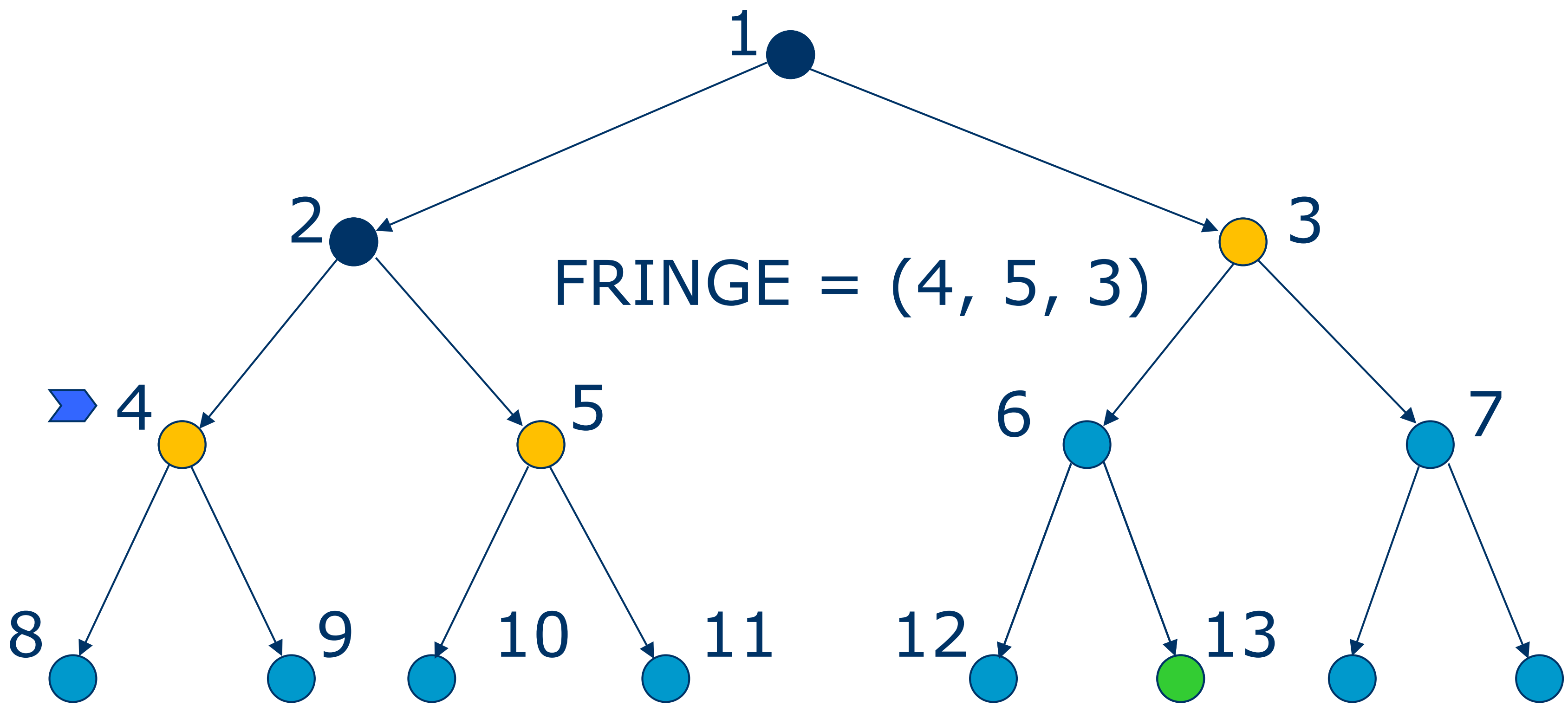
Depth-First Strategy

- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)
- new nodes are inserted at the front of the queue



Depth-First Strategy

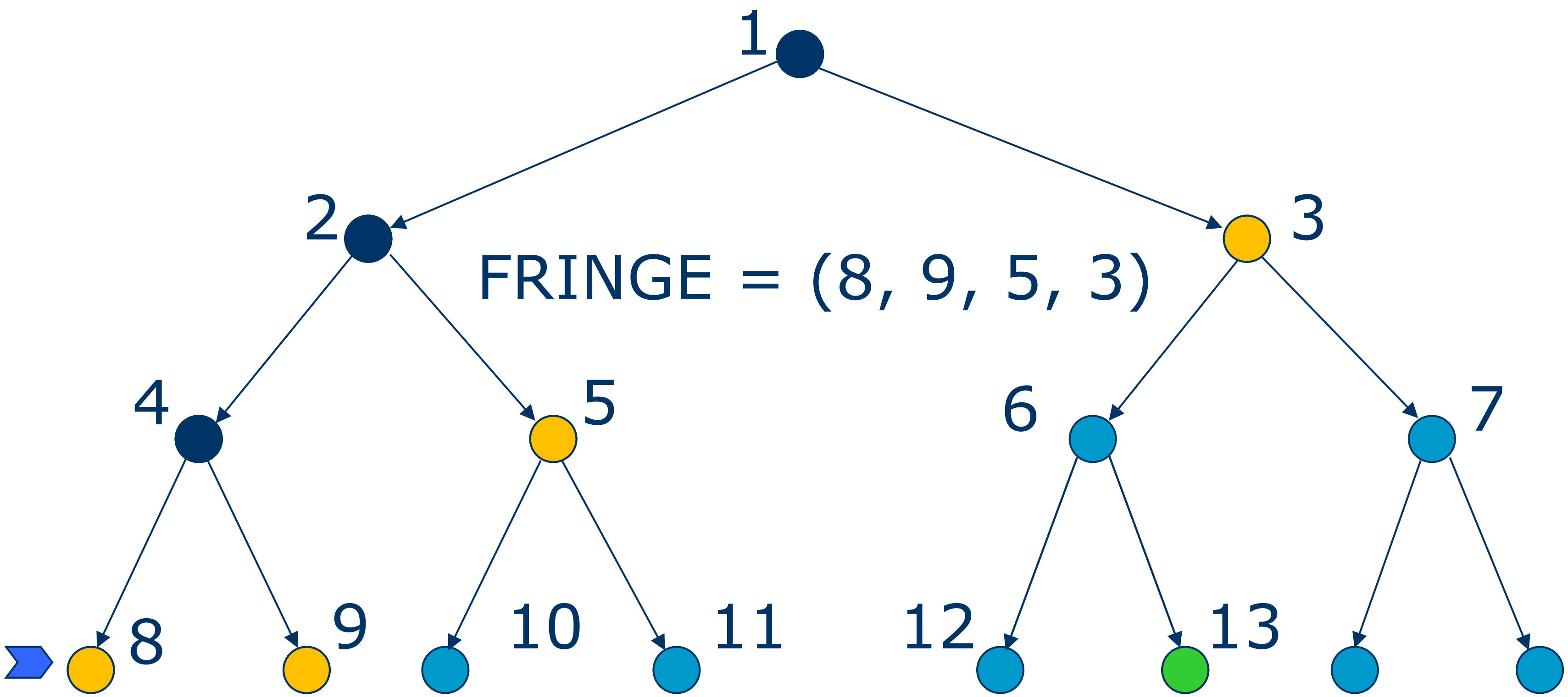
- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



EXPLORED = (1, 2)

Depth-First Strategy

- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)

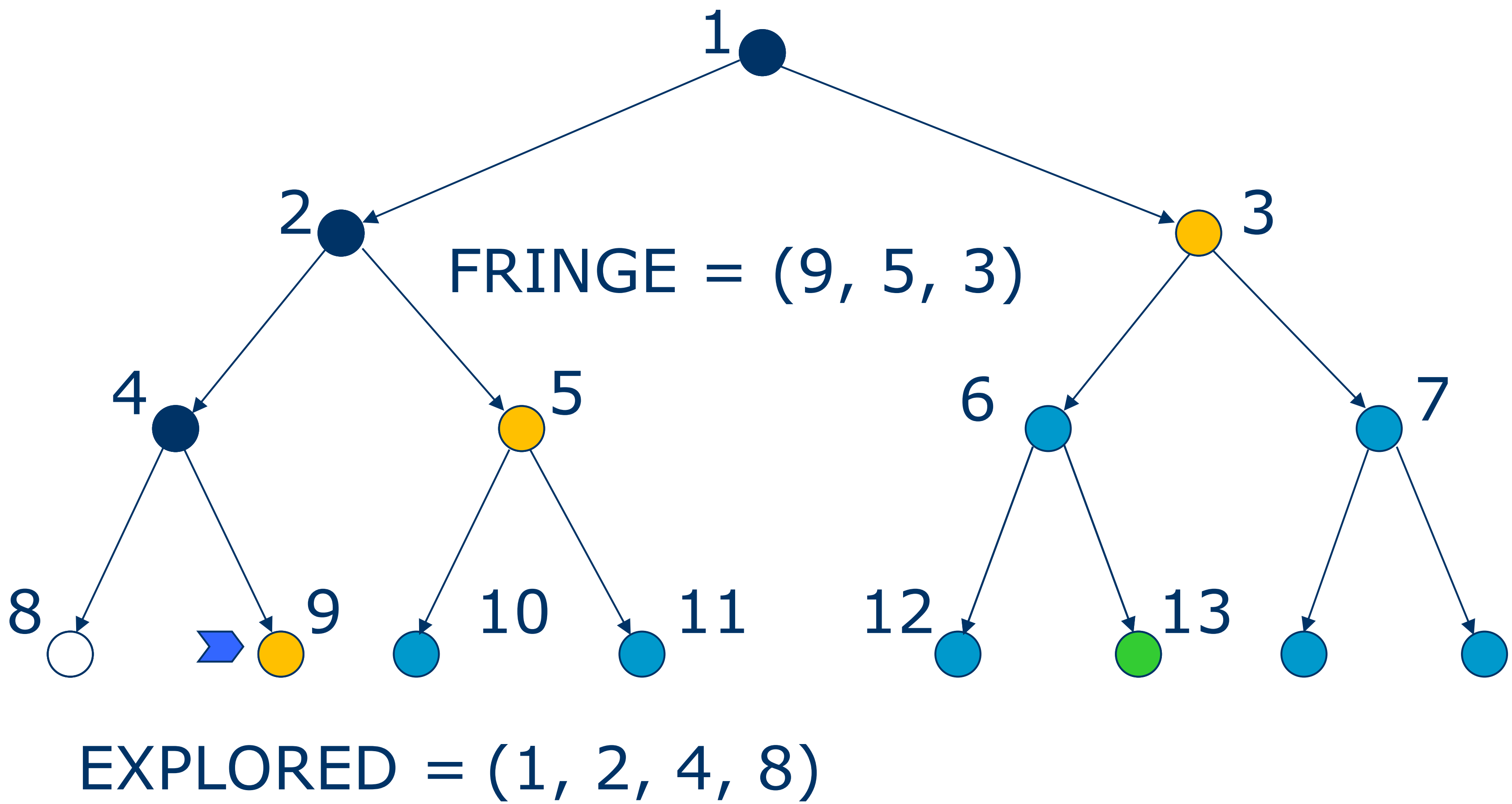


FRINGE = (8, 9, 5, 3)

EXPLORED = (1, 2, 4)

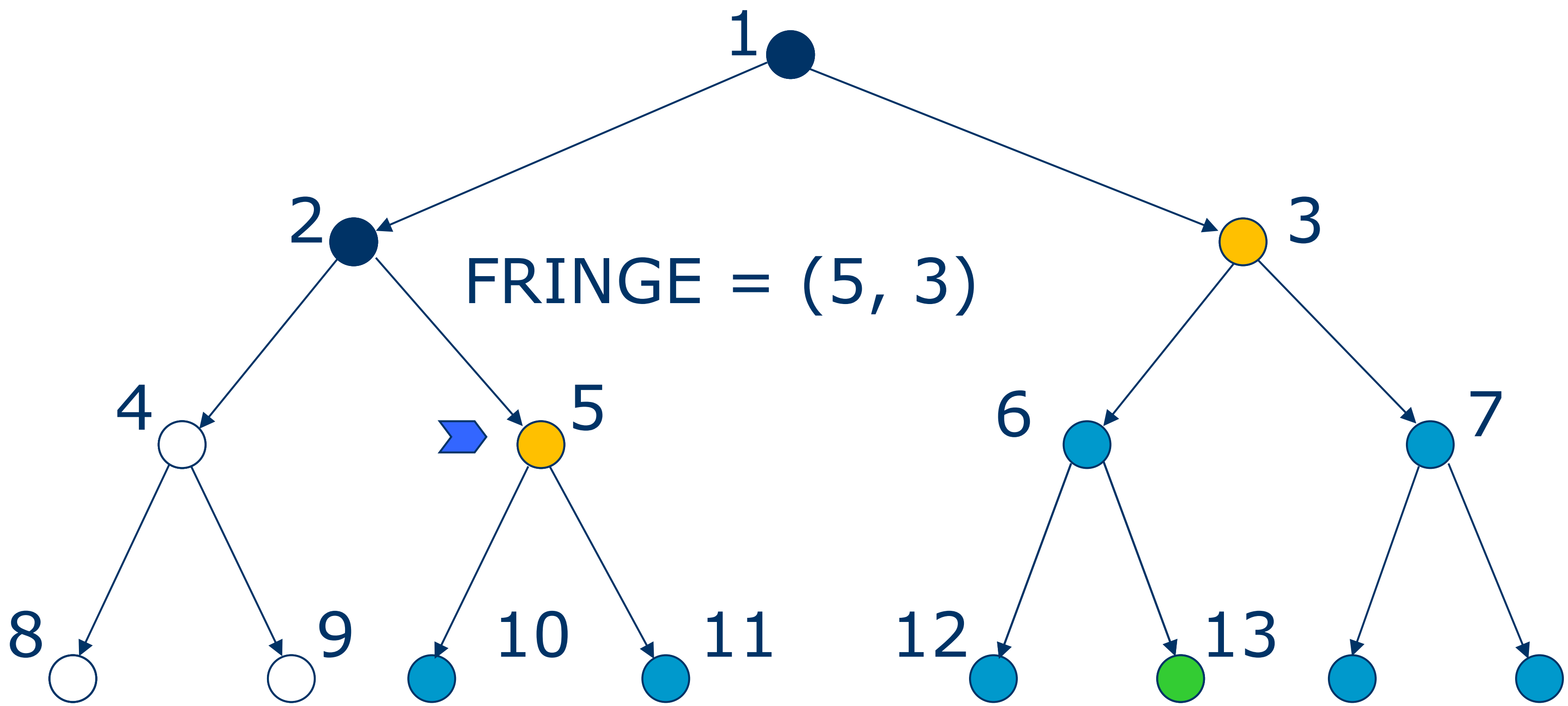
Depth-First Strategy

- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



Depth-First Strategy

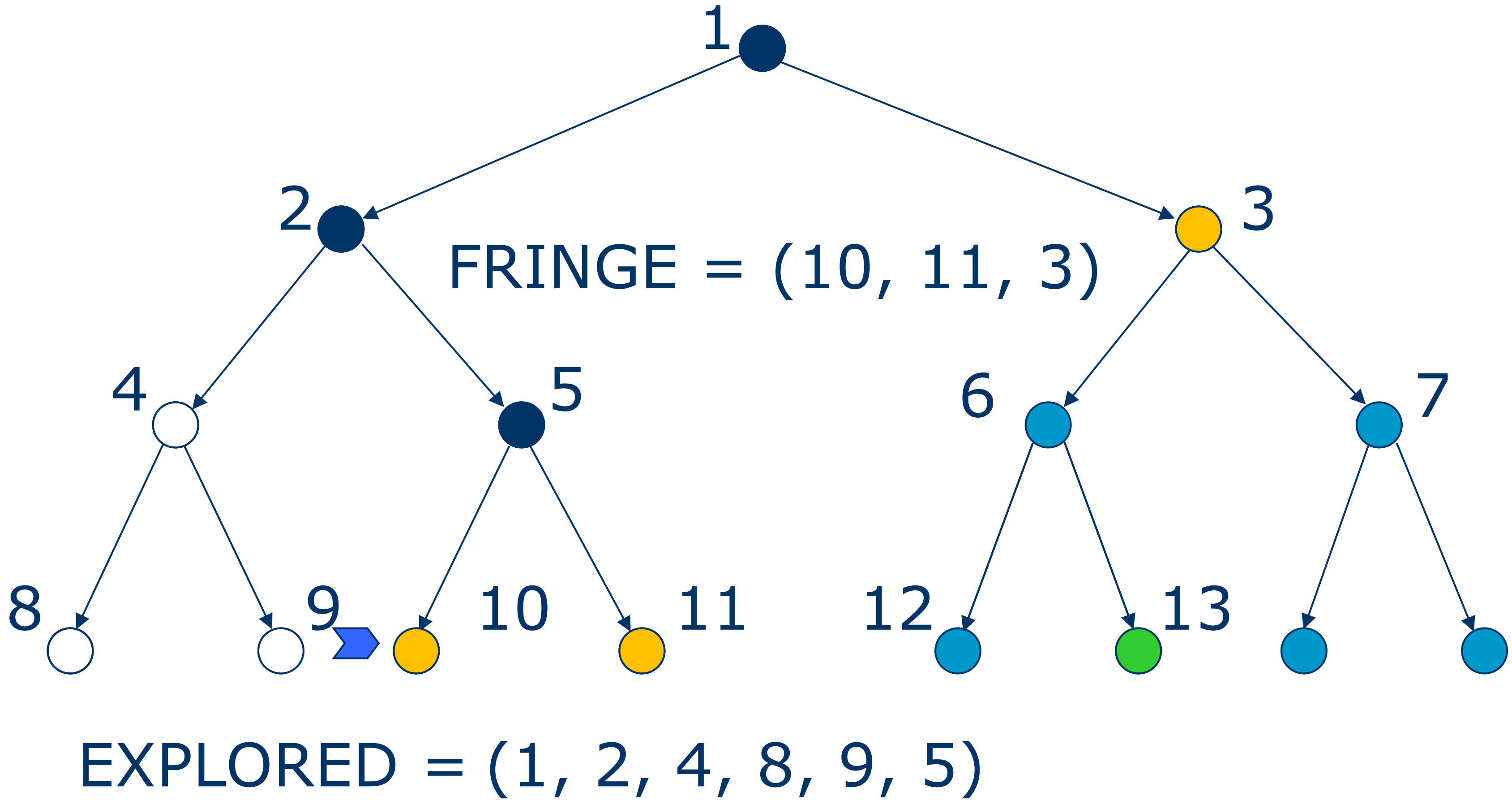
- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



EXPLORED = (1, 2, 4, 8, 9)

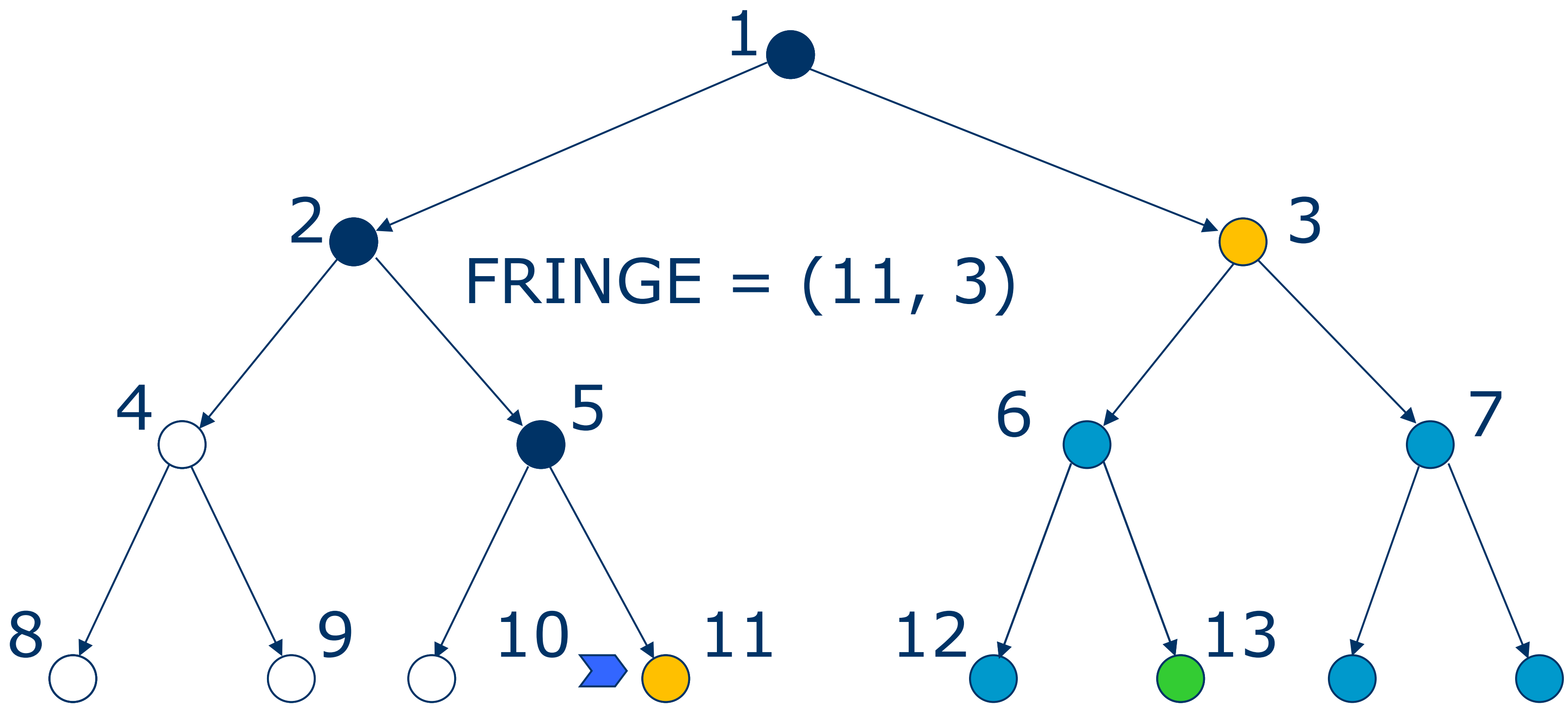
Depth-First Strategy

- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



Depth-First Strategy

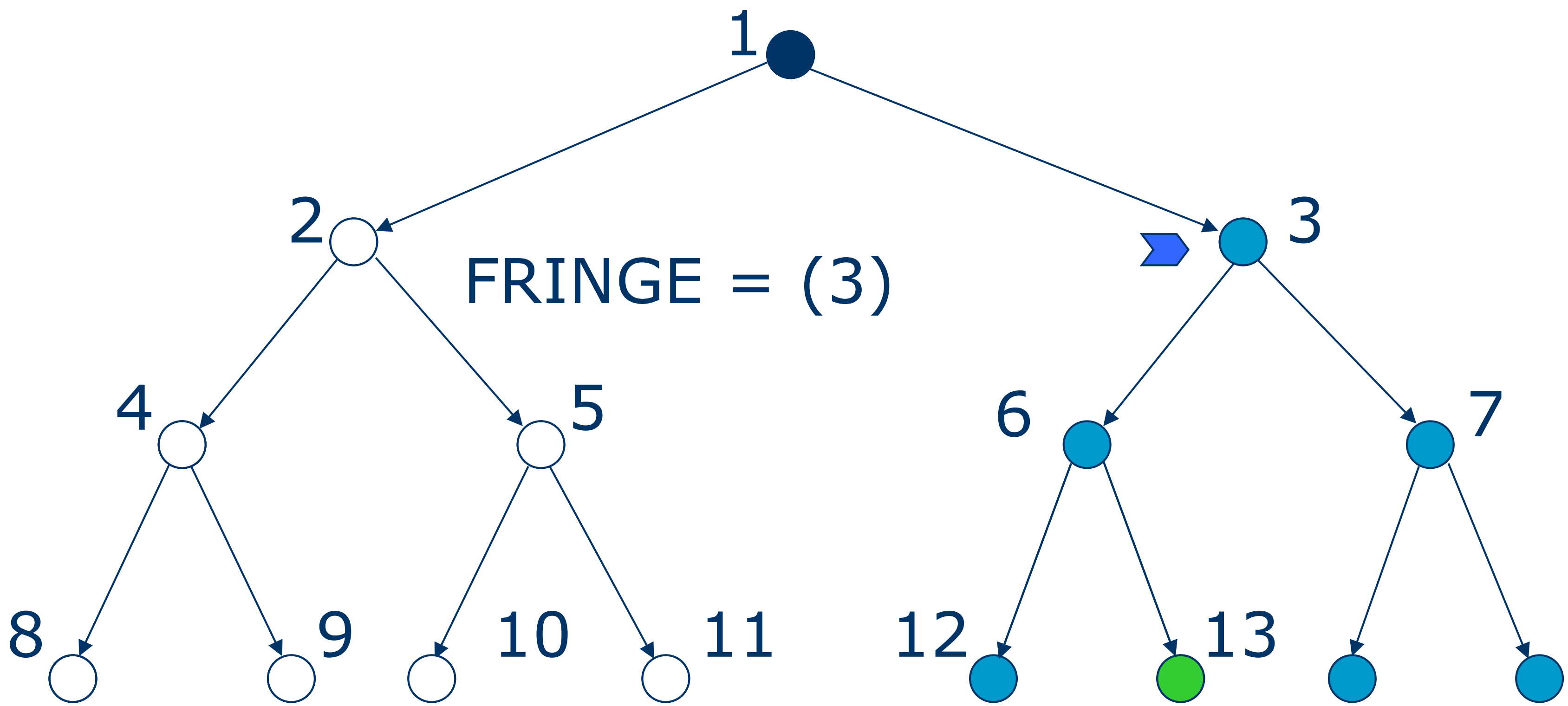
- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



EXPLORED = (1, 2, 4, 8, 9, 5, 10)

Depth-First Strategy

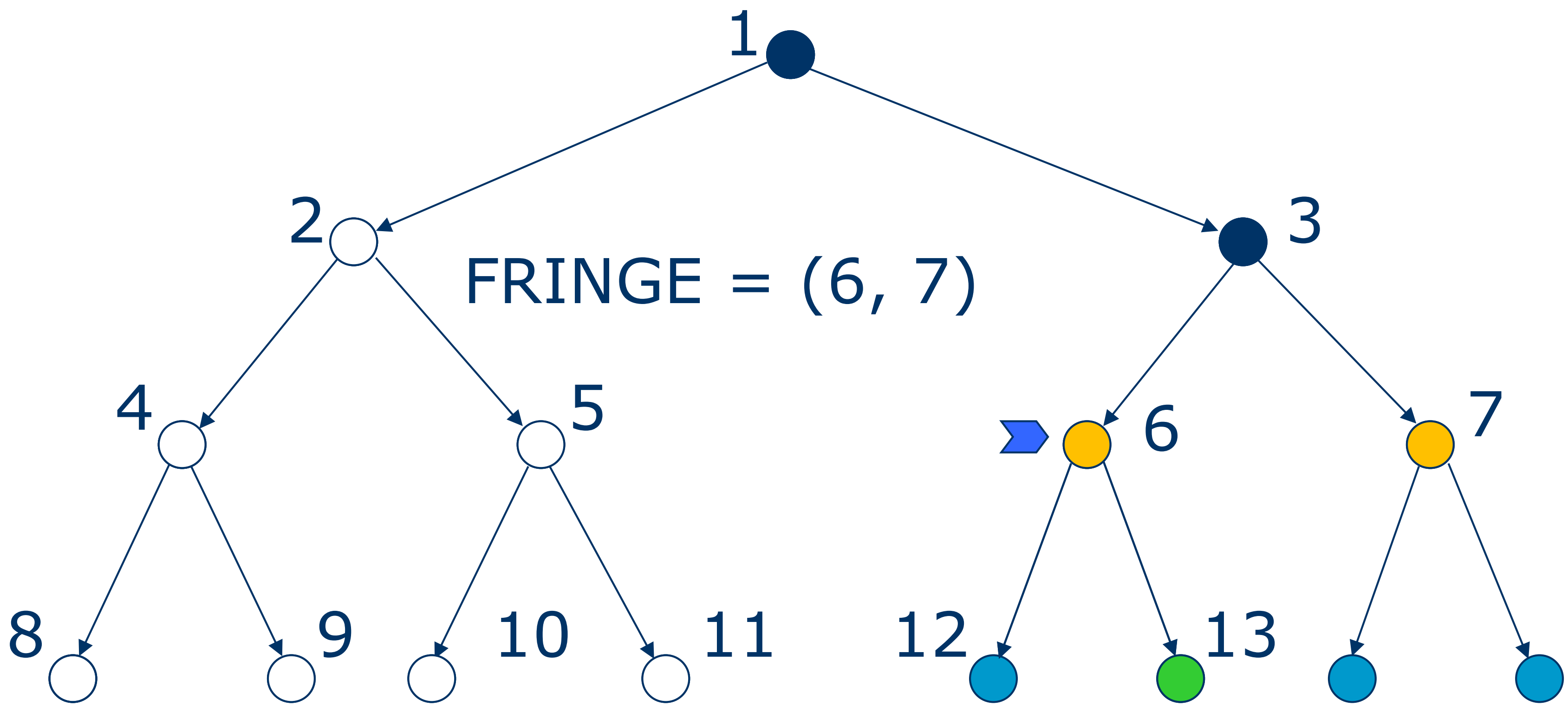
- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



EXPLORED = (1, 2, 4, 8, 9, 5, 10, 11)

Depth-First Strategy

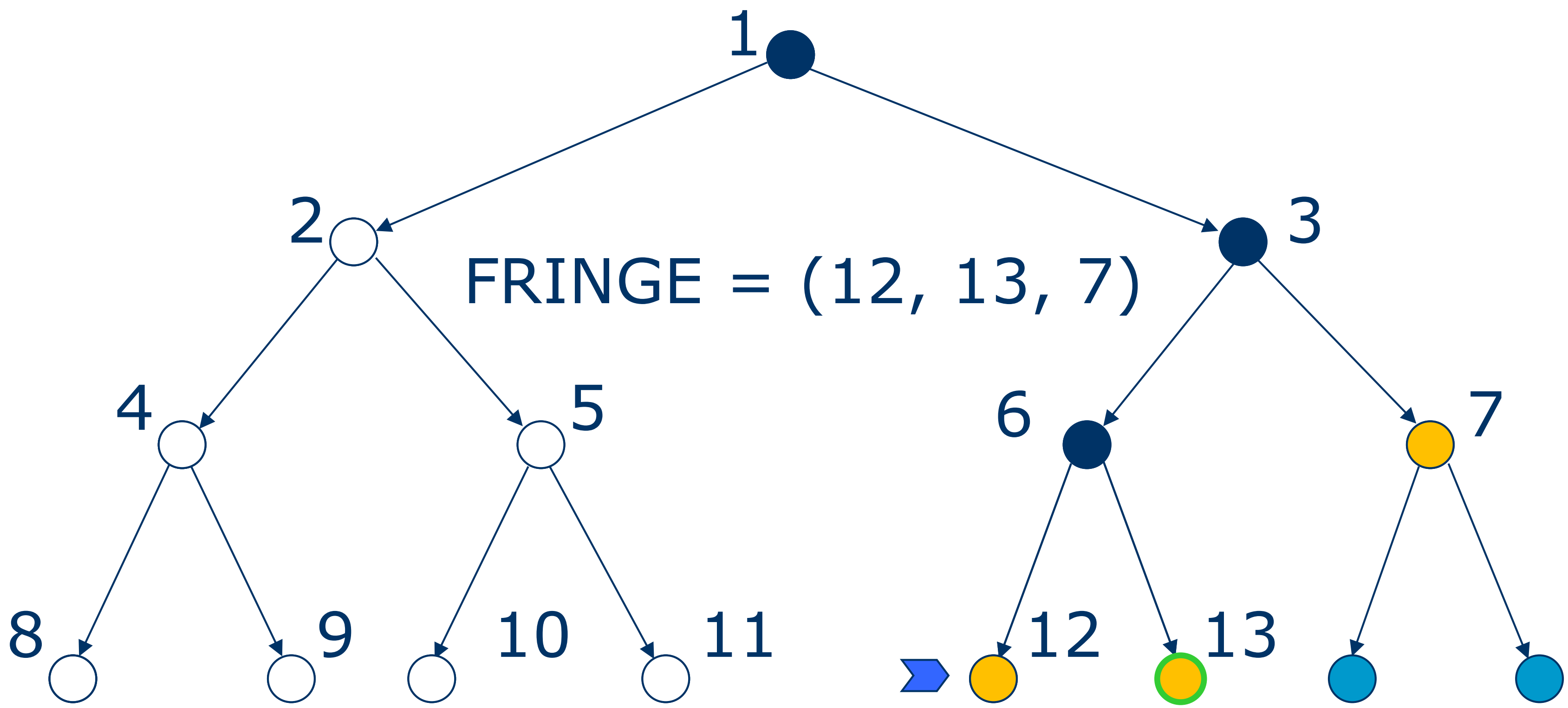
- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



EXPLORED = (1, 2, 4, 8, 9, 5, 10, 11, 3)

Depth-First Strategy

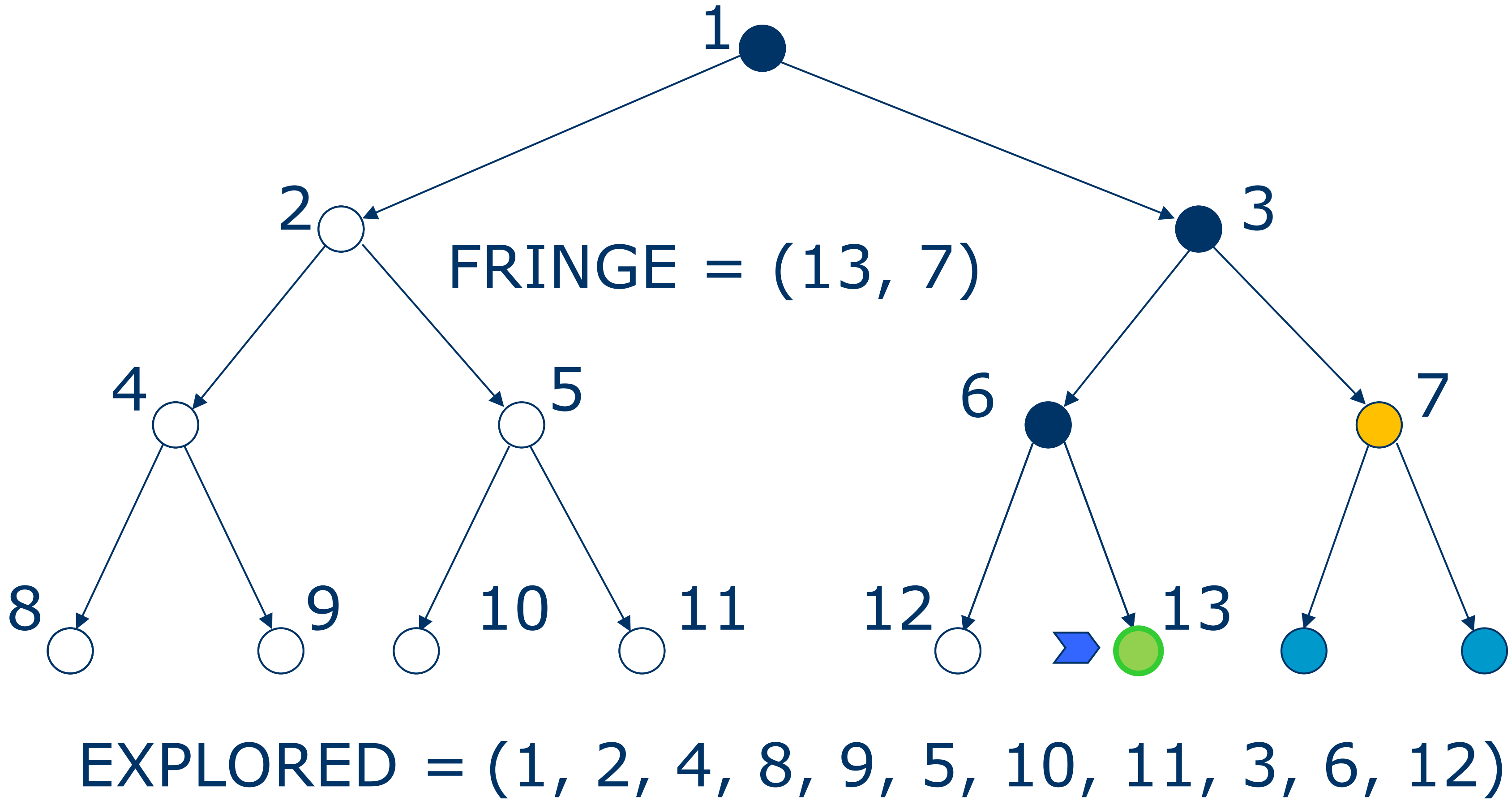
- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



EXPLORED = (1, 2, 4, 8, 9, 5, 10, 11, 3, 6)

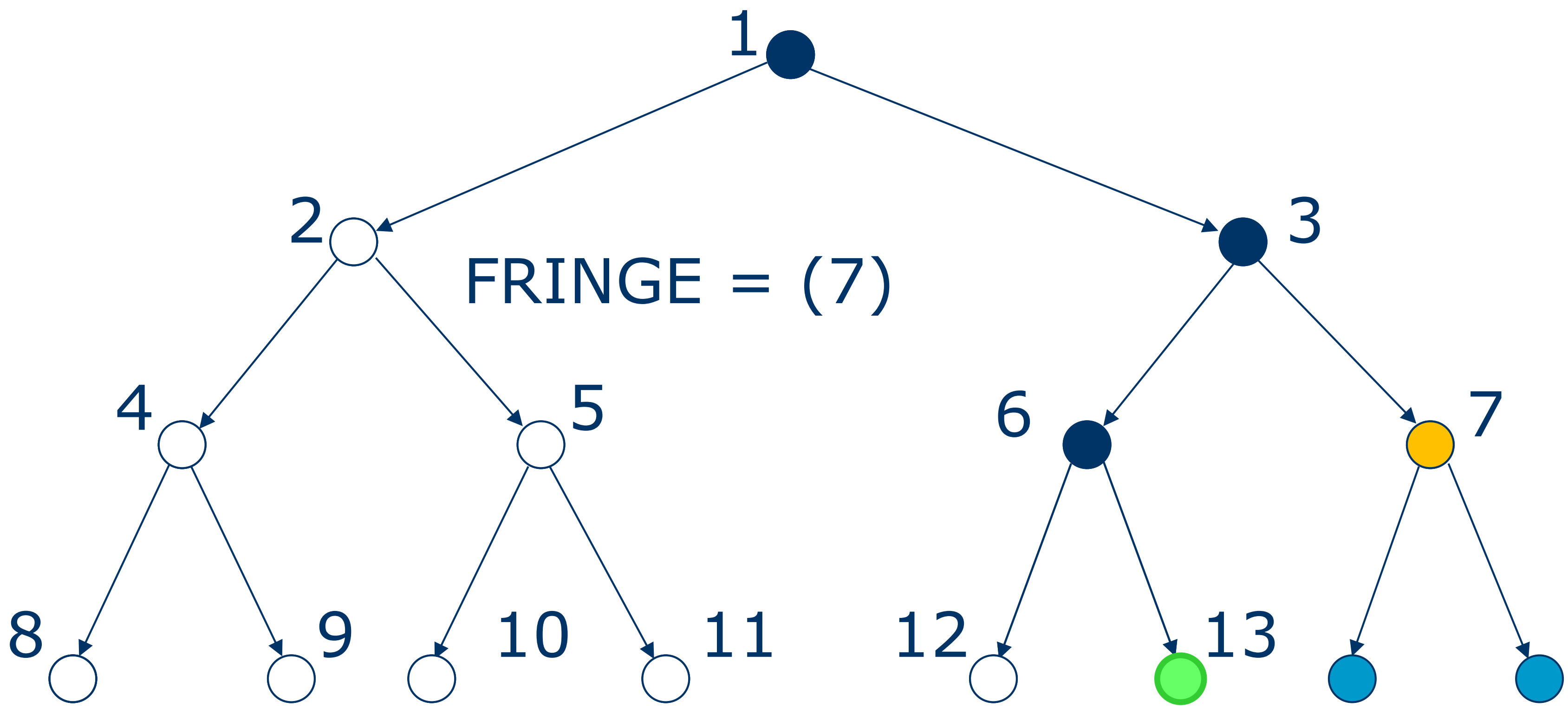
Depth-First Strategy

- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



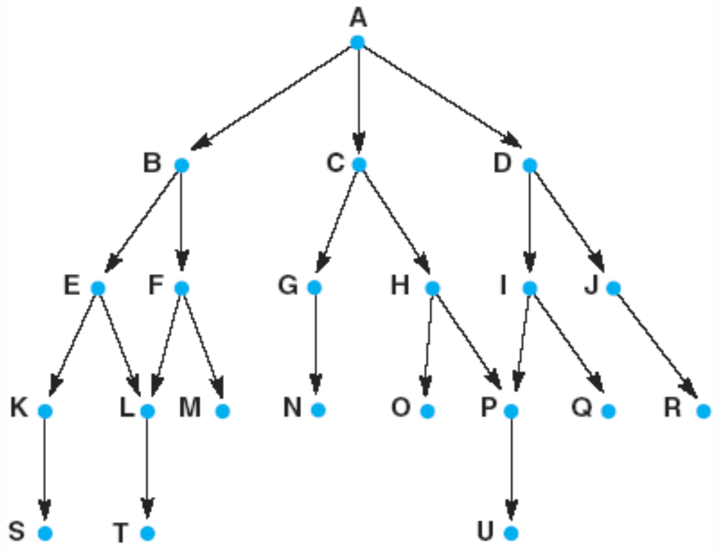
Depth-First Strategy

- explores a path all the way to a leaf before backtracking and exploring another path
- implementation: *fringe* is a LIFO queue (=stack)



EXPLORED = (1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13)

Another DFS Example (Exercise)



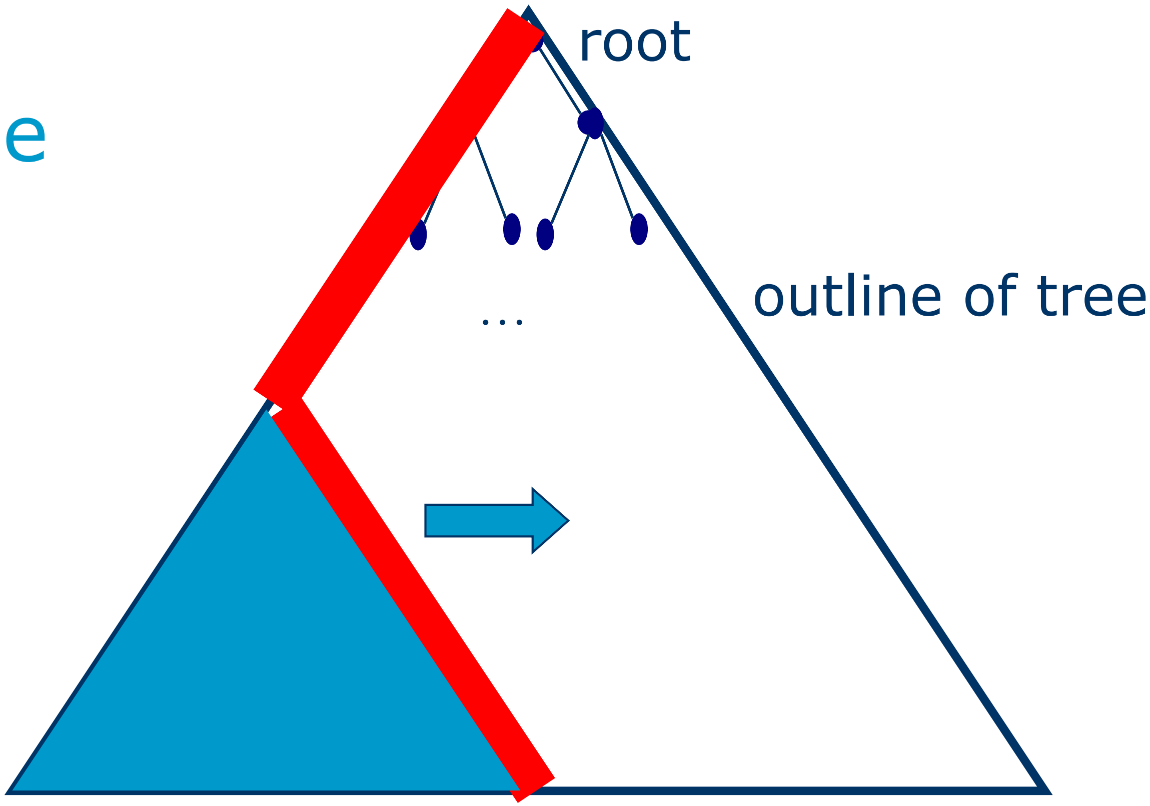
1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [E,F,C,D]; closed = [B,A]**
4. **open = [K,L,F,C,D]; closed = [E,B,A]**
5. **open = [S,L,F,C,D]; closed = [K,E,B,A]**
6. **open = [L,F,C,D]; closed = [S,K,E,B,A]**
7. **open = [T,F,C,D]; closed = [L,S,K,E,B,A]**
8. **open = [F,C,D]; closed = [T,L,S,K,E,B,A]**
9. **open = [M,C,D]**, as L is already on **closed**; **closed = [F,T,L,S,K,E,B,A]**
10. **open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]**
11. **open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]**

Depth-First Search Observations

- Only needs to store the path from the root to the leaf node as well as the unexpanded nodes. For a state space with a branching factor of b and a maximum depth of m, DFS requires **storage** of bm nodes
- Time complexity for DFS is b^m in the worst case
- If DFS goes down a infinite branch it will not terminate if it does not find a goal state
- If it does find a solution there may be a better solution at a lower level in the tree. Therefore, depth first search is **neither complete nor optimal**
- When this method succeeds, it doesn’t give the path to the goal

Depth-First Search

- Search Pattern: dive before spread
- Fringe in red
- Explored in blue
- Size of fringe
  - O(bd)
  - linear



Depth Limited Search (vs DFS)

- DFS may never terminate as it could follow a path that has no solution on it and is infinite

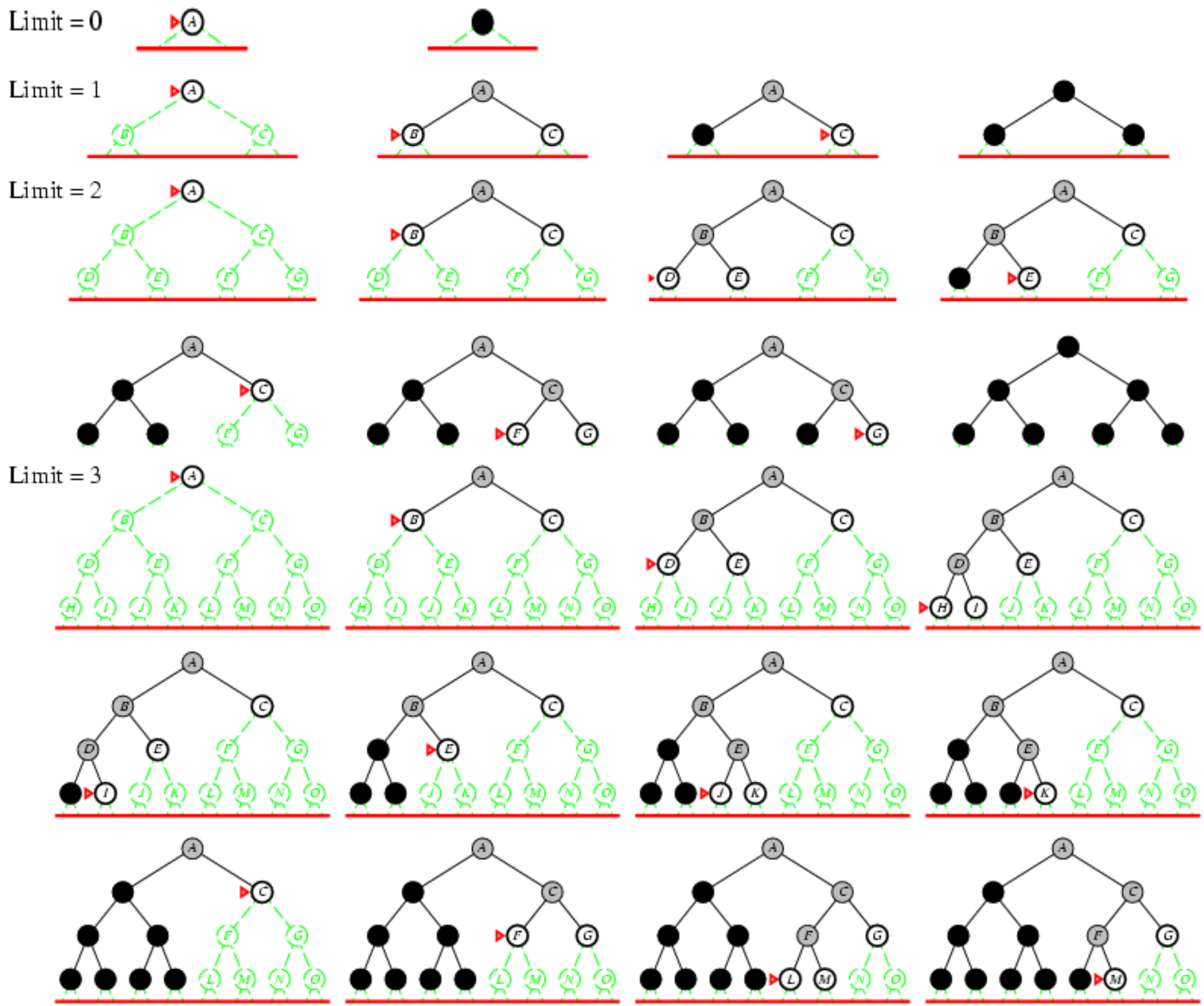
- DLS solves this by imposing a depth limit, at which point the search terminates for that particular branch

Depth Limited Search Observations

- Can be implemented by the general search algorithm using operators which keep track of the depth
- Choice of depth parameter is important
  - too deep is wasteful of time and space
  - too shallow and we may never reach a goal state
- If the depth parameter, l, is set deep enough then we are guaranteed to find a solution if one exists
  - therefore it is complete if l>=d (d=depth of solution)
- Space requirements are O(bl)
- Time requirements are $O(b^l)$
- DLS is not optimal

Iterative Deepening Search (vs DLS)

- On the Romania map there are 20 towns so any town is reachable in 19 steps
- In fact, any town is reachable in 9 steps
- Setting a depth parameter to 19 is obviously wasteful if using DLS
- **IDS** remedies the issue of choosing the right depth limit by **sequentially trying all depth limits**, first depth 0, then 1, then 2, and so on, **until a solution is found**.
- In effect it is **combining BFS and DFS**


Limit = 0
Limit = 1
Limit = 2
Limit = 3

Iterative Deepening Search Observations

- IDS may seem wasteful as it is expanding the same nodes many times. In fact, IDS expands just 11% more nodes than those by BFS or DLS when b=10
  - If b = 10, d = 5, N(IDS) = 123,450, N(BFS) = 111,110
  - Time Complexity = $O(b^d)$
  - Space Complexity = O(bd)
- For **large search spaces**, where the depth of the solution is not known, IDS is normally the preferred search method

State Space Search

- Two main approaches to searching a state space
  - **Data-driven search** which starts from an initial state and uses actions that are allowed to move forward until a goal is reached. Also known as ***forward chaining***
  - **Goal-driven search** which starts at the goal and work back toward a start state, by seeing what moves could have led to the goal state. Also known as ***backward chaining***

- Both search the same state space and produce the same result, however the order and actual number of states searched can be different

State Space Search

- Goal-driven search
  - **Goal** can be **clearly** and **easily formulated**, e.g. theorem prover; finding an exit path from a maze; medical diagnosis (with known conditions)
  - Problem data are not given but must be acquired by the problem solver

- Data-driven search
  - Goal is not clear or hard to formulate precisely
  - All or most of the data are given in the initial problem statement
  - Large number of potential goals

Example (Exercise)

Suppose the goal node for the state space tree in Figure 1 is M. List the *order* in which nodes will be *visited* for the following.

(i) breadth-first search; [3 marks]

(ii) depth-limited search with depth limit of 2 (assume the root of the tree is at depth of 0); and [3 marks]

(iii) depth-first iterative deepening search. [4 marks]

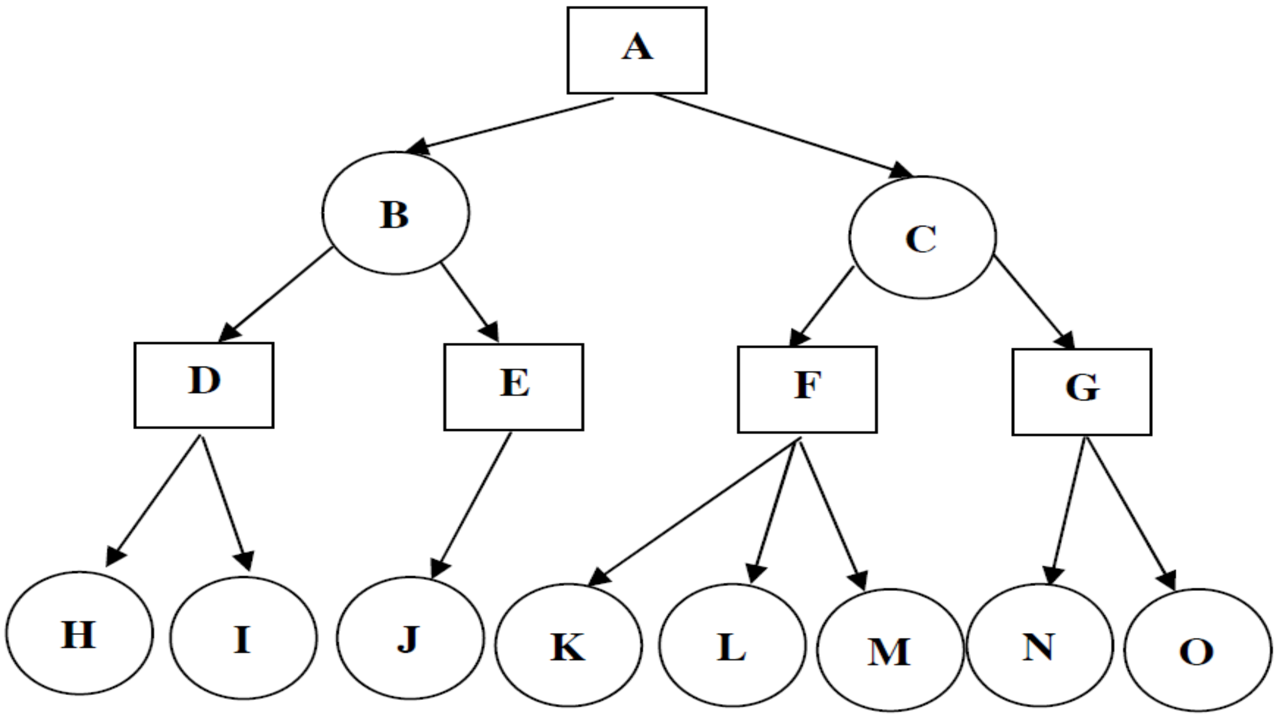


Figure 1.

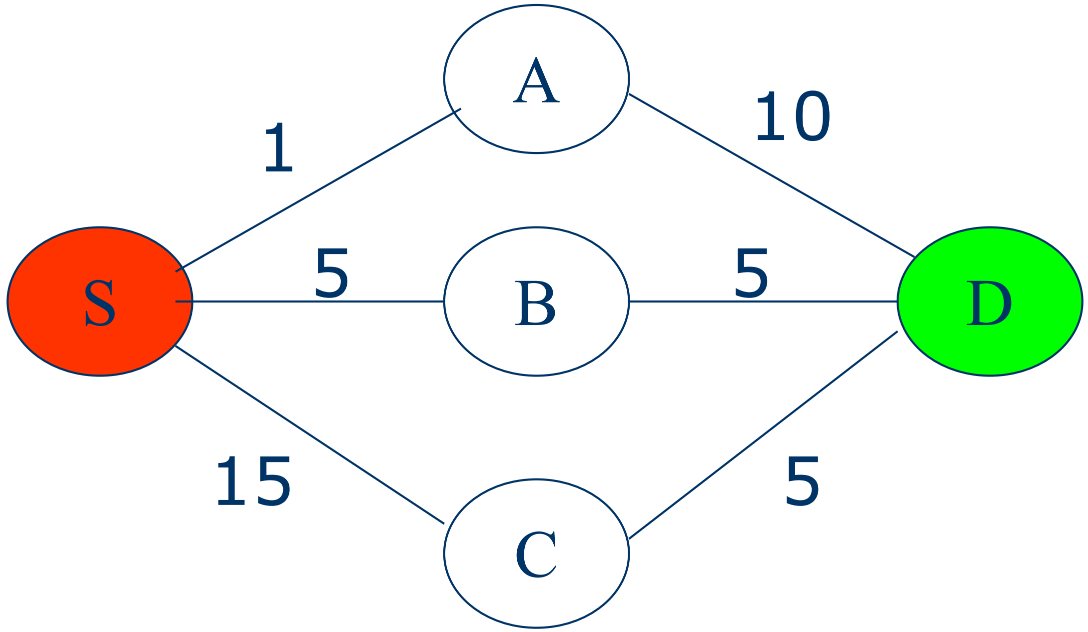
Uniform Cost Search

- **BFS** will find the **optimal** (shallowest) solution provided that ***all step costs are equal***.

- For other cases, **Uniform Cost Search (a variant of Dijkstra's algorithm for graph search)** can be used to find the cheapest solution provided that ***the path cost grows monotonically*** (i.e. never decreases as one proceeds along the path).

- Instead of expanding the shallowest node, Uniform Cost Search works by expanding the node *n* with the lowest path cost on the fringe.

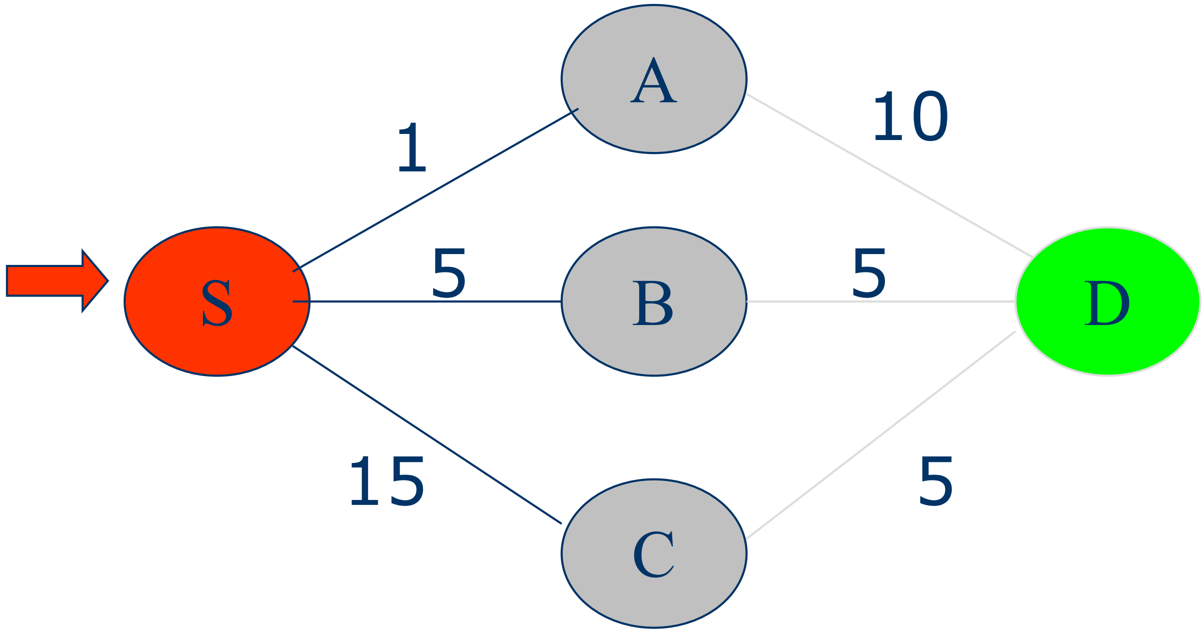
Uniform Cost Search



Similar to BFS except that it sorts (ascending order) the nodes in the fringe according to the cost of the node, where cost is the path cost, *g(n).*

Uniform Cost Search

Fringe = [S_0]



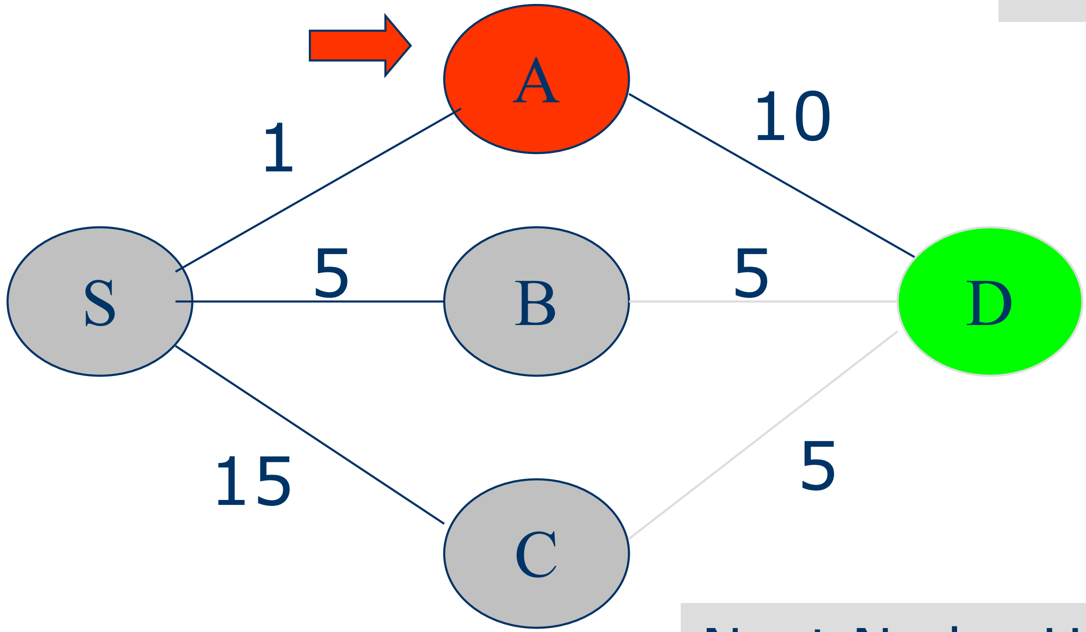
Updated Fringe=[A_1,B_5,C_{15}]

Next Node=Head of Fringe=S, S is not goal

Successor(S)={C,B,A}=expand(S) but sort them according to path cost.

Uniform Cost Search

Fringe = [A_1,B_5,C_{15}]



Updated Fringe=[B_5,D_{11},C_{15}]

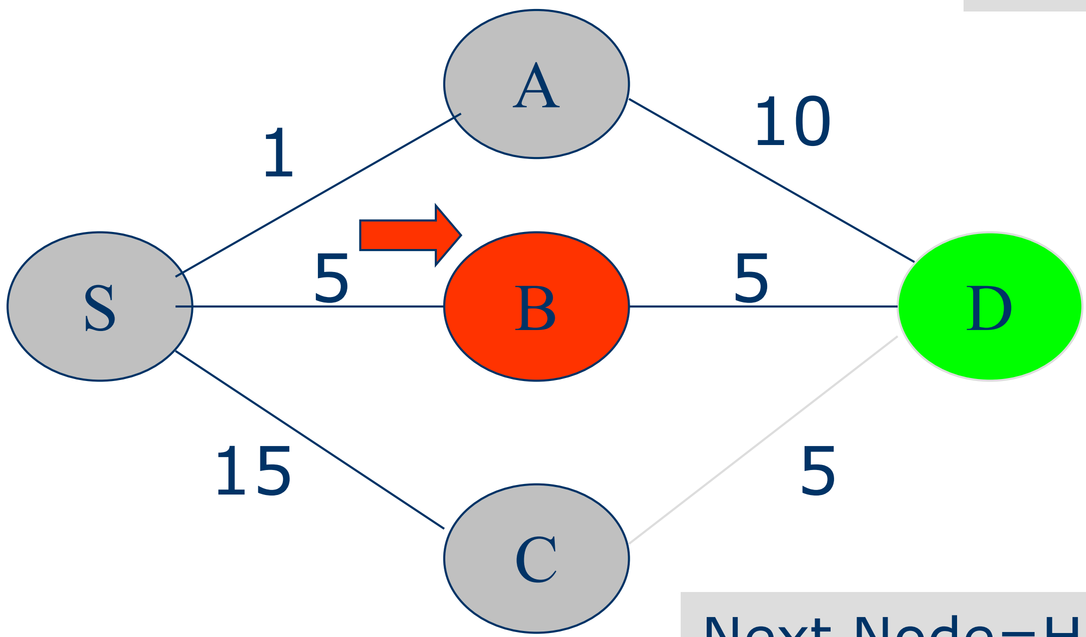
Next Node=Head of Fringe=A, A is not goal

Successor(A)={D}=expand(A)

Sort the queue according to path cost.

Uniform Cost Search

Fringe = [B_5,D_{11},C_{15}]



Next Node=Head of Fringe=B, B is not goal

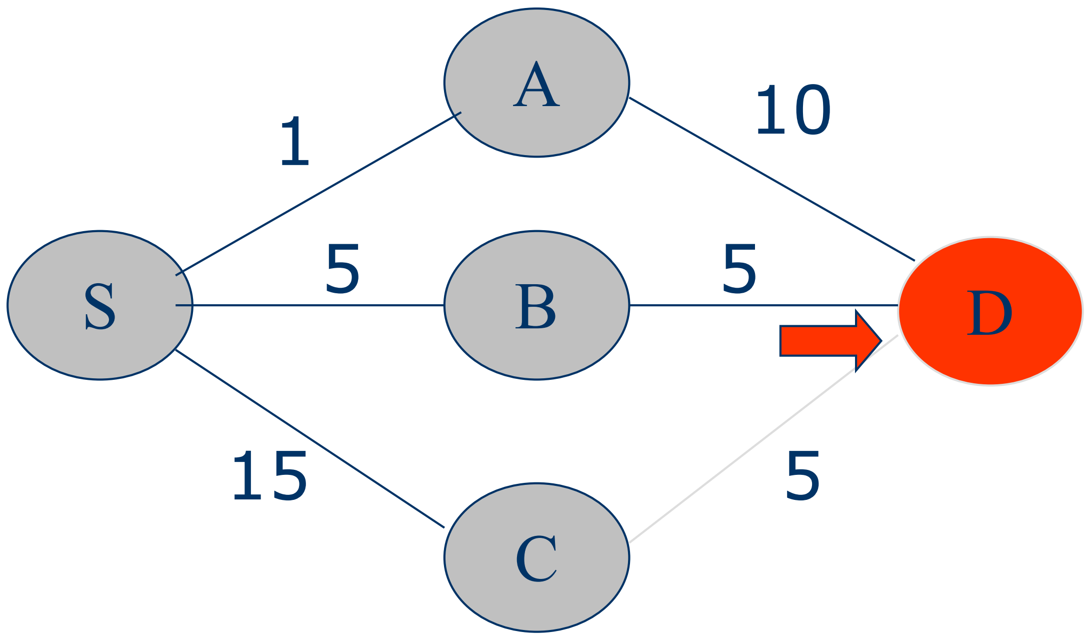
Successor(B)={D}=expand(B)

Sort the queue according to path cost.

Updated Fringe=[D_{10},D_{11},C_{15}]

Uniform Cost Search

Fringe = [$\mathbf{D_{10}}$, D_{11}, C_{15}]



Always finds the cheapest solution

Next Node=Head of Fringe=D,

D is a GOAL (cost 10 = 5+5)

S→B→D

Example (Exercise)

The start node and the goal node for the state space in Figure 1 are S and G respectively.

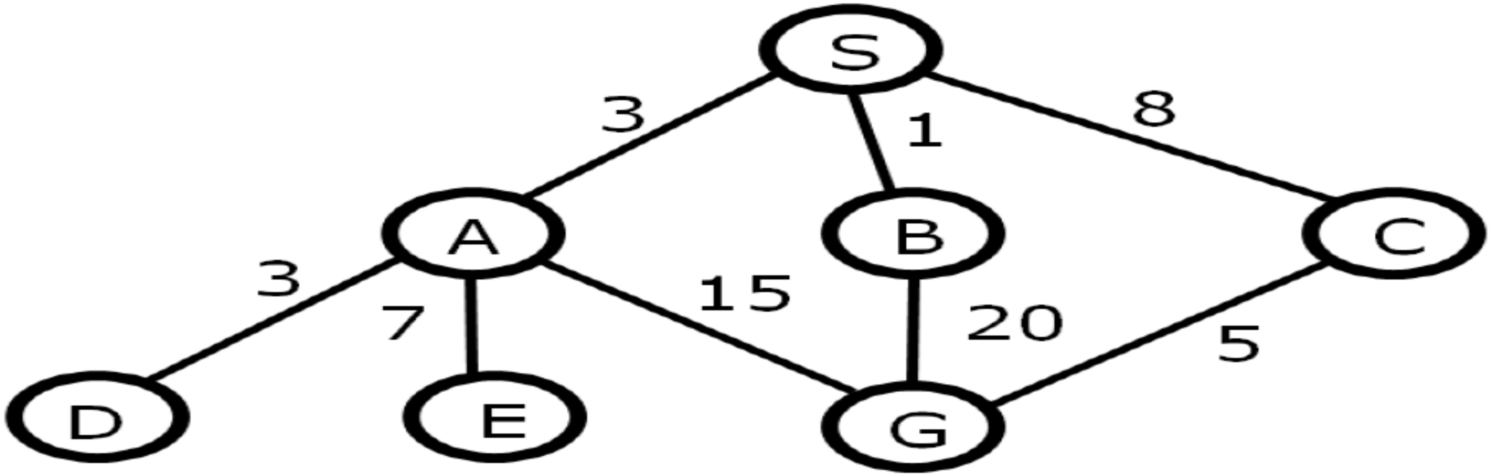


Figure 1.

For each of the search strategies below, work out the *solution path* and the *number of nodes expanded*. Show at each step what *nodes* are in the *queue*. Assume that processed nodes will be ignored.

(i) *depth-first search*;

(ii) *uniform cost search*.

Tree Search Algorithm

```
function TREE-SEARCH(problem, fringe) return a solution or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if EMPTY?(fringe) then return failure
        node ← REMOVE-FIRST(fringe)
        if GOAL-TEST[problem] applied to STATE[node] succeeds
                then return SOLUTION(node)
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function EXPAND(node, problem) return a set of nodes
    successors ← the empty set
    for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        STATE[s] ← result
        PARENT-NODE[s] ← node
        ACTION[s] ← action
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node]+1
        add s to successors
    return successors
```

Graph Search Algorithm

```
function GRAPH-SEARCH(problem, fringe) return a solution or failure
   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if EMPTY?(fringe) then return failure
       node ← REMOVE-FIRST(fringe)
       if GOAL-TEST[problem] applied to STATE[node] succeeds
               then return SOLUTION(node)
       child ← the empty set
       for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
           if result not in closed or fringe then do
               s ← a new NODE
               STATE[s] ← result
               PARENT-NODE[s] ← node
               ACTION[s] ← action
               PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
               DEPTH[s] ← DEPTH[node]+1
               add s to child
       fringe ← INSERT-ALL(child, fringe)
```

Summary

Evaluation	Breadth First	Uniform Cost	Depth First	Depth Limited	Iterative Deepening
Time	B^D	B^D	B^M	B^L	B^D
Space	B^D	B^D	BM	BL	BD
Optimal?	Yes	Yes	No	No	Yes
Complete?	Yes	Yes	No	Yes, if L >= D	Yes

B = Branching factor
D = Depth of solution
M = Maximum depth of the search tree
L = Depth Limit

Summary

- Repeated states
- Evaluation of various search strategies
  - Blind searches
    - ✓ Breath-first
    - ✓ Depth-first
    - ✓ Depth limited
    - ✓ Iterative deepening
    - ✓ Uniform cost
- Algorithmic Implementation

Acknowledgements

Most of the lecture slides are
adapted from the same module
taught in Nottingham UK
by
Professor Graham Kendall,
Dr. Rong Qu
and
Dr. Andrew Parker