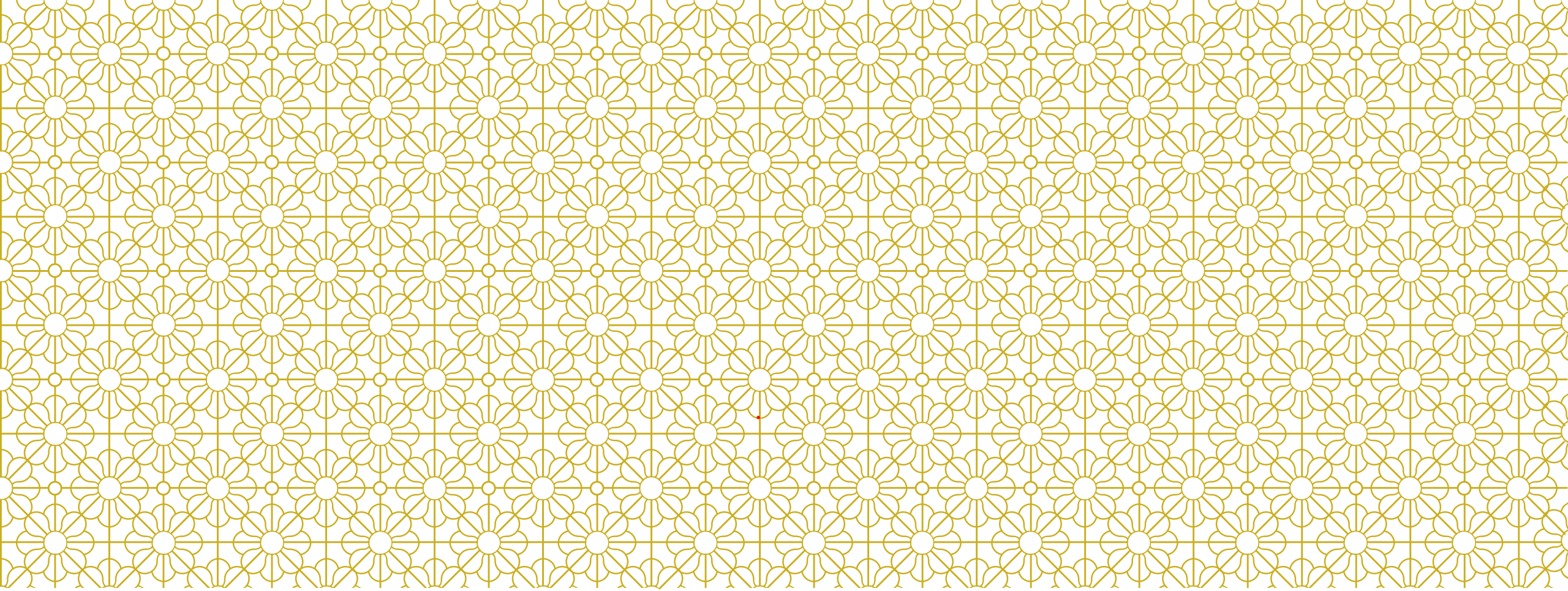
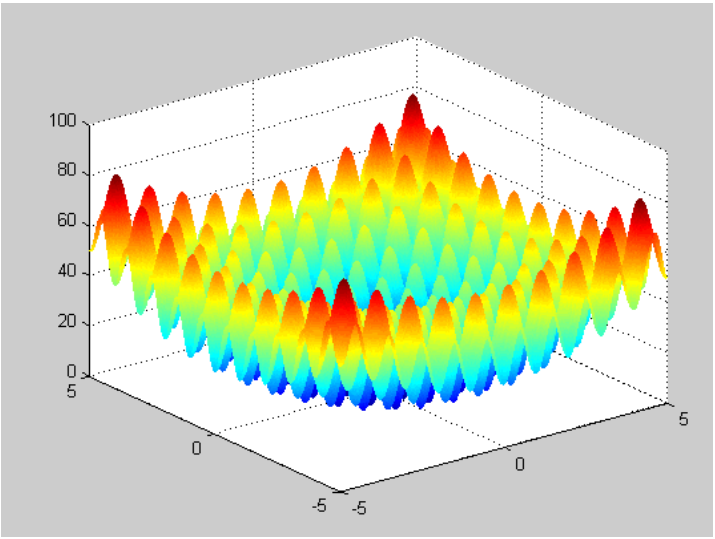


BLIND SEARCHES
FUNDAMENTALS OF AI(COMP1037)

Dr Qian Zhang
University of Nottingham,
Ningbo China 2023

HOW GOOD IS A SEARCH STRATEGY?

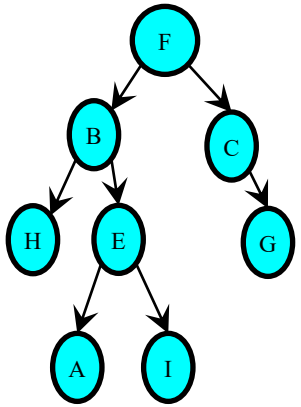
- Does our search method actually **find** a solution?
  - if **a goal exists** then the search will always **find it**
  - if **no goal exists** then the search will eventually **finish** and be able to say for sure that no goal exists
- Is it a good **solution** (optimal) ?
  - Path Cost
- What is the **search cost** ?
  - **Time**
  - **Memory**



$$x_i \in [-5.12, 5.12]$$

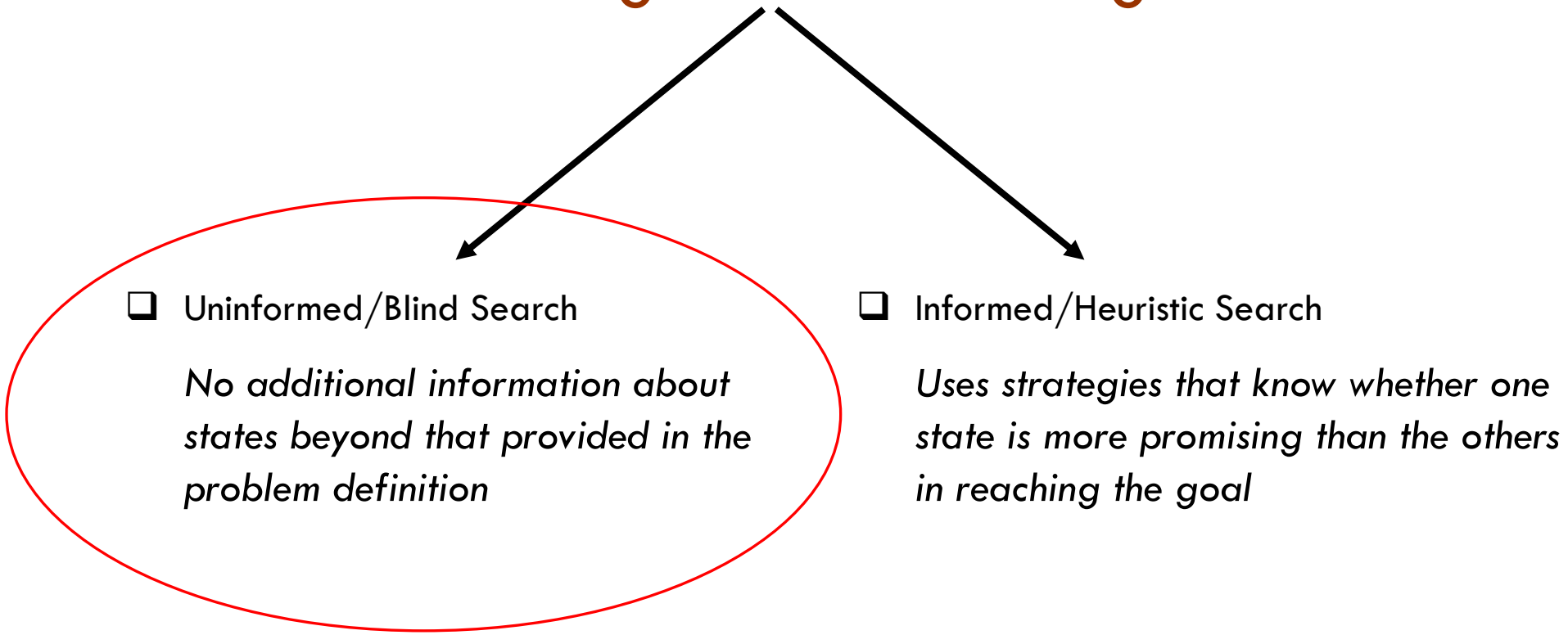
EVALUATION CRITERIA

- Strategies are evaluated along the following dimensions:
  - completeness—does it always find a solution if one exists?
  - time complexity—number of nodes generated/expanded
  - space complexity—maximum number of nodes in memory
  - optimality—does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - b—maximum branching factor of the search tree
  - d—depth of the least-cost solution
  - m—maximum depth of the state space (may be ∞)



SEARCH STRATEGIES

Two Categories of Strategies



- Uninformed/Blind Search

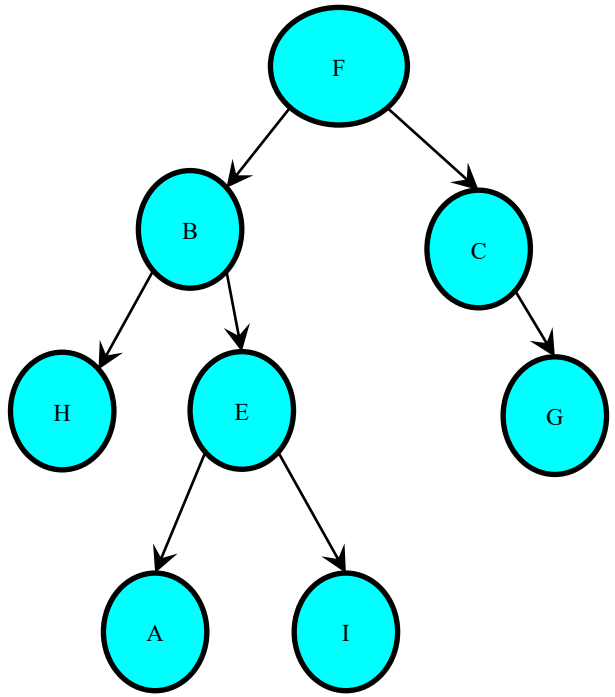
  No additional information about states beyond that provided in the problem definition

- Informed/Heuristic Search

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

BLIND SEARCHES - IMPLEMENTATION

- Fundamental actions (operators):
- "Expand"
  - Ask a node for its children
- "Test"
  - Test a node to see whether it is a goal



GENERAL TREE SEARCH

Function **General-Search**(p, **QUEUING-FN**) returns a solution or failure

frontier = Make-Queue(Make-Node(Initial-State[p]))

Loop do

    If frontier is empty then return failure

    node = Remove-Front(frontier)

    If **Goal-Test[p]** on State(node) succeeds then return node

    frontier = **QUEUING-FN**(frontier, (**Expand(node, Actions[p])**)

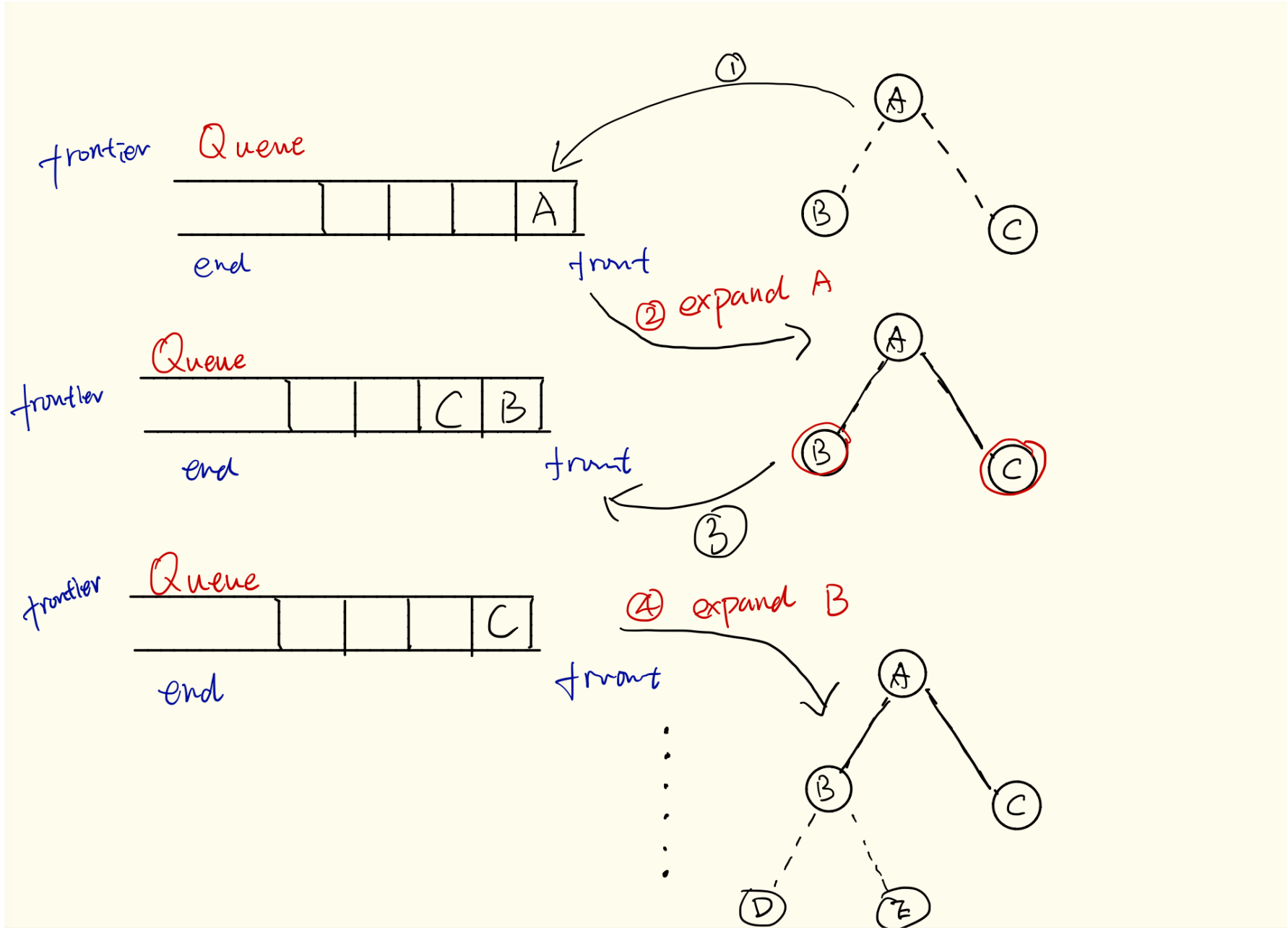
End

BREADTH FIRST SEARCH (BFS)

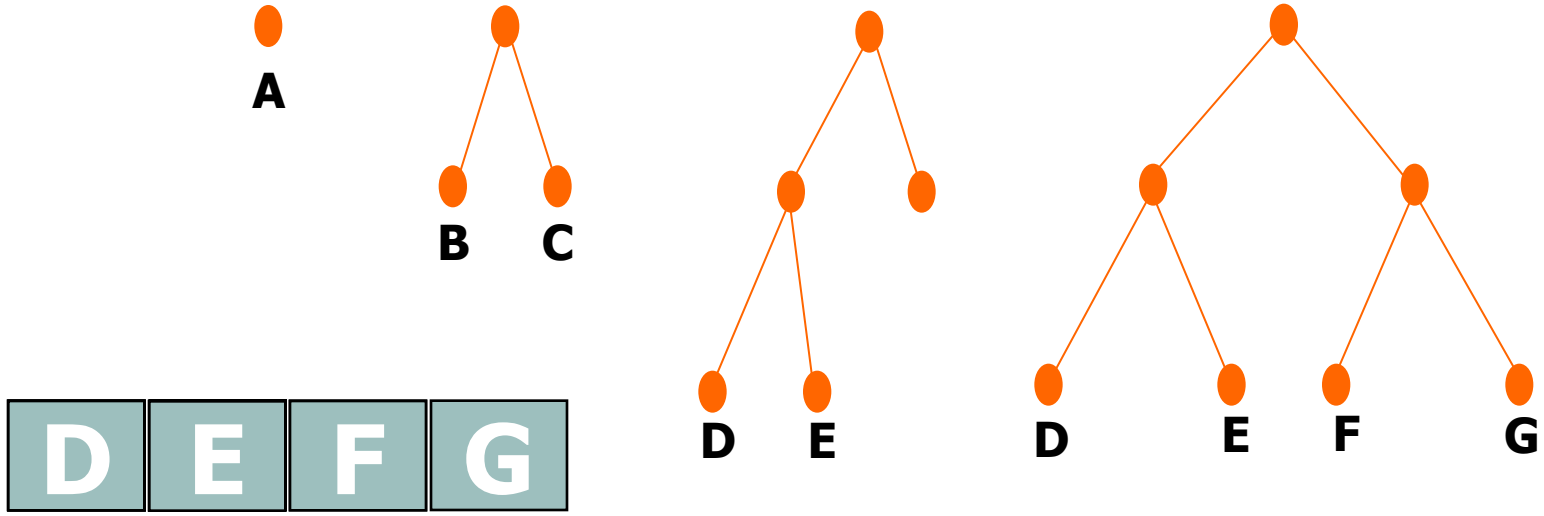
BFS - METHOD

- Expand shallowest unexpanded node
  - Expand Root Node First
  - Expand all nodes at level 1 before expanding level 2
  - …
  - Expand all nodes at level d before expanding nodes at level d+1

- Queuing function: adds nodes to the end of the queue **(FIFO)**
  - **General-Search(problem, ENQUEUE-AT-END)**


frontier
Queue
A
end
front
1
A
B
C
2 expand A
frontier
Queue
C
B
end
front
3
frontier
Queue
C
end
front
4 expand B
A
B
C
D
E

BFS - IMPLEMENTATION

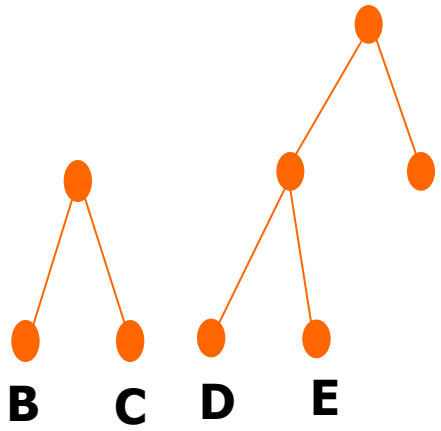


node = **Remove-Front**(**frontier**)
If Goal-Test[p] on State(node) succeeds
    then return node
frontier = **QUEUING-FN**(**frontier**, Expand(node, Actions[p]))

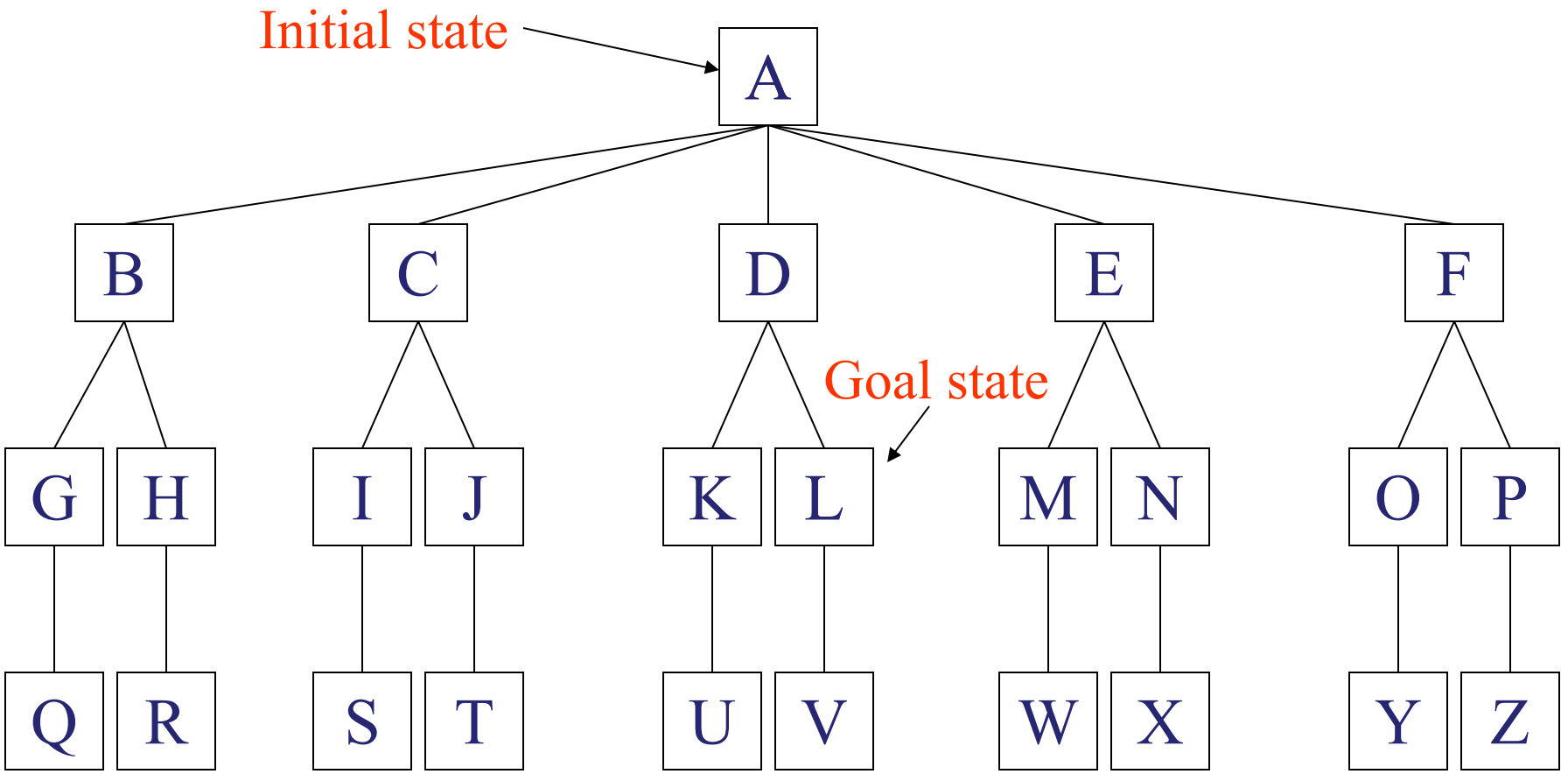
SEARCHES - IMPLEMENTATION

Three types of nodes during the search in tree search

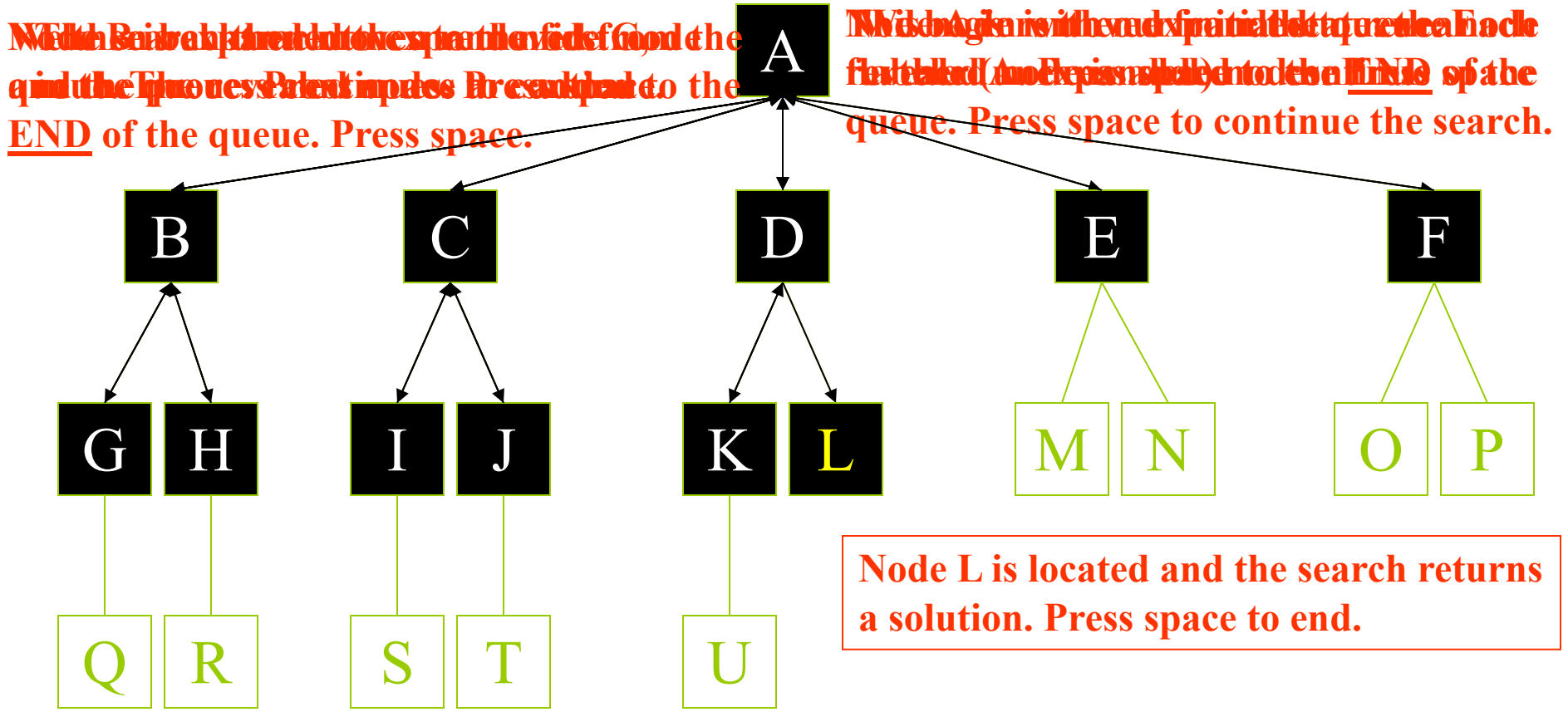
- **Frontier nodes (open nodes, leaves)**
  - have been discovered
  - have not yet been processed
    - Children not yet explored; not yet tested if they are goal
- **Visited nodes (closed nodes)**
  - have been discovered
  - have been processed
    - Children explored; tested if they match a goal
- **Undiscovered nodes**
  - have not yet been discovered



The example node set



Press space to see a BFS of the example node set



Press space to begin the search

Size of Queue: 0	**Queue: Empty**	
Nodes expanded: 11	**FINISHED SEARCH**	**Current level: 2**

Breadth-First Search Pattern

EVALUATING A SEARCH

We'll evaluate all the tree search techniques w.r.t the below 4 criteria

- 1. **Completeness**

Guaranteed to find a solution if one exist?

- 2. **Time Complexity**

How long does it take to find a solution?

- 3. **Space Complexity**

How much memory required to perform the search?

- 4. **Optimality**

Find the optimal solution (one or all optimal solutions)?

EVALUATING BFS

Evaluating against four criteria

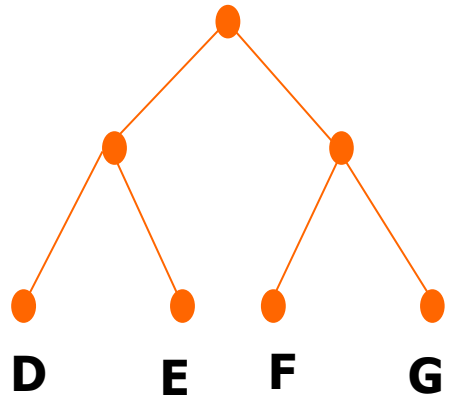
- Optimal
- Complete
- Time complexity
- Space complexity

EVALUATING BFS

- Space Complexity
  $O(b^d)$ i.e. number of leaves

- Time Complexity
  $O(b^d)$ i.e. total number of nodes in the tree

  - b: the maximum branching factor
  - d: is the depth of the search tree



- The space / time complexity could be less as the solution may be found before the dth level.

- Big O: notation in complexity theory. We use it to measure how the problem size affects the algorithm's computational resource (time or memory).

Combinatorial explosion!

EXPONENTIAL GROWTH

- Combinatorial explosion: the number of problem solutions grows exponentially with its size
- Time and memory requirements for breadth-first search, assuming a branching factor of 10, memory of 100 bytes per node and searching 1000 nodes/second

Depth	Nodes	Time		Memory	
0	1	1	millisecond	100	bytes
2	111	0.1	second	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

GENERAL TREE SEARCH

Function **General-Search**(p, **QUEUING-FN**) returns a solution or failure

frontier = Make-Queue(Make-Node(Initial-State[p]))

Loop do

    If frontier is empty then return failure

    node = Remove-Front(frontier)

    If **Goal-Test[p]** on State(node) succeeds then return node

    frontier = **QUEUING-FN**(frontier, (**Expand(node, Actions[p])**)

End

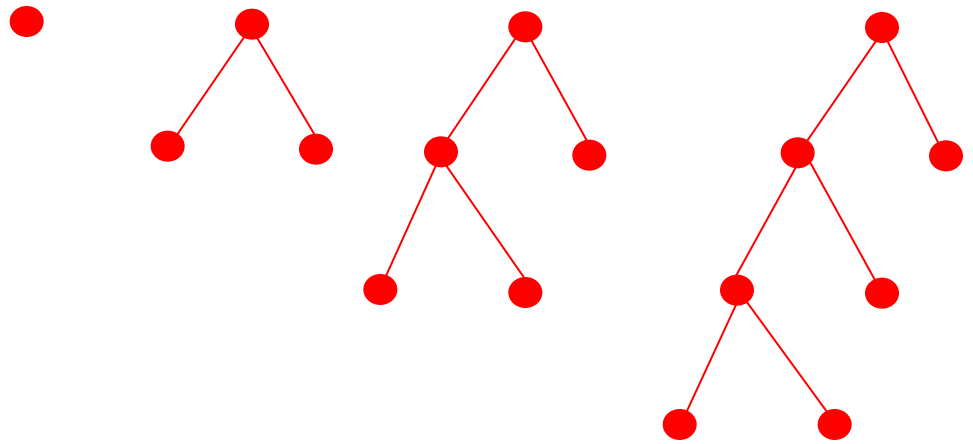
DEPTH FIRST SEARCH

DEPTH FIRST SEARCH - METHOD

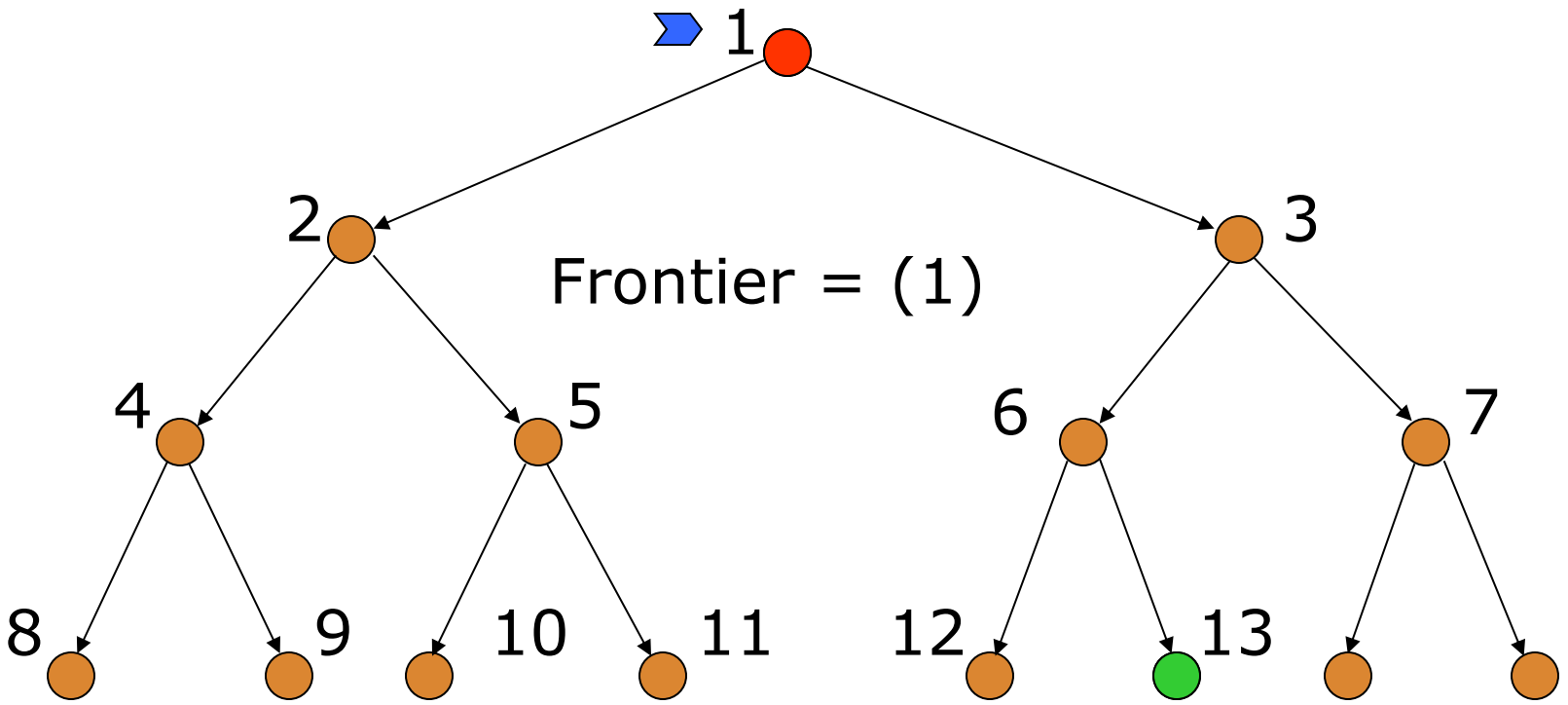
- Expand Root Node First
- Explore one branch before exploring another branch
- Queuing function: adds nodes to the **front** of the queue **(LIFO)**

Function DFS(problem) returns a solution or failure

Return General-Search(problem, ENQUEUE-AT-FRONT)

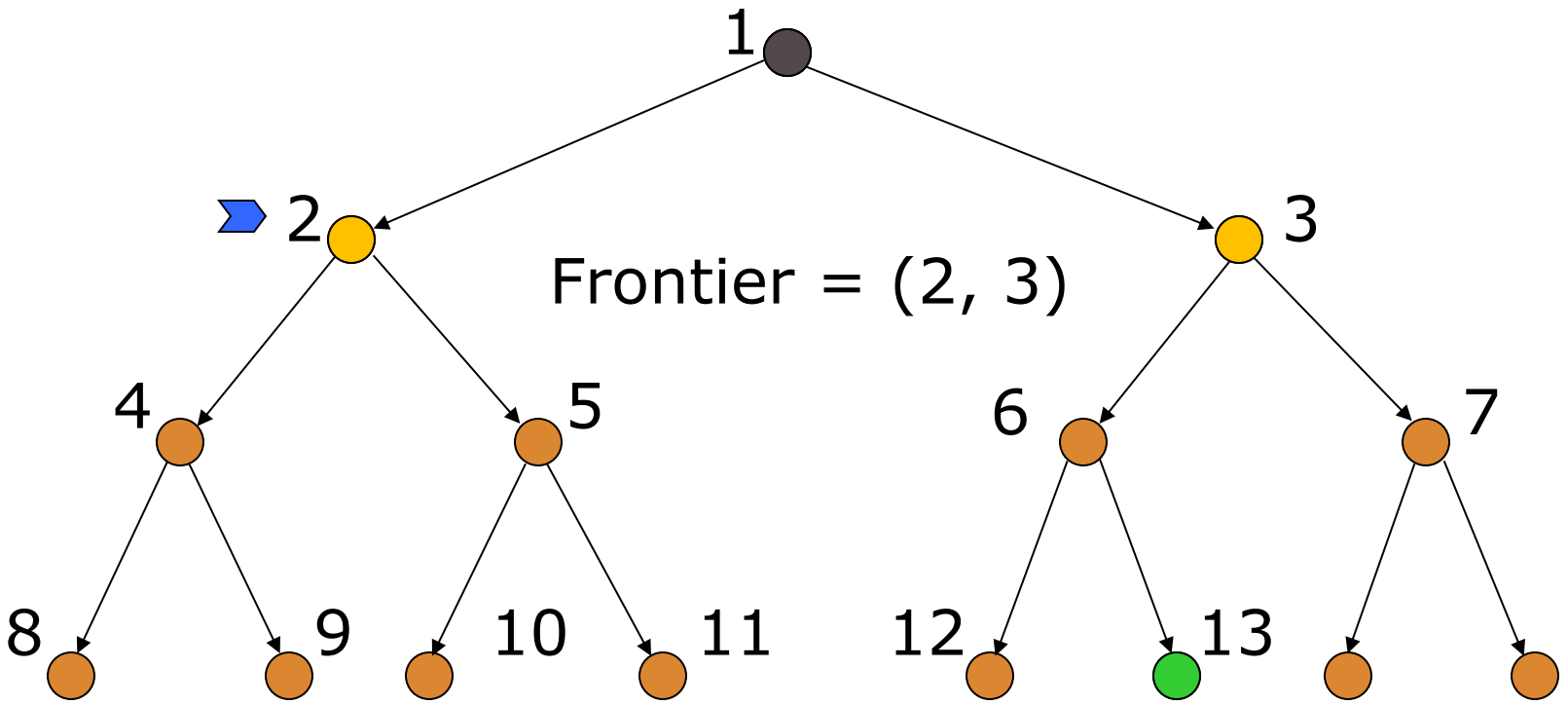


DEPTH-FIRST STRATEGY



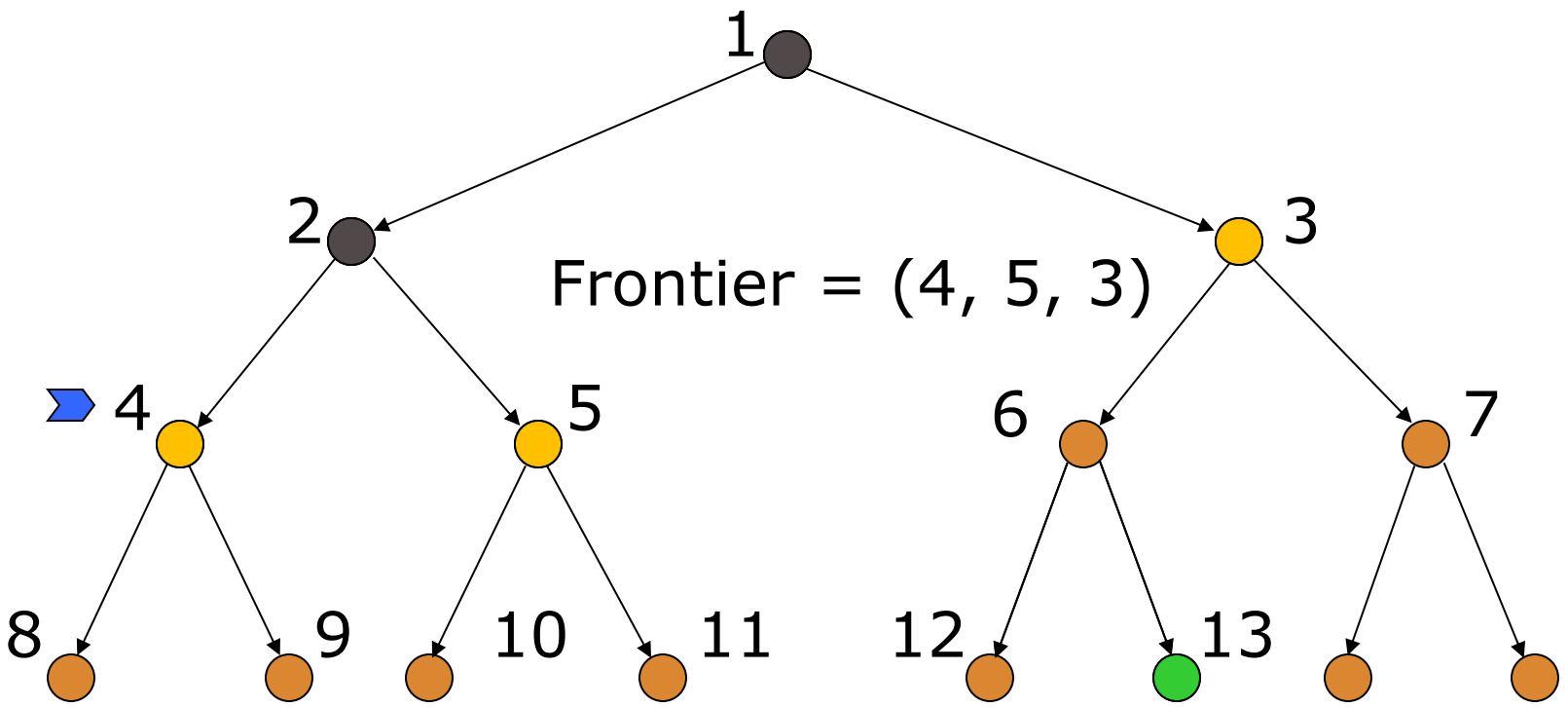
EXPLORED =

DEPTH-FIRST STRATEGY



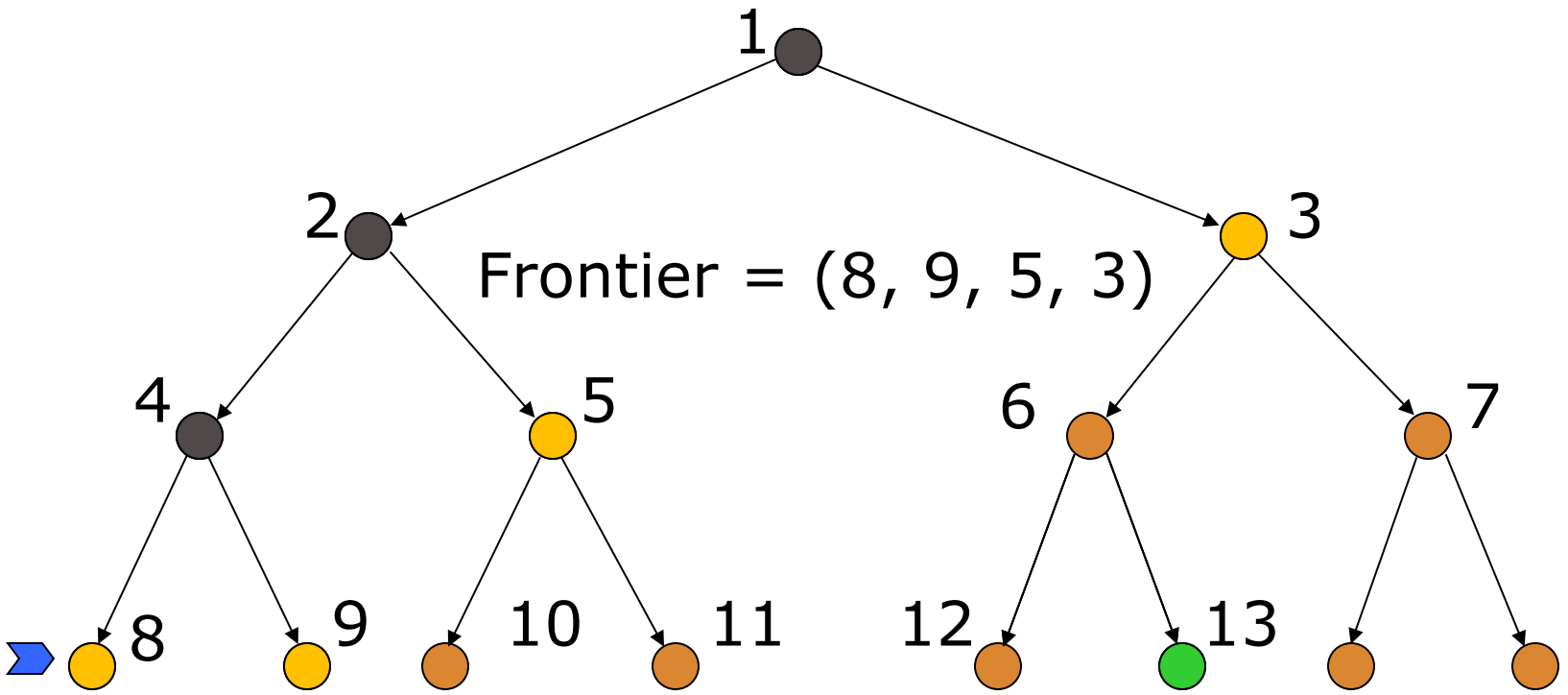
EXPLORED = (1)

DEPTH-FIRST STRATEGY



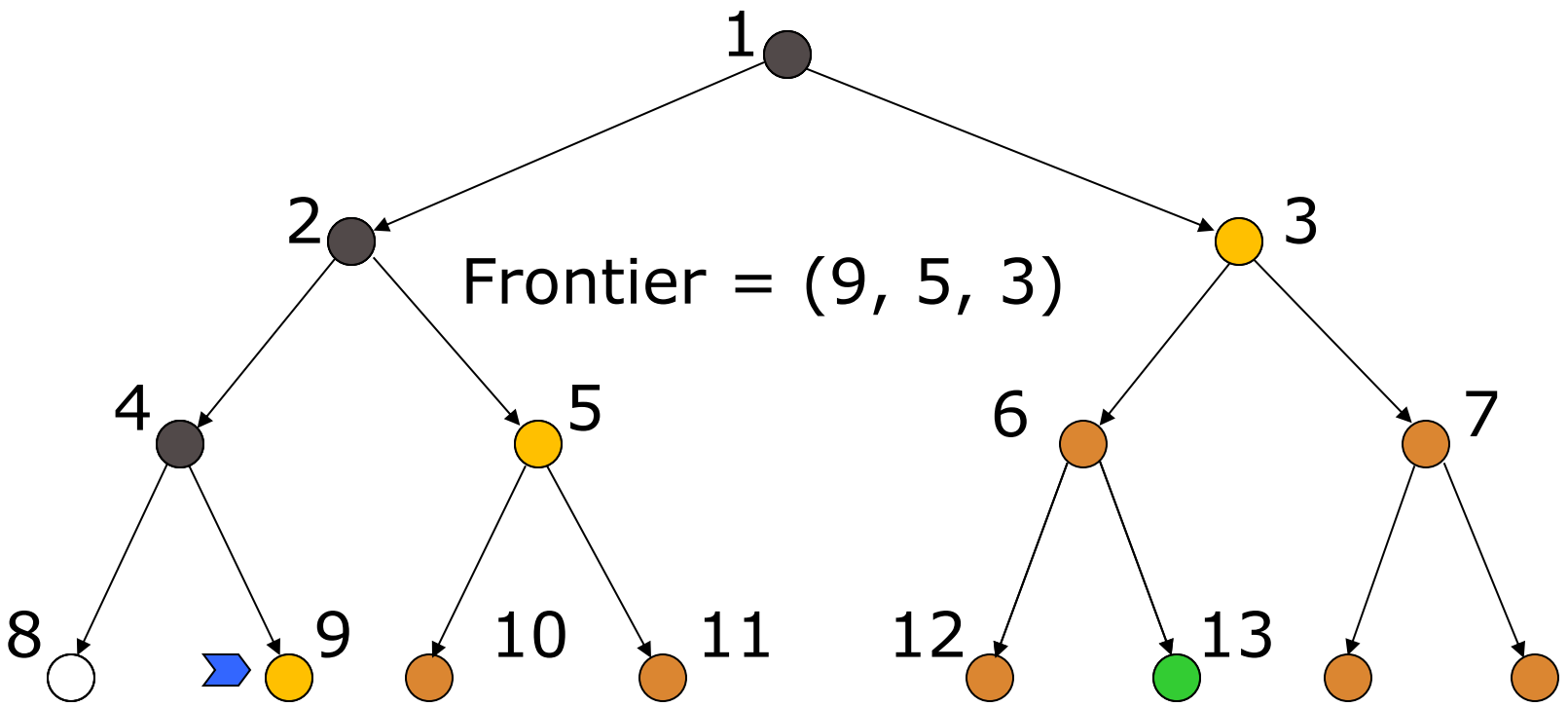
EXPLORED = (1, 2)

DEPTH-FIRST STRATEGY



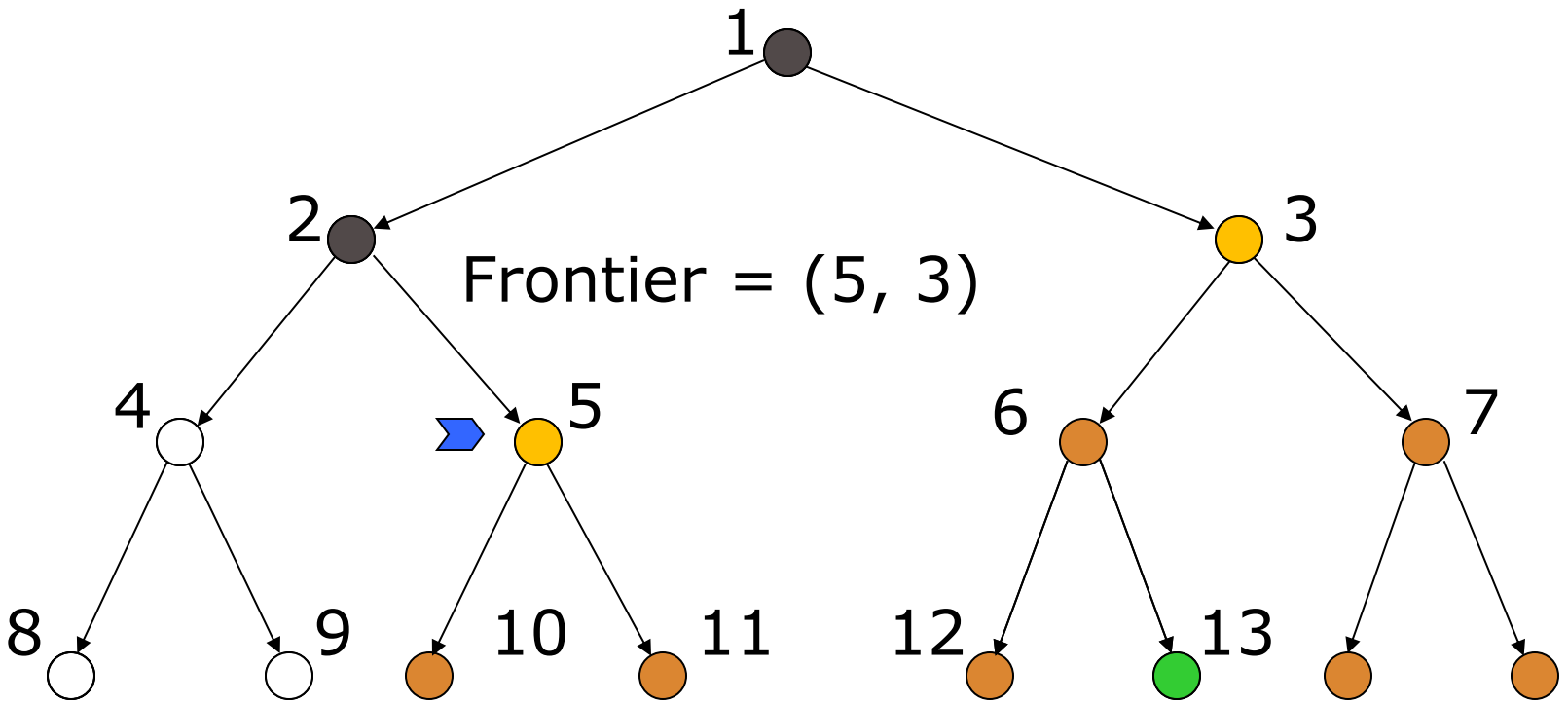
EXPLORED = (1, 2, 4)

DEPTH-FIRST STRATEGY



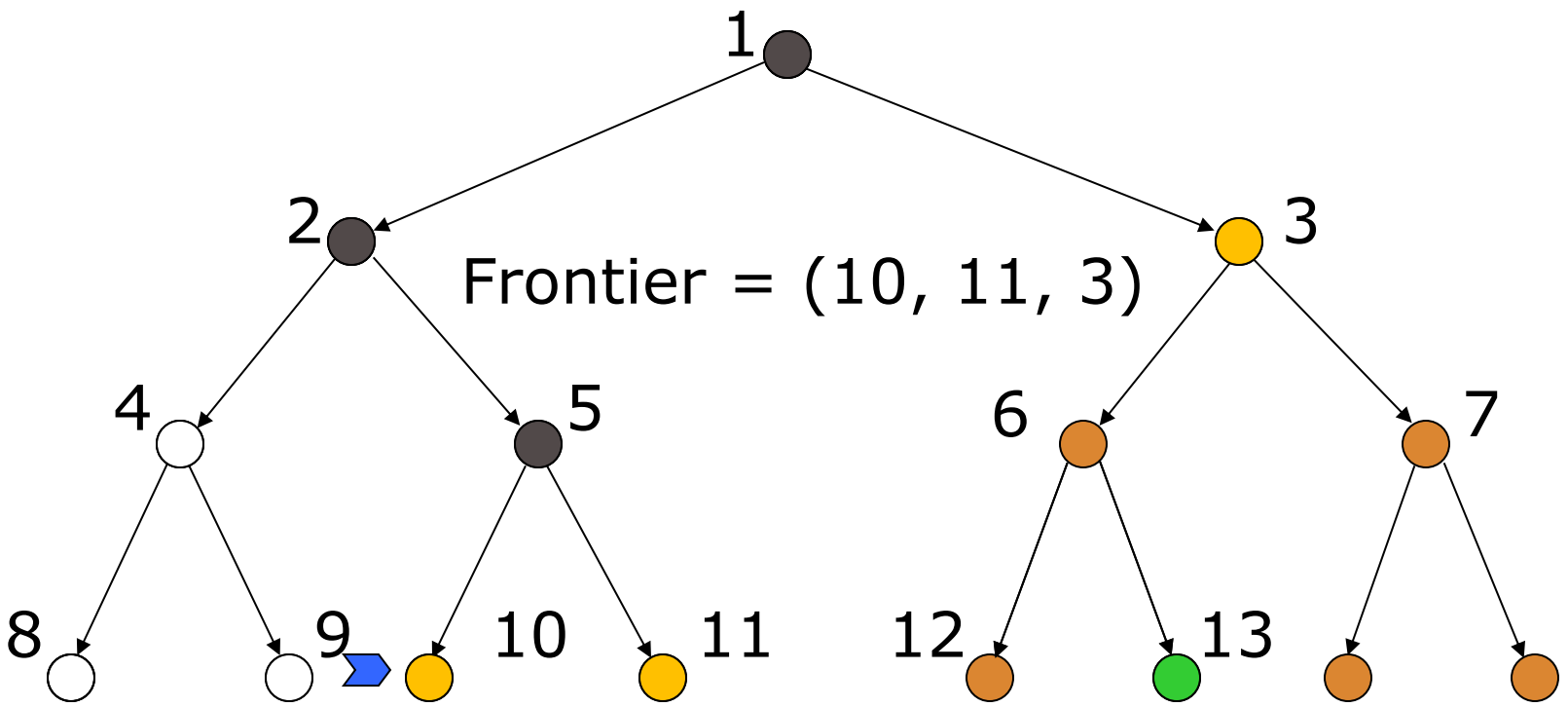
EXPLORED = (1, 2, 4, 8)

DEPTH-FIRST STRATEGY



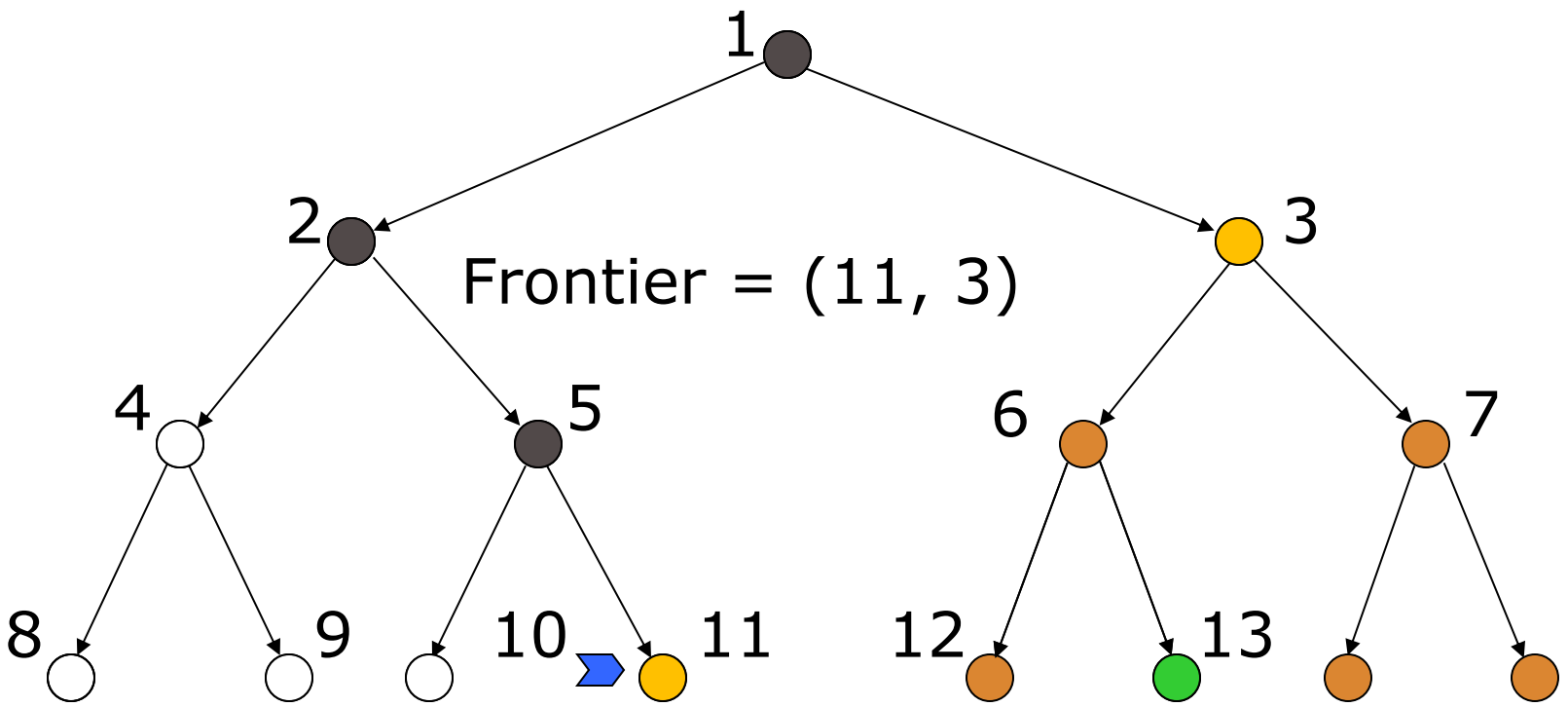
EXPLORED = (1, 2, 4, 8, 9)

DEPTH-FIRST STRATEGY



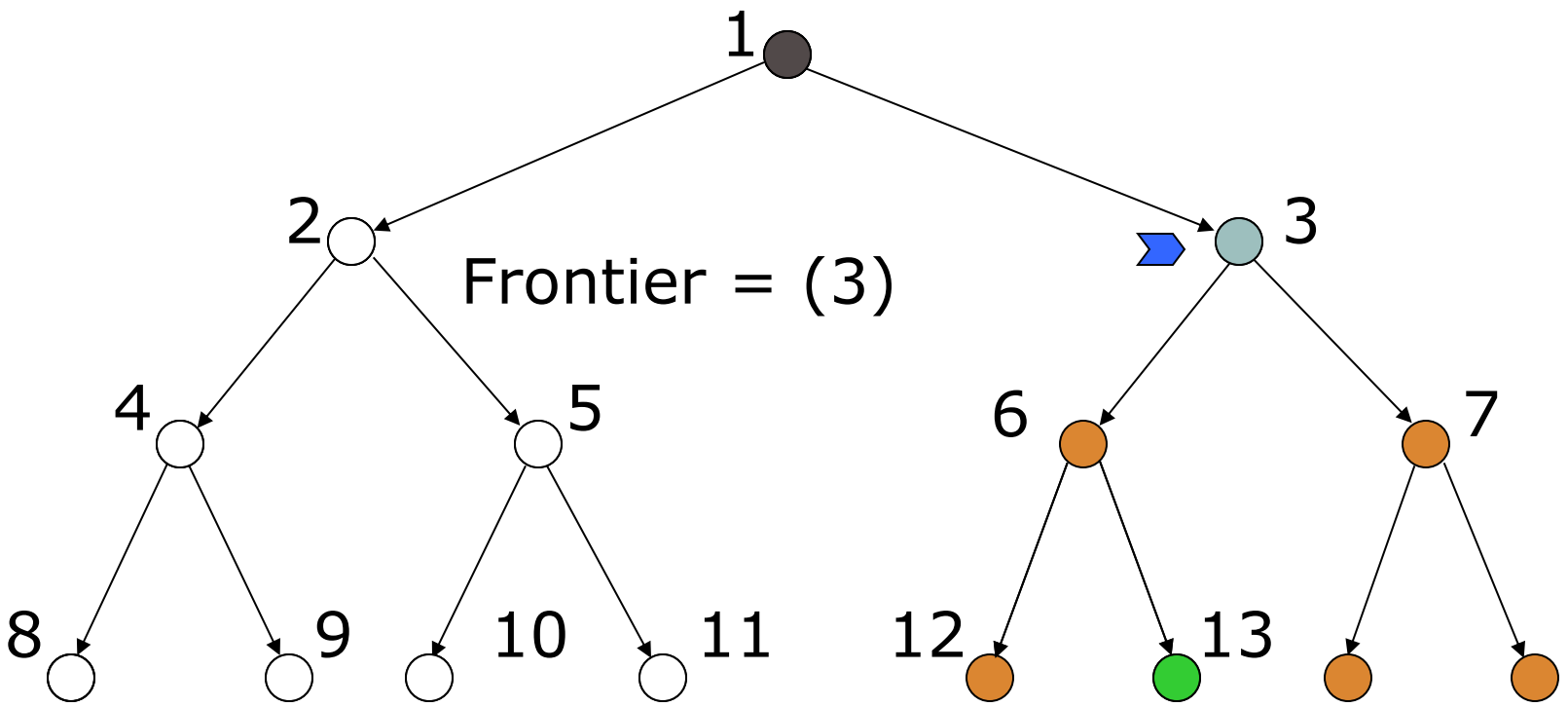
EXPLORED = (1, 2, 4, 8, 9, 5)

DEPTH-FIRST STRATEGY



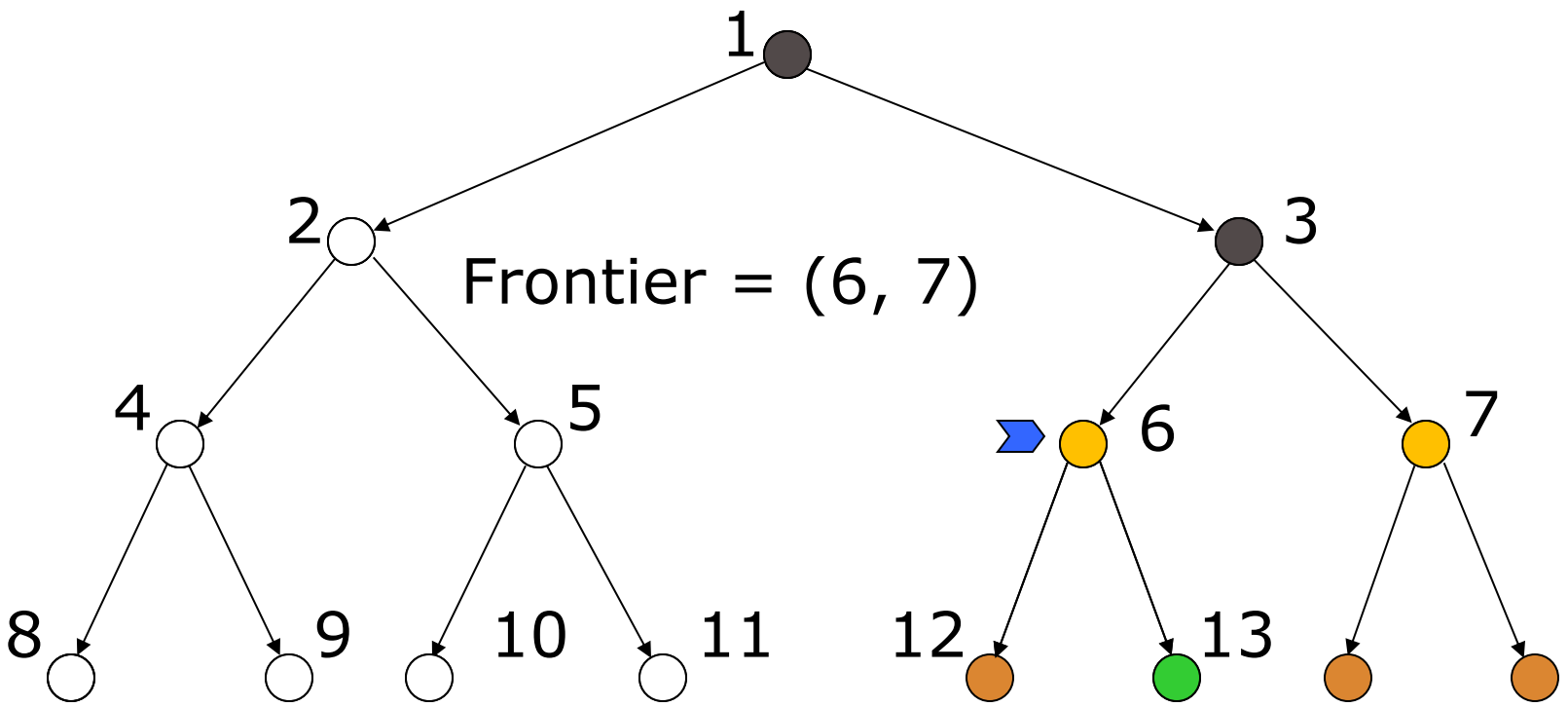
EXPLORED = (1, 2, 4, 8, 9, 5, 10)

DEPTH-FIRST STRATEGY



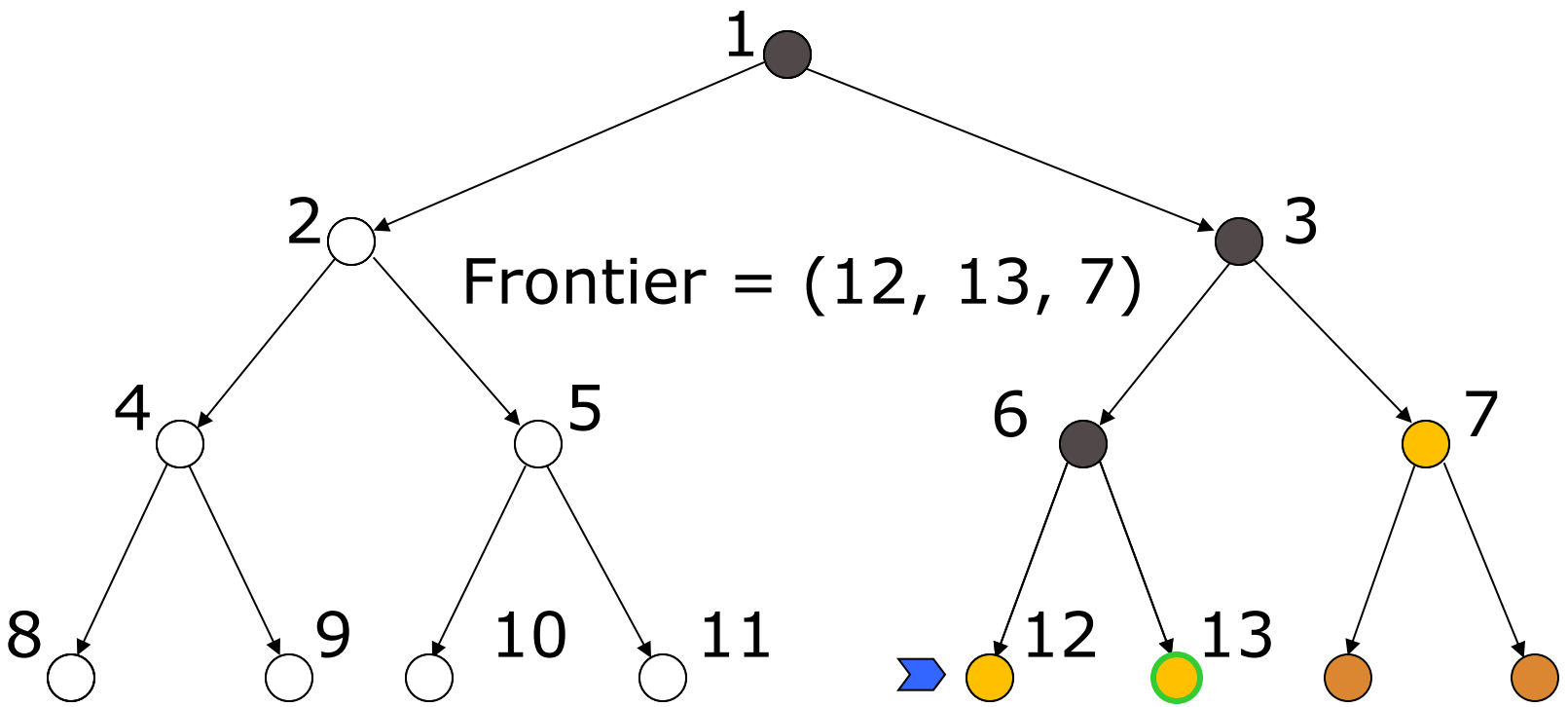
EXPLORED = (1, 2, 4, 8, 9, 5, 10, 11)

DEPTH-FIRST STRATEGY



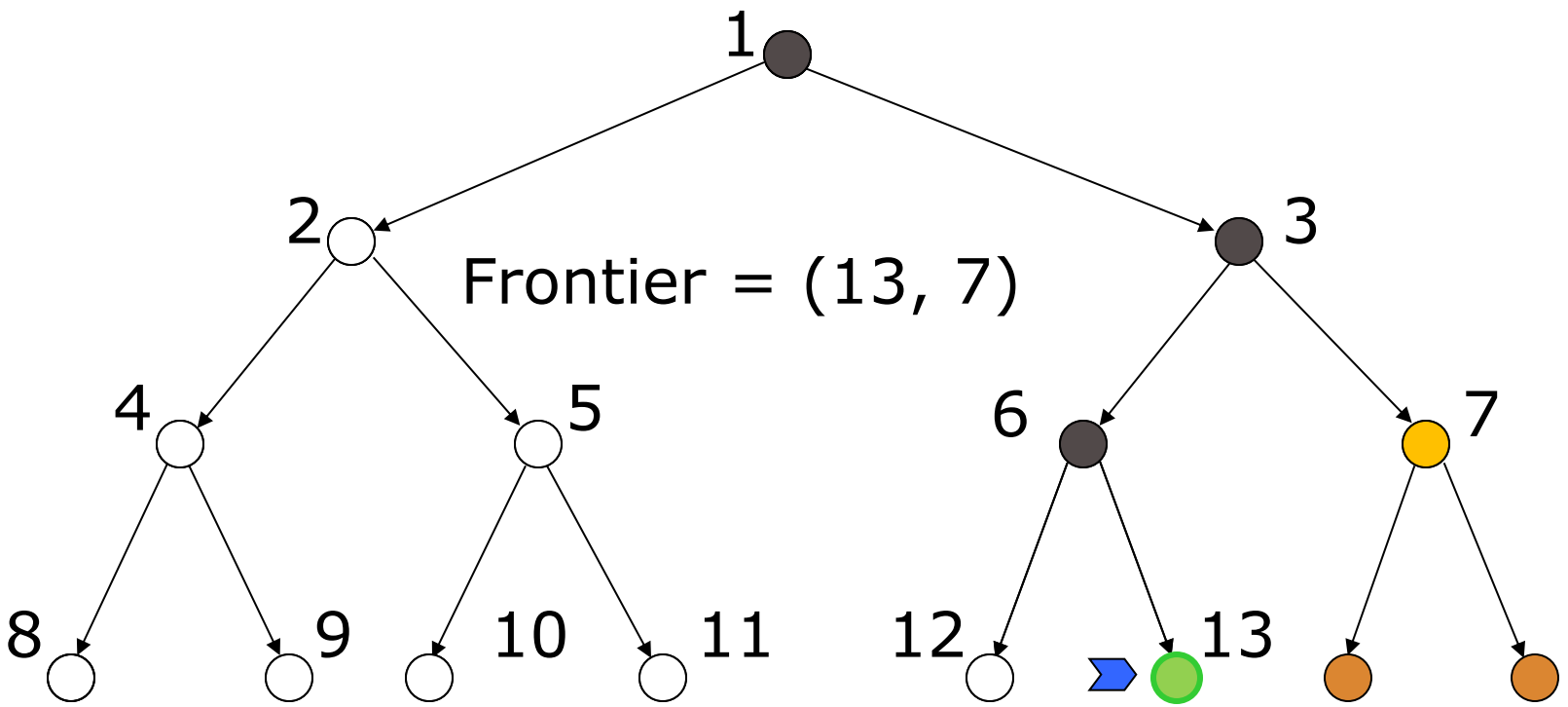
EXPLORED = (1, 2, 4, 8, 9, 5, 10, 11, 3)

DEPTH-FIRST STRATEGY



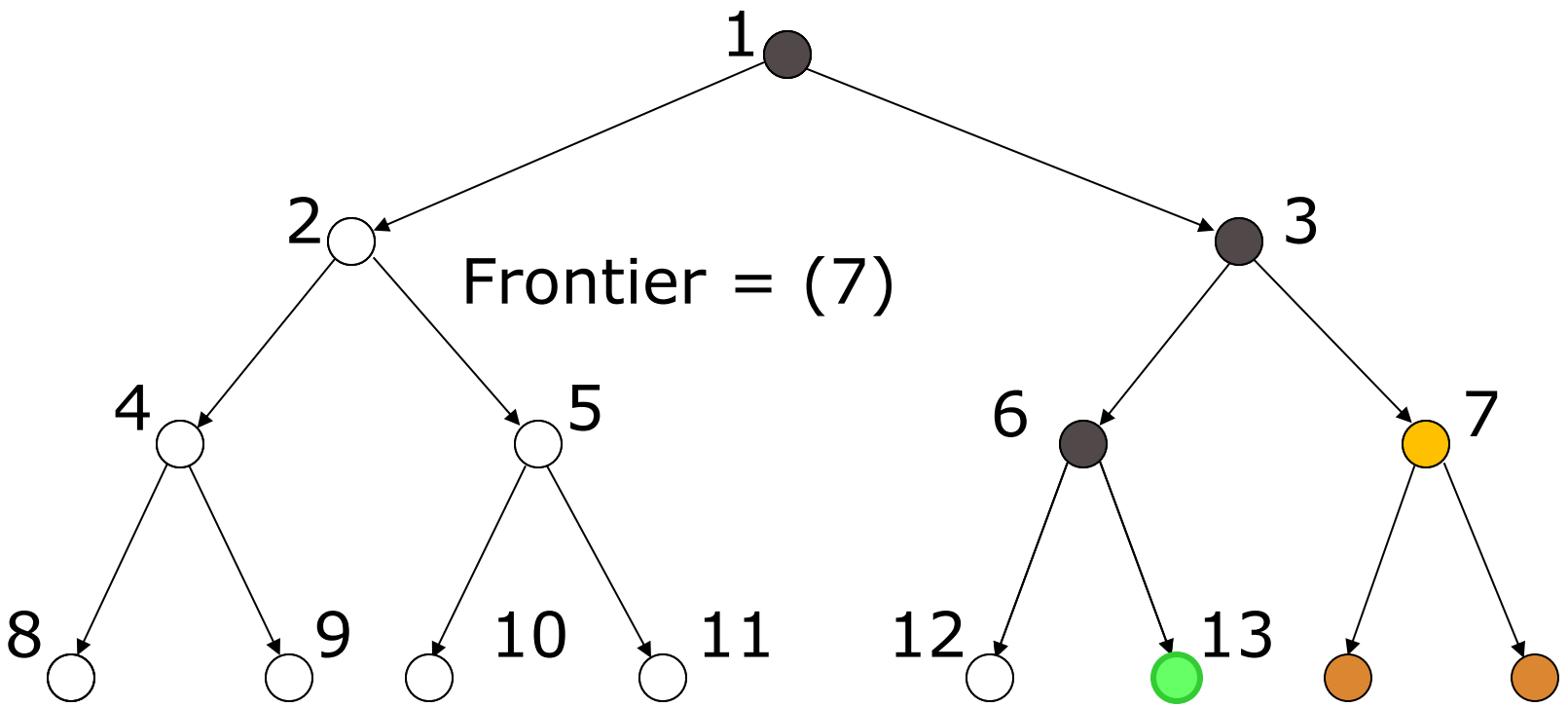
EXPLORED = (1, 2, 4, 8, 9, 5, 10, 11, 3, 6)

DEPTH-FIRST STRATEGY



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

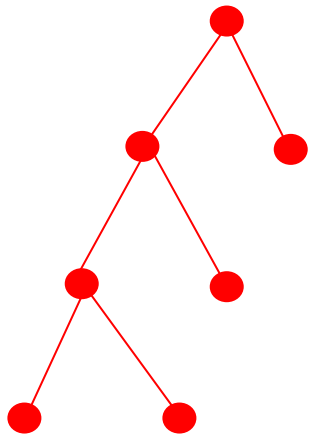
DEPTH-FIRST STRATEGY



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

DEPTH FIRST SEARCH – EVALUATION

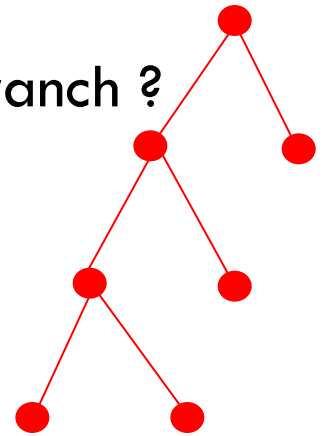
- Space complexity
  - Store: the path from the root to the leaf node as well as the unexpanded neighbor nodes
  - DFS requires storage of **O(bm)** nodes
    - *b*: a branching factor
    - *m*: maximum depth



- Time complexity
  - **$O(b^m)$** in the worst case

DEPTH FIRST SEARCH – EVALUATION

- Completeness
  - An infinite branch: never terminate => ? no goal state exist in that branch ?

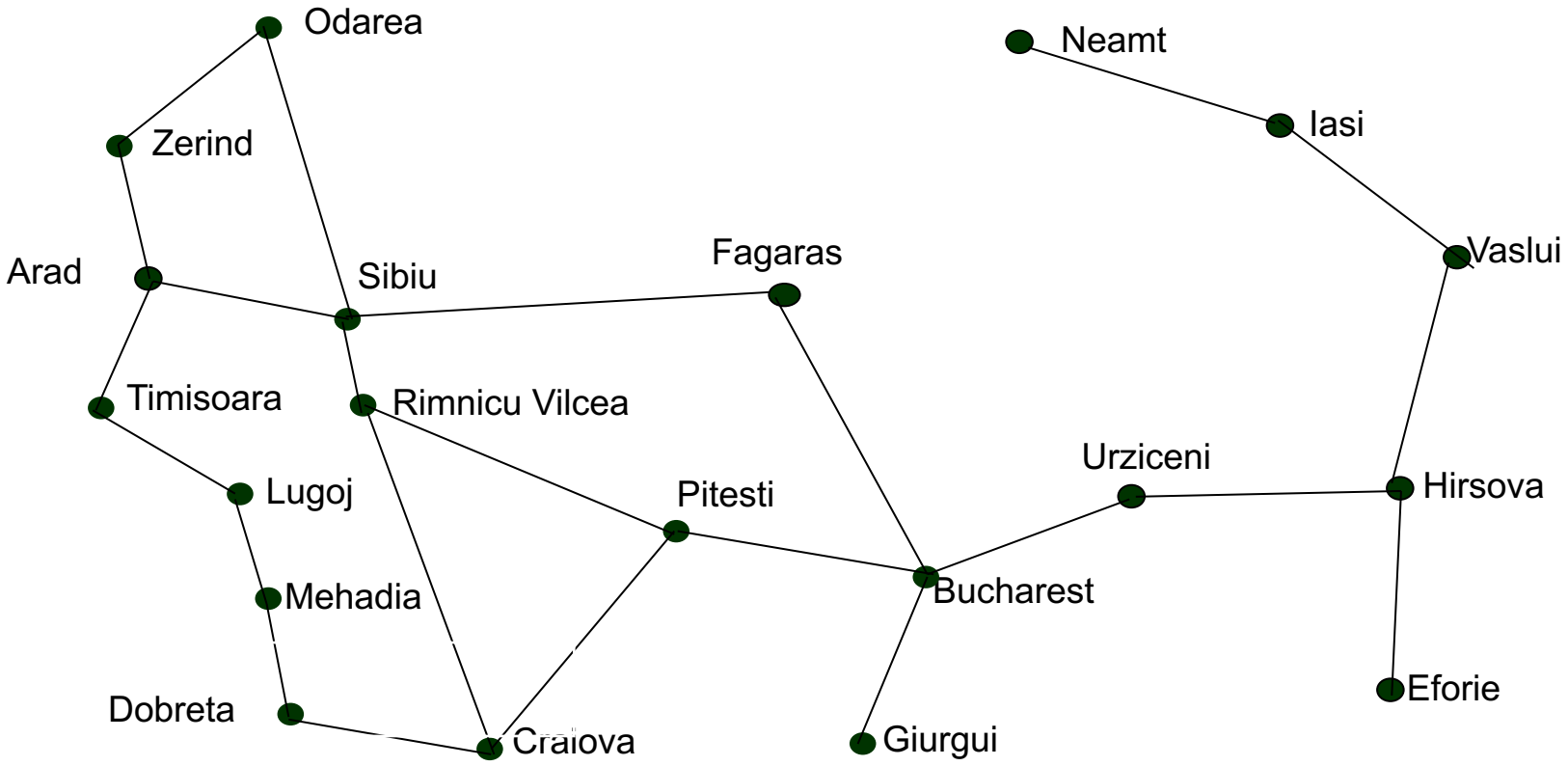


- Optimality
  - Finds a solution: is there a better solution at a lower level?

DFS: neither complete nor optimal

1	4	7
2	5	8
3	6	

MAP OF ROMANIA



UNIFORM COST SEARCH (UCS VS. BFS)

UNIFORM COST SEARCH (VS. BFS)

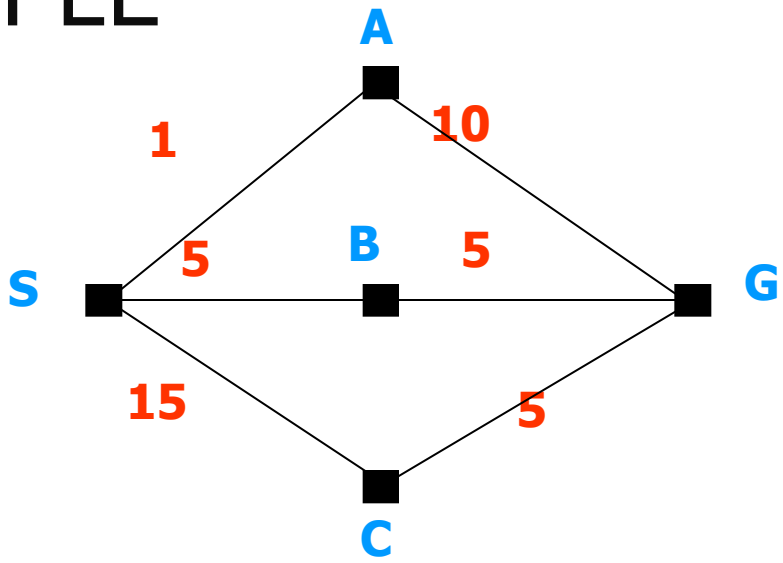
	Optimal condition	Order of nodes explored
BFS	Same cost for all branches	Shallower nodes first
UCS	Cost never decreases	Lowest cost node first

- Cost: total cost of the path from the root to node *n*
- UCS: always remove the **smallest cost node** first
  - Sort the queue in an increasing order, or
  - Search the queue and remove the node of smallest cost

Function UCS(problem) returns a solution or failure

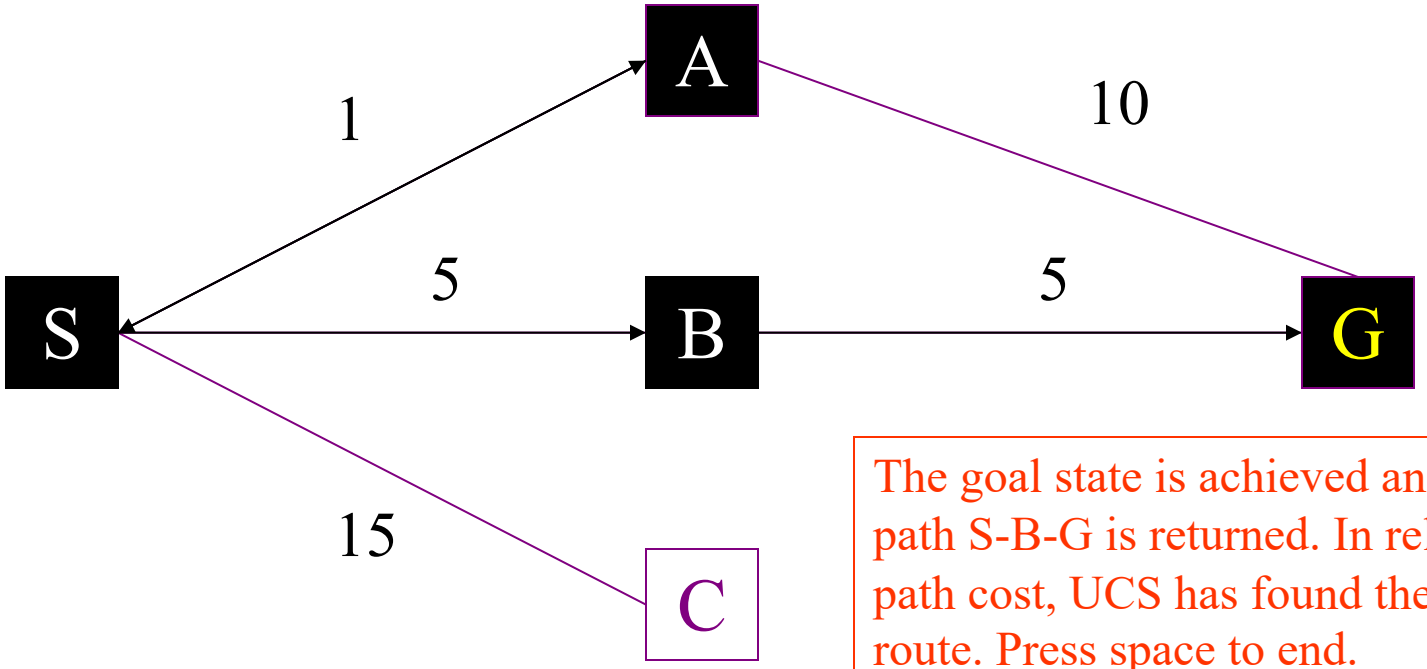
Return General-Search(problem, **ENQUEUE-BY-COST**)

UCS - EXAMPLE



- BFS: finds the path SAG, of a cost of 11
- UCS: finds the cheaper solution (SBG). It will find SAG but will not see it as it is not at the head (cheapest) of the queue

proceed to goal state. Press space.



The goal state is achieved and the path S-B-G is returned. In relation to path cost, UCS has found the optimal route. Press space to end.

Press space to begin the search

Size of Queue: 0	**Queue: Empty**	
Nodes expanded: 3	**FINISHED SEARCH**	**Current level: 2**

UNIFORM COST SEARCH PATTERN

UCS VS. BFS – EVALUATION

	Optimality	Completeness
BFS	Only if the branch costs are the same	Systematically search the whole tree (in the worst case)
UCS	Even branch costs are different	

	Time complexity	Space complexity
BFS & UCS	b^d	b^d

- UCS = BFS
  - When all branches have the same cost
  - We are talking about the worst case scenario
- UCS is usually better than BFS

BLIND SEARCHES – SUMMARY

Evaluation	BFS & UCS	DFS
Time	b^d	b^m
Space	b^d	bm
Optimal?	Yes	No
Complete?	Yes	No

b = Average branching factor
d = Depth of solution
m = Maximum depth of the search tree

BLIND (UNINFORMED) SEARCHES

- Simply searches the State Space
  - No preference as to which node to expand next

- The different blind searches
  - Characterised by **the order** which expands the nodes
    - Different node orderings: **shape** of the frontier
    - Different shapes of the frontier: **very different memory usages**

- An uninformed search
  - Has no knowledge about its domain

BLIND SEARCHES – SUMMARY

- Blind searches (Chapter 4.1-4.3 AIMA (solving problem by searching))
  - Breadth first
  - Depth first
  - Uniform cost search (vs. BFS)
  - Against the four evaluation criteria

- Know how to define a search problem
- Know how to implement UCS (in the lab)