Blind (Uninformed) Search

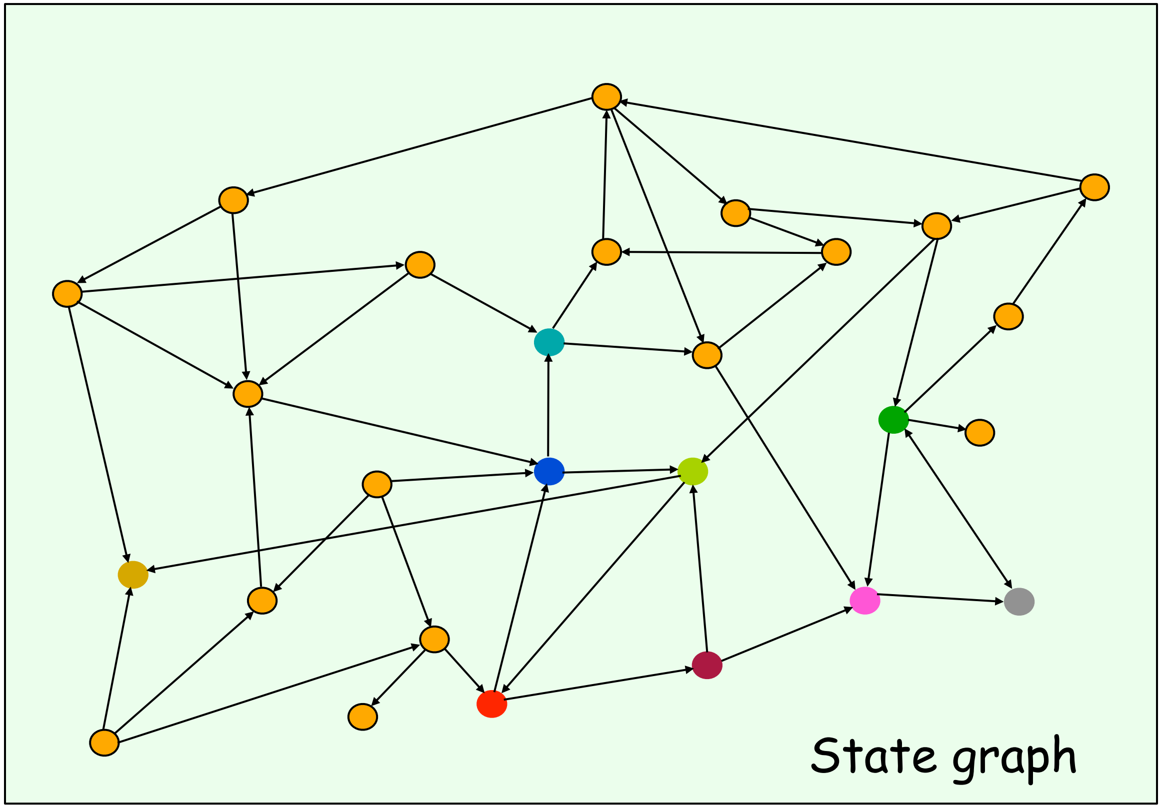
(Where we systematically explore alternatives)

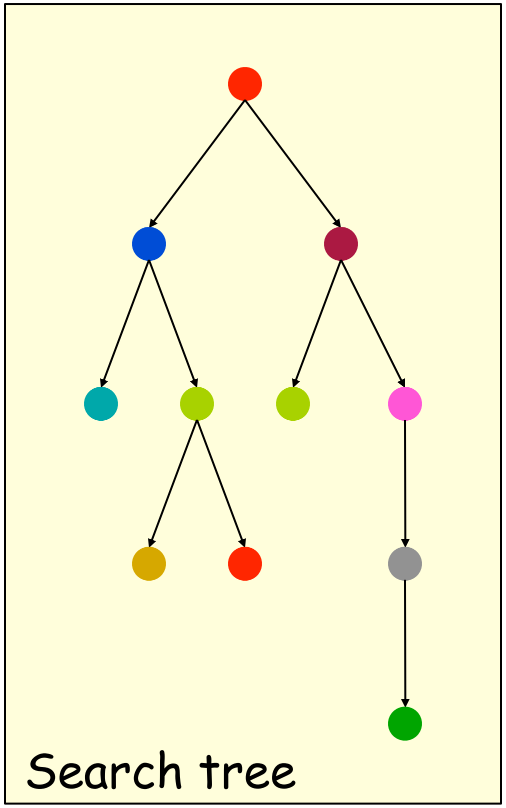
R&N: Chap. 3, Sect. 3.3–5

Simple Problem-Solving-Agent Agent Algorithm

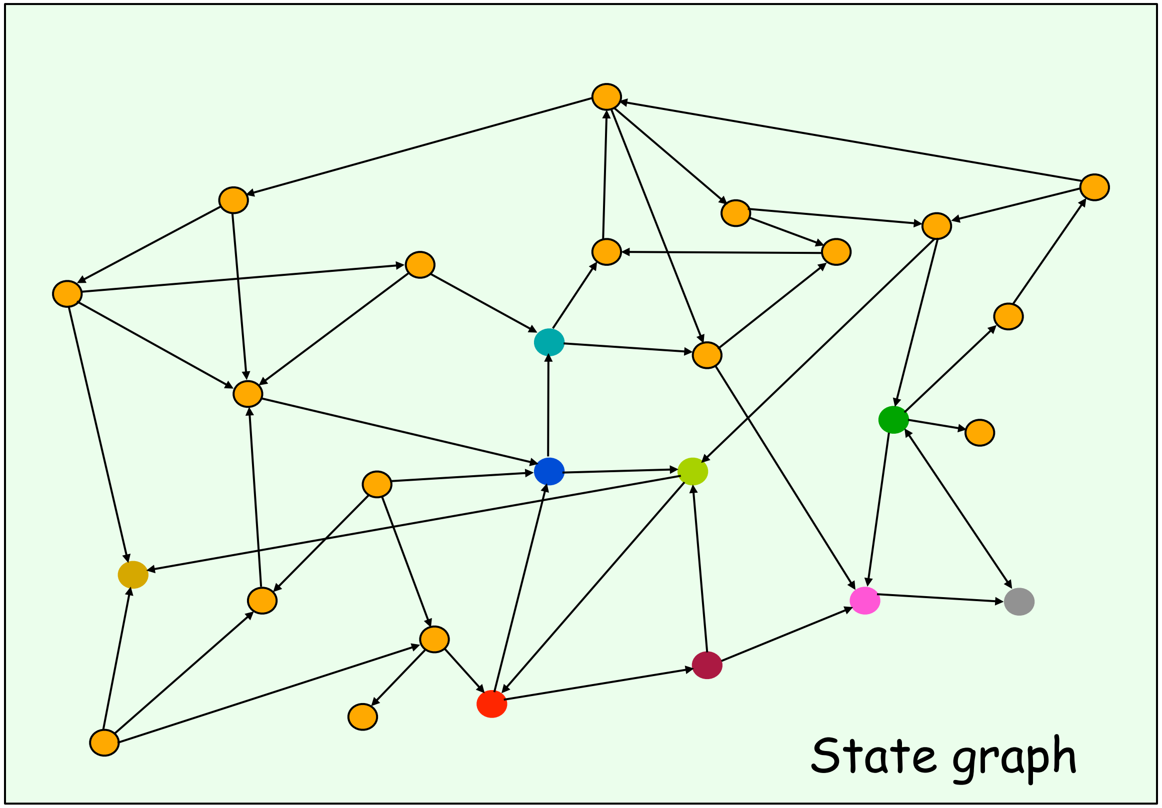
1. s_0 ← sense/read initial state
2. GOAL? ← select/read goal test
3. Succ ← read successor function
4. solution ← **search**(s_0, GOAL?, Succ)
5. perform(solution)

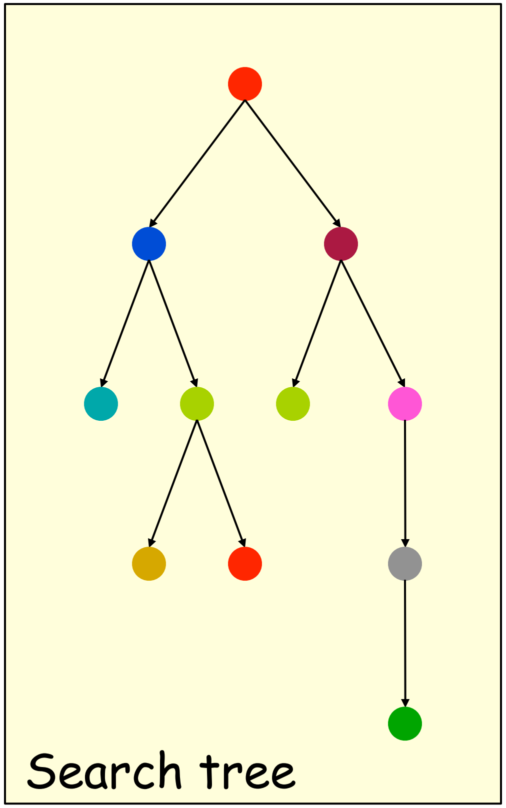
Search Tree





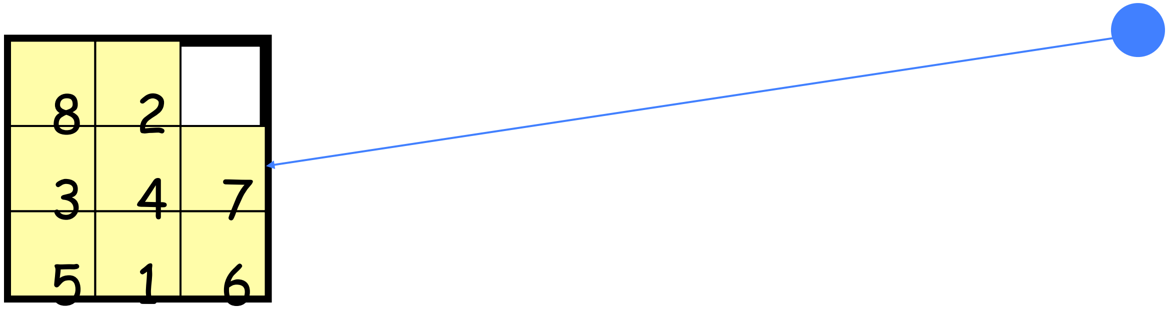
Search Tree



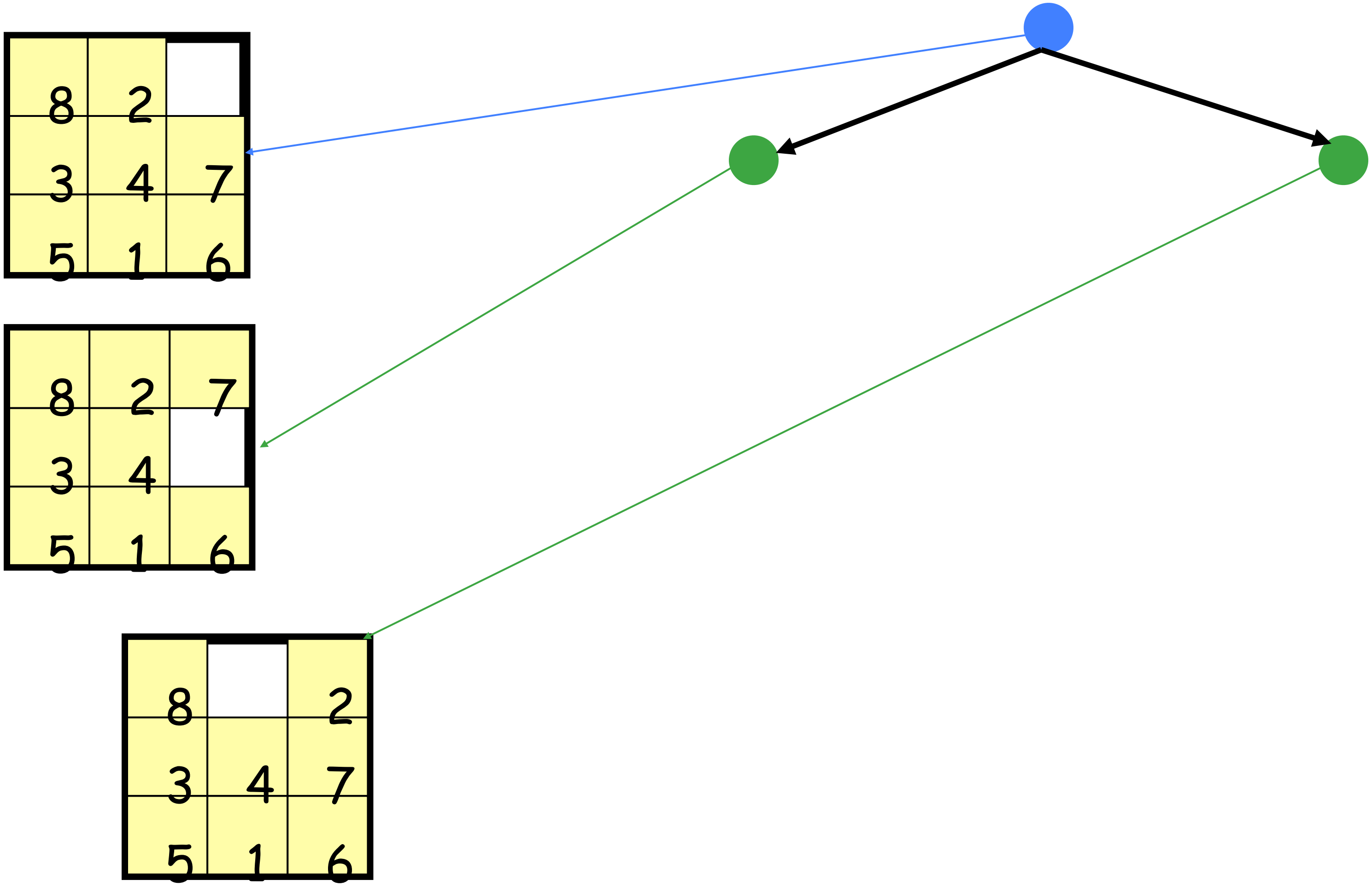


Note that some states may
be visited multiple times

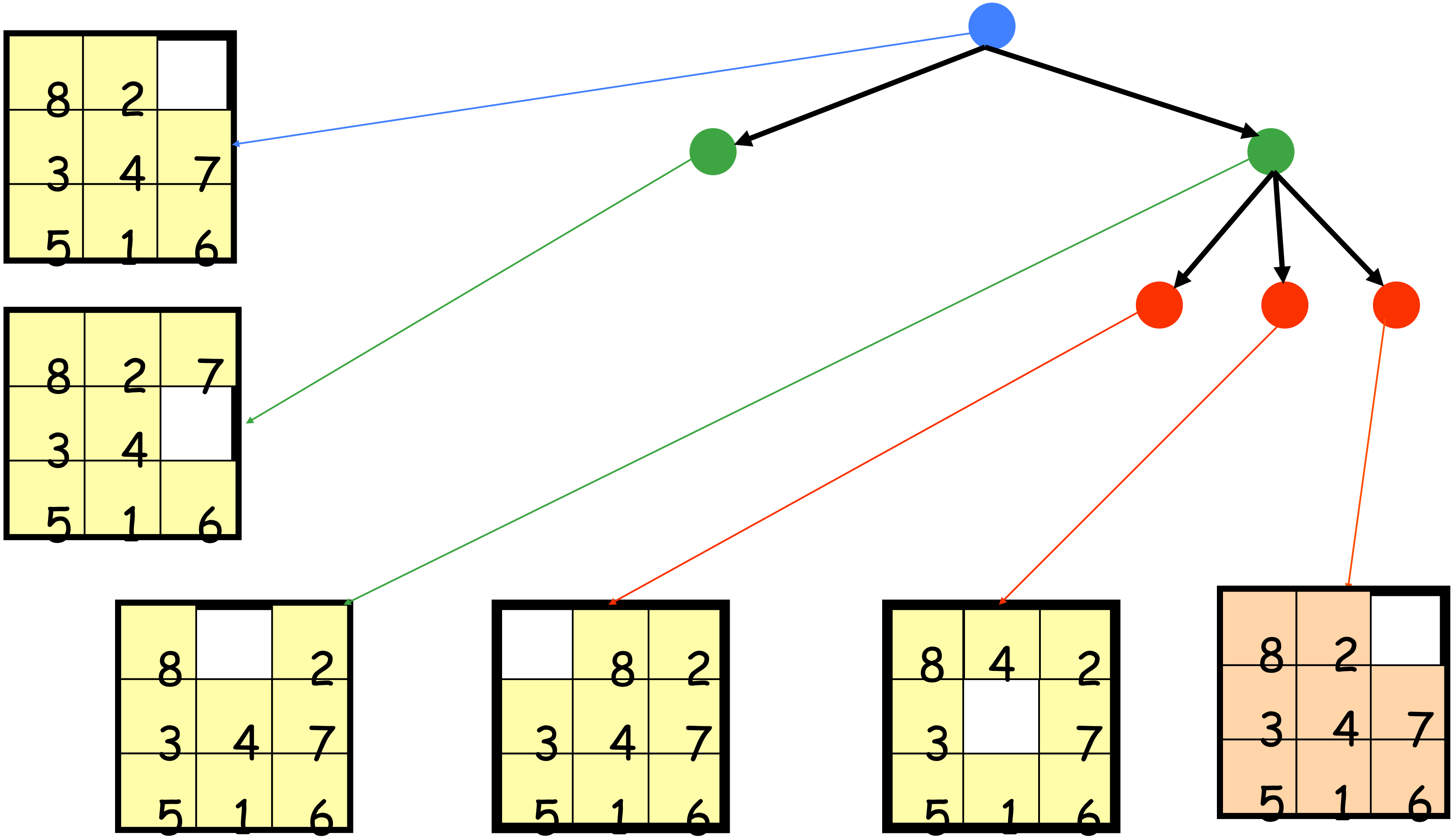
Search Nodes and States



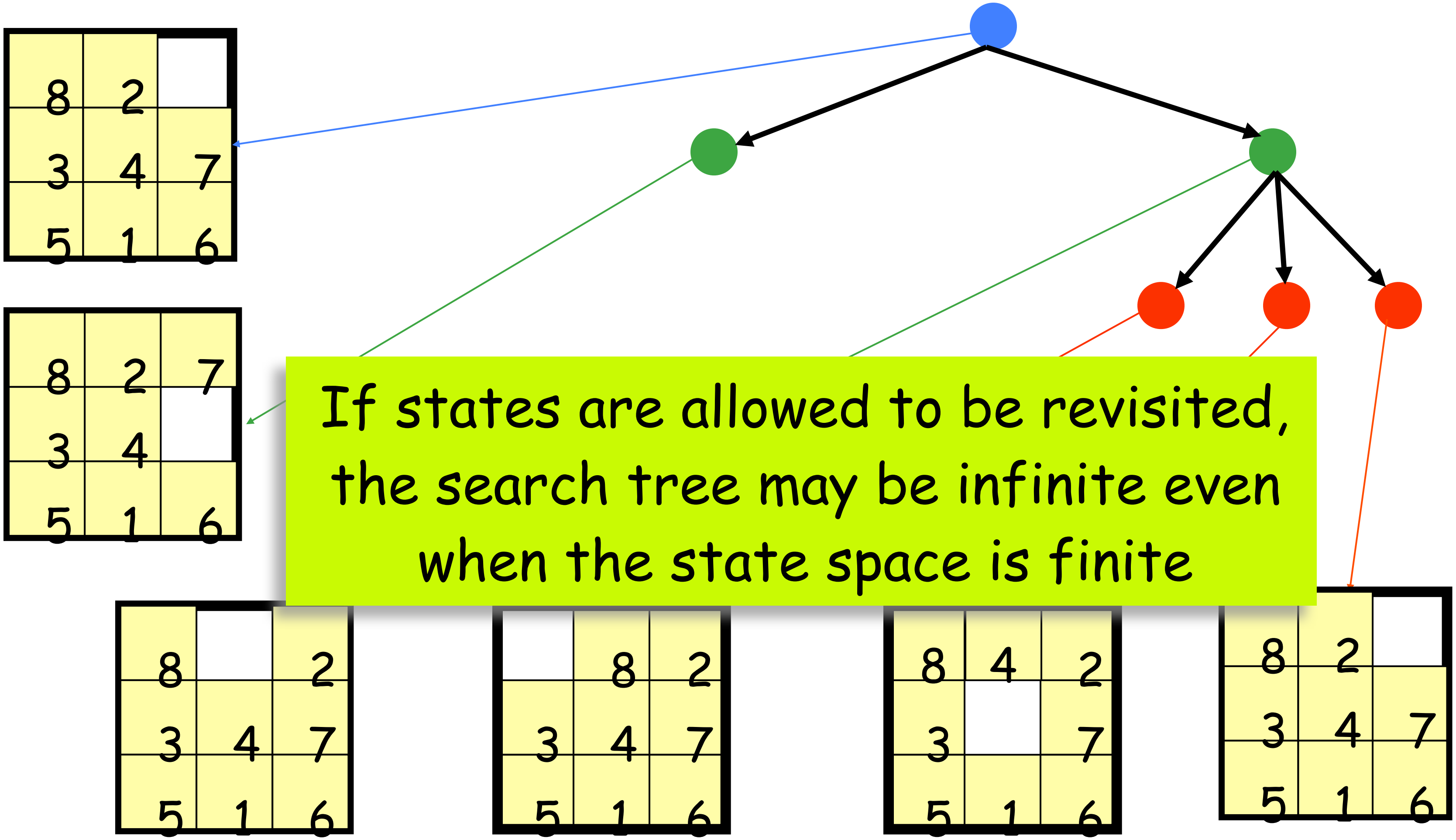
Search Nodes and States



Search Nodes and States

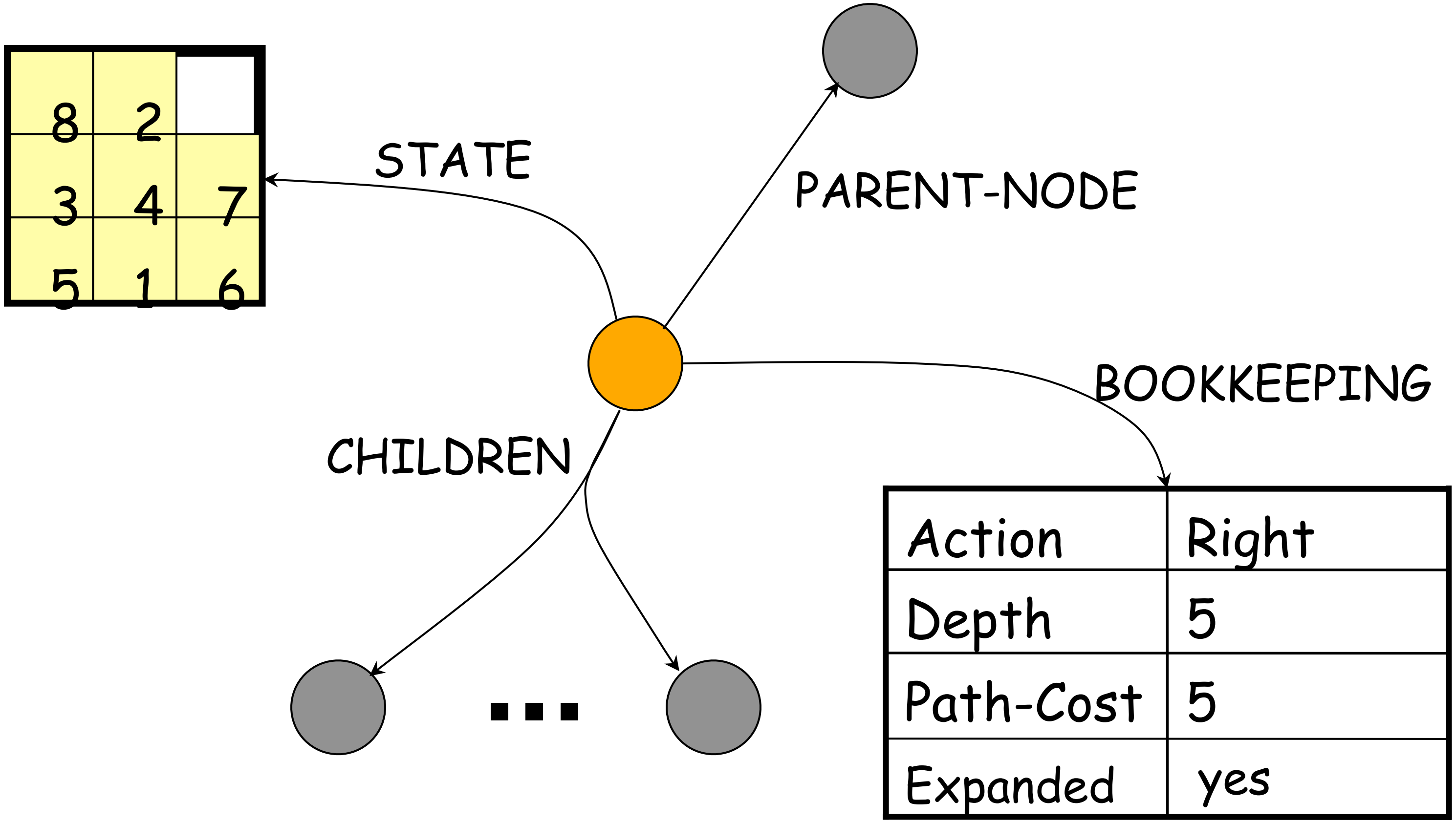


Search Nodes and States



If states are allowed to be revisited, the search tree may be infinite even when the state space is finite

Data Structure of a Node



8	2	
3	4	7
5	1	6

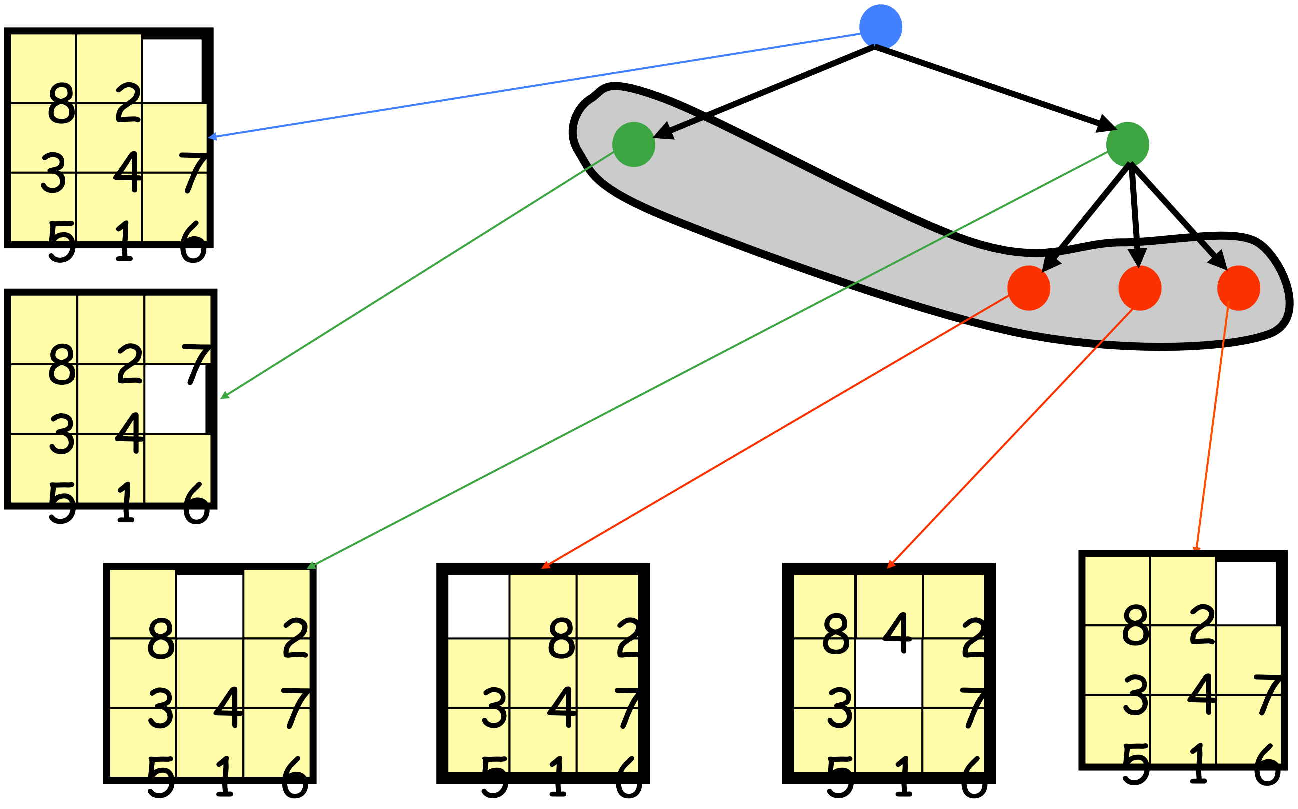
Action	Right
Depth	5
Path-Cost	5
Expanded	yes

Depth of a node N
= length of path from root to N

(depth of the root = 0)

Open List (OL) of Search Tree

- The OL is the set of all search nodes that haven’t been expanded yet



Search Strategy

- The OL is the set of all search nodes that haven't been expanded yet
- The OL is implemented as a priority queue
  - INSERT(node, OL)
  - REMOVE(OL)
- The ordering of the nodes in OL defines the search strategy

Search Algorithm #1

```
SEARCH#1
1. If GOAL?(initial-state) then return initial-state
2. INSERT(initial-node,OL)
3. Repeat:
    a. If empty(OL) then return failure
    b. N ← REMOVE(OL)
    c. s ← STATE(N)
    d. For every state s’ in SUCCESSORS(s)
        i.   Create a new node N’ as a child of N
        ii.  If GOAL?(s’) then return path or goal state
        iii. INSERT(N’,OL)
```

Performance Measures

- Completeness

  A search algorithm is complete if it finds a solution whenever one exists

  [What about the case when no solution exists?]
- Optimality

  A search algorithm is optimal if it returns a minimum-cost path whenever a solution exists
- Complexity

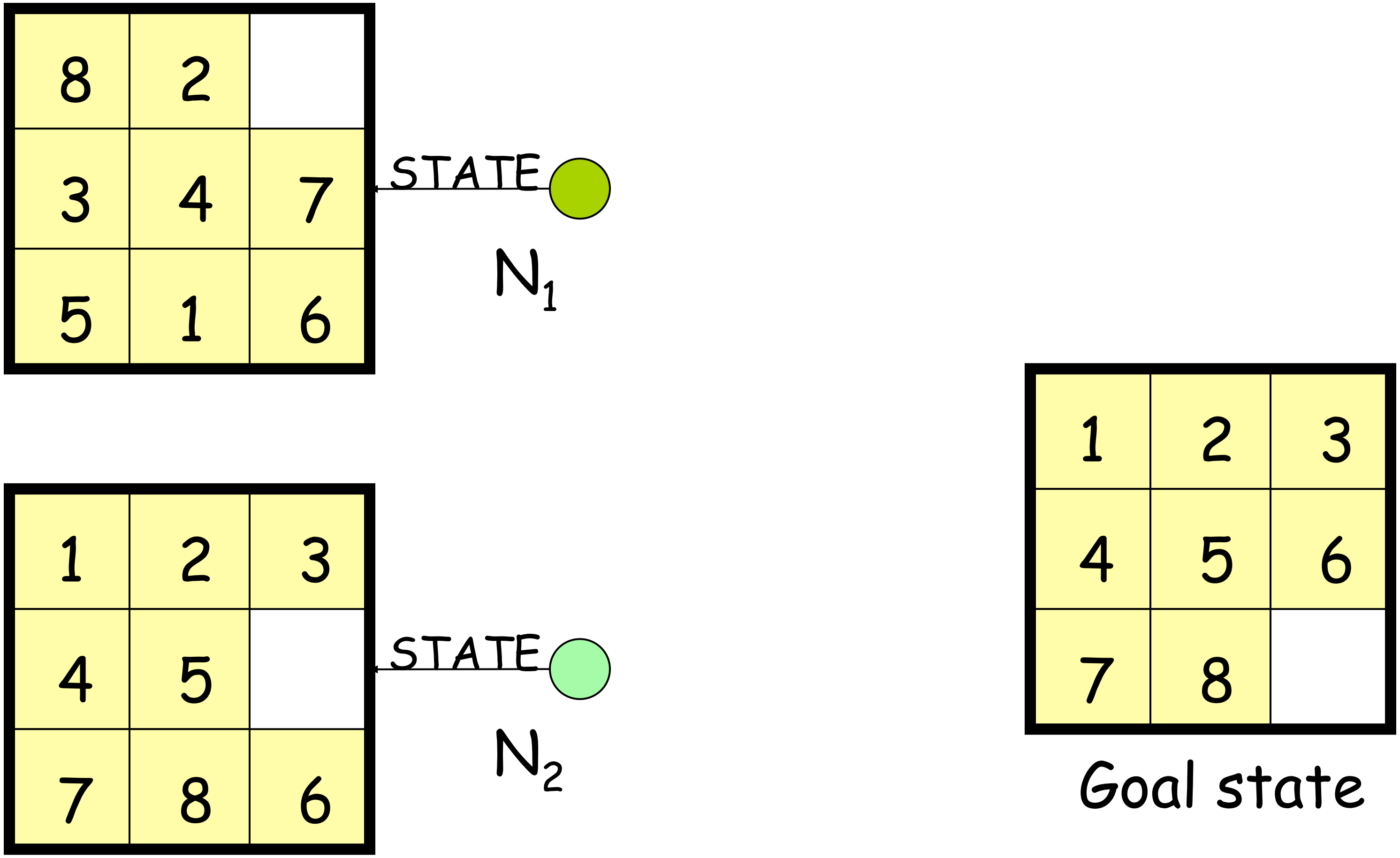
  It measures the time and amount of memory required by the algorithm

Blind vs. Heuristic Strategies

- Blind (or un-informed) strategies do not exploit state descriptions to order OL. They only exploit the positions of the nodes in the search tree

- Heuristic (or informed) strategies exploit state descriptions to order OL (the most "promising" nodes are placed at the beginning of OL)

Example



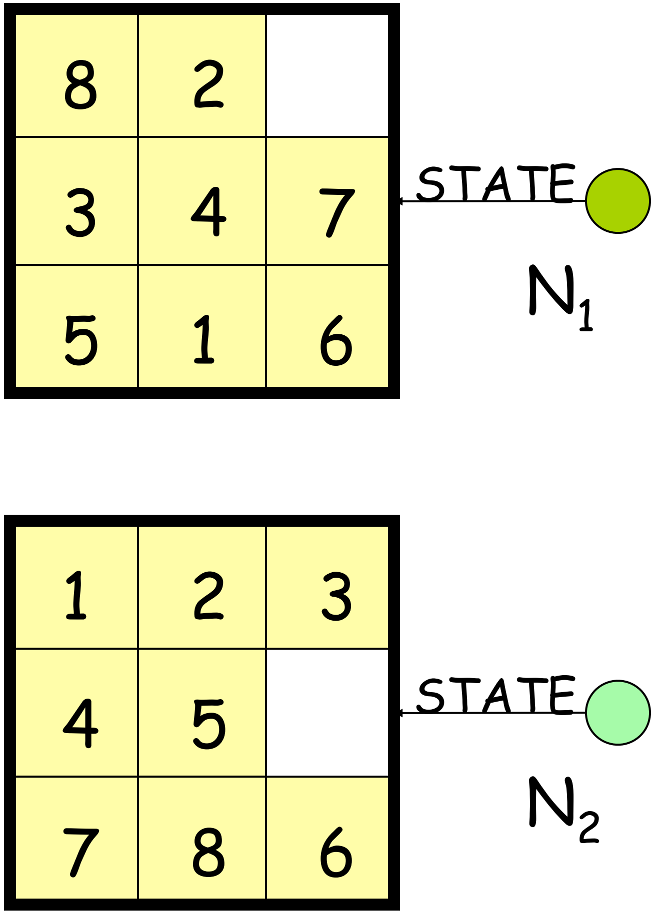
8	2	
3	4	7
5	1	6

1	2	3
4	5	
7	8	6

1	2	3
4	5	6
7	8	

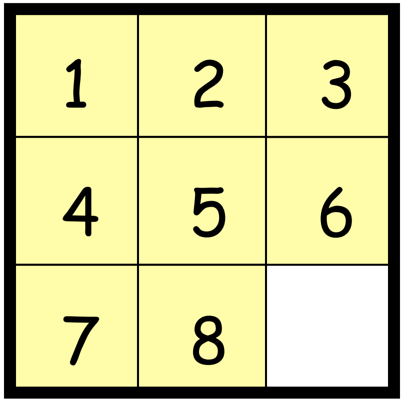
Example

For a blind strategy, N_1 and N_2 are just two nodes (at some position in the search tree)



8	2	
3	4	7
5	1	6

1	2	3
4	5	
7	8	6

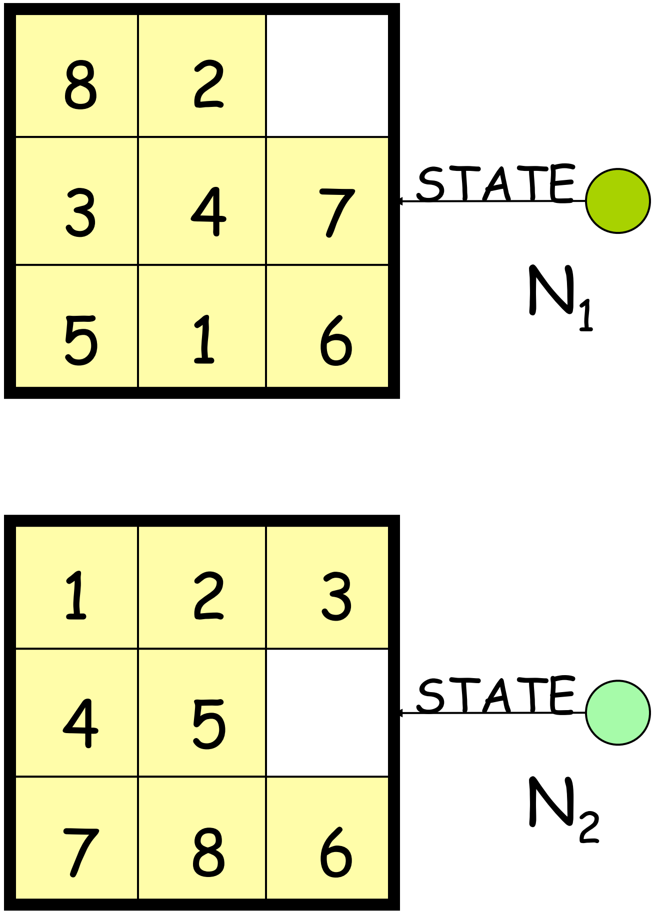


1	2	3
4	5	6
7	8	

Goal state

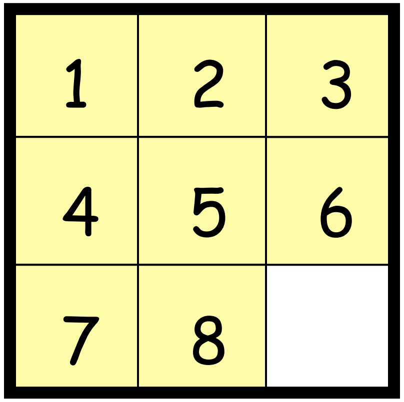
Example

For a heuristic strategy counting the number of misplaced tiles, N_2 is more promising than N_1



8	2	
3	4	7
5	1	6

1	2	3
4	5	
7	8	6



1	2	3
4	5	6
7	8	

Goal state

Remark

- Some search problems, such as the (n^2-1)-puzzle, are NP-hard
- One can't expect to solve all instances of such problems in less than exponential time (in n)
- One may still strive to solve each instance as efficiently as possible

This is the purpose of the search strategy

Blind Strategies

Blind Strategies

Arc cost = 1

Blind Strategies

- Breadth-first
  - Bidirectional

} Arc cost = 1

Blind Strategies

- Breadth-first
  - Bidirectional
- Depth-first
  - Depth-limited
  - Iterative deepening

Arc cost = 1

Blind Strategies

- Breadth-first
  - Bidirectional
- Depth-first
  - Depth-limited
  - Iterative deepening

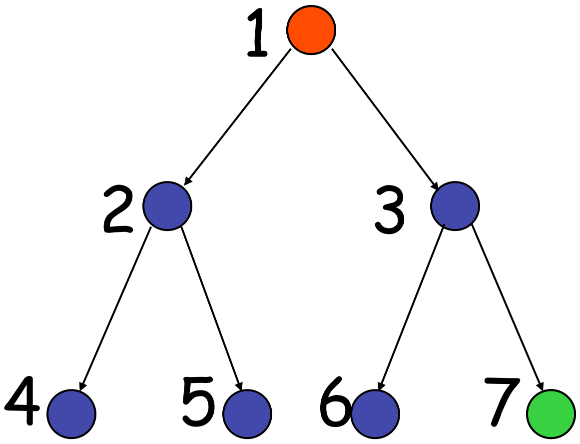
Arc cost = 1

- Uniform-Cost
  (variant of breadth-first)

Arc cost = c(action) $\geq \varepsilon > 0$

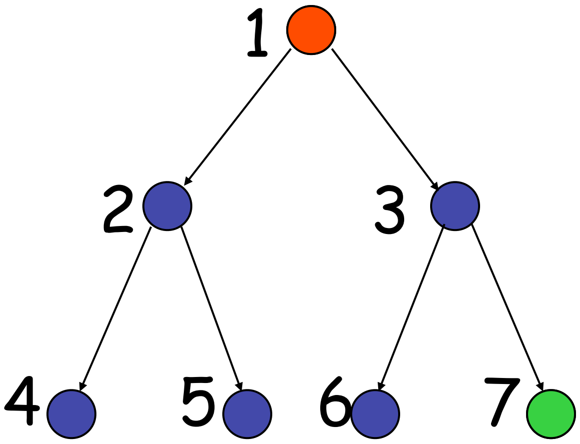
Breadth-First Strategy

New nodes are inserted at the end of OL



Breadth-First Strategy

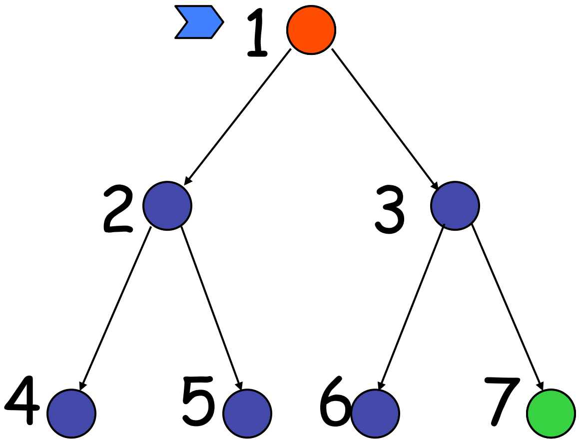
New nodes are inserted at the end of OL



OL = (1)

Breadth-First Strategy

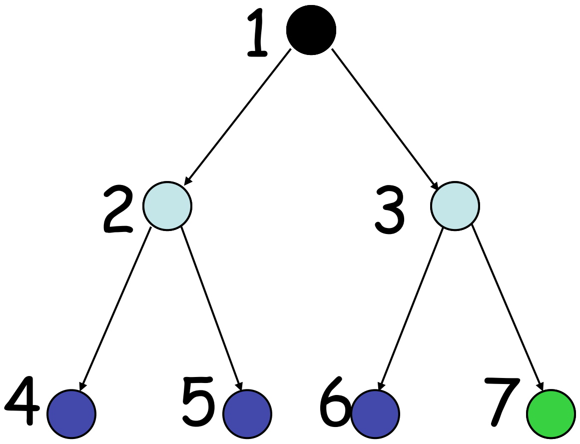
New nodes are inserted at the end of OL



OL = (1)

Breadth-First Strategy

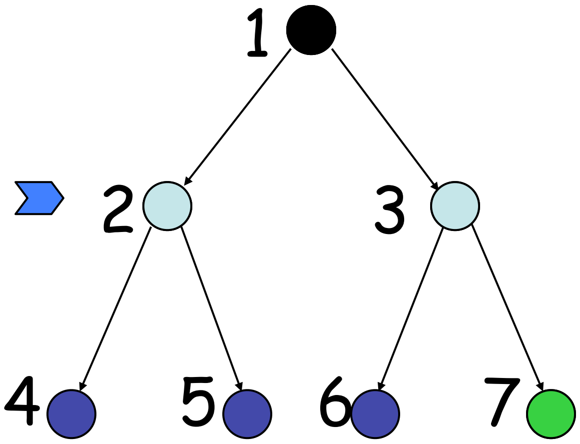
New nodes are inserted at the end of OL



OL = (2, 3)

Breadth-First Strategy

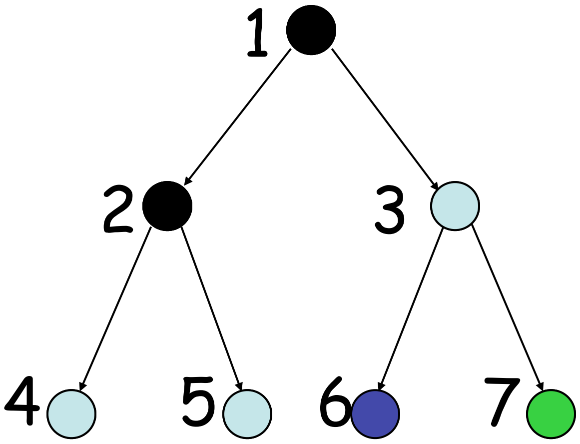
New nodes are inserted at the end of OL



OL = (2, 3)

Breadth-First Strategy

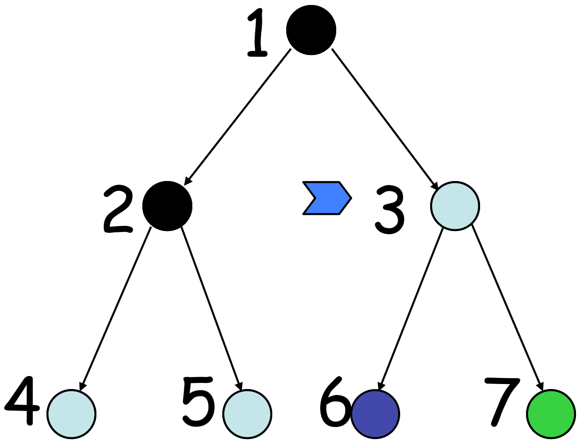
New nodes are inserted at the end of OL



OL = (3, 4, 5)

Breadth-First Strategy

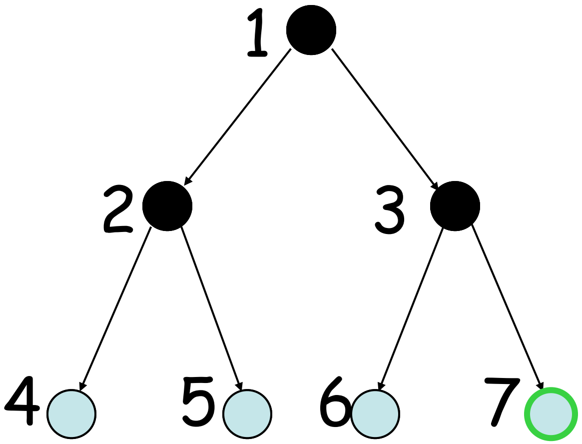
New nodes are inserted at the end of OL



OL = (3, 4, 5)

Breadth-First Strategy

New nodes are inserted at the end of OL



OL = (4, 5, 6, 7)

Important Parameters

1) Maximum number of successors of any state

→ branching factor b of the search tree

2) Minimal length (≠ cost) of a path between the initial and a goal state

→ depth d of the shallowest goal node in the search tree

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- Breadth-first search is:
  - Complete? Not complete?
  - Optimal? Not optimal?

Evaluation

Evaluation

- b: branching factor

Evaluation

- b: branching factor
- d: depth of shallowest goal node

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- Breadth-first search is:
  - Complete
  - Optimal if step cost is 1

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- Breadth-first search is:
  - Complete
  - Optimal if step cost is 1
- Number of nodes generated:

  ???

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- Breadth-first search is:
  - Complete
  - Optimal if step cost is 1
- Number of nodes generated:
  $1 + b + b^2 + \ldots + b^d \quad = \quad ???$

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- Breadth-first search is:
  - Complete
  - Optimal if step cost is 1
- Number of nodes generated:
  $1 + b + b^2 + \ldots + b^d \;=\; (b^{d+1}-1)/(b-1) \;=\; O(b^d)$
- → Time and space complexity is $O(b^d)$

Big O Notation

$g(n) = O(f(n))$ if there exist two positive constants a and N such that:

for all $n > N$: $g(n) \leq af(n)$

Time and Memory Requirements

d	# Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	~10^6	1 sec	100 Mb
8	~10^8	100 sec	10 Gbytes
10	~10^{10}	2.8 hours	1 Tbyte
12	~10^{12}	11.6 days	100 Tbytes
14	~10^{14}	3.2 years	10,000 Tbytes

Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

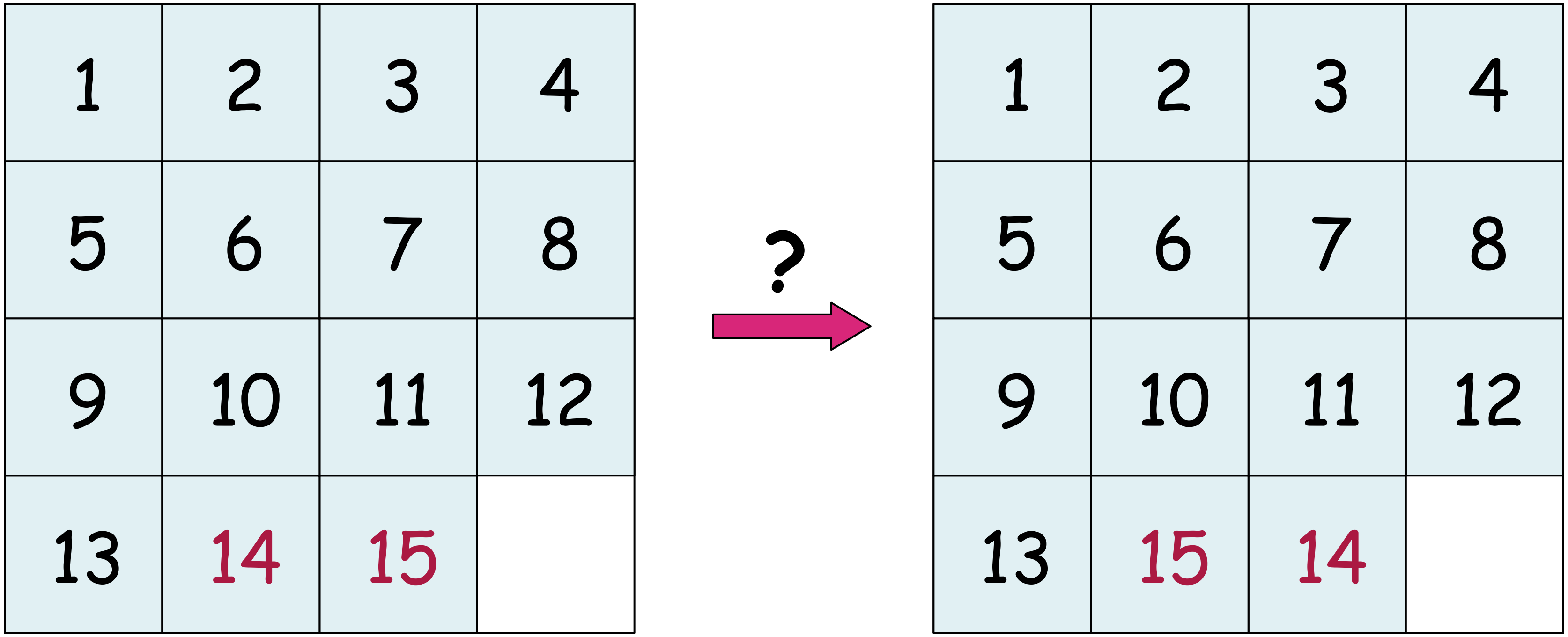
Time and Memory Requirements

d	# Nodes	Time	Memory
2	111	.01 msec	11 Kbytes
4	11,111	1 msec	1 Mbyte
6	$\sim 10^6$	1 sec	100 Mb
8	$\sim 10^8$	100 sec	10 Gbytes
10	$\sim 10^{10}$	2.8 hours	1 Tbyte
12	$\sim 10^{12}$	11.6 days	100 Tbytes
14	$\sim 10^{14}$	3.2 years	10,000 Tbytes

Assumptions: b = 10; 1,000,000 nodes/sec; 100bytes/node

Remark

If a problem has no solution, breadth-first may run for ever (if the state space is infinite or states can be revisited arbitrary many times)



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Bidirectional Strategy

Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2

Bidirectional Strategy

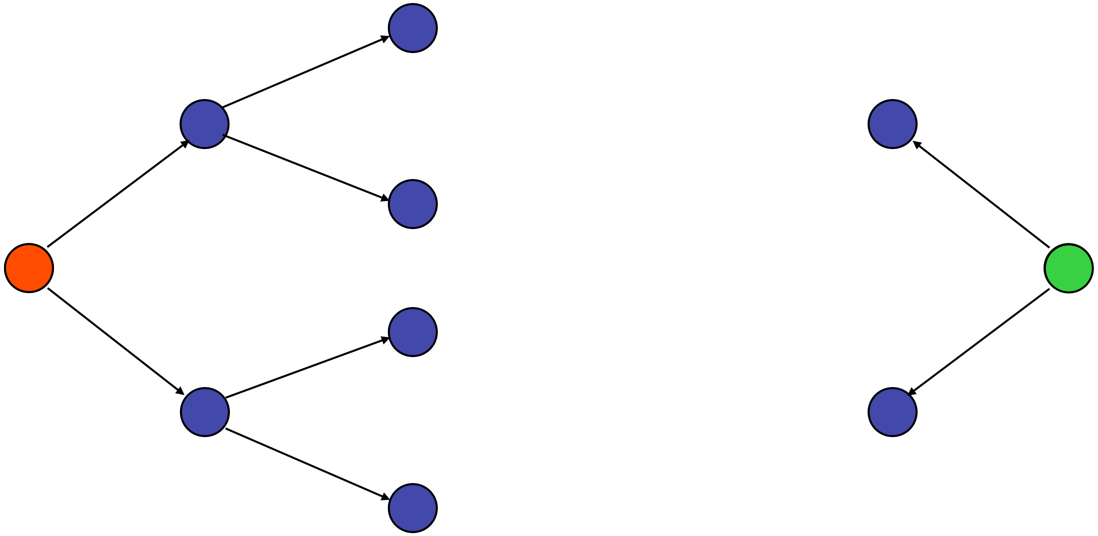
2 fringe queues: FRINGE1 and FRINGE2

Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2

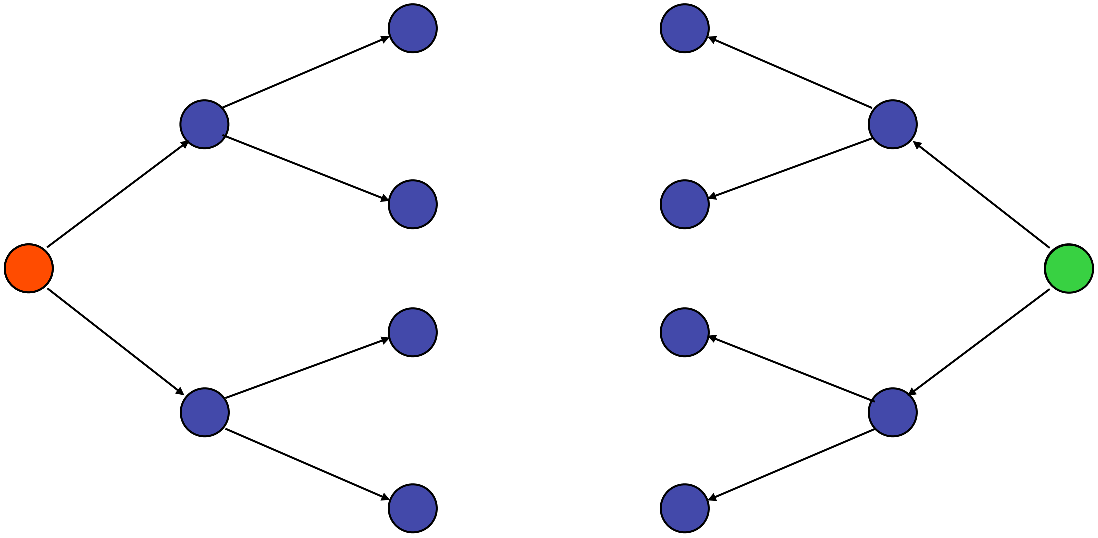
Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2



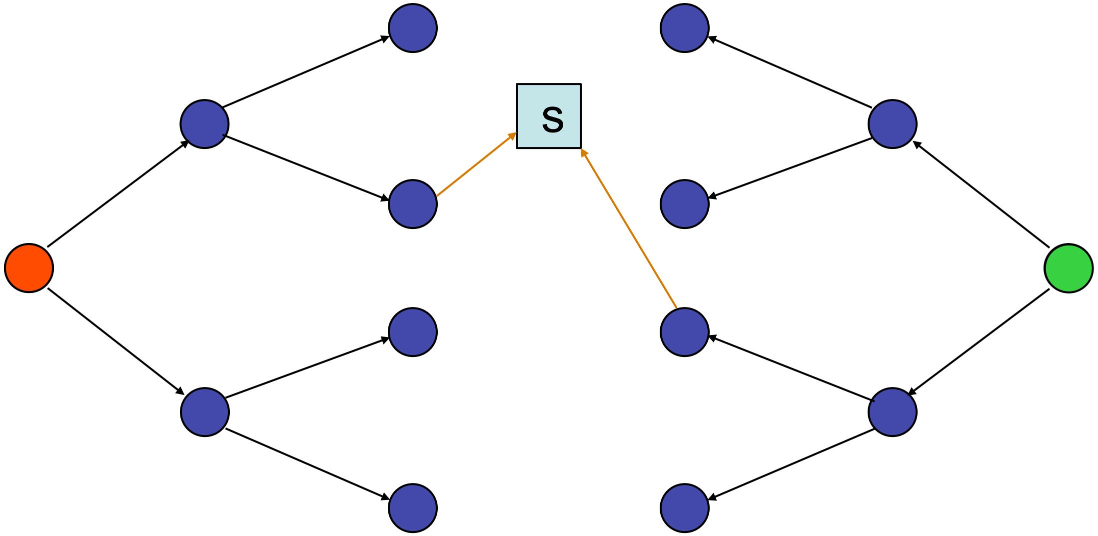
Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2



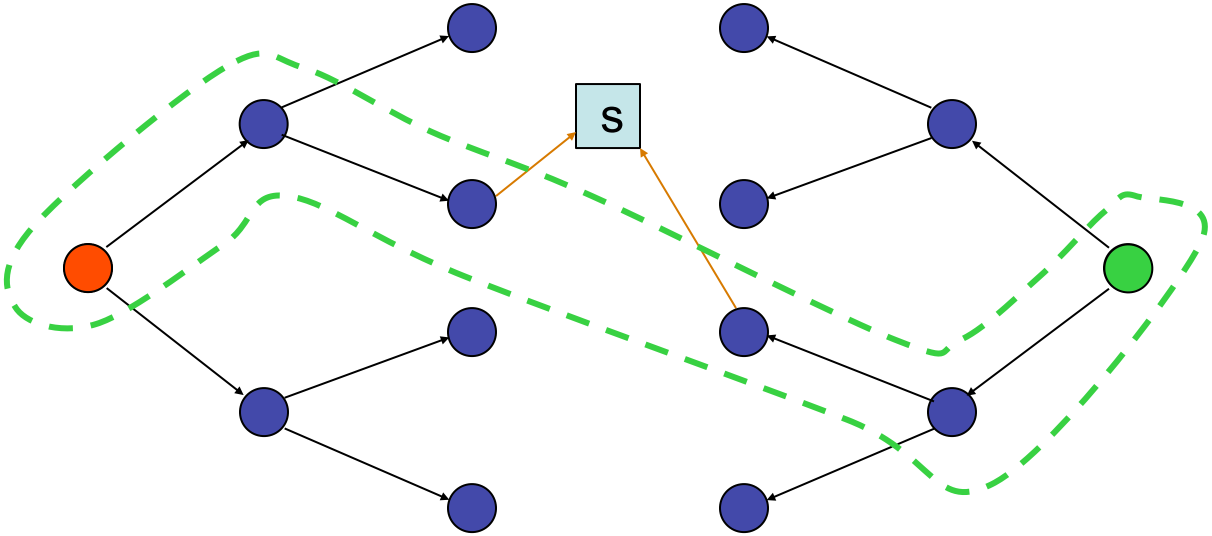
Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2



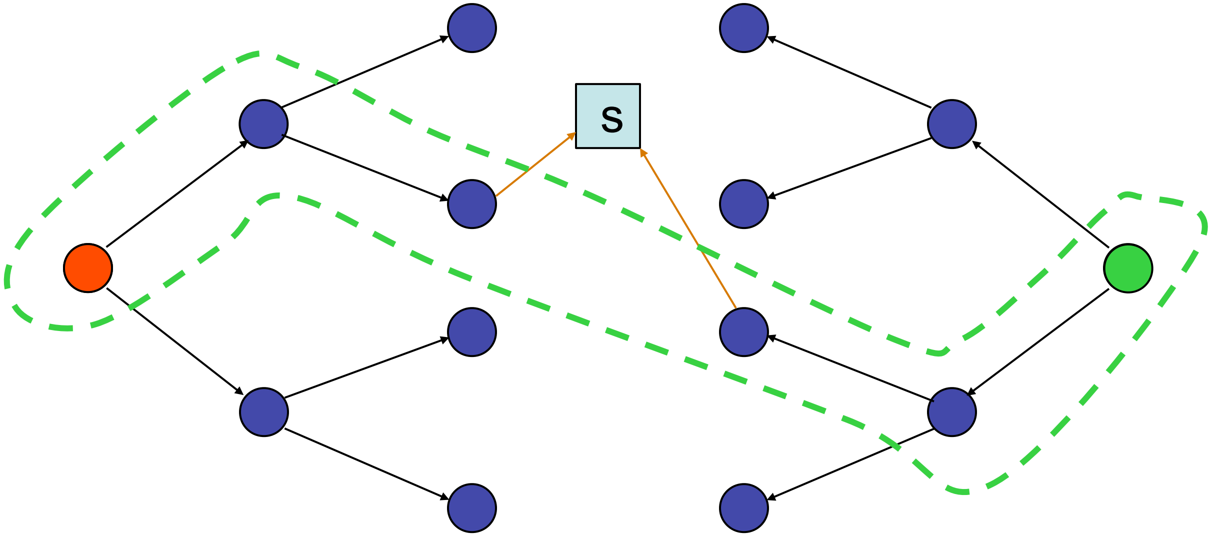
Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2



Bidirectional Strategy

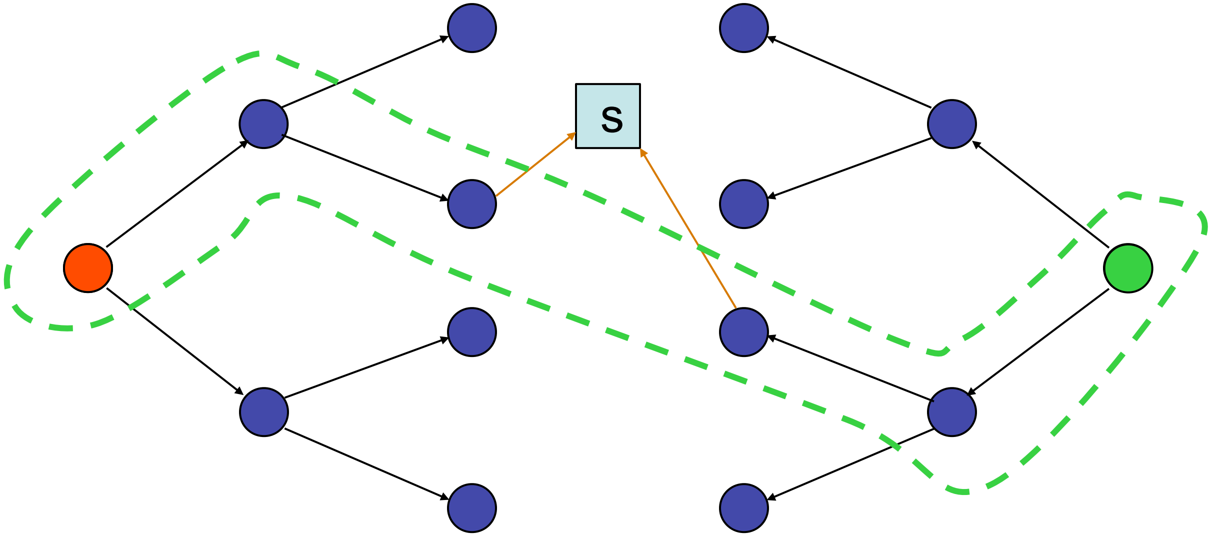
2 fringe queues: FRINGE1 and FRINGE2



Time and space complexity is $O(b^{d/2}) << O(b^d)$ if both trees have the same branching factor b

Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2

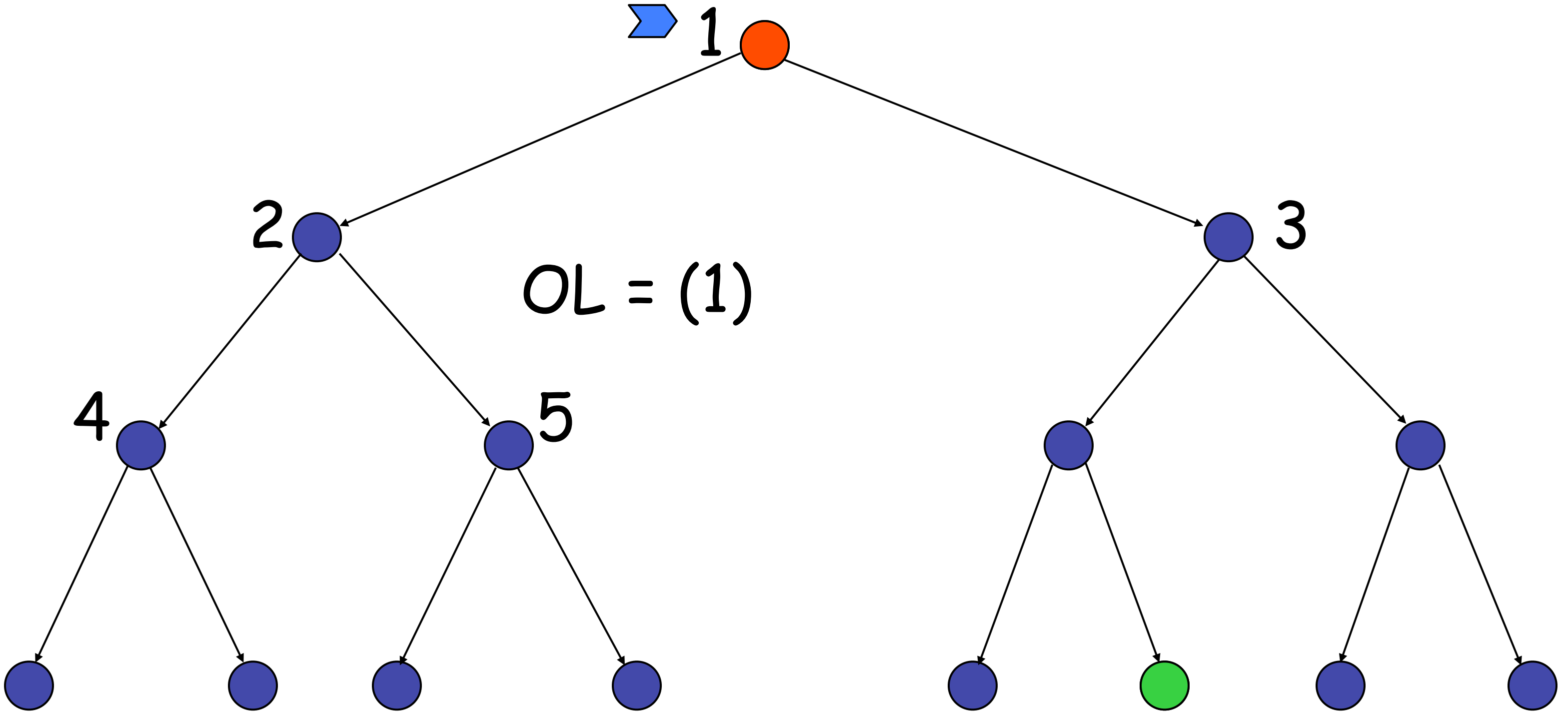


Time and space complexity is $O(b^{d/2}) << O(b^d)$ if both trees have the same branching factor b

Question: What happens if the branching factor is different in each direction?

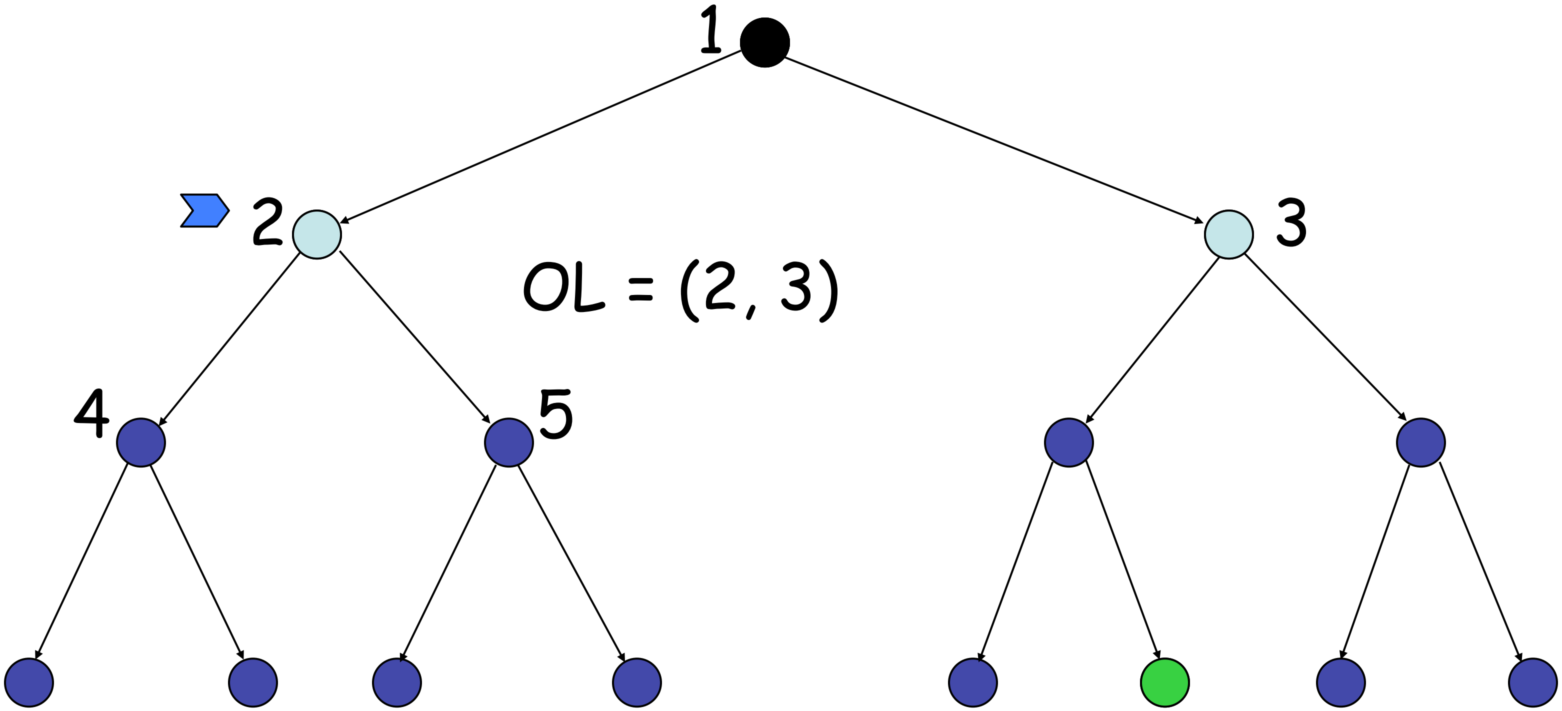
Depth-First Strategy

New nodes are inserted at the front of OL



Depth-First Strategy

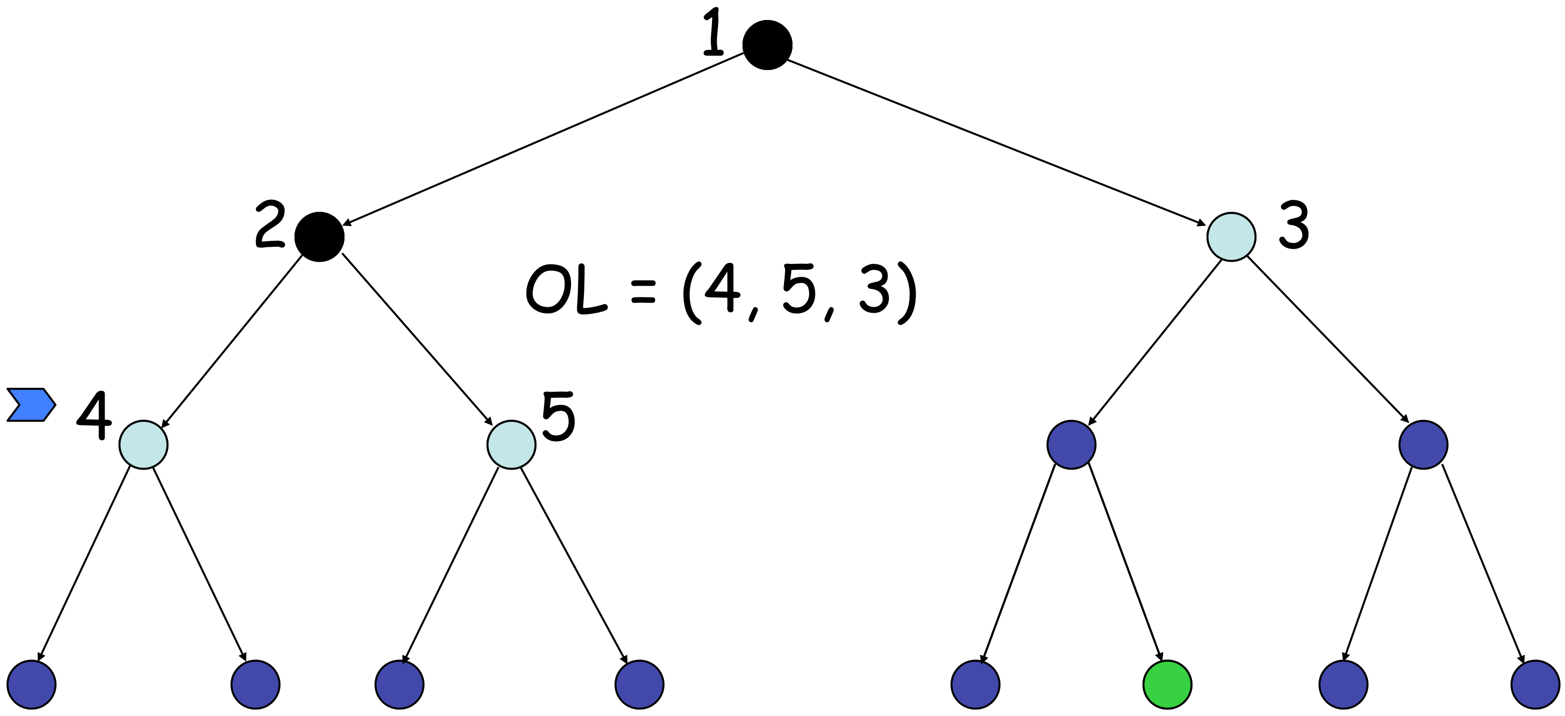
New nodes are inserted at the front of OL



OL = (2, 3)

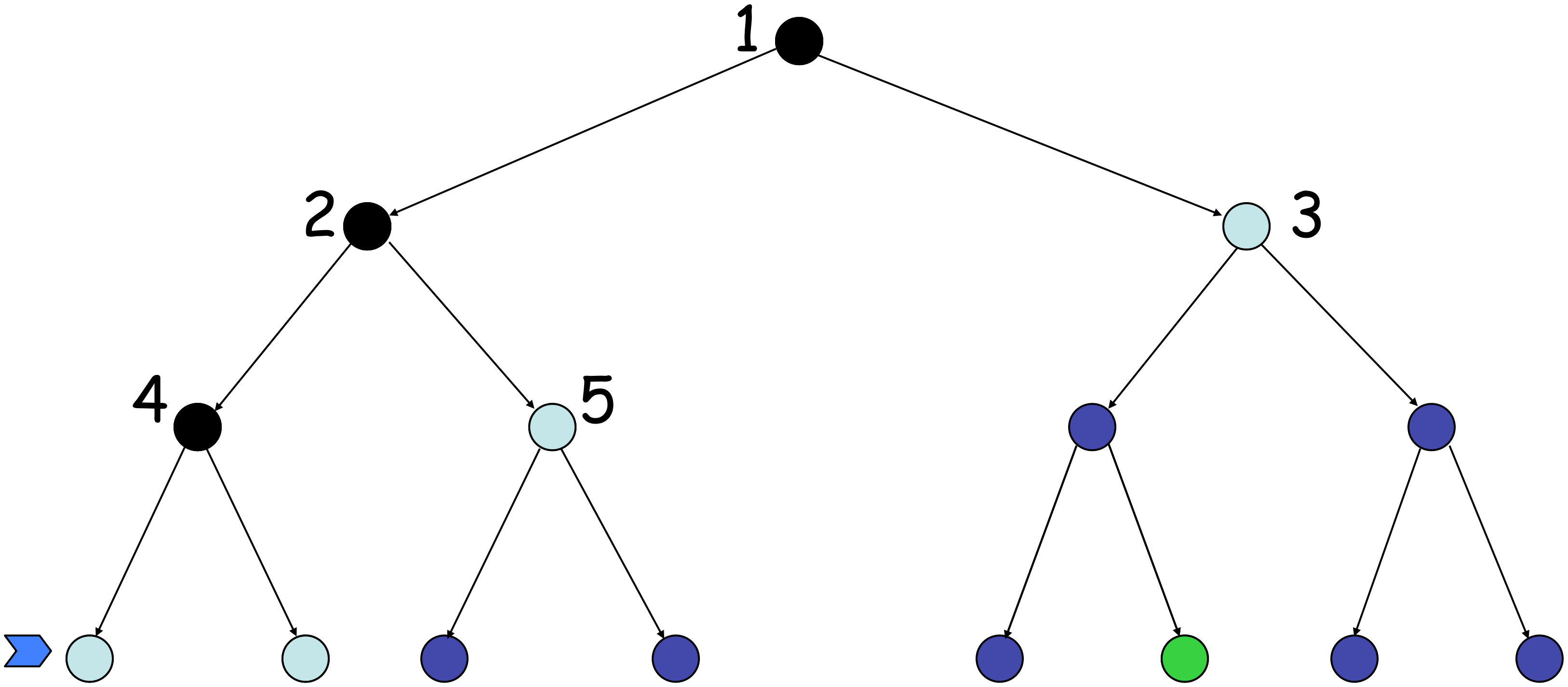
Depth-First Strategy

New nodes are inserted at the front of OL



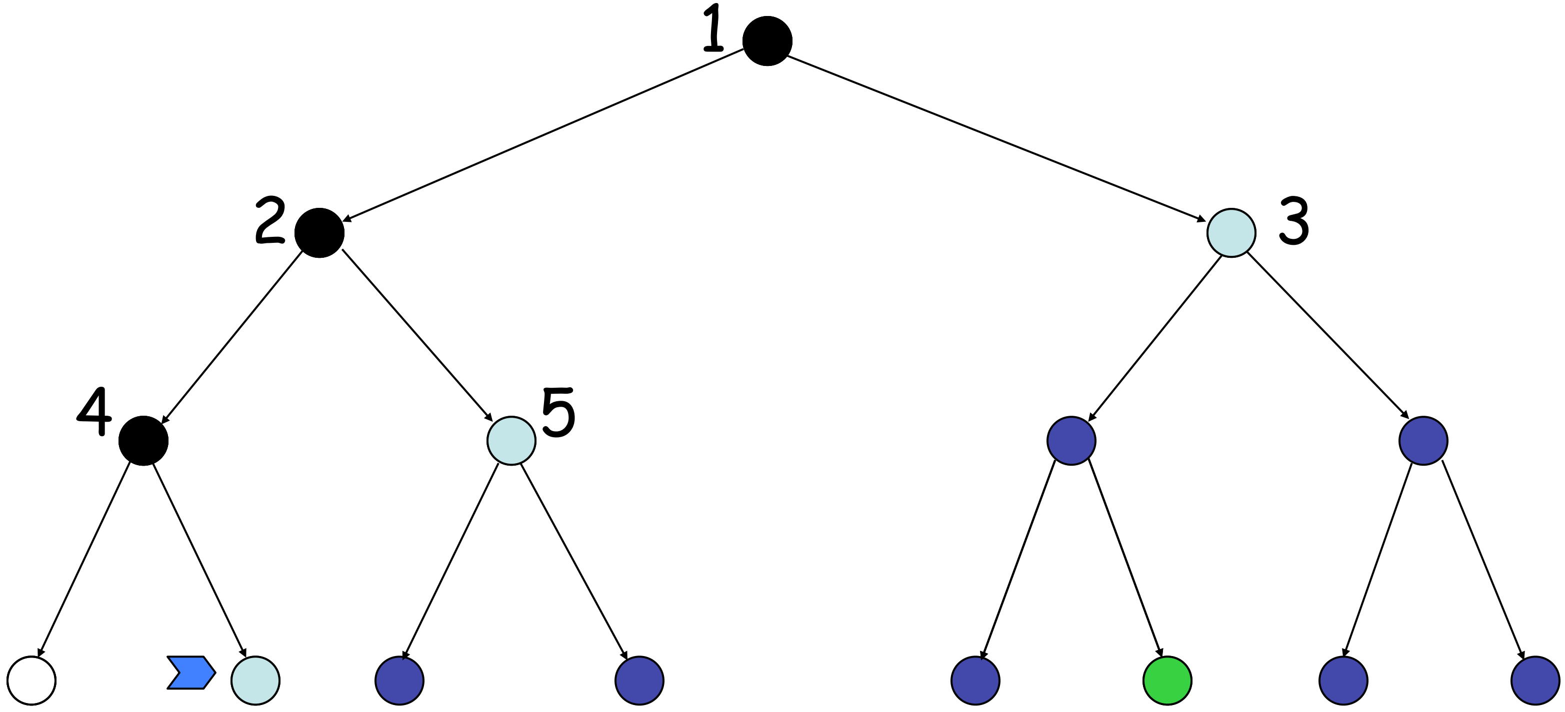
Depth-First Strategy

New nodes are inserted at the front of OL



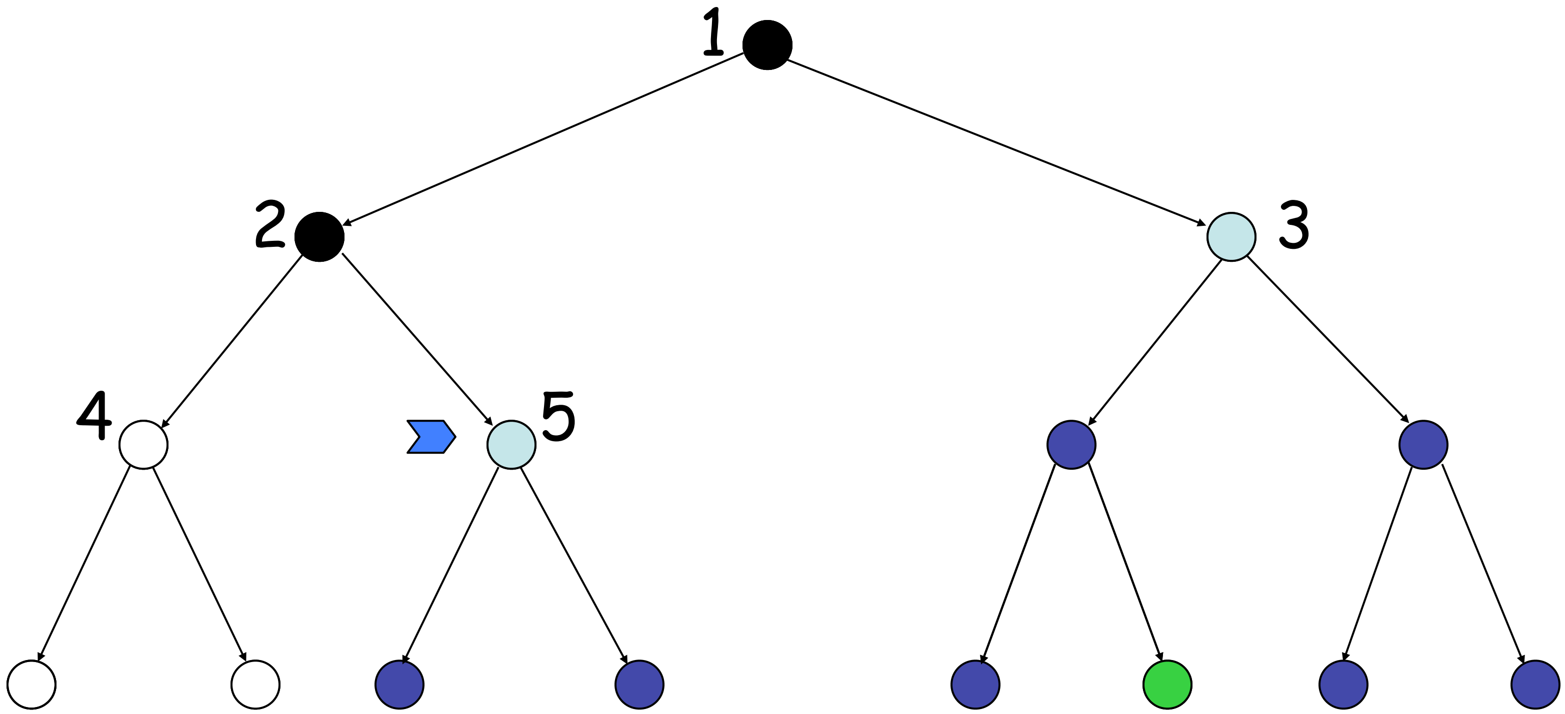
Depth-First Strategy

New nodes are inserted at the front of OL



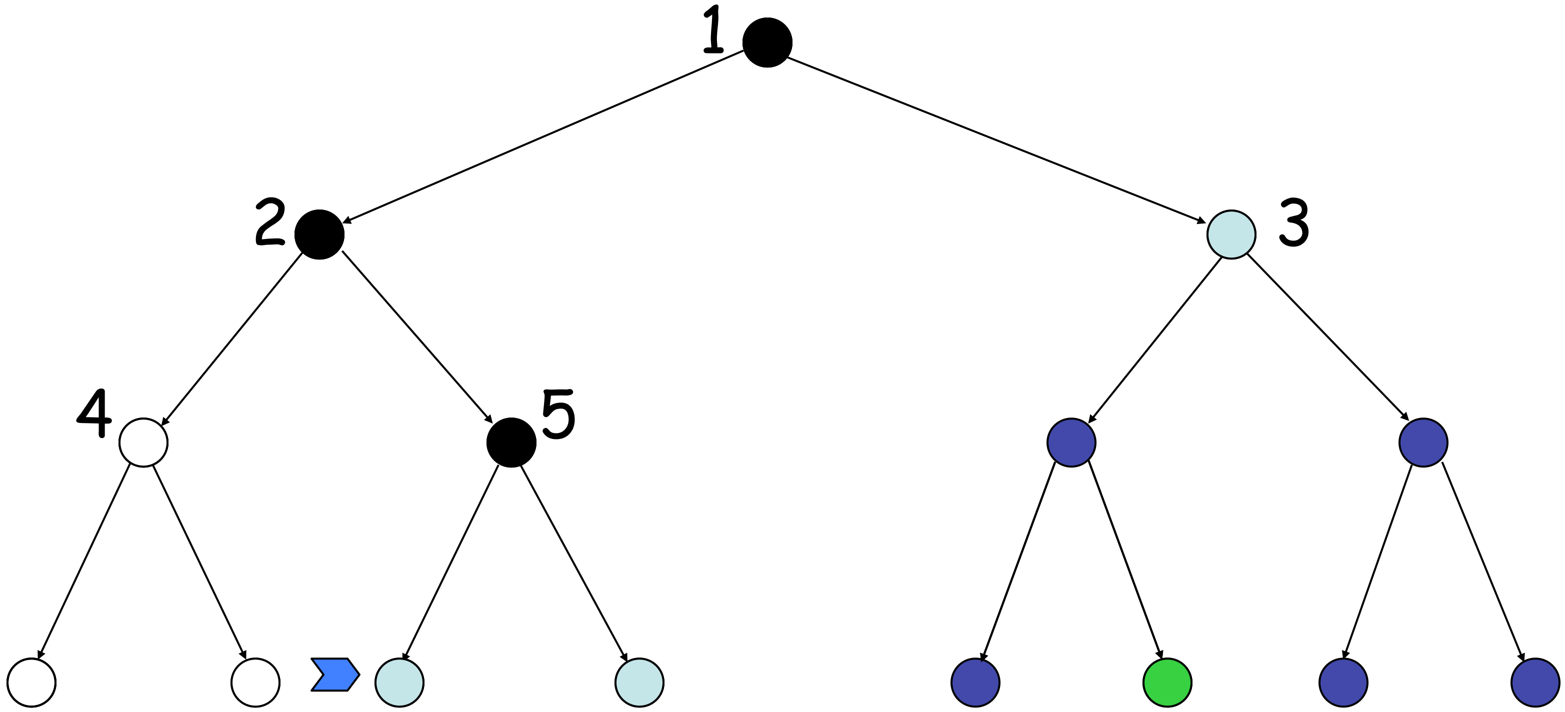
Depth-First Strategy

New nodes are inserted at the front of OL



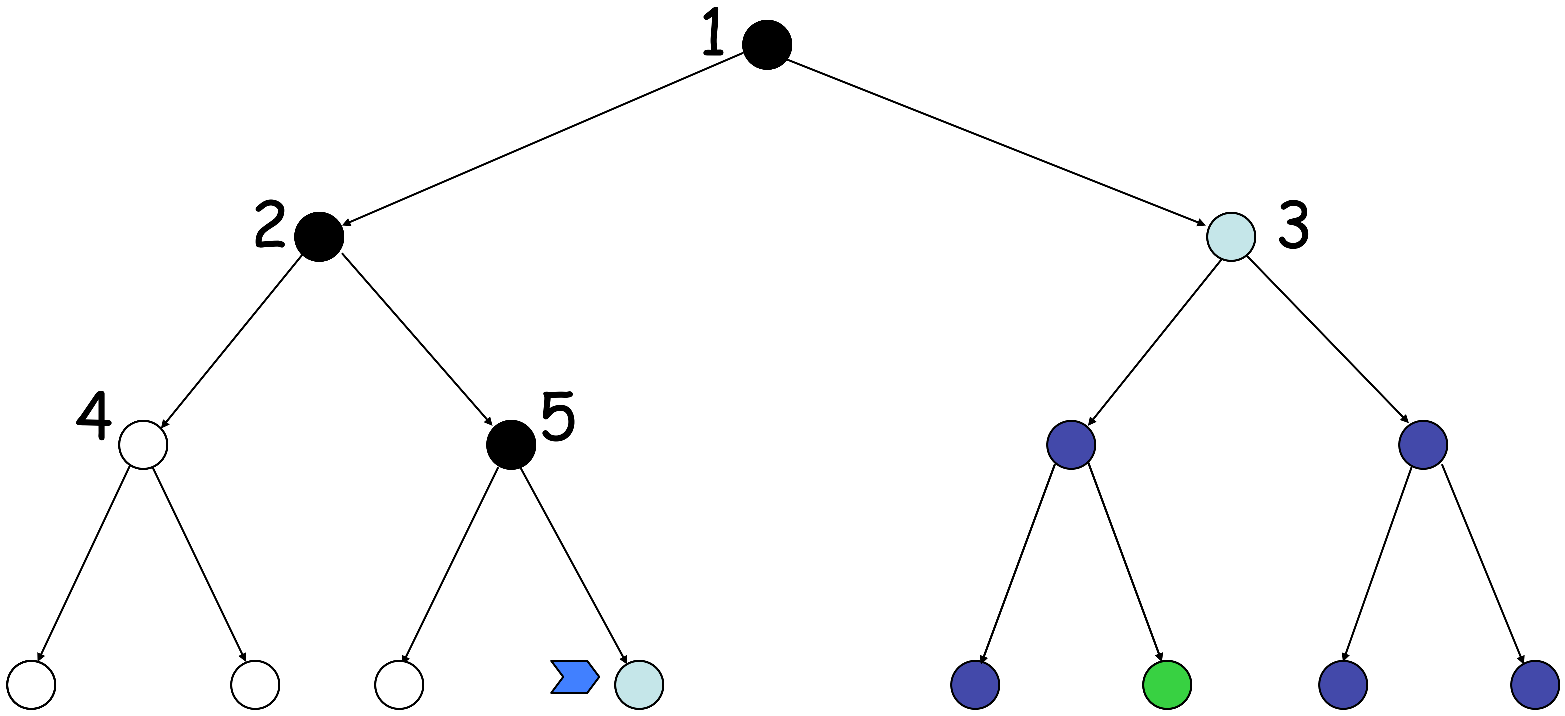
Depth-First Strategy

New nodes are inserted at the front of OL



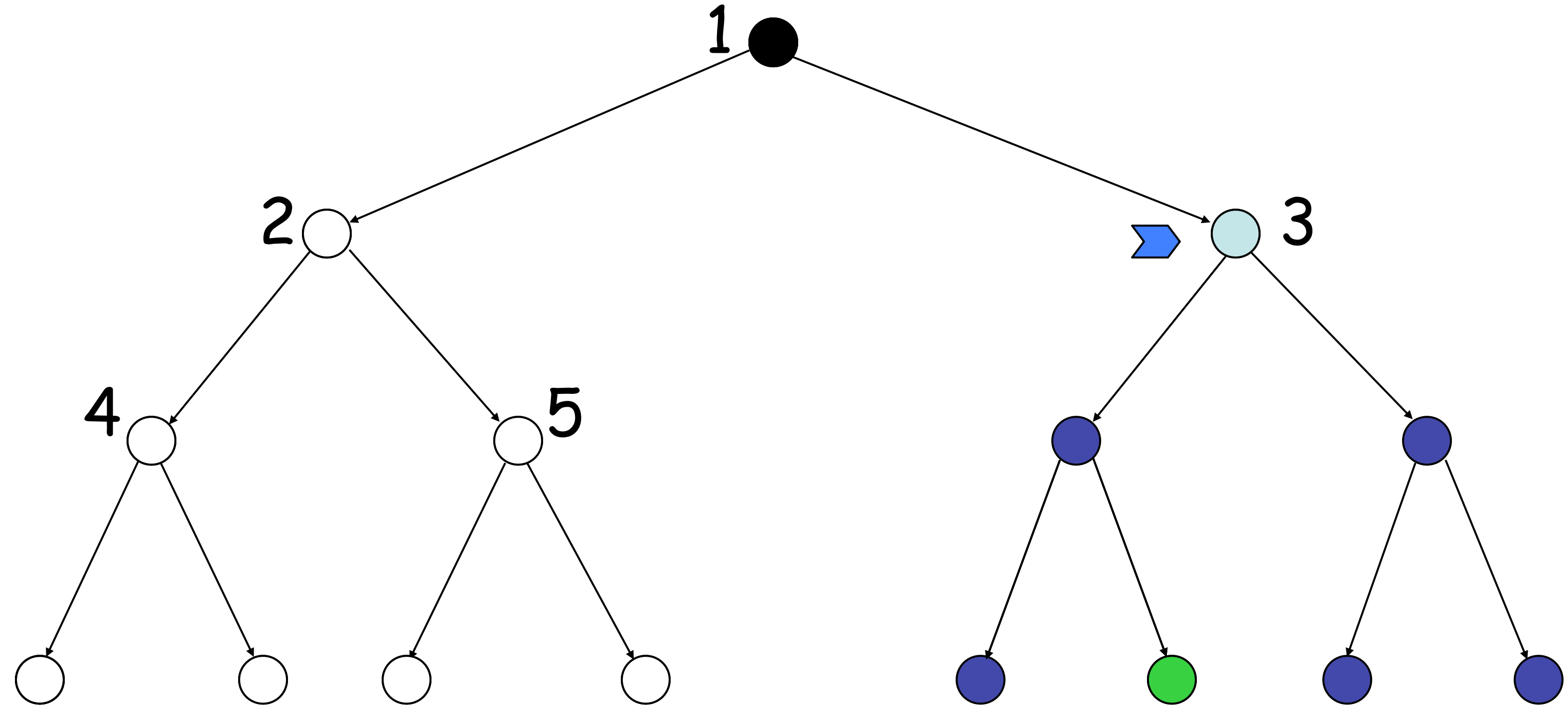
Depth-First Strategy

New nodes are inserted at the front of OL



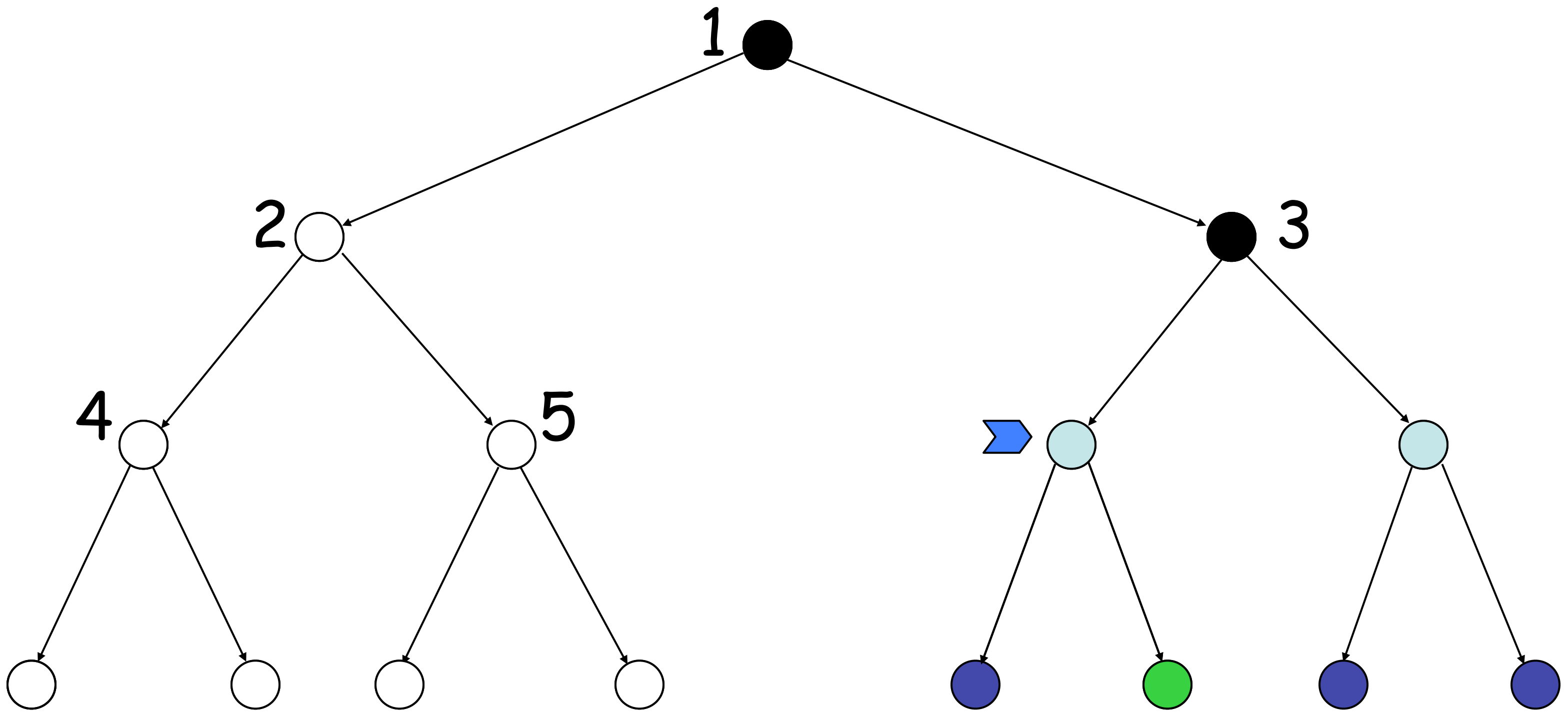
Depth-First Strategy

New nodes are inserted at the front of OL



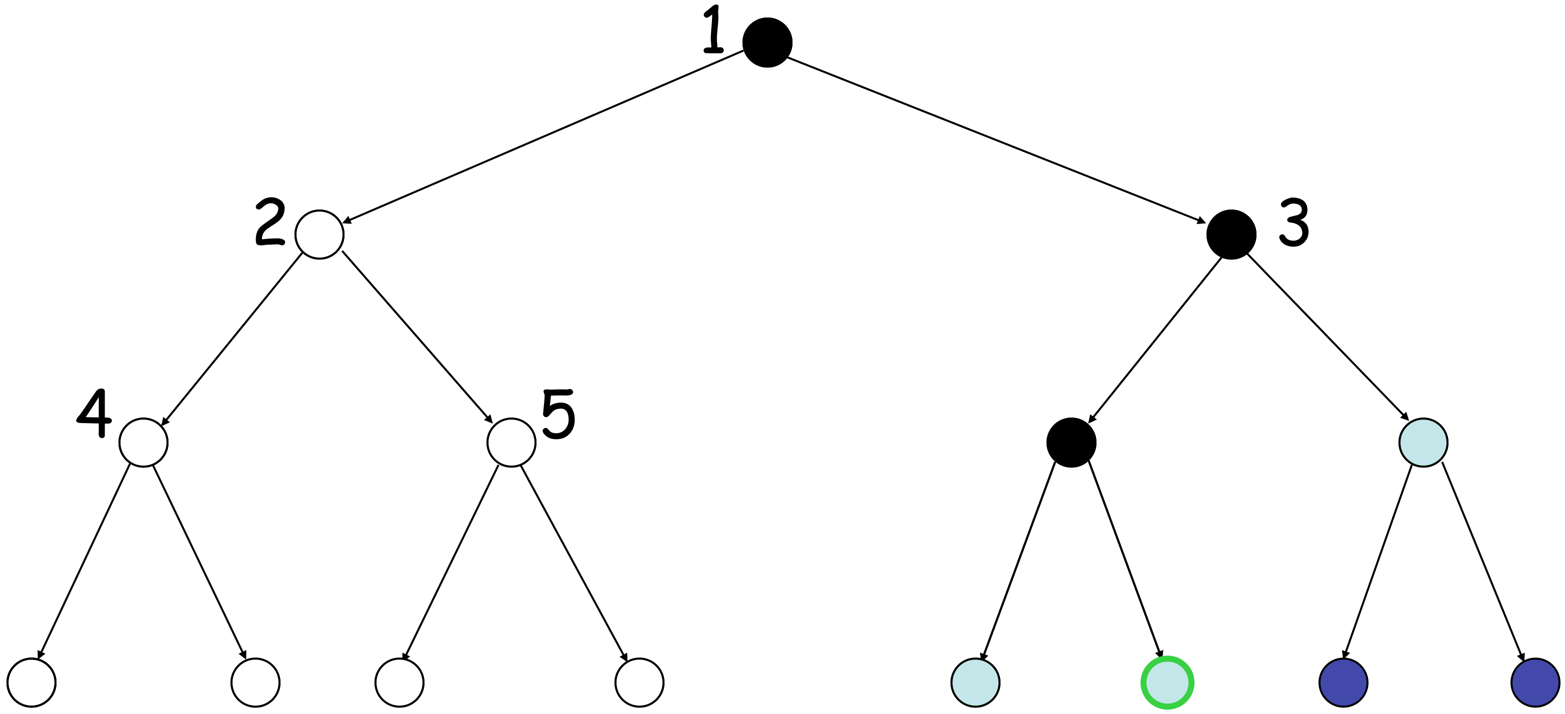
Depth-First Strategy

New nodes are inserted at the front of OL



Depth-First Strategy

New nodes are inserted at the front of OL



Evaluation

- b: branching factor
- d: depth of shallowest goal node
- m: maximal depth of a leaf node
- Depth-first search is:
  - Complete?
  - Optimal?

Evaluation

Evaluation

- b: branching factor
- d: depth of shallowest goal node

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- m: maximal depth of a leaf node
- Depth-first search is:
  - Complete only for finite search tree
  - Not optimal

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- m: maximal depth of a leaf node
- Depth-first search is:
  - Complete only for finite search tree
  - Not optimal
- Number of nodes generated (worst case):
  $1 + b + b^2 + \ldots + b^m = O(b^m)$

Evaluation

- b: branching factor
- d: depth of shallowest goal node
- m: maximal depth of a leaf node
- Depth-first search is:
  - Complete only for finite search tree
  - Not optimal
- Number of nodes generated (worst case): $1 + b + b^2 + \ldots + b^m = O(b^m)$
- Time complexity is $O(b^m)$

Depth-Limited Search

- Depth-first with depth cutoff k (depth at which nodes are not expanded)

- Three possible outcomes:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)

Iterative Deepening Search

Provides the best of both breadth-first and depth-first search

Main idea: Totally horrifying !

Iterative Deepening Search

Provides the best of both breadth-first and depth-first search

Main idea: Totally horrifying !

Iterative Deepening Search

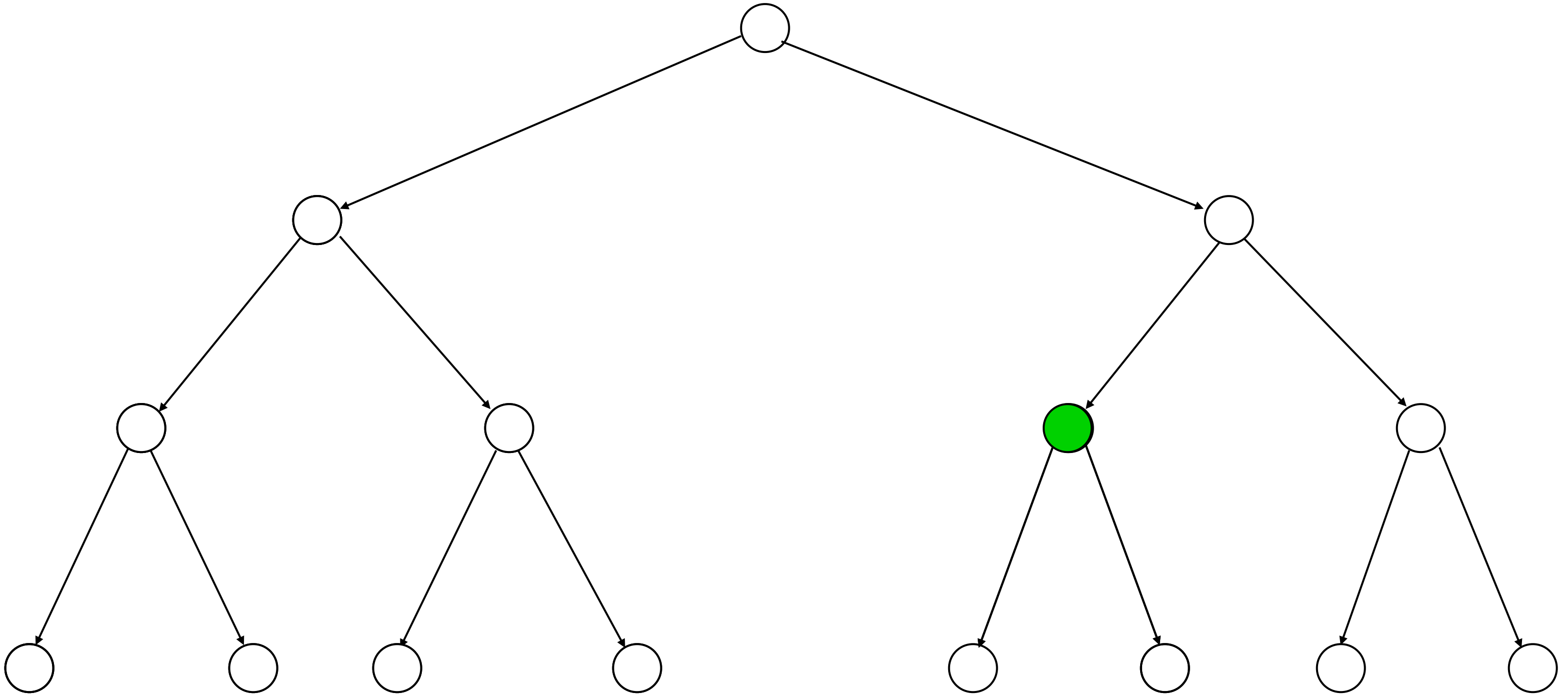
Provides the best of both breadth-first and depth-first search

Main idea: Totally horrifying !

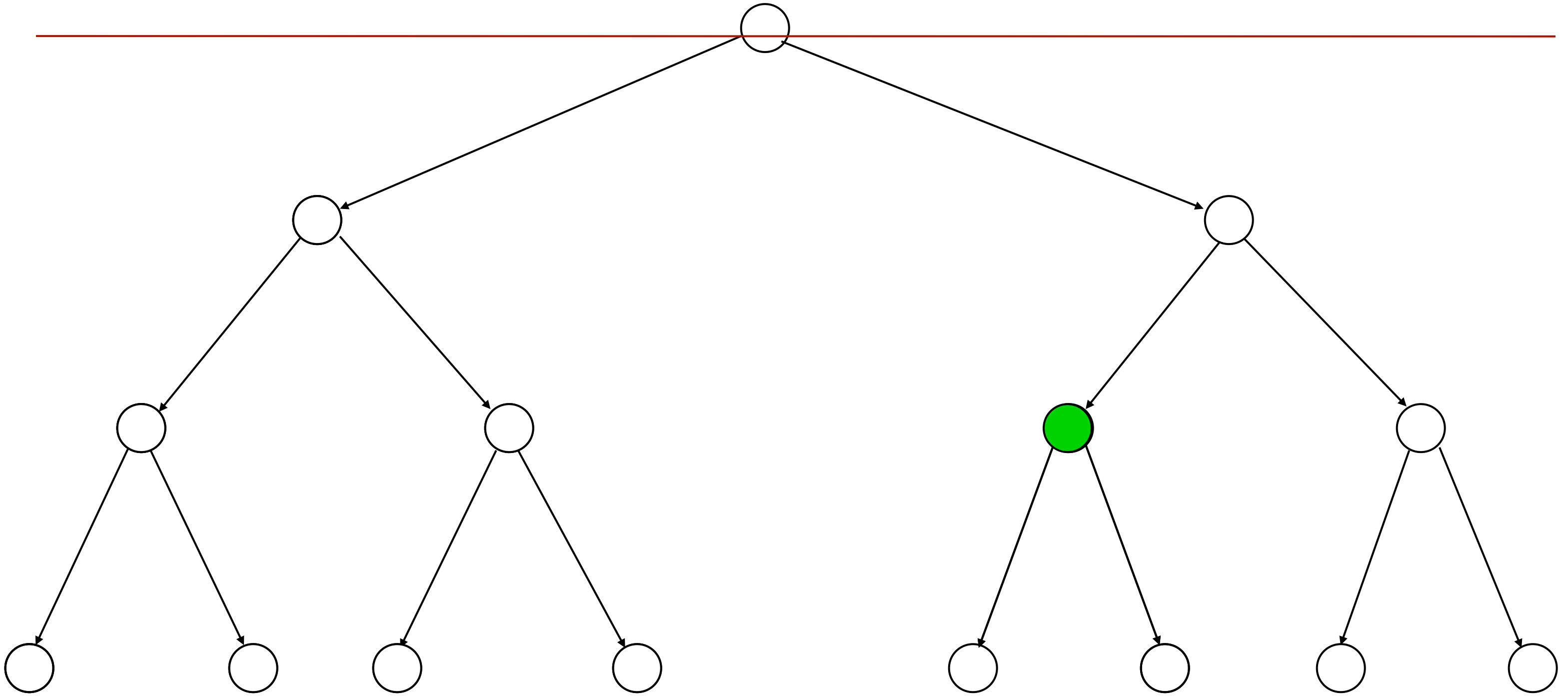
IDS

```
For k = 0, 1, 2, … do:
    Perform depth-first search with
    depth cutoff k
    (i.e., only generate nodes with depth ≤ k)
```

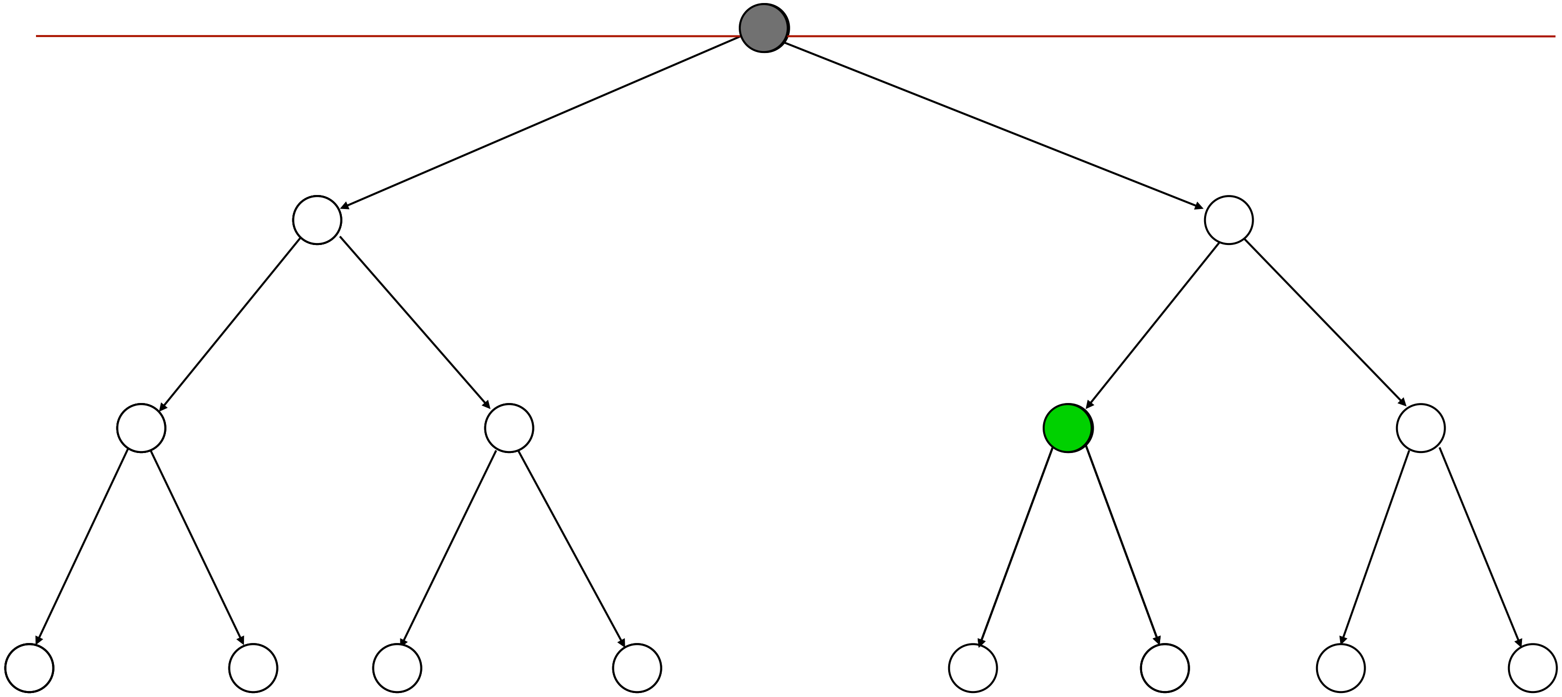
Iterative Deepening



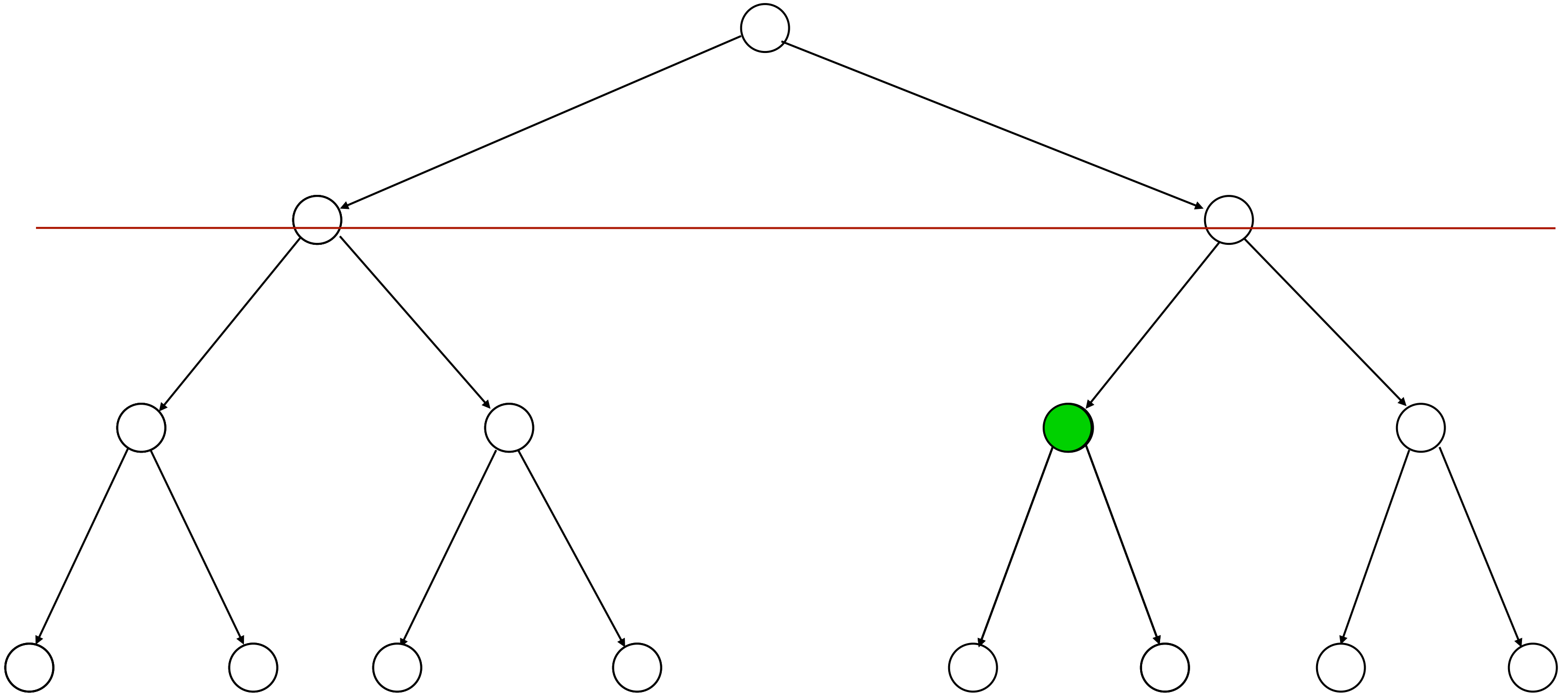
Iterative Deepening



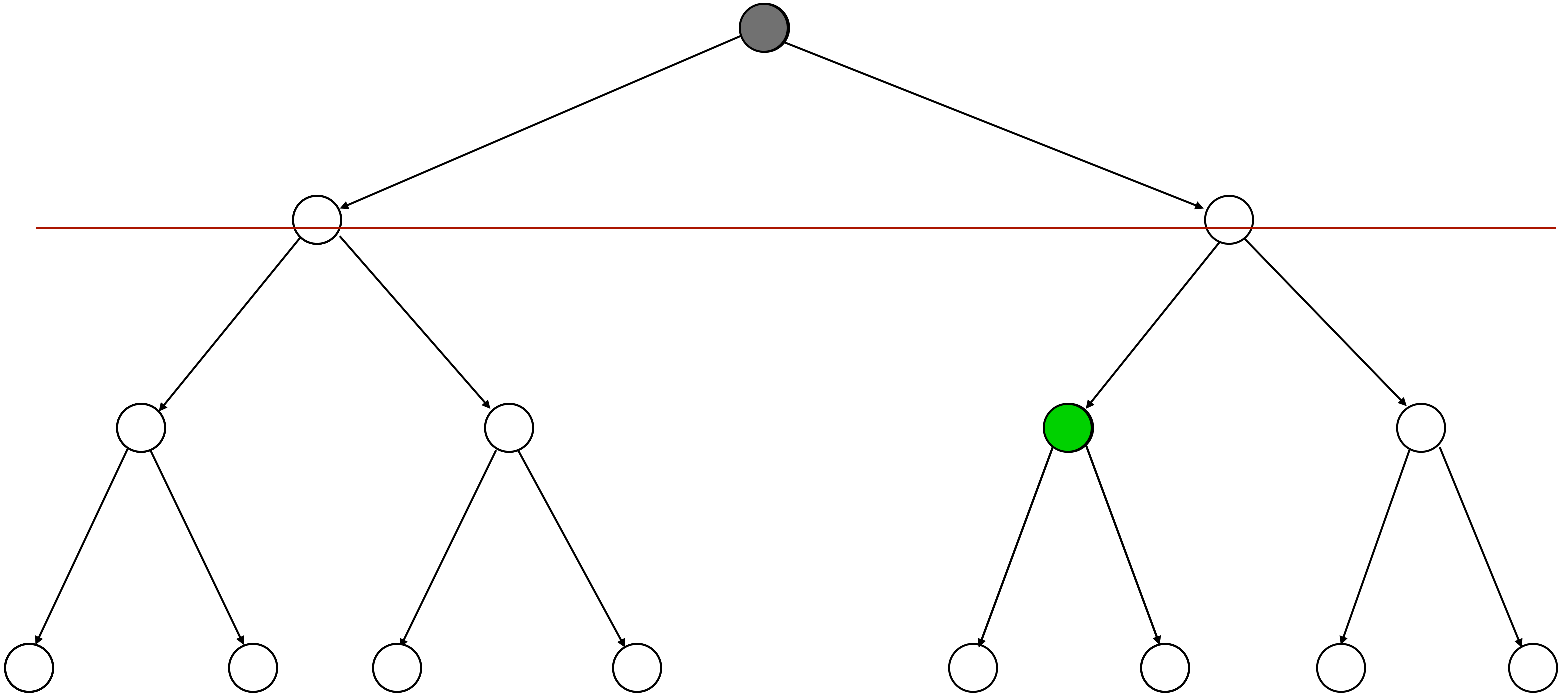
Iterative Deepening



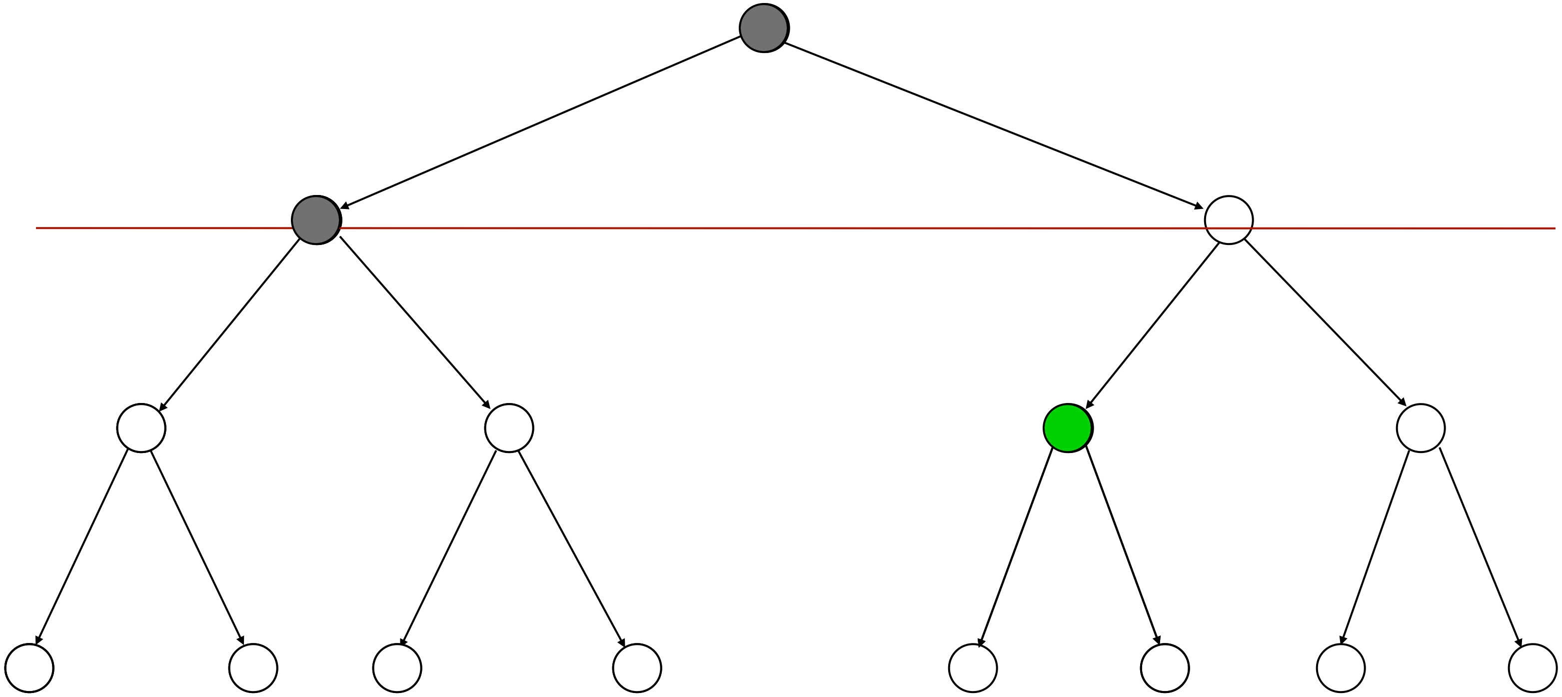
Iterative Deepening



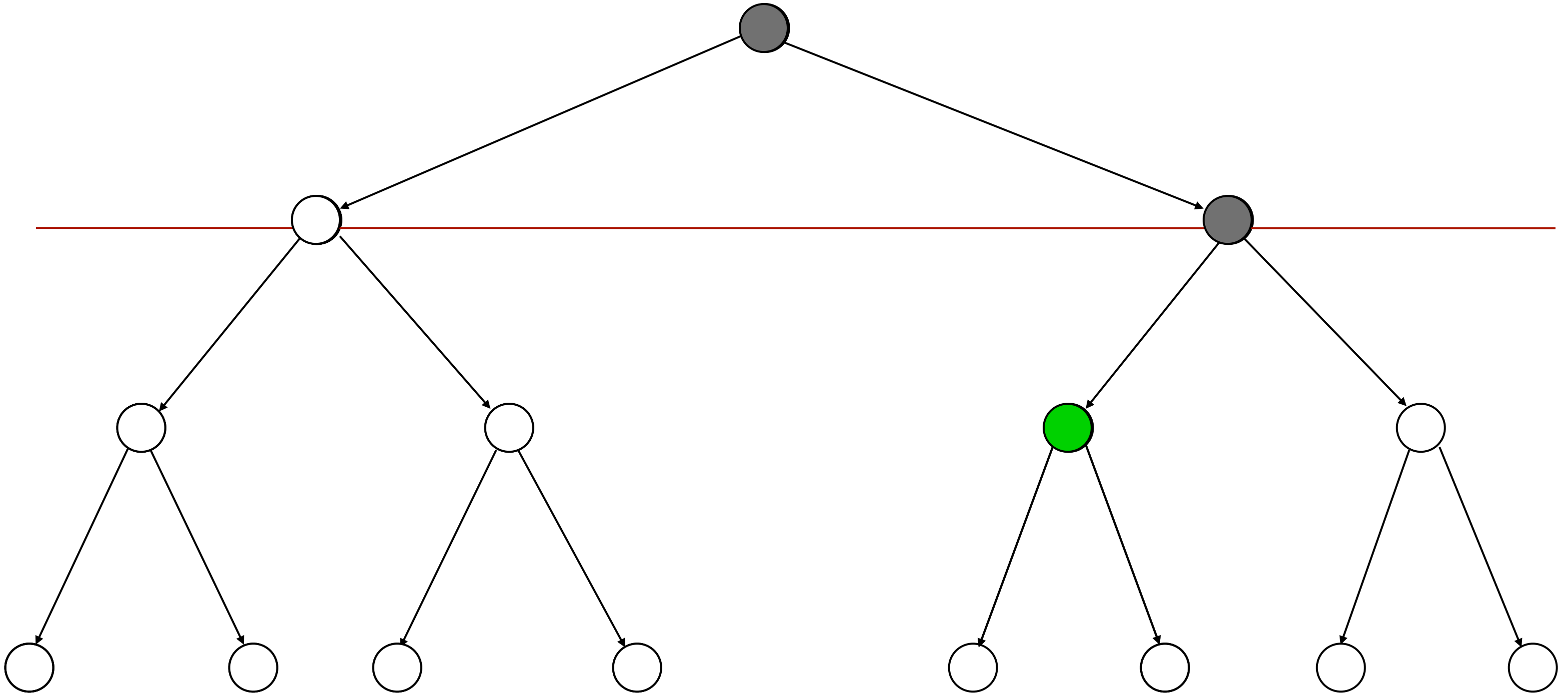
Iterative Deepening



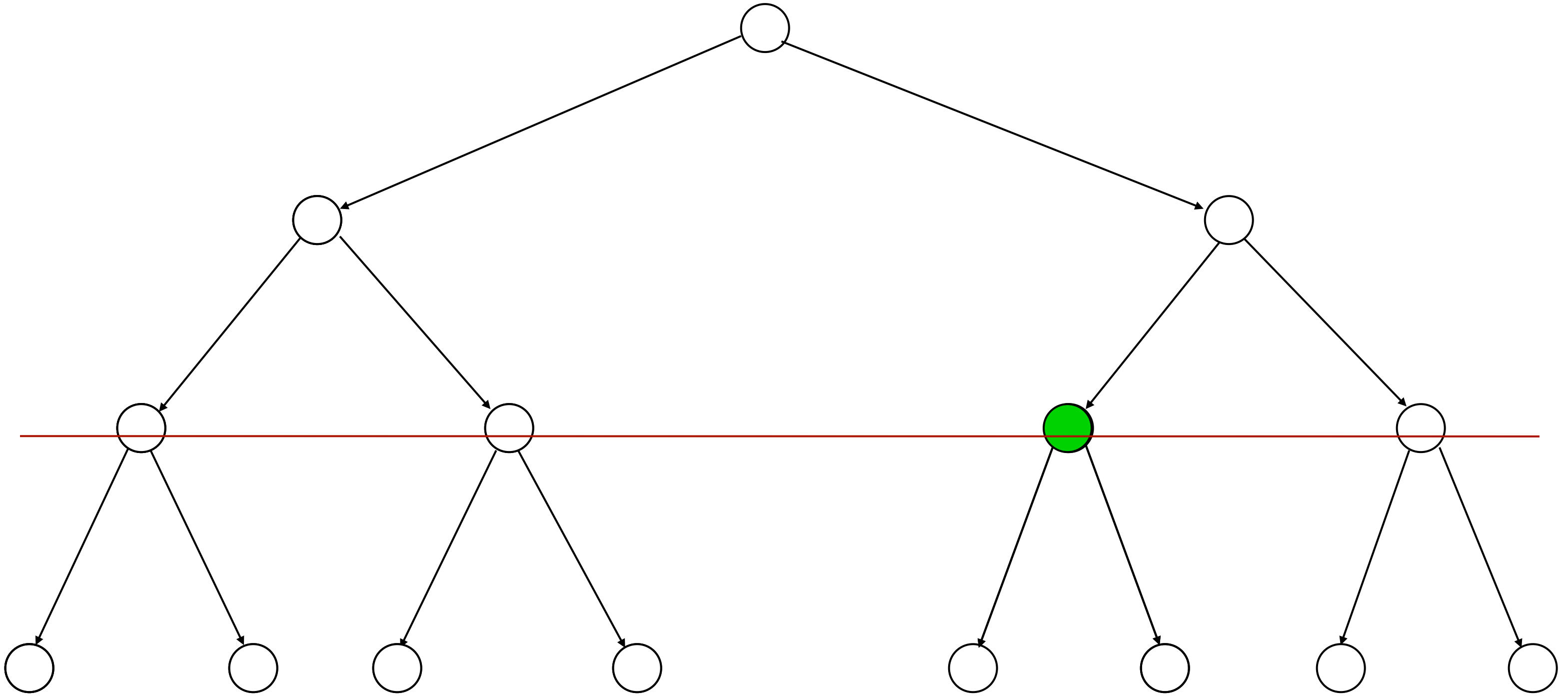
Iterative Deepening



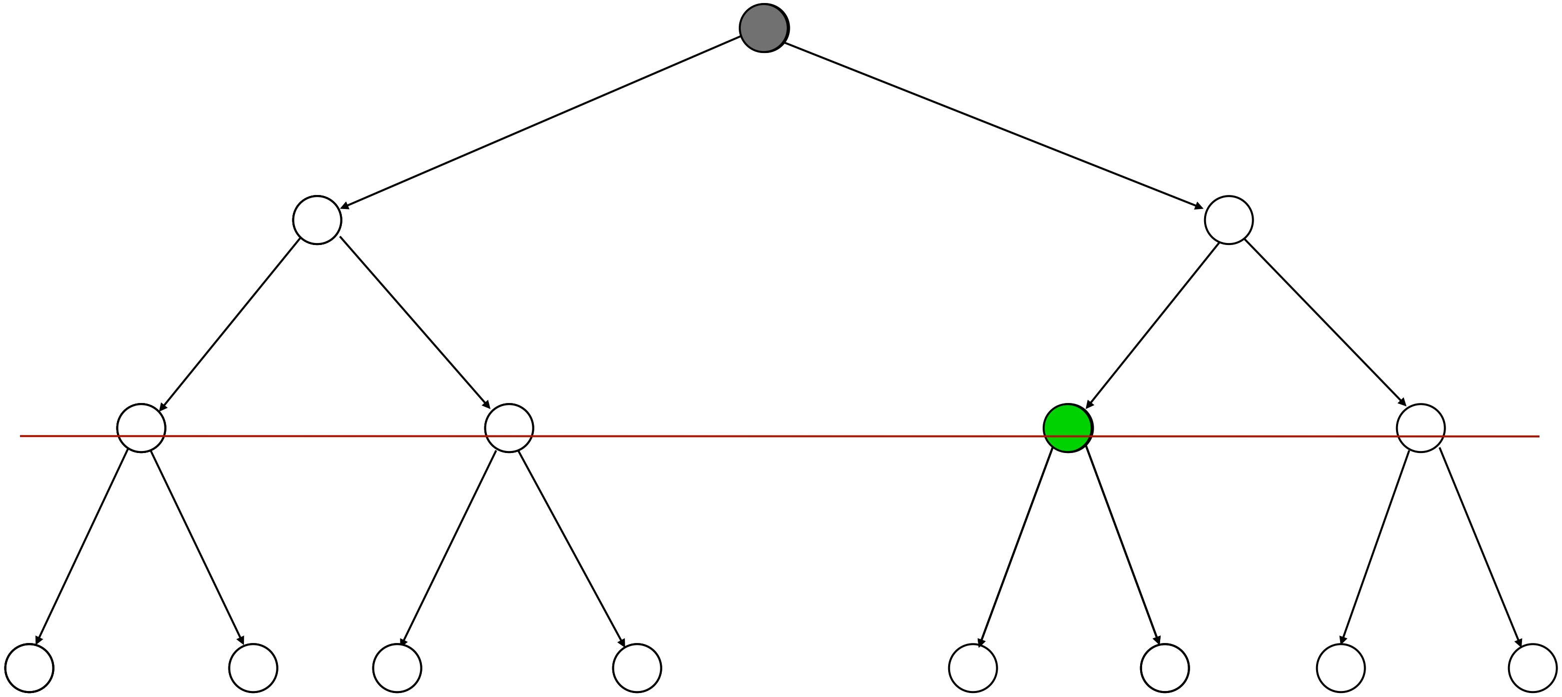
Iterative Deepening



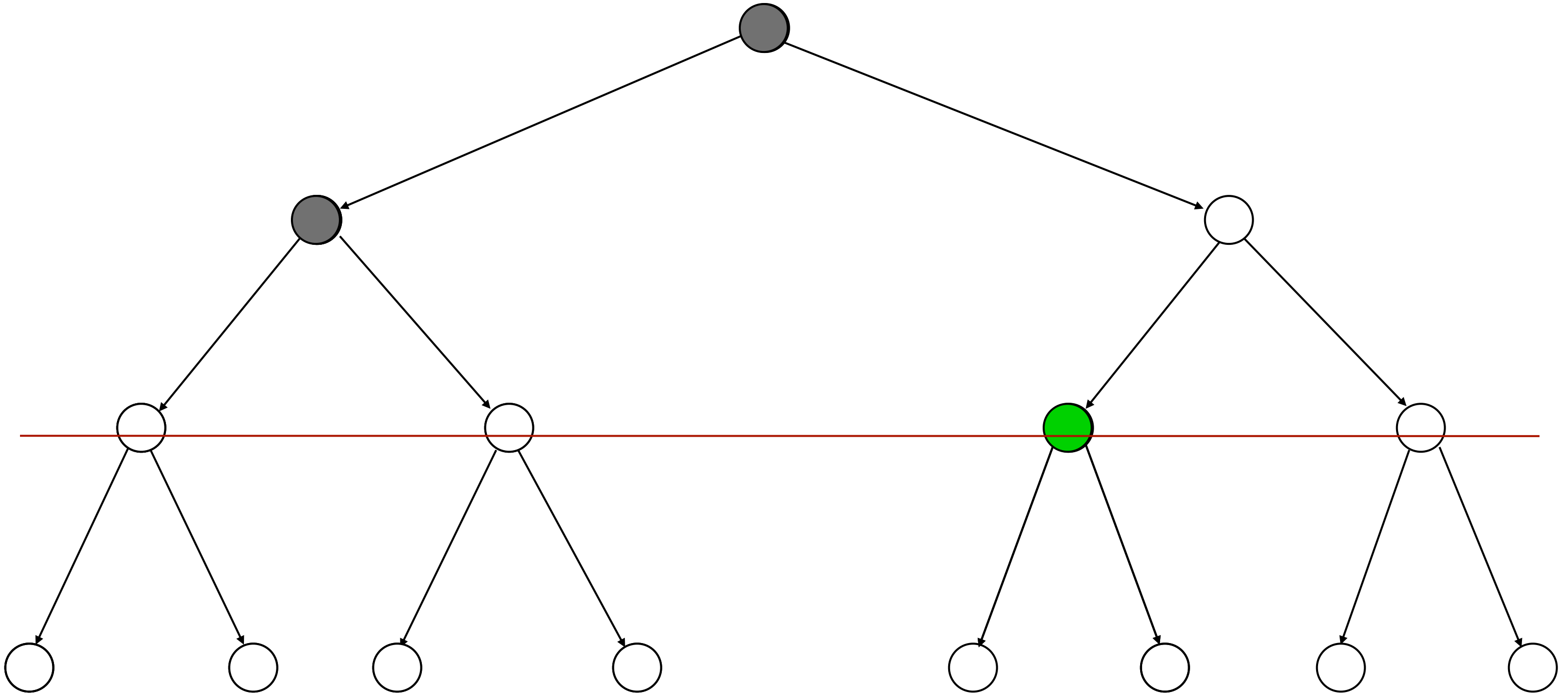
Iterative Deepening



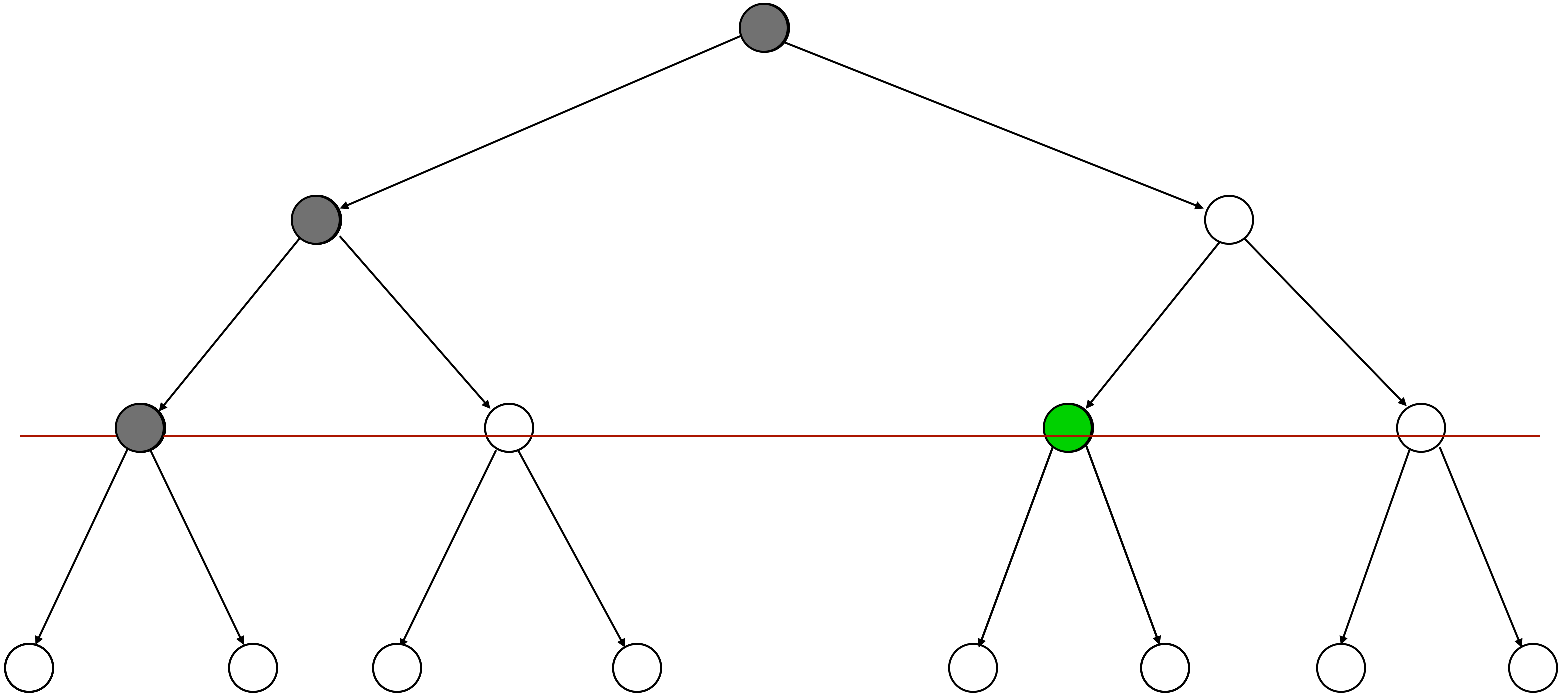
Iterative Deepening



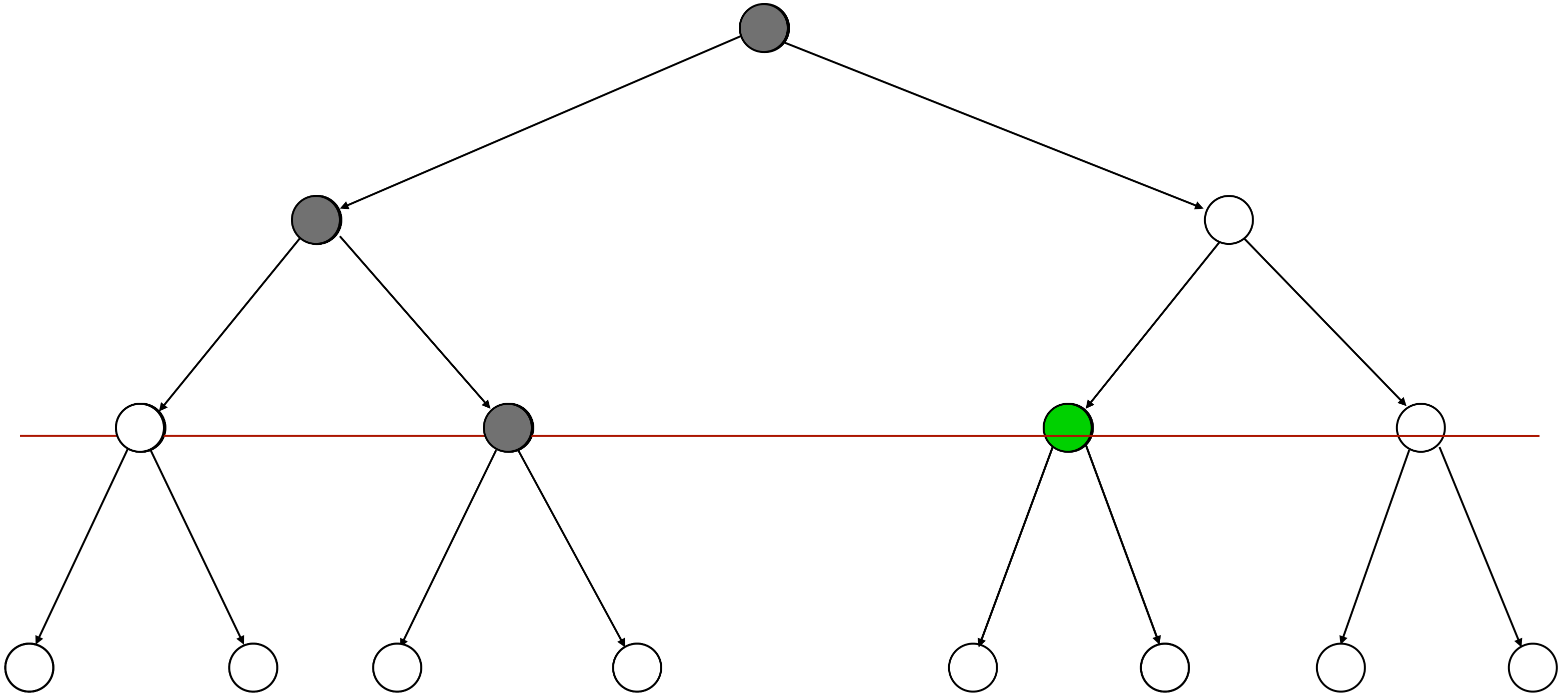
Iterative Deepening



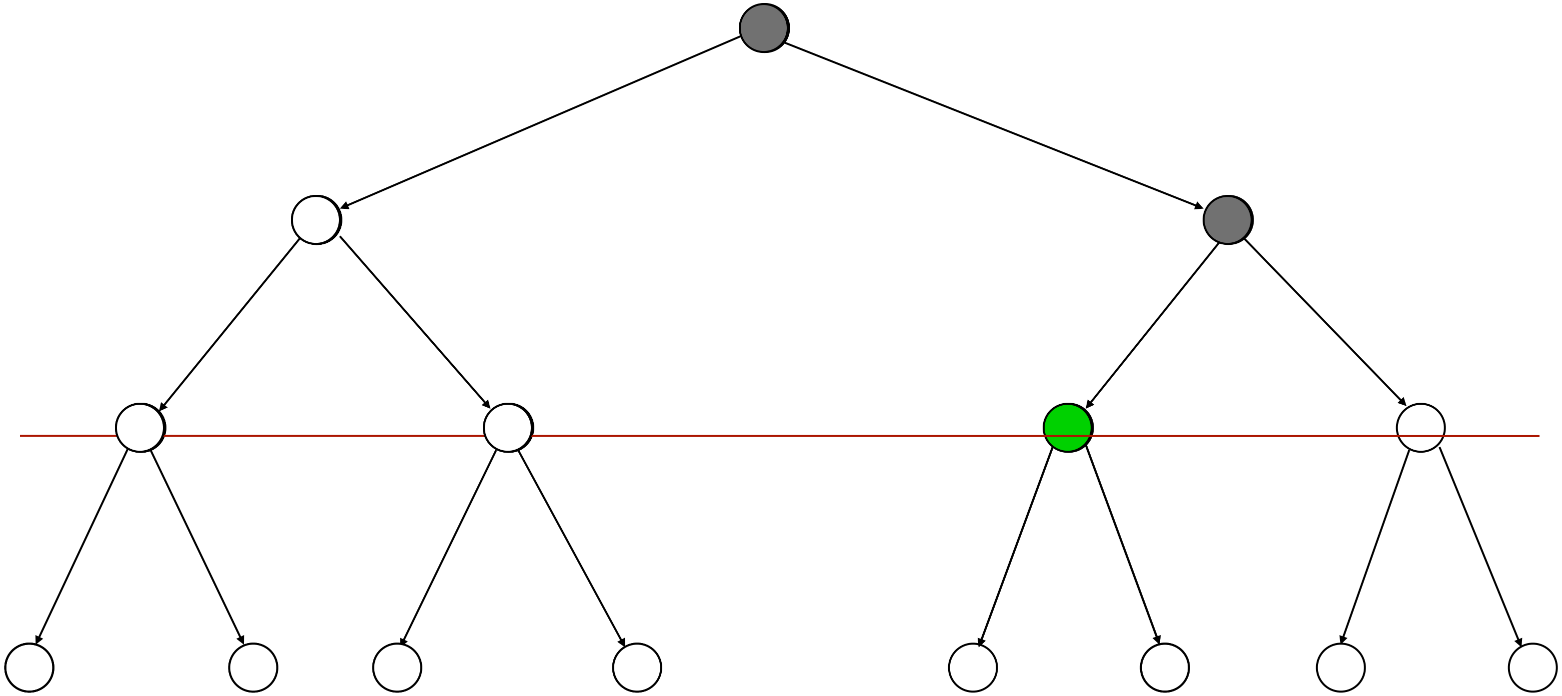
Iterative Deepening



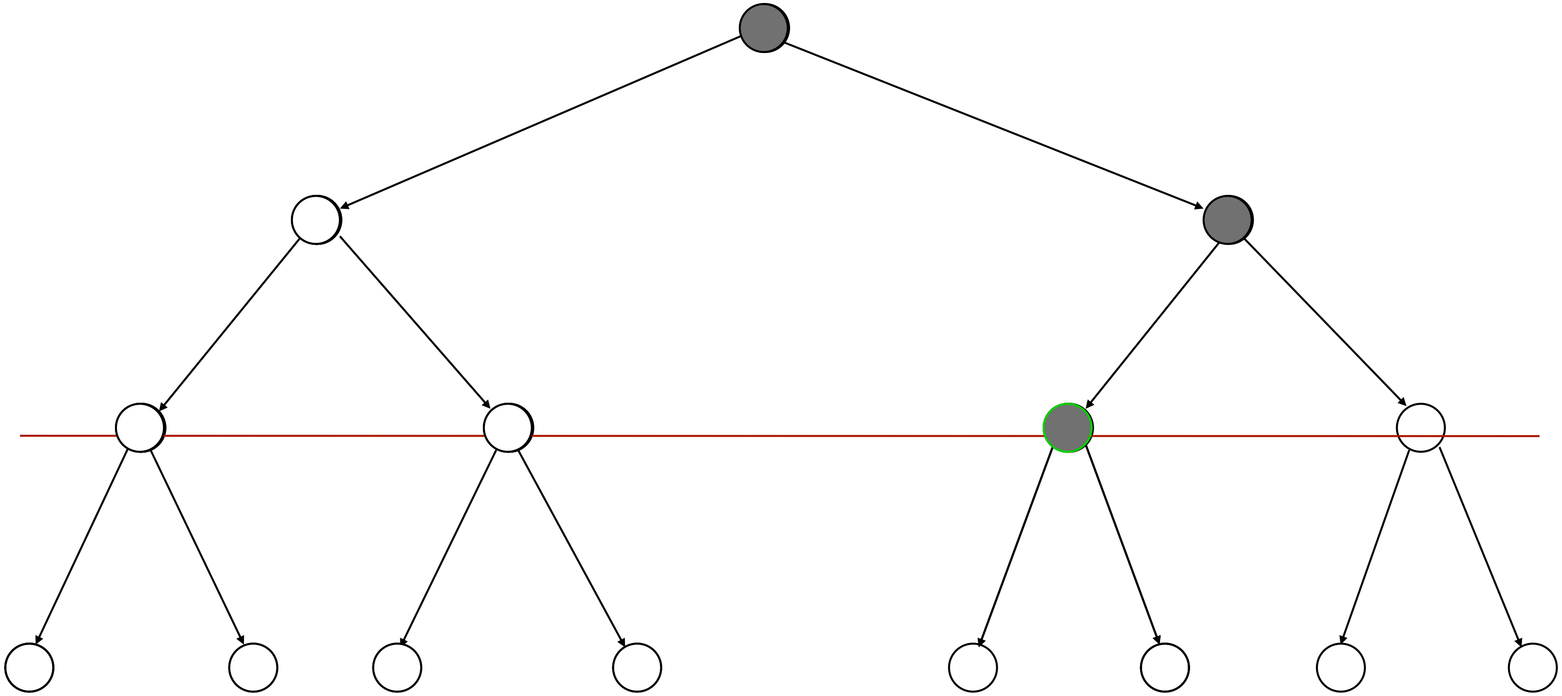
Iterative Deepening



Iterative Deepening



Iterative Deepening



Performance

- Iterative deepening search is:
  - Complete
  - Optimal if step cost = 1
- Time complexity is:
  $(d+1)(1) + db + (d-1)b^2 + \dots + (1)\, b^d = O(b^d)$
- Space complexity is: O(bd) or O(d)

Calculation

$$db + (d-1)b^2 + \ldots + (1)\, b^d$$

$$= b^d + 2b^{d-1} + 3b^{d-2} + \ldots + db$$

$$= (1 + 2b^{-1} + 3b^{-2} + \ldots + db^{-d}) \times b^d$$

$$\leq \left(\sum_{i=1,\ldots,\infty} ib^{(1-i)}\right) \times b^d = b^d\, (b/(b-1))^2$$

Number of Generated Nodes
(Breadth-First & Iterative Deepening)

d = 5 and b = 2

BF	ID
1	1 x 6 = 6
2	2 x 5 = 10
4	4 x 4 = 16
8	8 x 3 = 24
16	16 x 2 = 32
32	32 x 1 = 32
63	120

120/63 ~ 2

Number of Generated Nodes (Breadth-First & Iterative Deepening)

d = 5 and b = 10

BF	ID
1	6
10	50
100	400
1,000	3,000
10,000	20,000
100,000	100,000
111,111	123,456

123,456/111,111 ~ 1.111

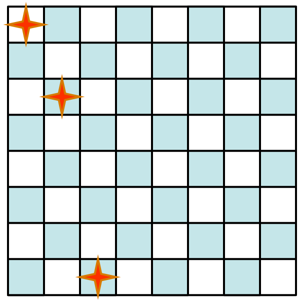
Comparison of Strategies

- Breadth-first is complete and optimal, but has high space complexity
- Depth-first is space efficient, but is neither complete, nor optimal
- Iterative deepening is complete and optimal, with the same space complexity as depth-first and almost the same time complexity as breadth-first

Revisited States

Revisited States

No

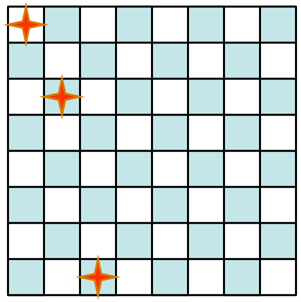


8-queens

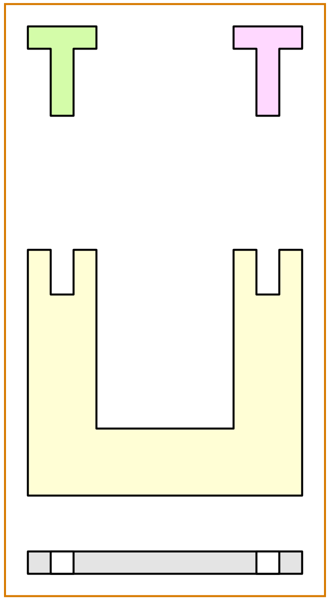
Revisited States

No

Few



8-queens



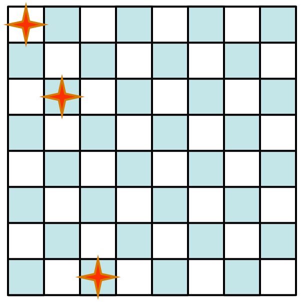
assembly planning

Revisited States

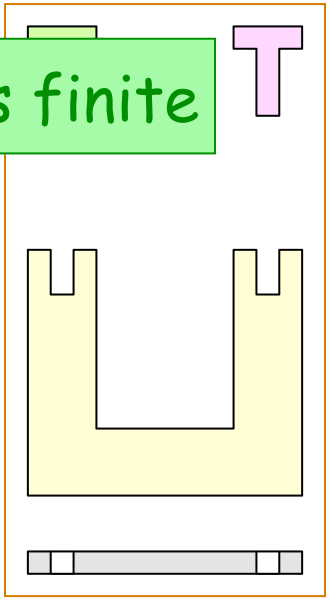
No

Few

search tree is finite



8-queens



assembly planning

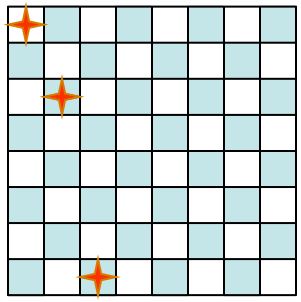
Revisited States

No

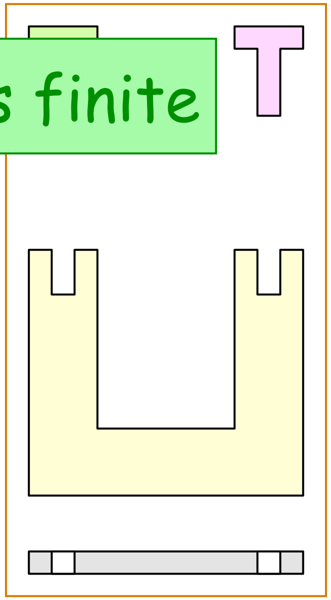
Few

Many

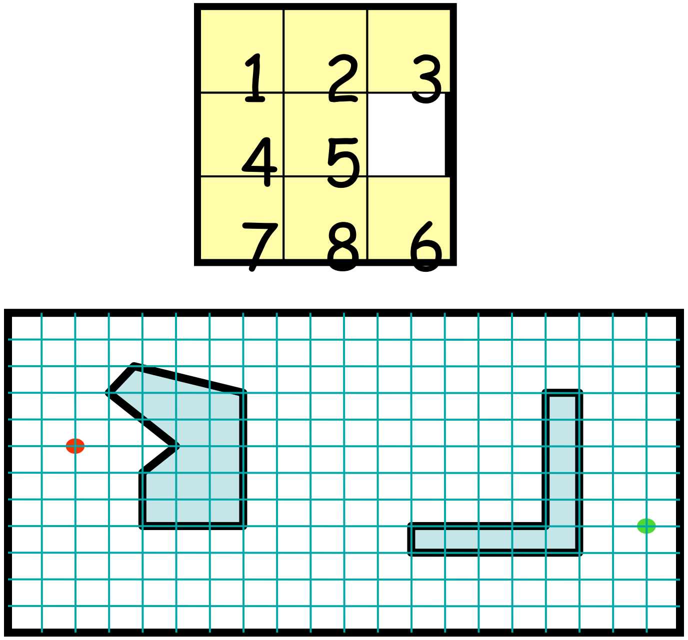
search tree is finite



8-queens



assembly planning



8-puzzle and robot navigation

Revisited States

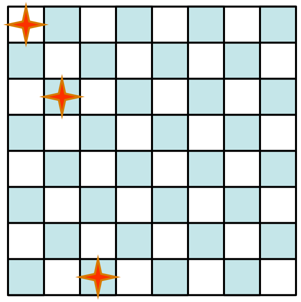
No

Few

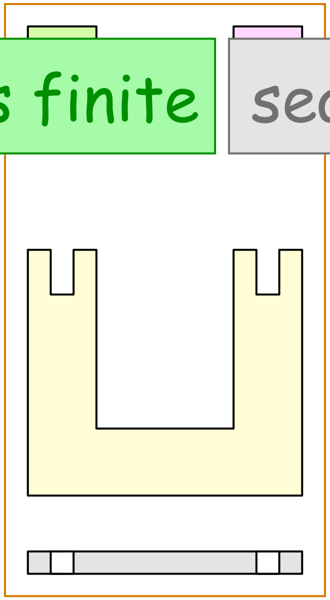
Many

search tree is finite

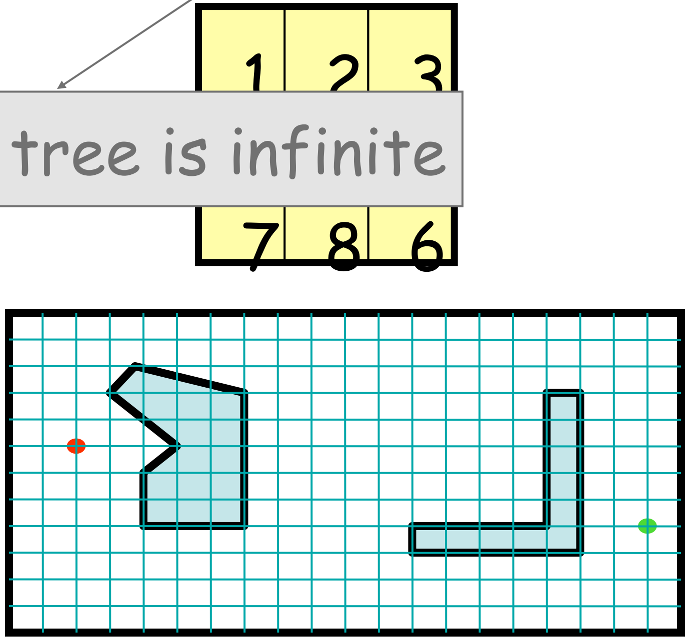
search tree is infinite



8-queens



assembly planning



8-puzzle and robot navigation

Avoiding Revisited States

- Requires comparing state descriptions
- Breadth-first search:
  - Store all states associated with **generated** nodes in CLOSED LIST (CL)
  - If the state of a new node is in CL, then discard the node

Avoiding Revisited States

Avoiding Revisited States

- Depth-first search:

  Solution 1:
  - Store all generated states in CL

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

Avoiding Revisited States

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

  Solution 2:

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

  Solution 2:

  - Store all states associated with nodes in current path in CL

Avoiding Revisited States

- Depth-first search:

  Solution 1:
  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

  Solution 2:
  - Store all states associated with nodes in current path in CL
  - If the state of a new node is in CL, then discard the node

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

  Solution 2:

  - Store all states associated with nodes in current path in CL
  - If the state of a new node is in CL, then discard the node

  → Only avoids loops

Avoiding Revisited States

- Depth-first search:

  Solution 1:

  - Store all generated states in CL
  - If the state of a new node is in CL, then discard the node

  → Same space complexity as breadth-first !

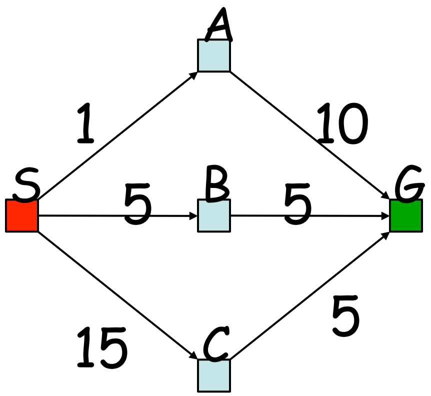
  Solution 2:

  - Store all states associated with nodes in current path in CL
  - If the state of a new node is in CL, then discard the node

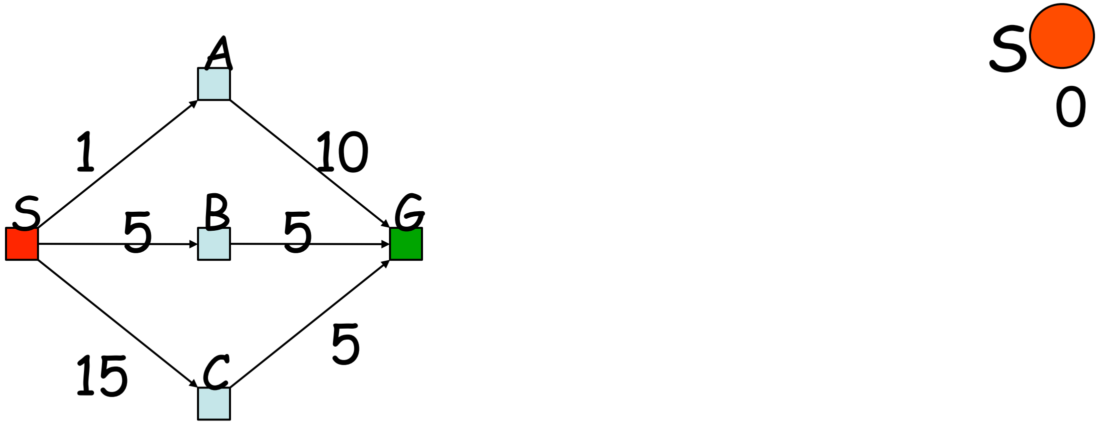
  → Only avoids loops

Uniform-Cost Search

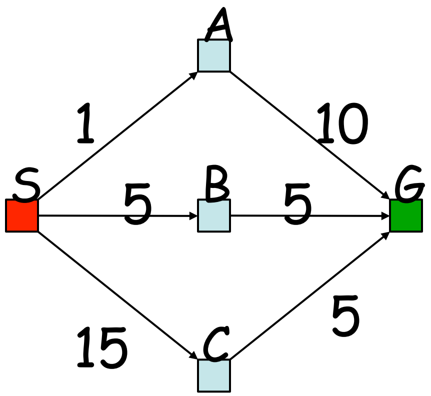
Uniform-Cost Search

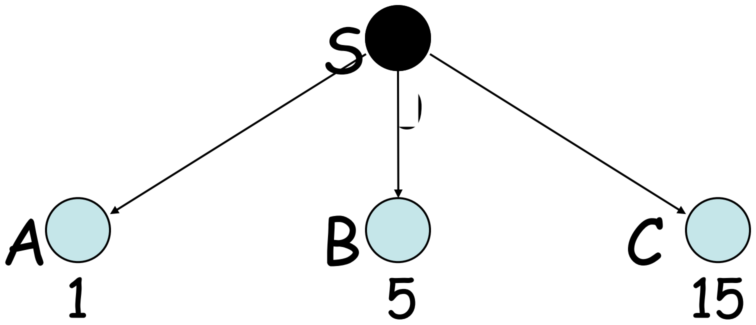


Uniform-Cost Search

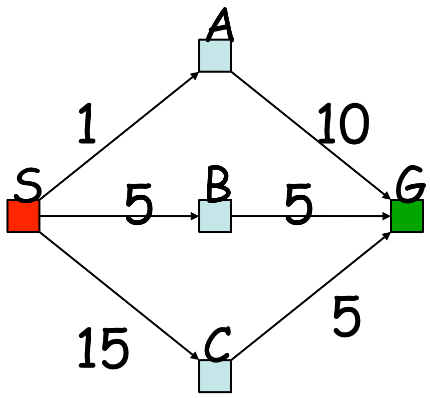


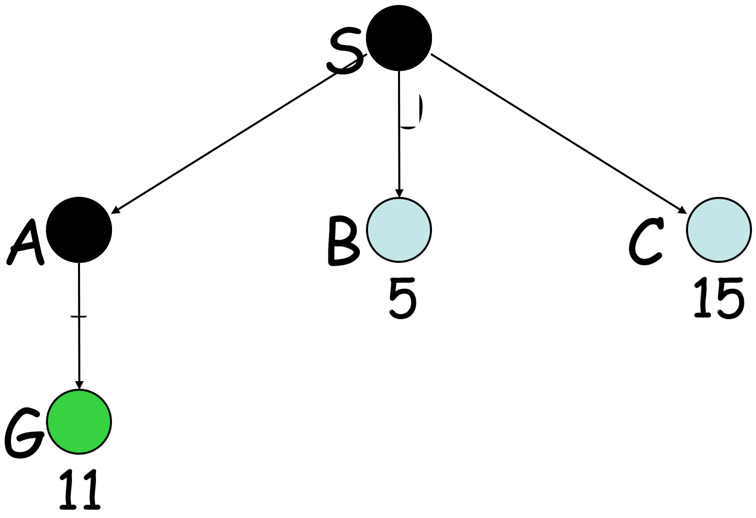
Uniform-Cost Search



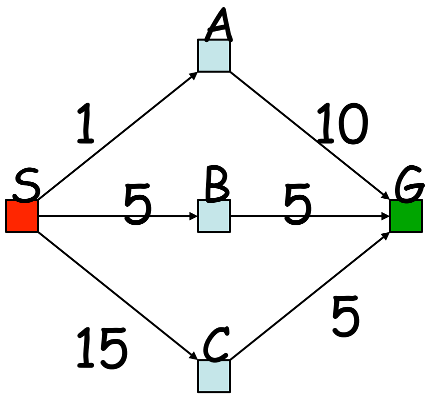


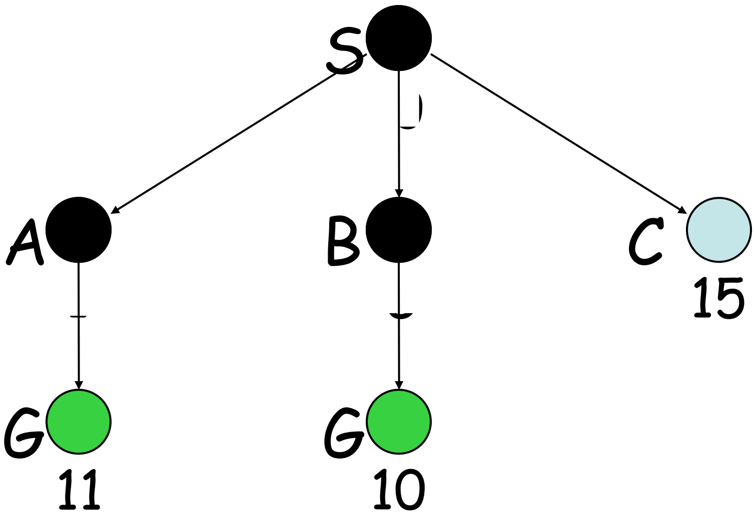
Uniform-Cost Search





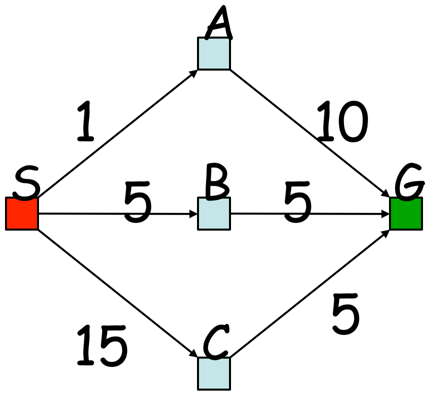
Uniform-Cost Search

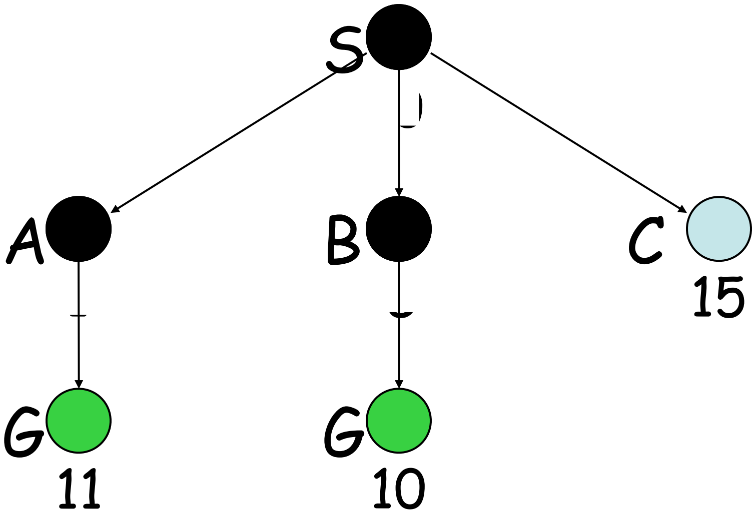




Uniform-Cost Search

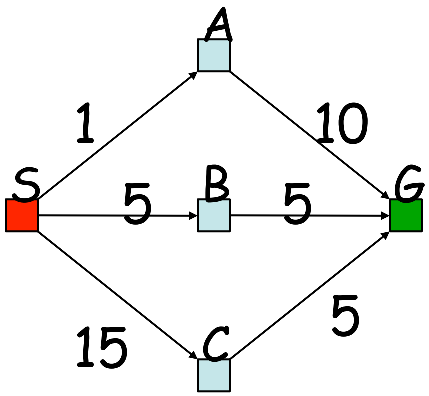
- Each arc has some cost c ≥ ε > 0

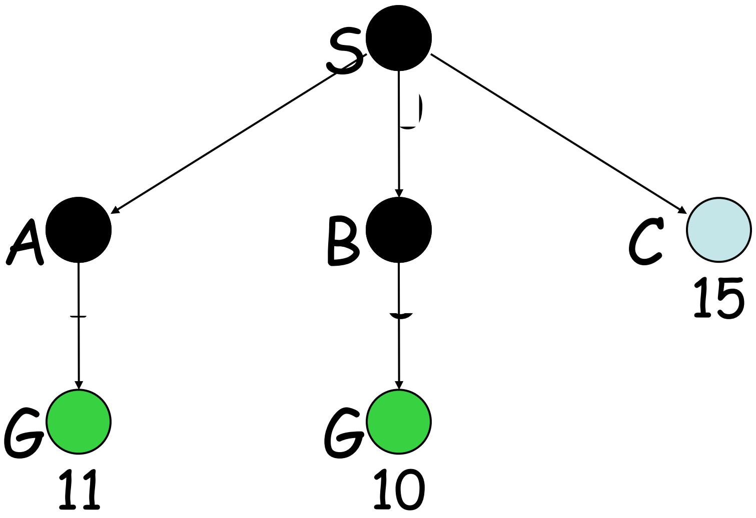




Uniform-Cost Search

- Each arc has some cost c ≥ ε > 0
- The cost of the path to each node N is

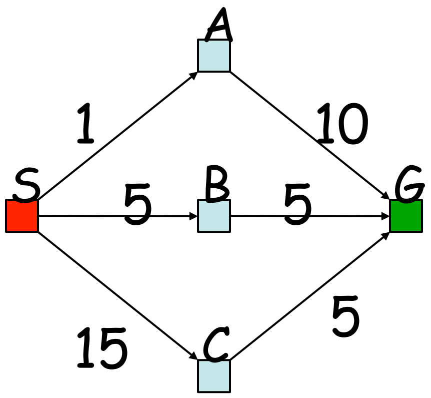


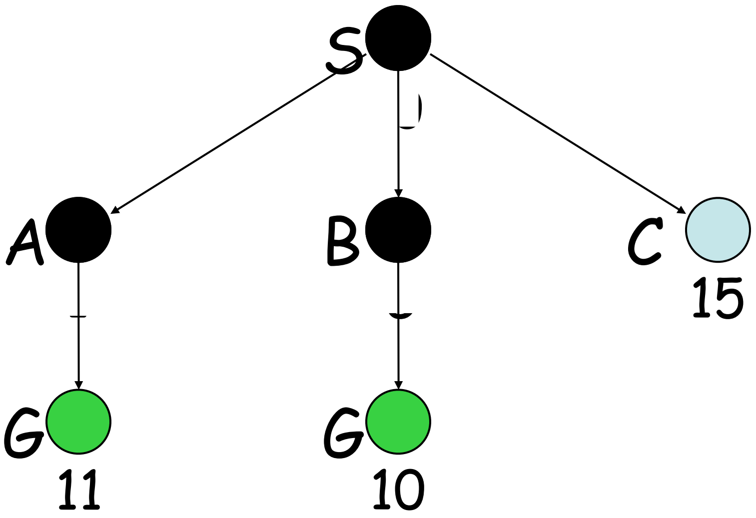


Uniform-Cost Search

- Each arc has some cost $c \geq \varepsilon > 0$
- The cost of the path to each node N is

$$g(N) = \Sigma \text{ costs of arcs}$$

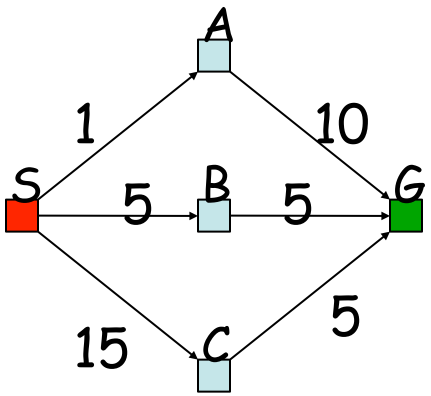


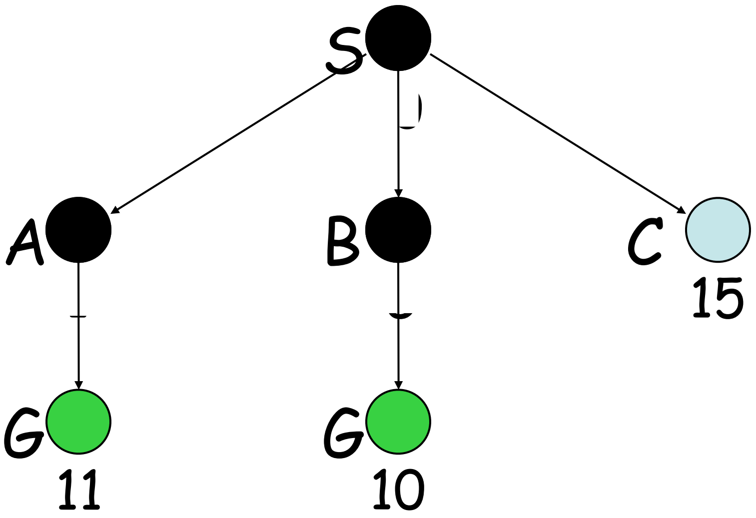


Uniform-Cost Search

- Each arc has some cost c ≥ ε > 0
- The cost of the path to each node N is

  g(N) = Σ costs of arcs
- The goal is to generate a solution path of minimal cost

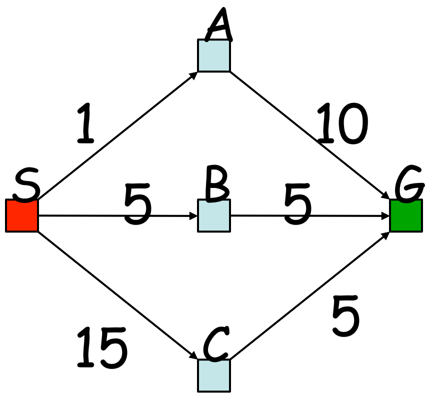


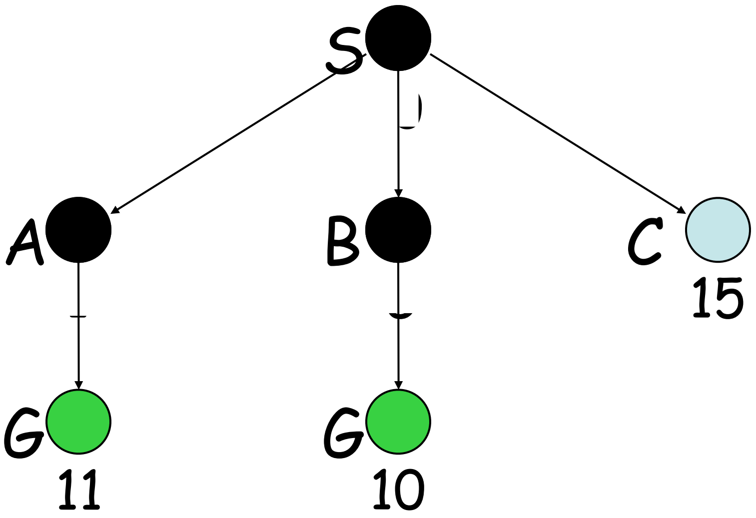


Uniform-Cost Search

- Each arc has some cost c ≥ ε > 0
- The cost of the path to each node N is

  $g(N) = \Sigma$ costs of arcs
- The goal is to generate a solution path of minimal cost
- The nodes N in the queue OL are sorted in increasing g(N)



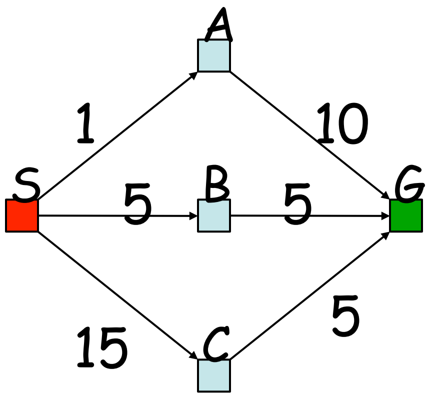


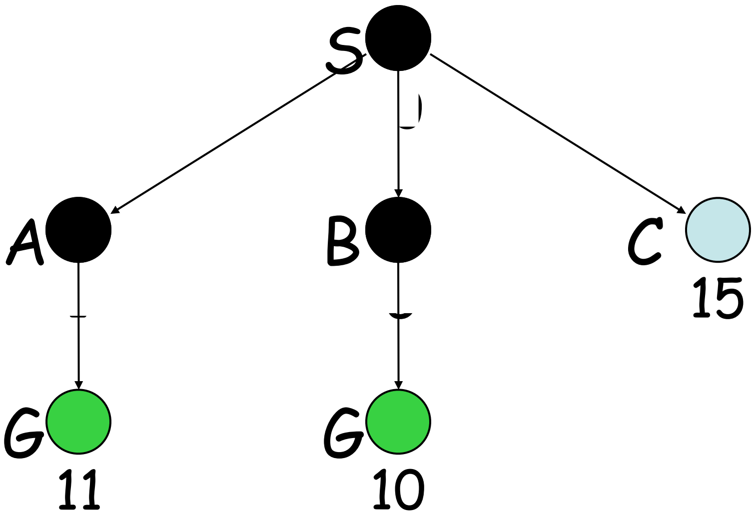
Uniform-Cost Search

- Each arc has some cost c ≥ ε > 0
- The cost of the path to each node N is

  g(N) = Σ costs of arcs

- The goal is to generate a solution path of minimal cost
- The nodes N in the queue OL are sorted in increasing g(N)





- Need to modify search algorithm

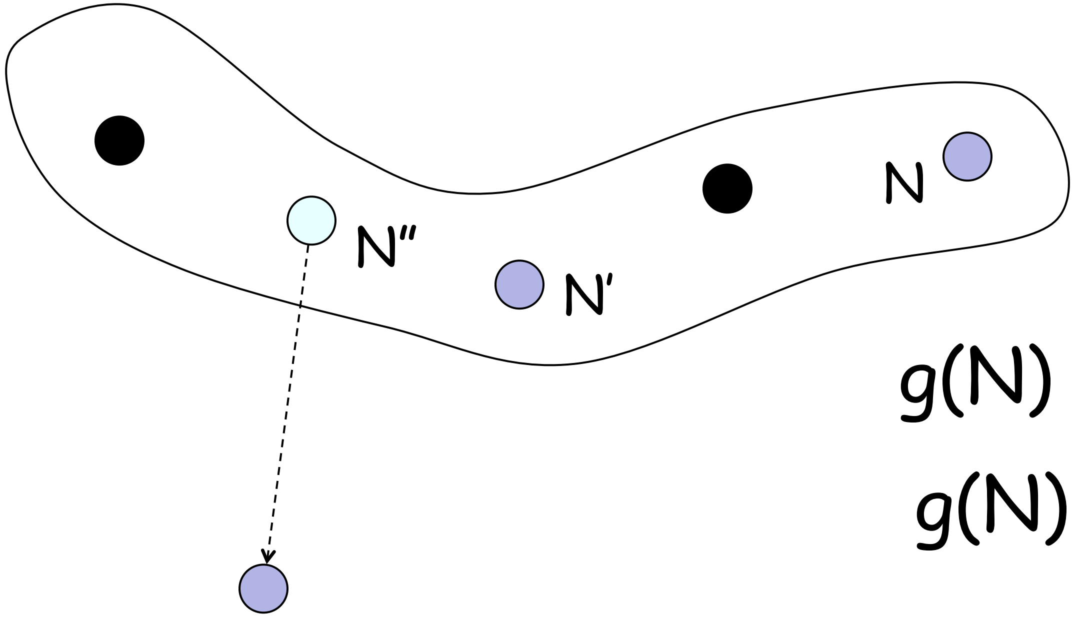
Search Algorithm #2

The goal test is applied to a node when this node is **expanded**, not when it is generated.

```
SEARCH#2
1. INSERT(initial-node,OL)
2. Repeat:
     a. If empty(OL) then return failure
     b. N ← REMOVE(OL)
     c. s ← STATE(N)
   ➤ d. If GOAL?(s) then return path or goal state
     e. For every state s’ in SUCCESSORS(s)
          i.  Create a new node N’ as a child of N
          ii. INSERT(N’,OL)
```

Avoiding Revisited States in Uniform-Cost Search

- For any state S, when the first node N such that STATE(N) = S is expanded, the path to N is the best path from the initial state to S



$$g(N) \leq g(N')$$

$$g(N) \leq g(N'')$$

Avoiding Revisited States in Uniform-Cost Search

- For any state S, when the first node N such that STATE(N) = S is expanded, the path to N is the best path from the initial state to S

- So:
  - When a node is **expanded**, store its state into CL
  - When a new node N is generated:
    - If STATE(N) is in CL, discard N
    - If there exits a node N' in OL such that STATE(N') = STATE(N), discard the node -- N or N' -- with the highest-cost path