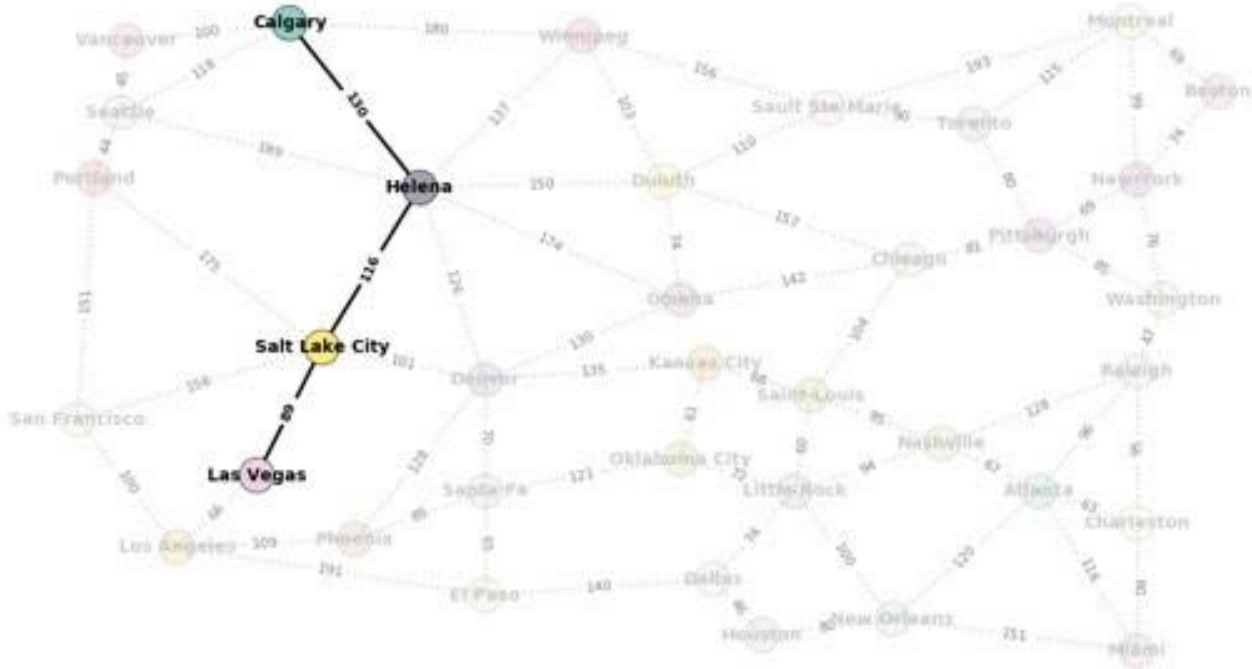
Artificial Intelligence

Heuristic

(Informed) Search



Heuristic (Informed) search

Use domain knowledge!

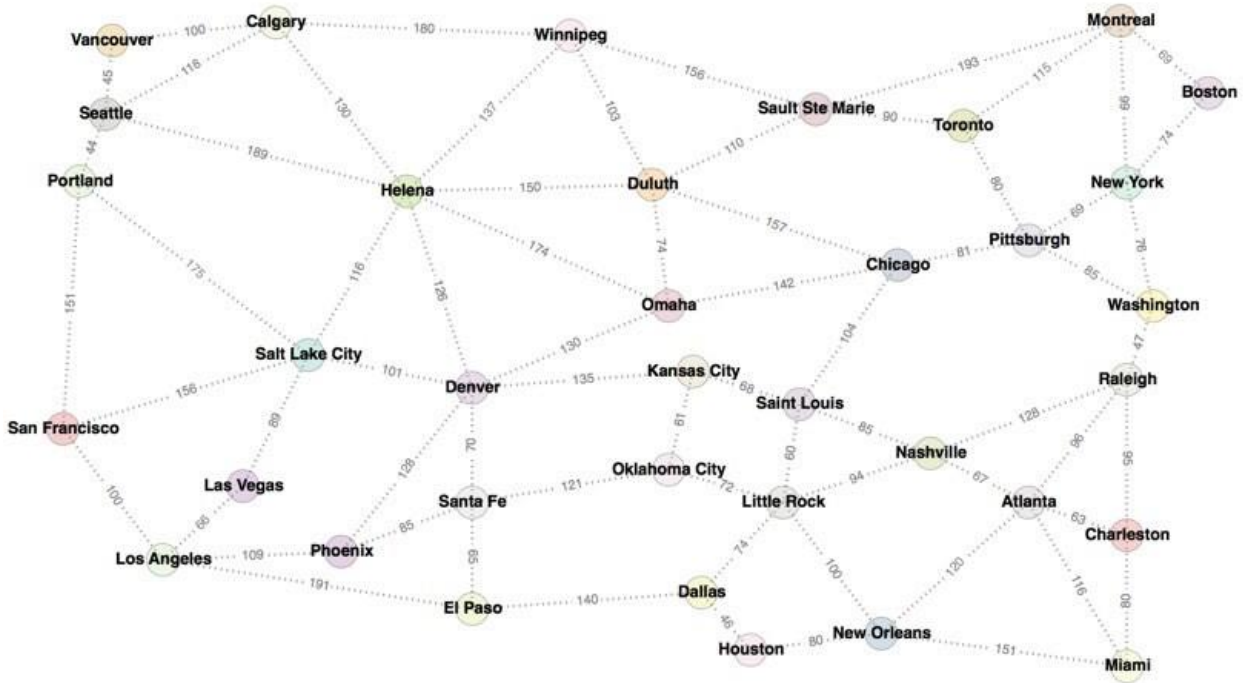
- Are we getting close to the goal?
- Use a heuristic function that estimates how close a state is to the goal
- A heuristic does NOT have to be perfect!

Informed search

Use domain knowledge!

- Are we getting close to the goal?
- Use a heuristic function that estimates how close a state is to the goal
- A heuristic does NOT have to be perfect!
- Example of strategies:
- **Greedy best-first search**
- **A* search**

Informed search



Atlanta	272
Boston	240
Calgary	334
Charleston	322
Chicago	107
Dallas	303
Denver	270
Duluth	110
El Paso	370
Helena	254
Houston	332
Kansas City	176
Las Vegas	418
Little Rock	240
Los Angeles	484
Miami	389
Montreal	193
Nashville	221
New Orleans	322
New York	195
Oklahoma City	237
Omaha	150
Phoenix	396
Pittsburgh	152
Portland	452
Raleigh	251
Saint Louis	180
Salt Lake City	344
San Francisco	499
Santa Fe	318
Sault Ste Marie	0
Seattle	434
Toronto	90
Vancouver	432
Washington	238
Winnipeg	156

Heuristic!

The distance is the straight line distance. The goal is to get to Sault Ste Marie, so all the distances are from each city to Sault Ste Marie.

Greedy Best-first Search

- Evaluation function $h(n)$ (*h*euristic)
- $h(n)$ estimates the cost from n to the closest goal
- Example: $h_{SLD}(n)$ = straight-line distance from n to Sault Ste Marie
- Greedy search expands the node that **appears** to be closest to goal

Greedy search

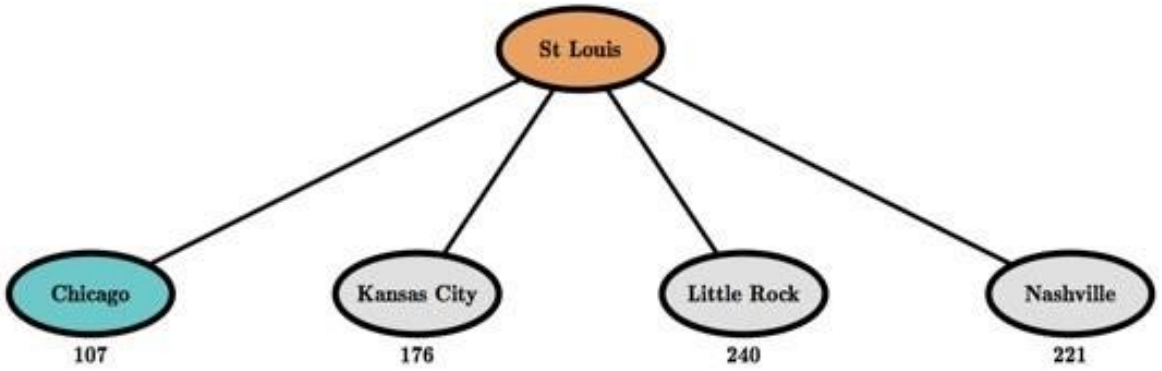
```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Greedy search example

The initial state:

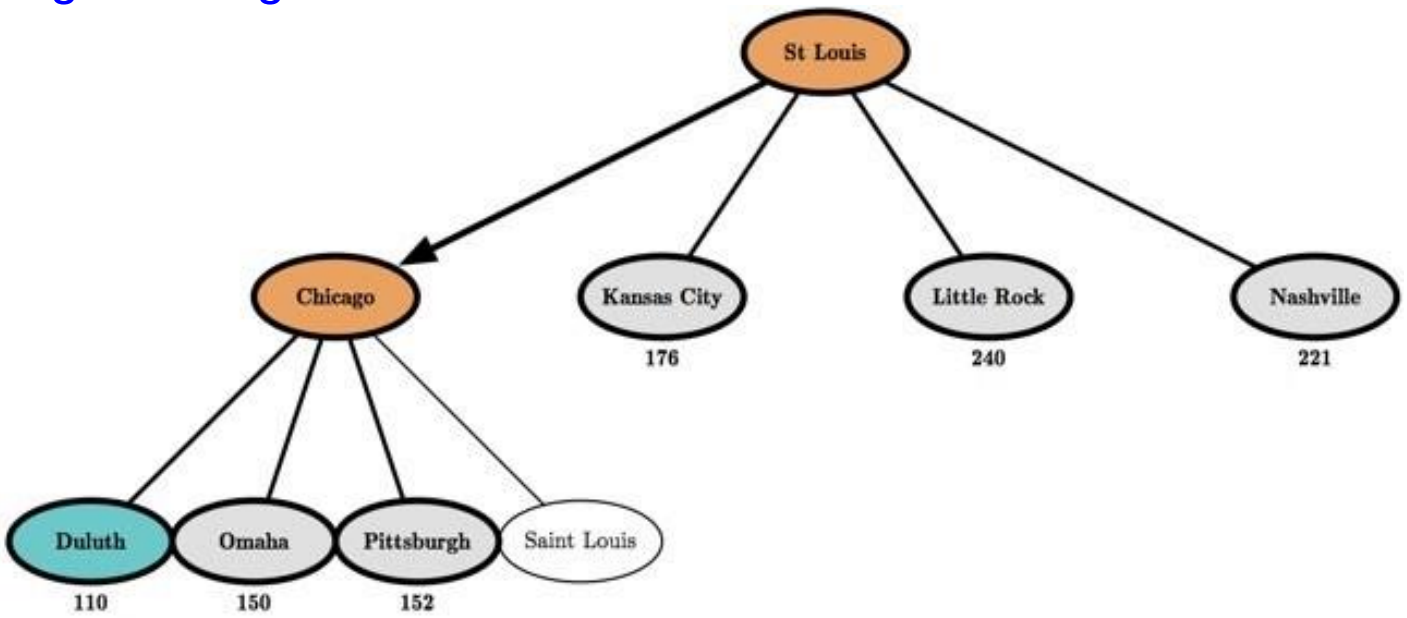
Greedy search example

After expanding St Louis:



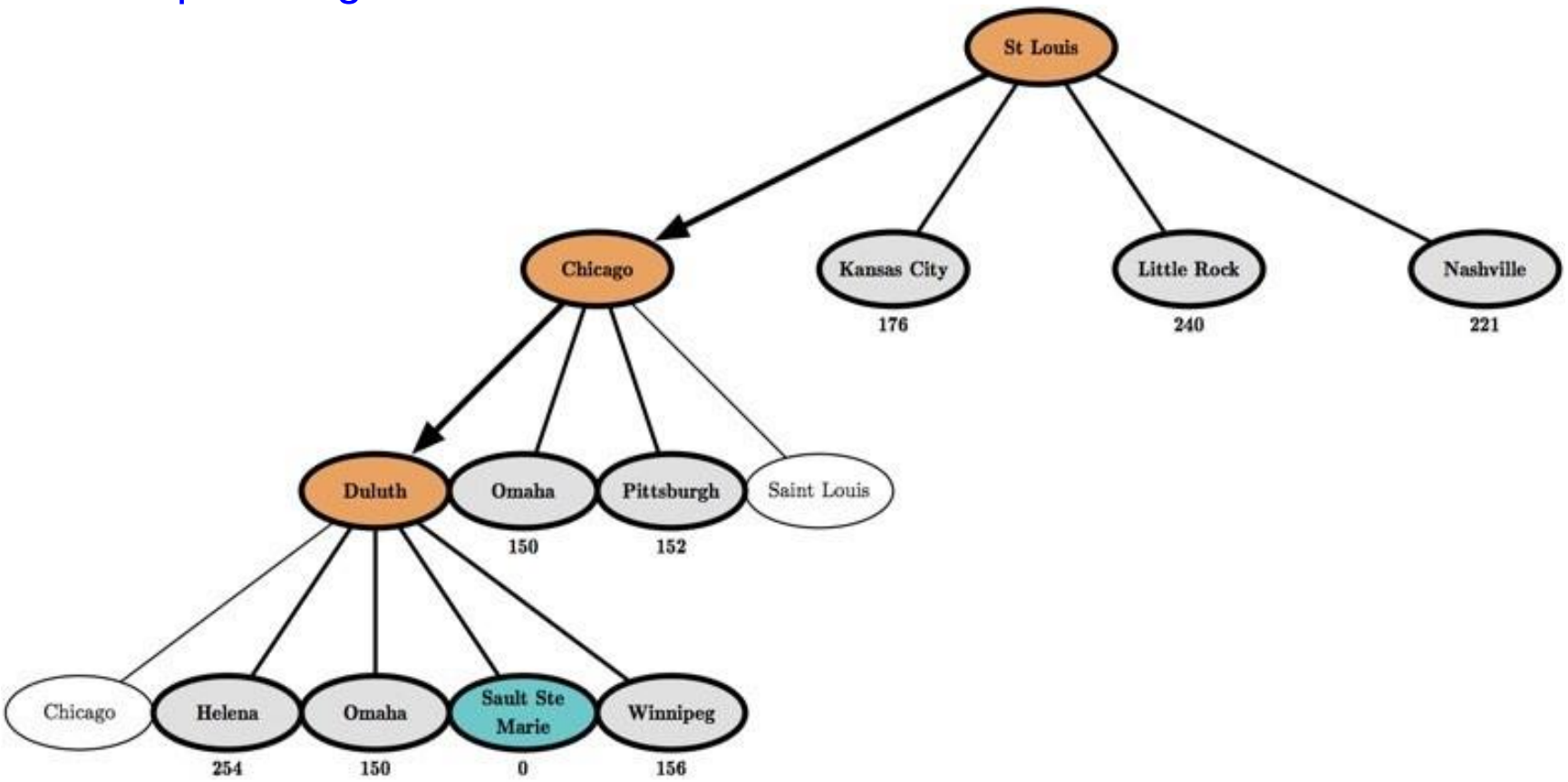
Greedy search example

After expanding Chicago:



Greedy search example

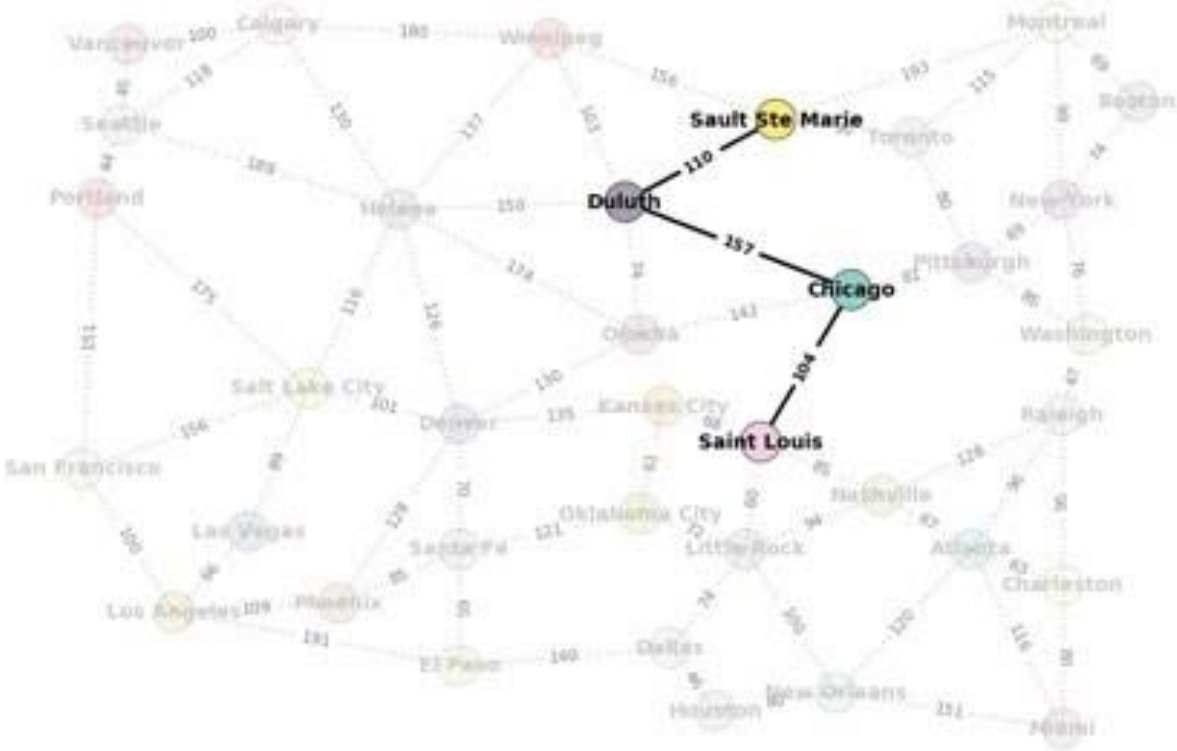
After expanding Duluth:



Examples using the map

Start: Saint Louis

Goal: Sault Ste Marie

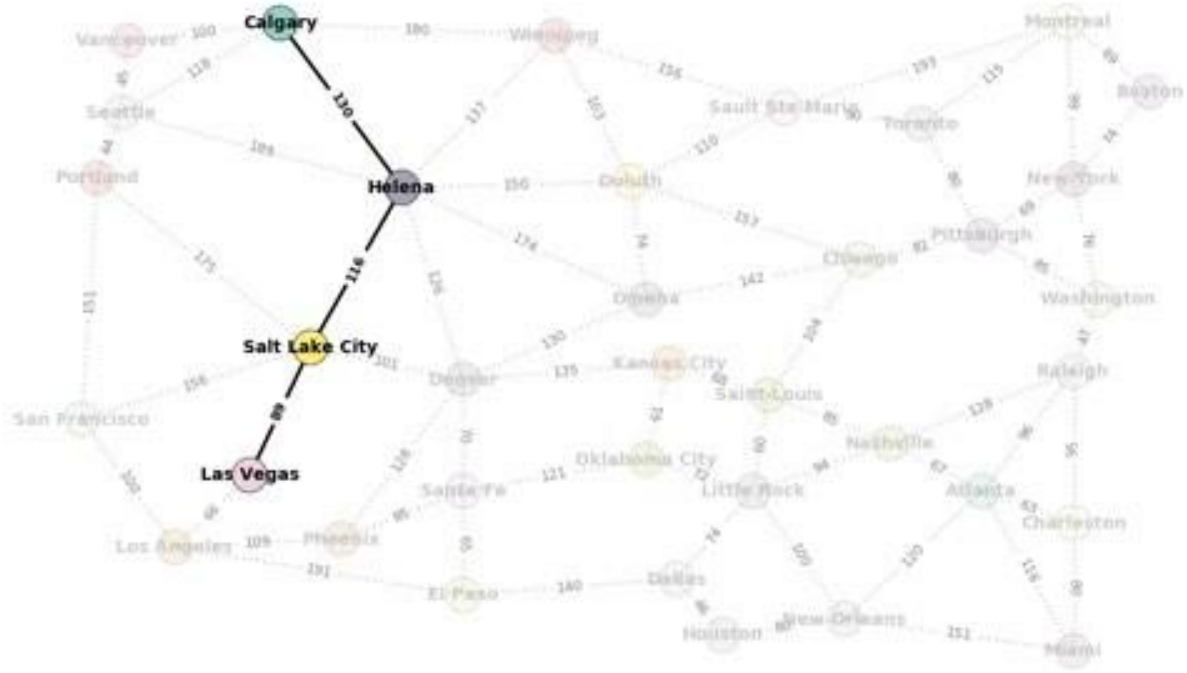


Greedy search

Examples using the map

Start: Las Vegas

Goal: Calgary



Greedy search

Greedy Best-First Search Criteria

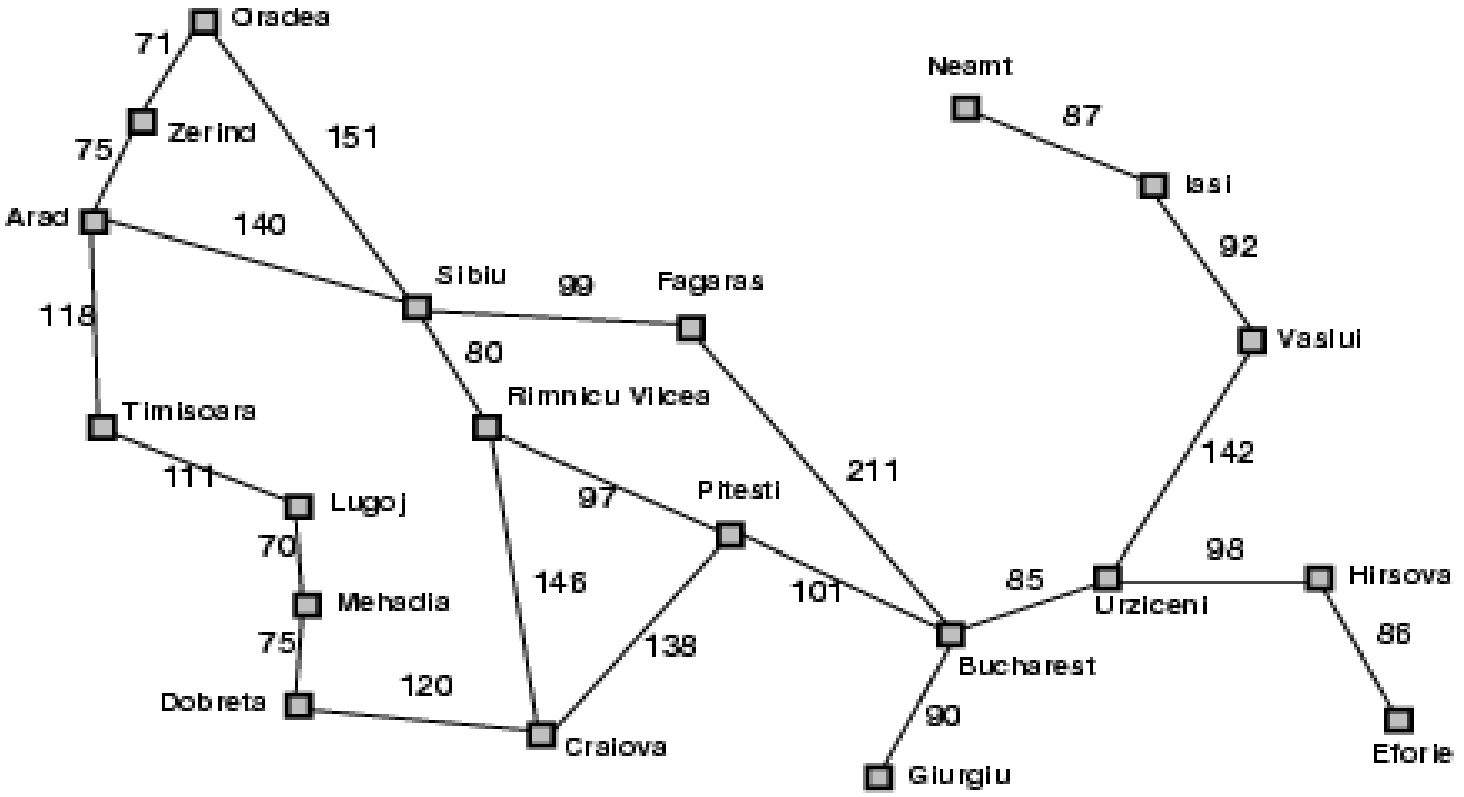
Think of an example!

Complete? **No – can get stuck in loops.**
Time: **$O(b^m)$, but a good heuristic can give dramatic improvement.**
Space: **$O(b^m)$ – keeps all nodes in memory.**
Optimal? **No**

Example: In the following graph, Greedy best-first search will yield:

Arad → Sibiu → Rimnicu

But Virea → Pitesti → Bucharest is shorter!



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search

- Minimize the total estimated solution cost
- Combines:
  - $g(n)$: cost to reach node n
  - $h(n)$: cost to get from n to the goal
  - $f(n) = g(n) + h(n)$

$f(n)$ is the estimated cost of the cheapest solution through n

A* search

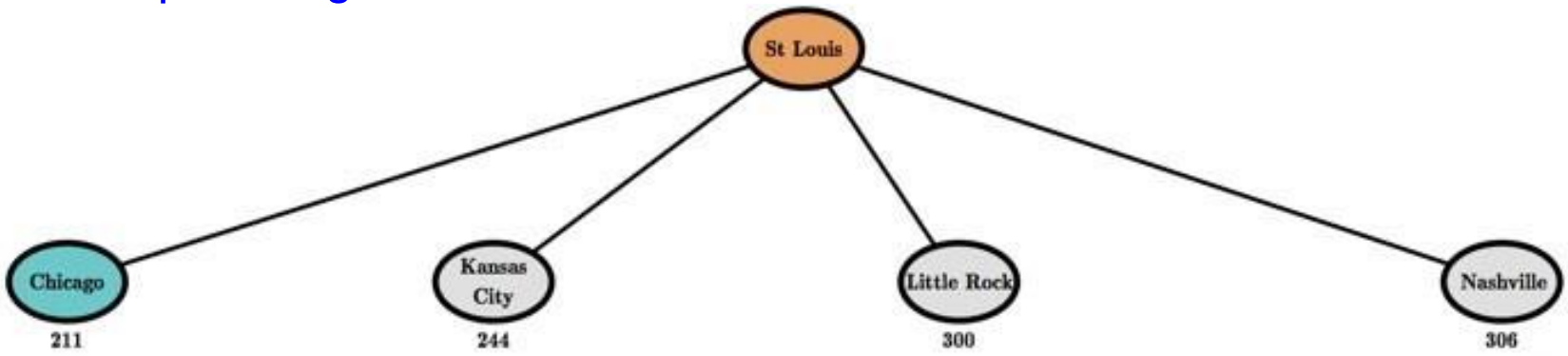
```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

A* search example

The initial state:

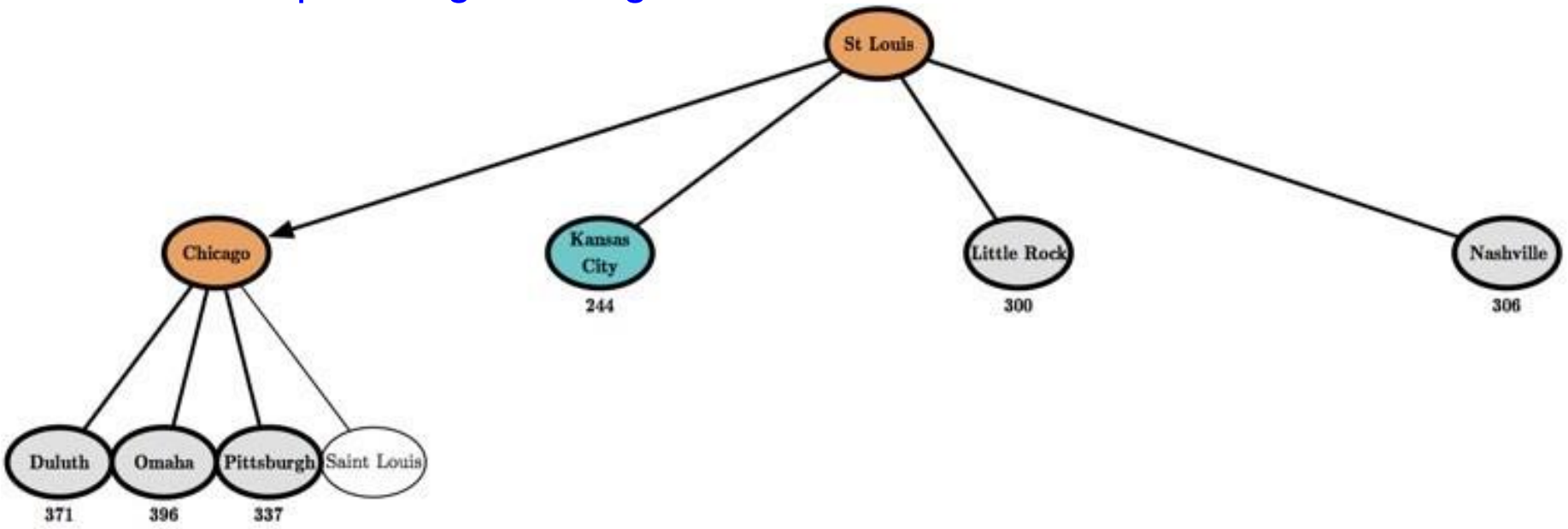
A* search example

After expanding St Louis:



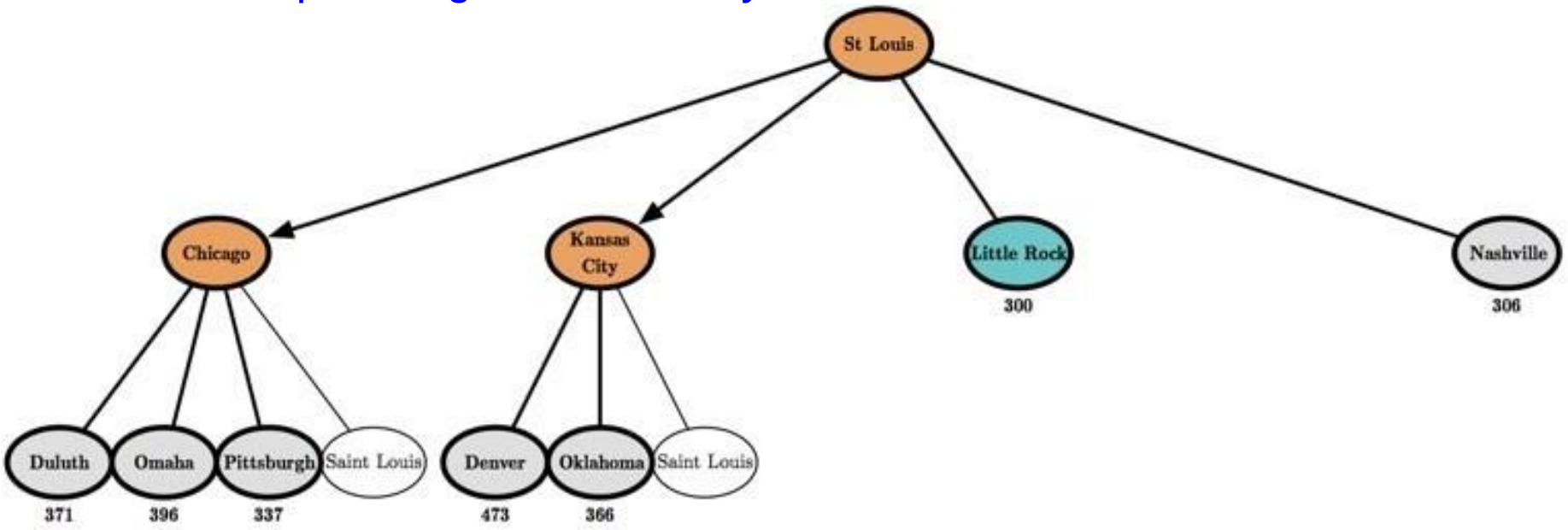
A* search example

After expanding Chicago:



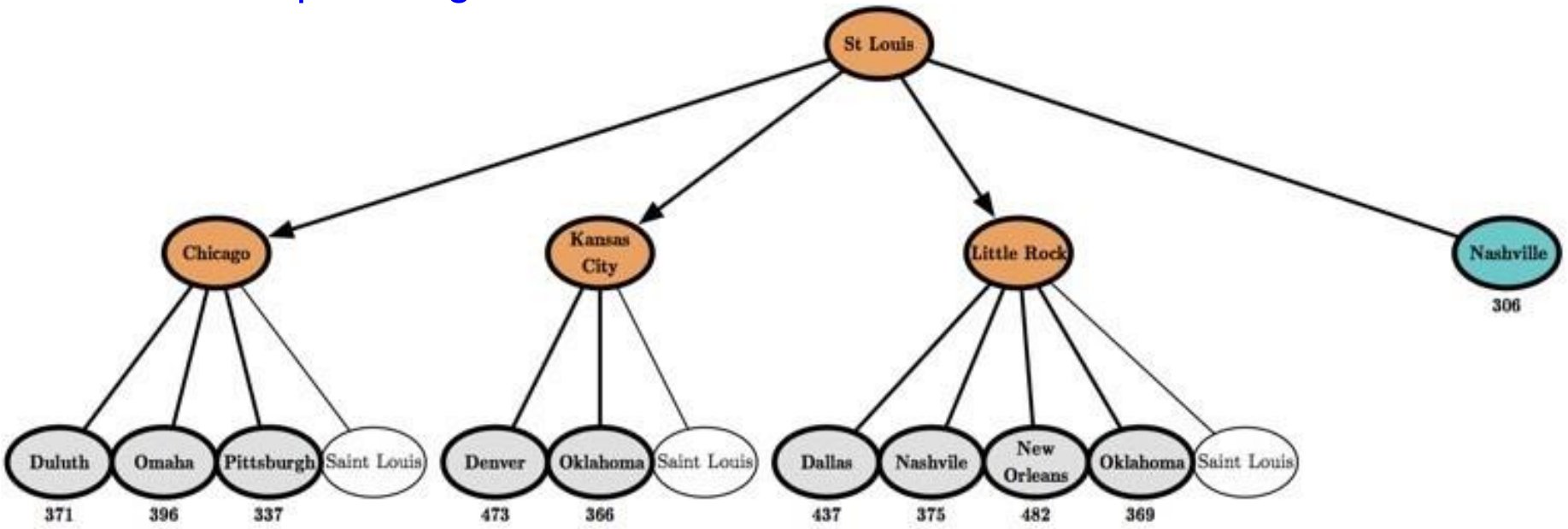
A* search example

After expanding Kansas City:



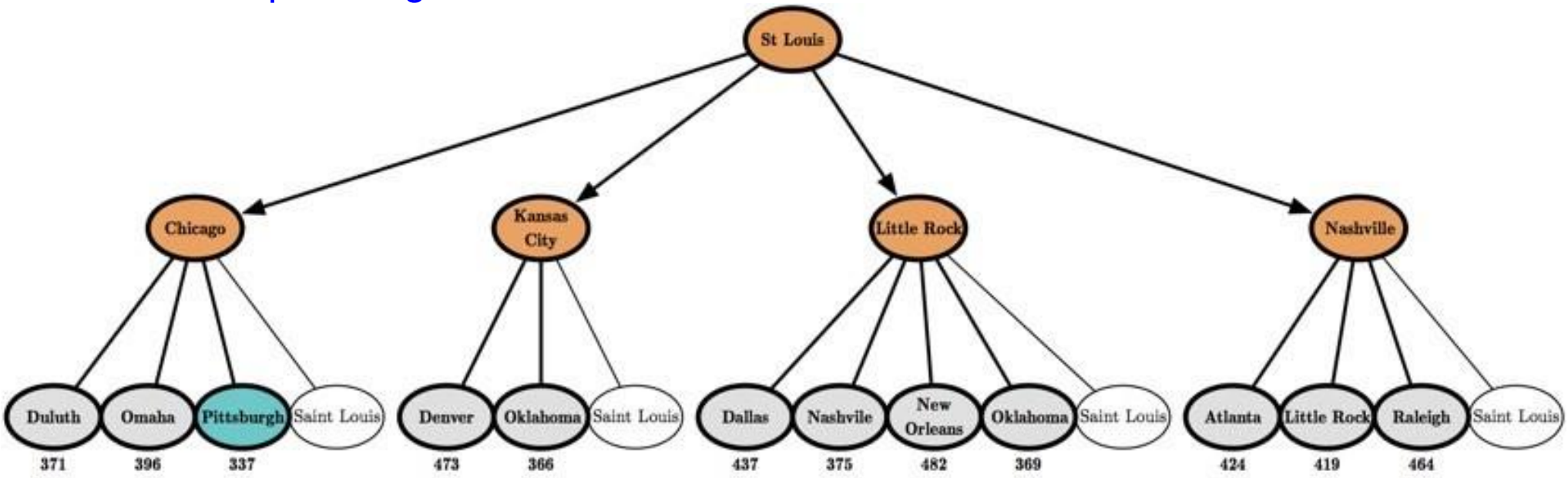
A* search example

After expanding Little Rock:



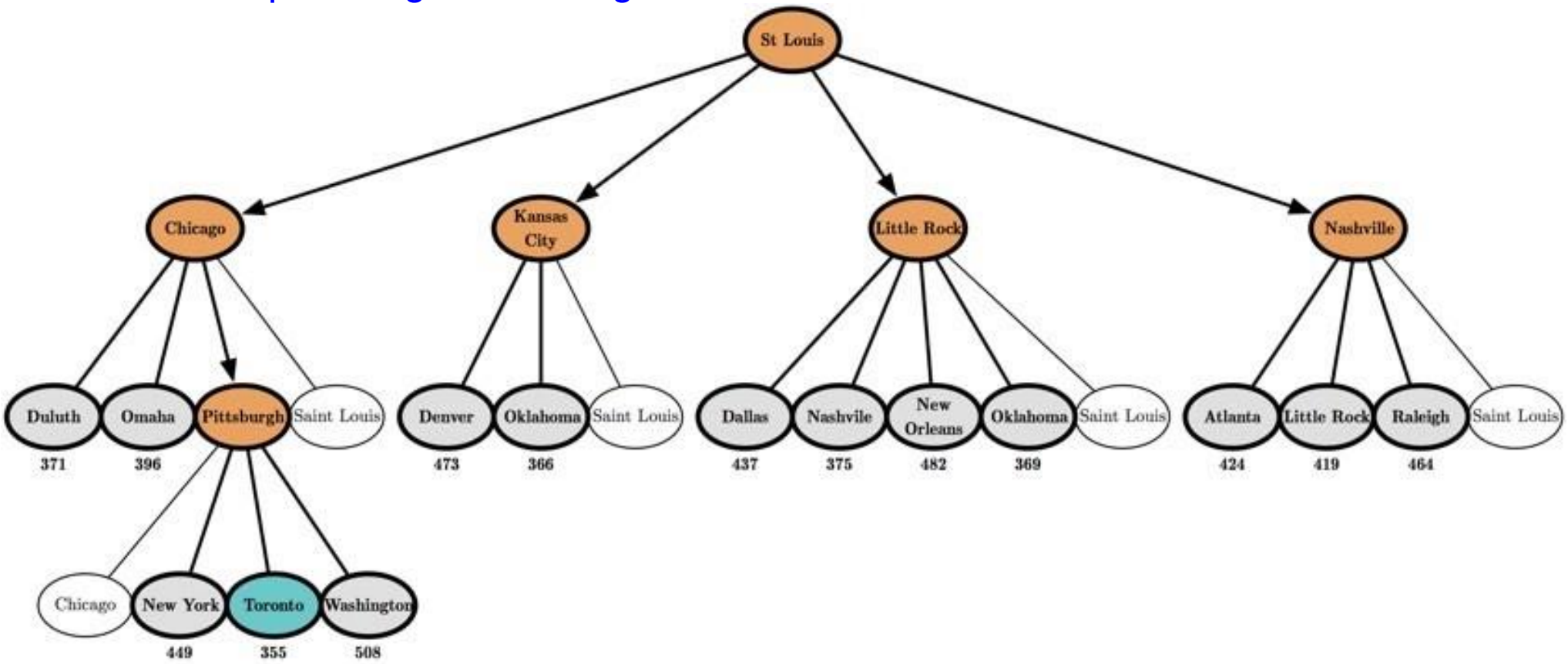
A* search example

After expanding Nashville:



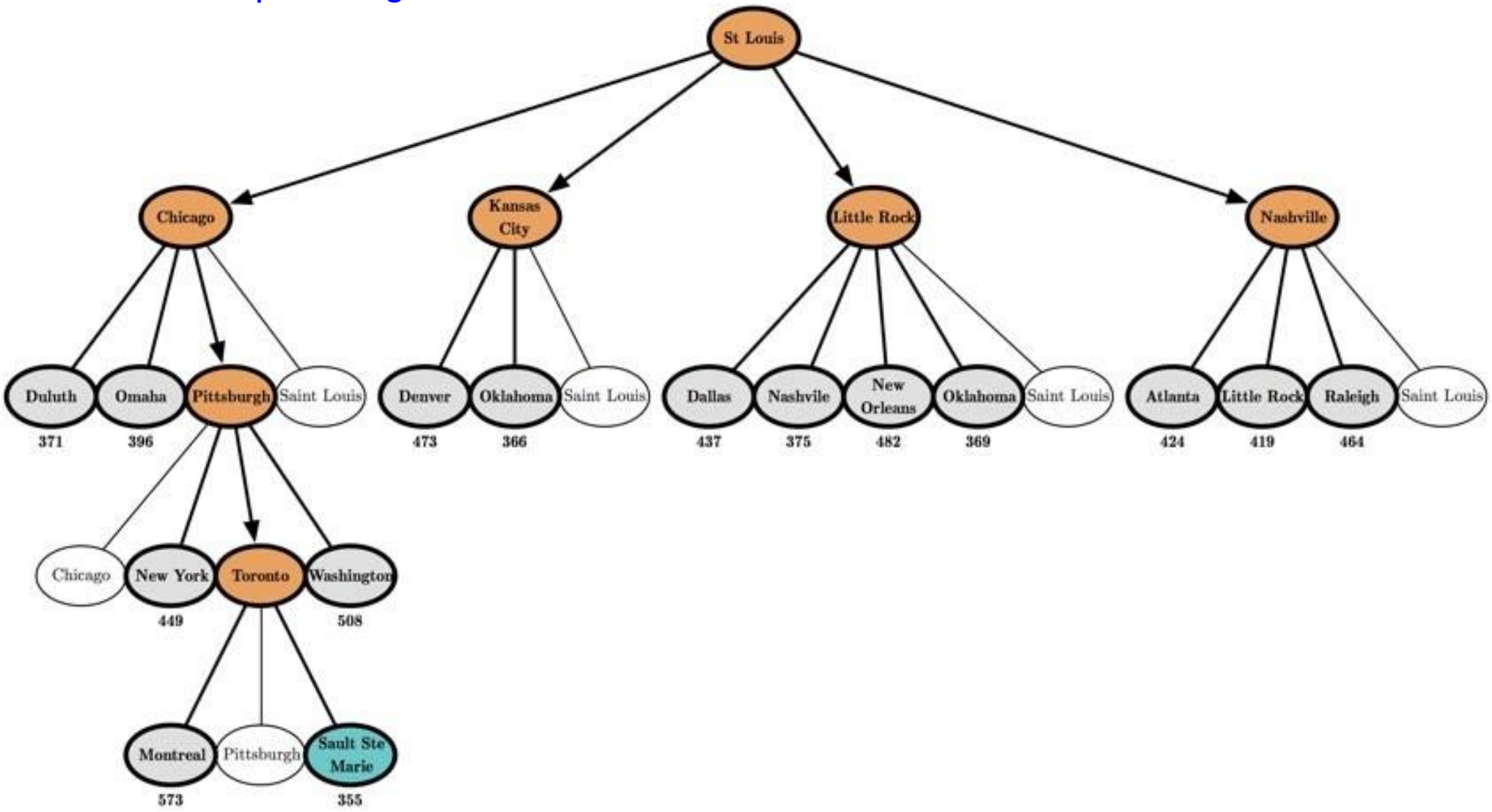
A* search example

After expanding Pittsburgh:



A* search example

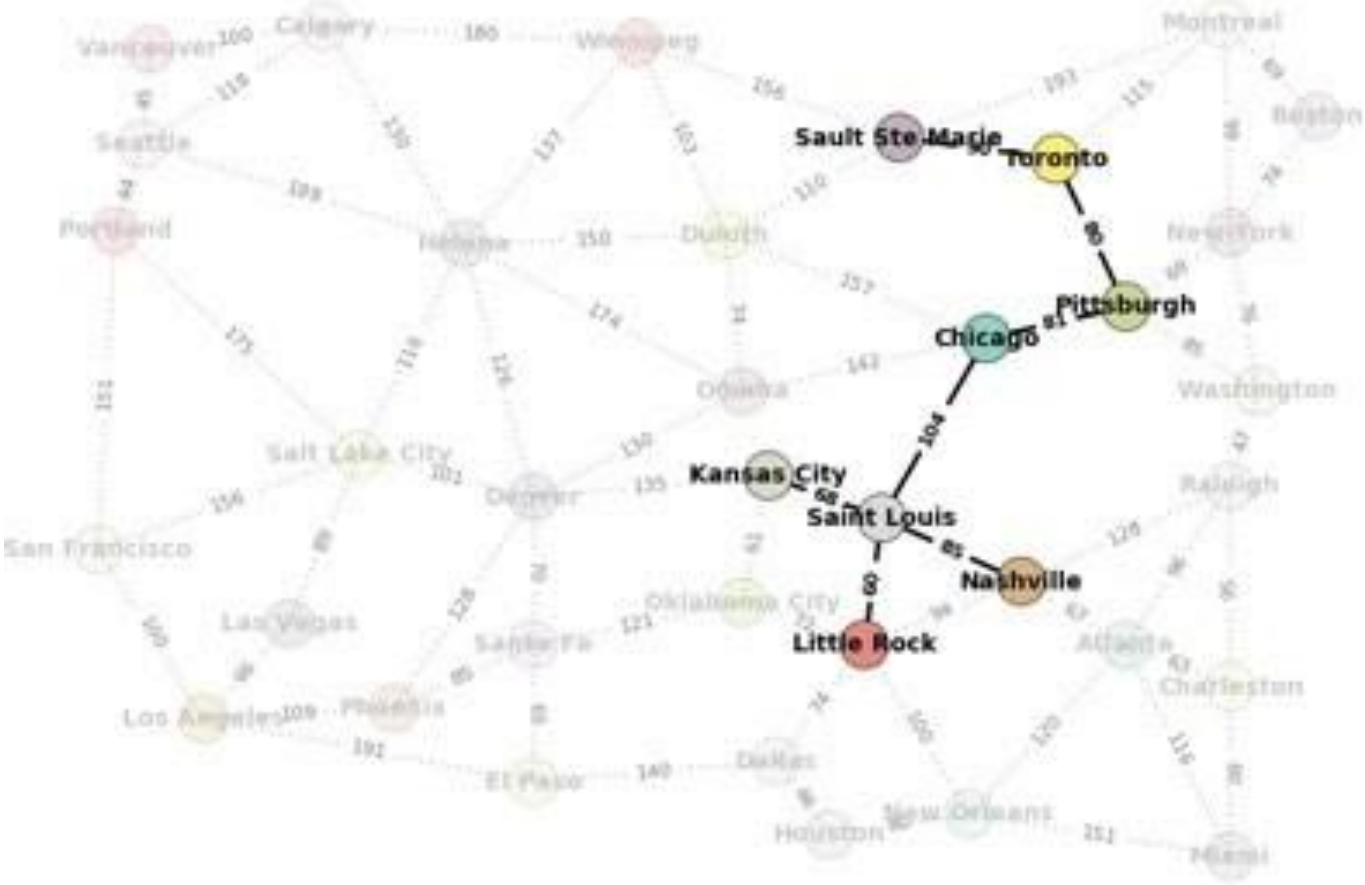
After expanding Toronto:



Examples using the map

Start: Saint Louis

Goal: Sault Ste Marie

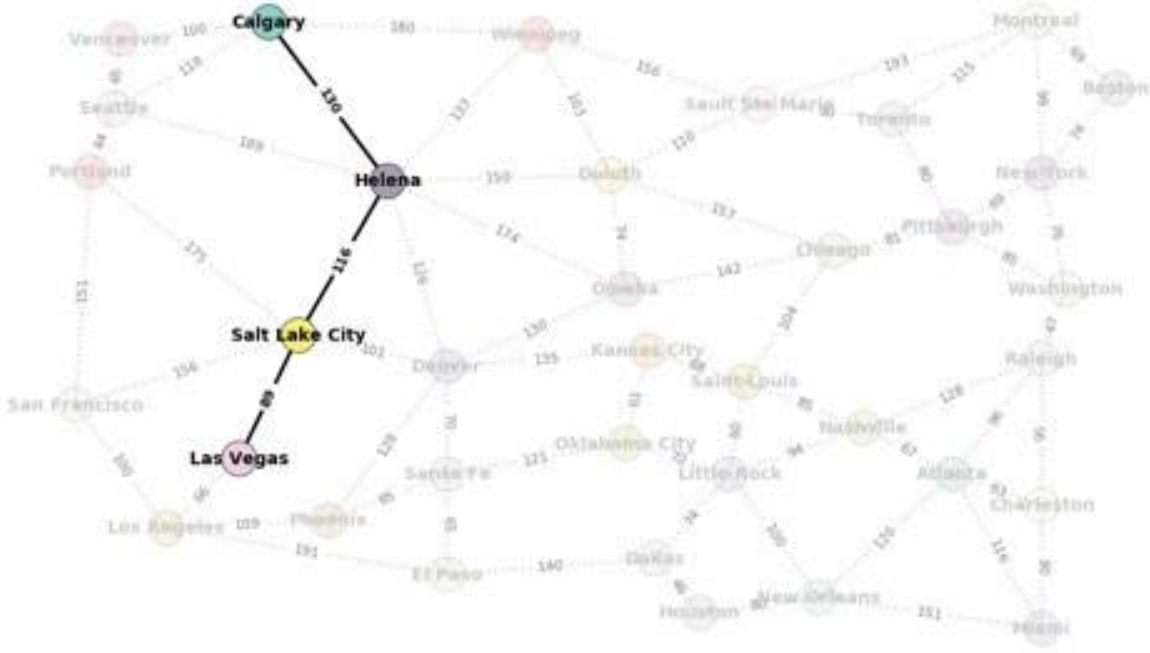


A*

Examples using the map

Start: Las Vegas

Goal: Calgary



A*

Admissible heuristics

A good heuristic can be powerful.

Only if it is of a “good quality”

Admissible heuristics

A good heuristic can be powerful.

Only if it is of a “good quality”

A good heuristic must be admissible.

Admissible heuristics

- An **admissible** heuristic never overestimates the cost to reach the goal, that is it is **optimistic**
- A heuristic h is admissible if for all

$$node\ n, h(n) \leq h^*(n)$$

  where h^* is true cost to reach the goal from n.

- h_{SLD} (used as a heuristic in the map example) is admissible because it is by definition the shortest distance between two points.

A* Optimality

If $h(n)$ is admissible, A* using tree search is optimal.

A* Optimality

If $h(n)$ is admissible, A* using tree search is optimal.

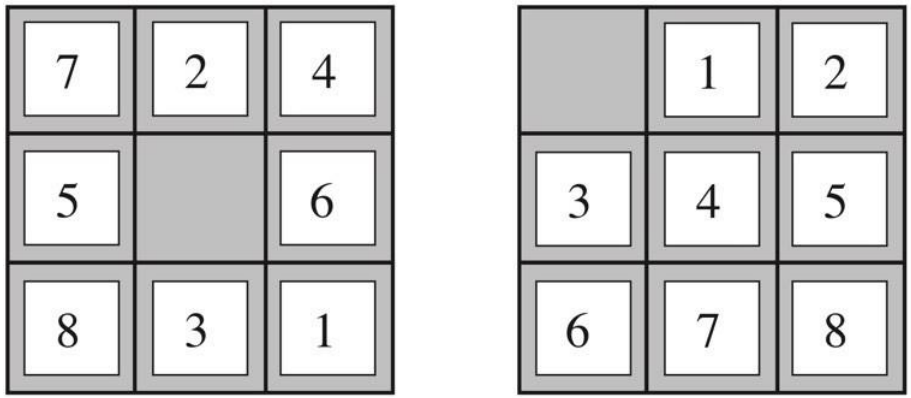
Rationale:

- Suppose tt_o is the optimal goal.
- Suppose tt_s is some suboptimal goal.
- Suppose n is an unexpanded node in the fringe such that n is on the shortest path to tt_o.
- $f(tt_s) = g(tt_s)$ since $h(tt_s) = 0$
  $f(tt_o) = g(tt_o)$ since $h(tt_o) = 0$
  $f(tt_s) > g(tt_o)$ since tt_s is suboptimal
  Then $f(tt_s) > f(tt_o)$ ...(1)
- $h(n) \leq h^*(n)$ since h is admissible
  $g(n) + h(n) \leq g(n) + h^*(n) = g(tt_o) = f(tt_o)$
  Then, $f(n) \leq f(tt_o)$ ...(2)
  From (1) and (2) $f(tt_s) > f(n)$
  so A* will never select tt_s during the search and hence A* is optimal.

A* search criteria

- **Complete**: Yes
- **Time**: exponential
- **Space**: keeps every node in memory, the biggest problem
- **Optimal**: Yes!

Heuristics



- The solution is 26 steps long.
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (sum of the horizontal and vertical distances).
- $h_1(n) = 8$
- Tiles 1 to 8 in the start state gives: $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$ which does not overestimate the true solution.

Search Algorithms: Recap

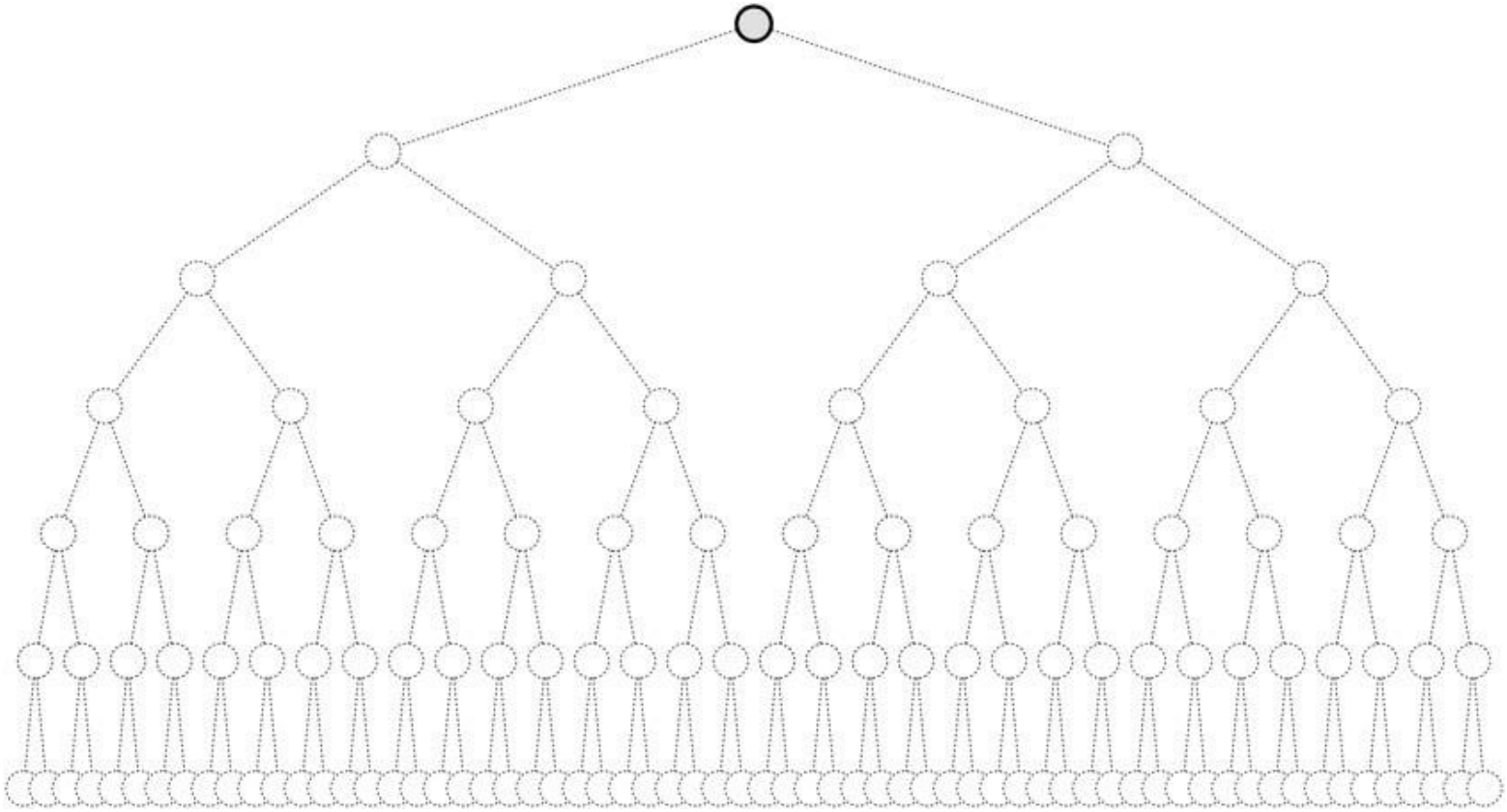
- **Uninformed Search**: Use no domain knowledge.

  BFS, DFS, DLS, IDS, UCS.

- **Informed Search**: Use a heuristic function that estimates how close a state is to the goal.

  Greedy search, A*

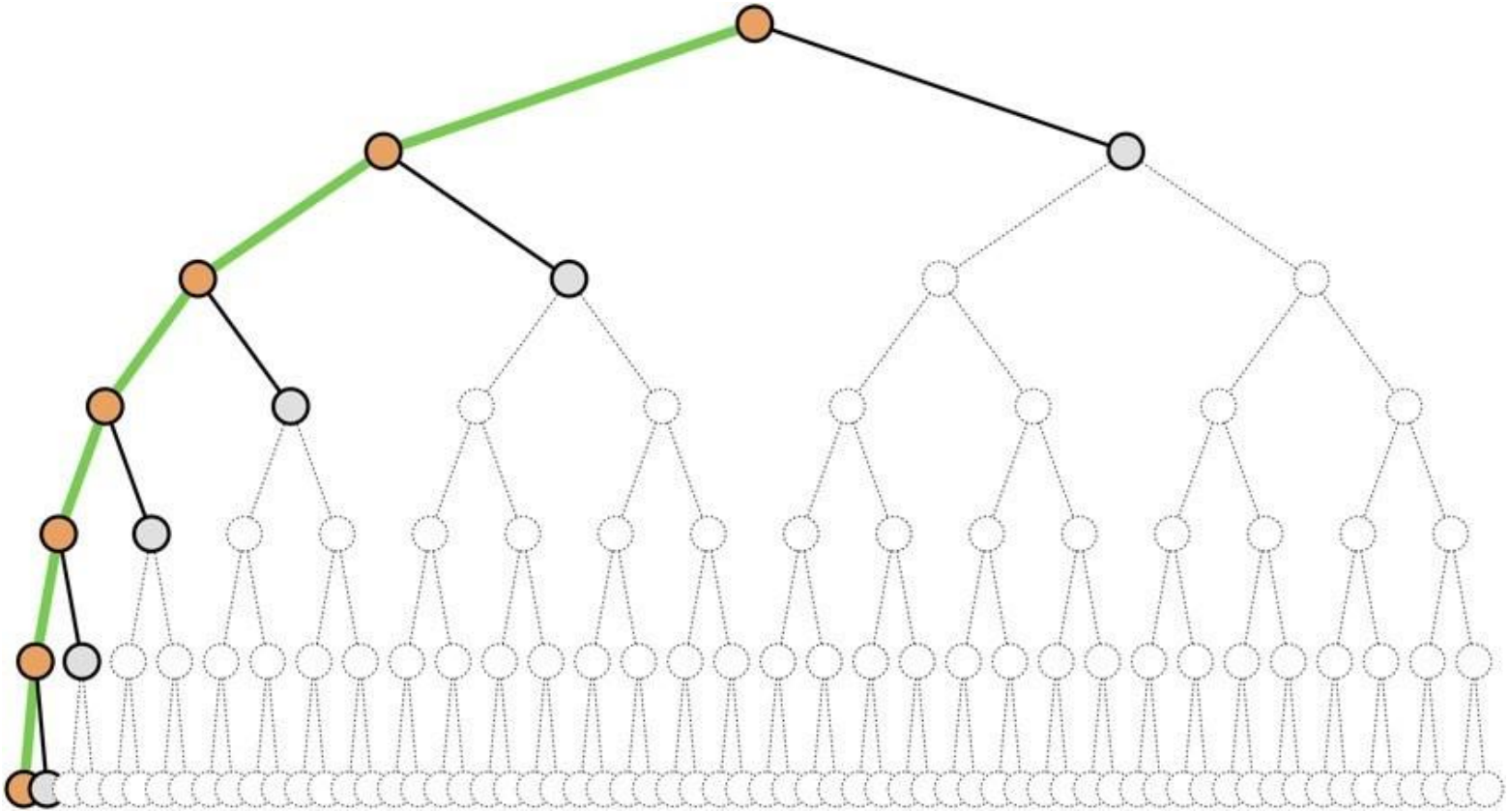
DFS



Searches branch by branch ...

... with each branch pursued to maximum depth.

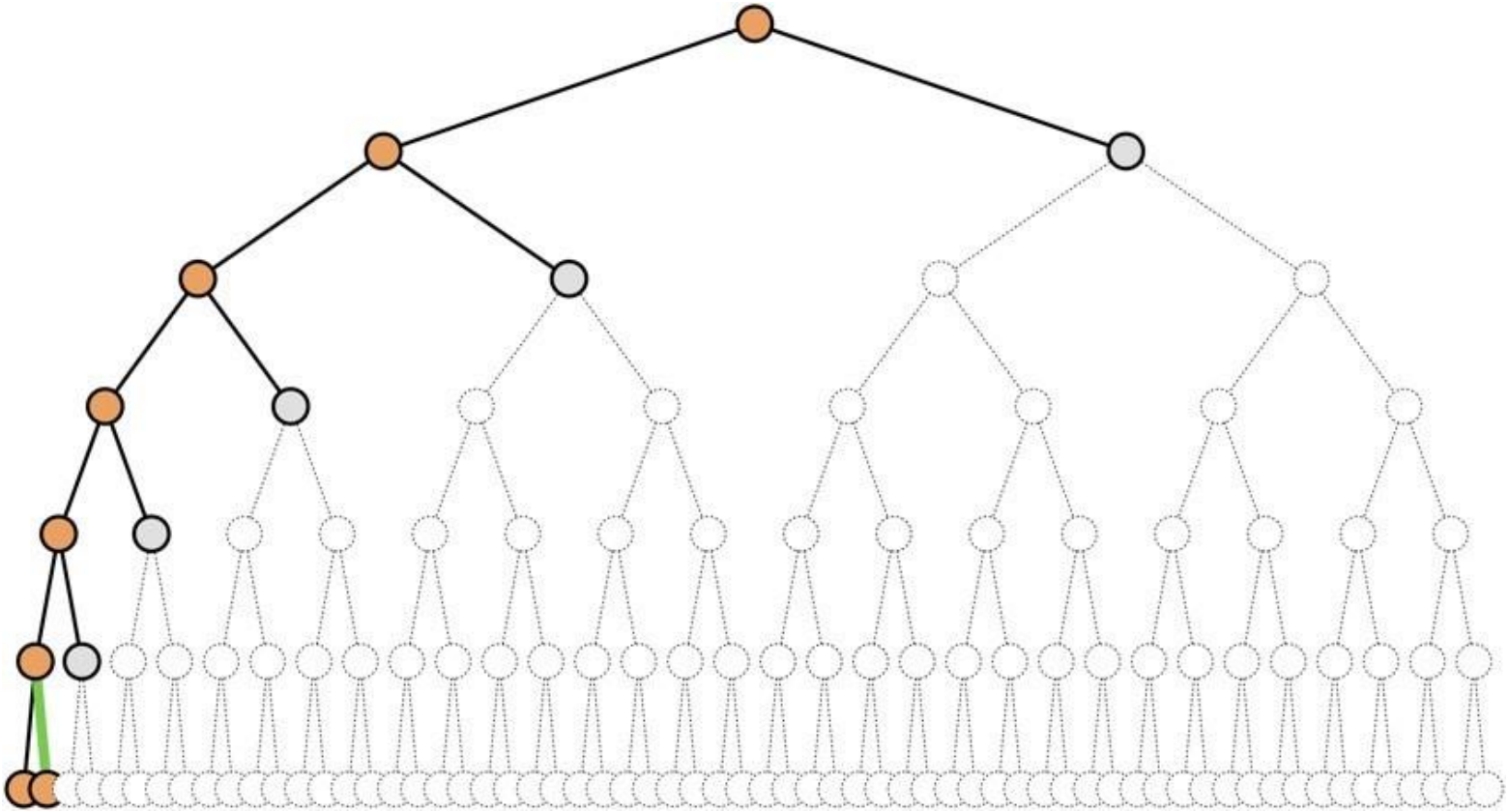
DFS



Searches branch by branch ...

... with each branch pursued to maximum depth.

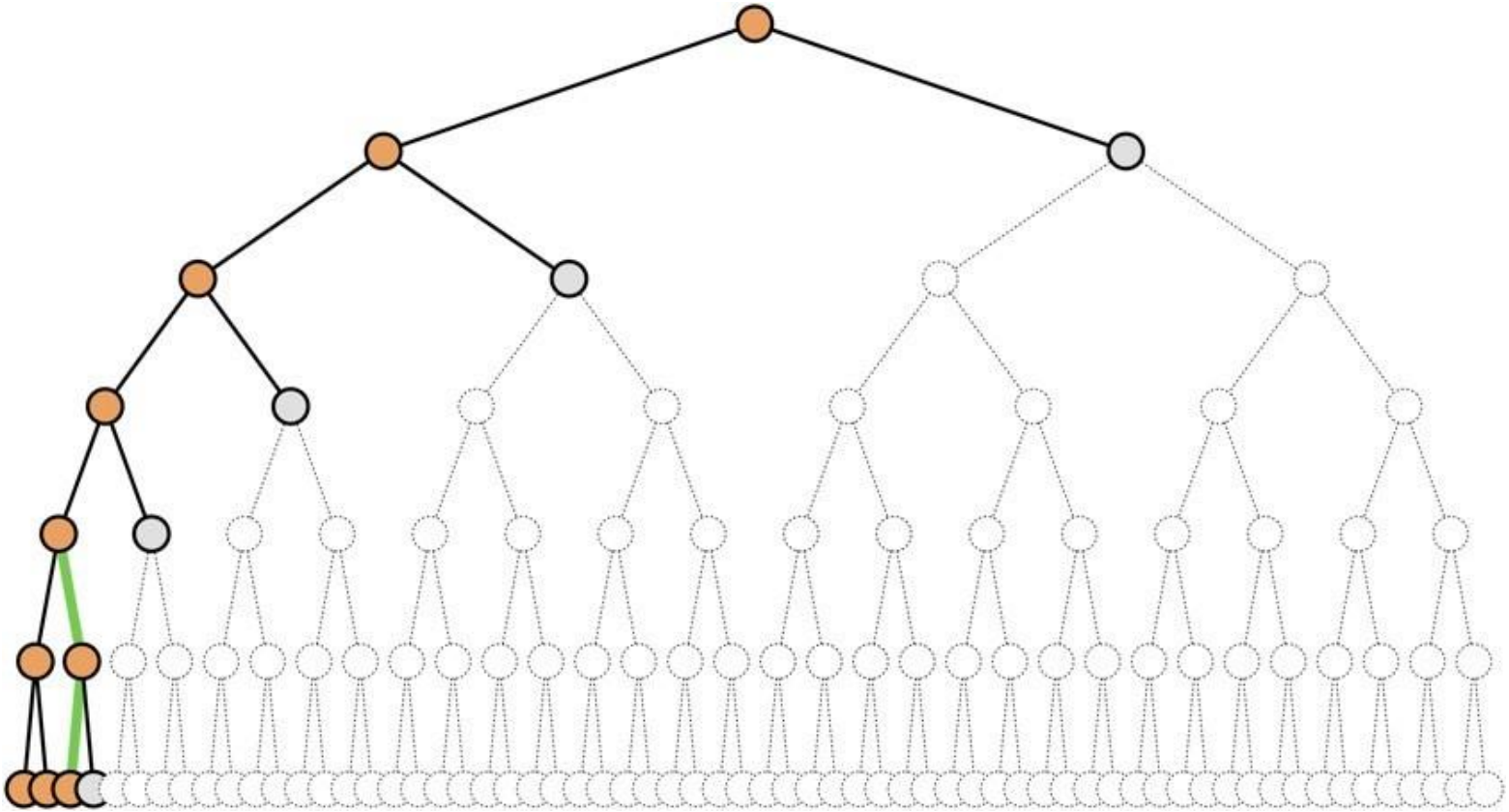
DFS



Searches branch by branch ...

... with each branch pursued to maximum depth.

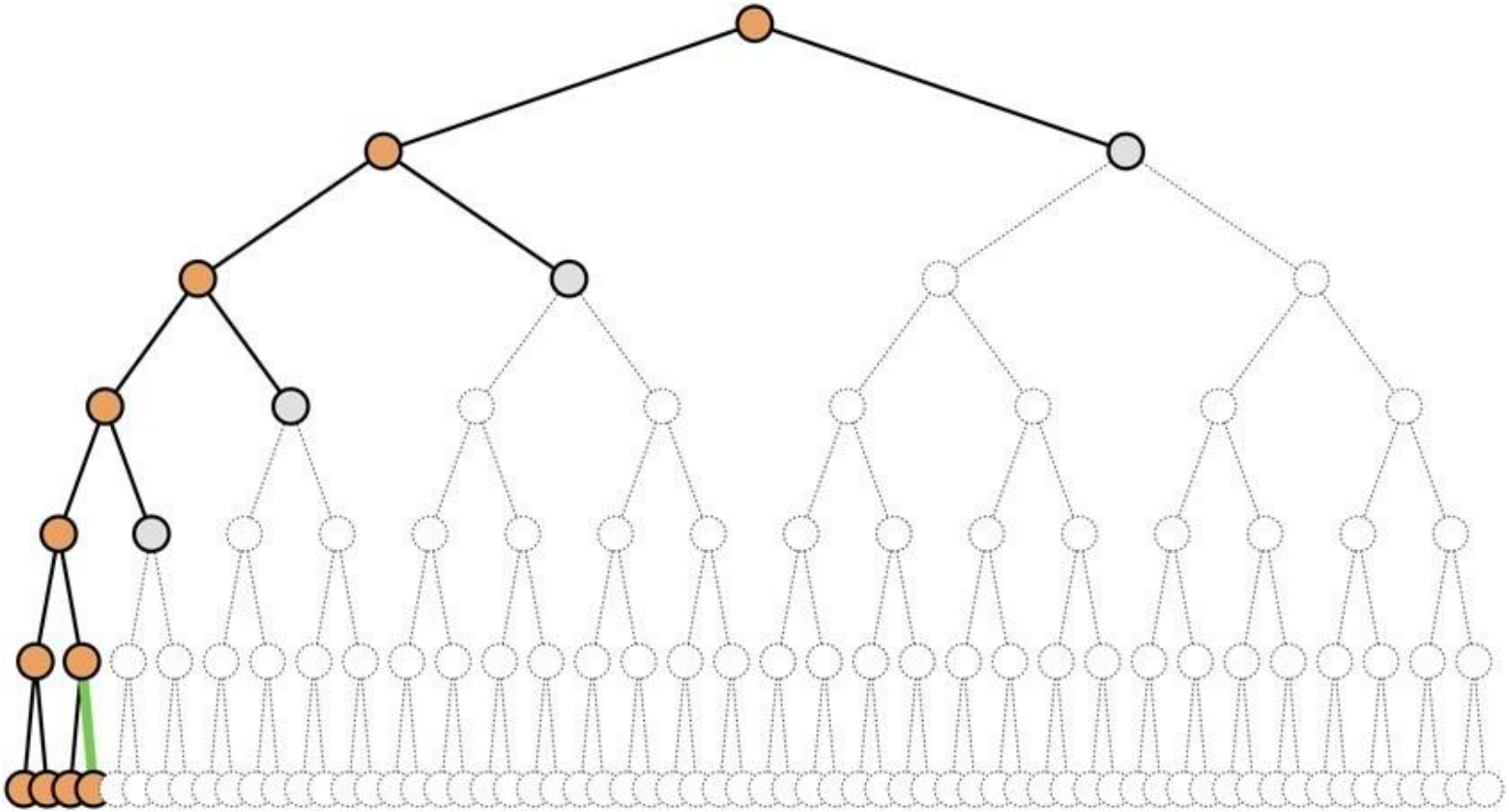
DFS



Searches branch by branch ...

... with each branch pursued to maximum depth.

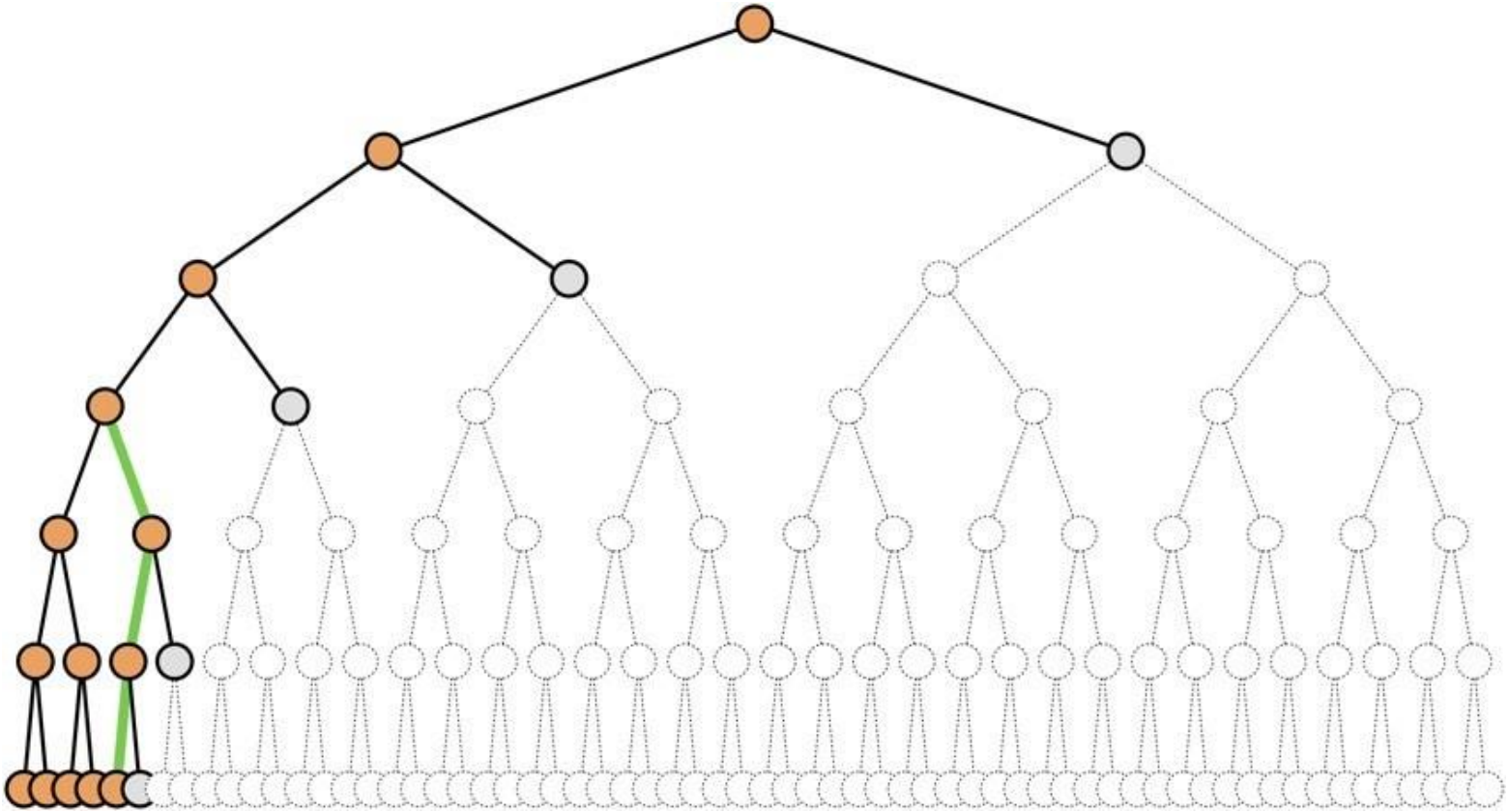
DFS



Searches branch by branch ...

... with each branch pursued to maximum depth.

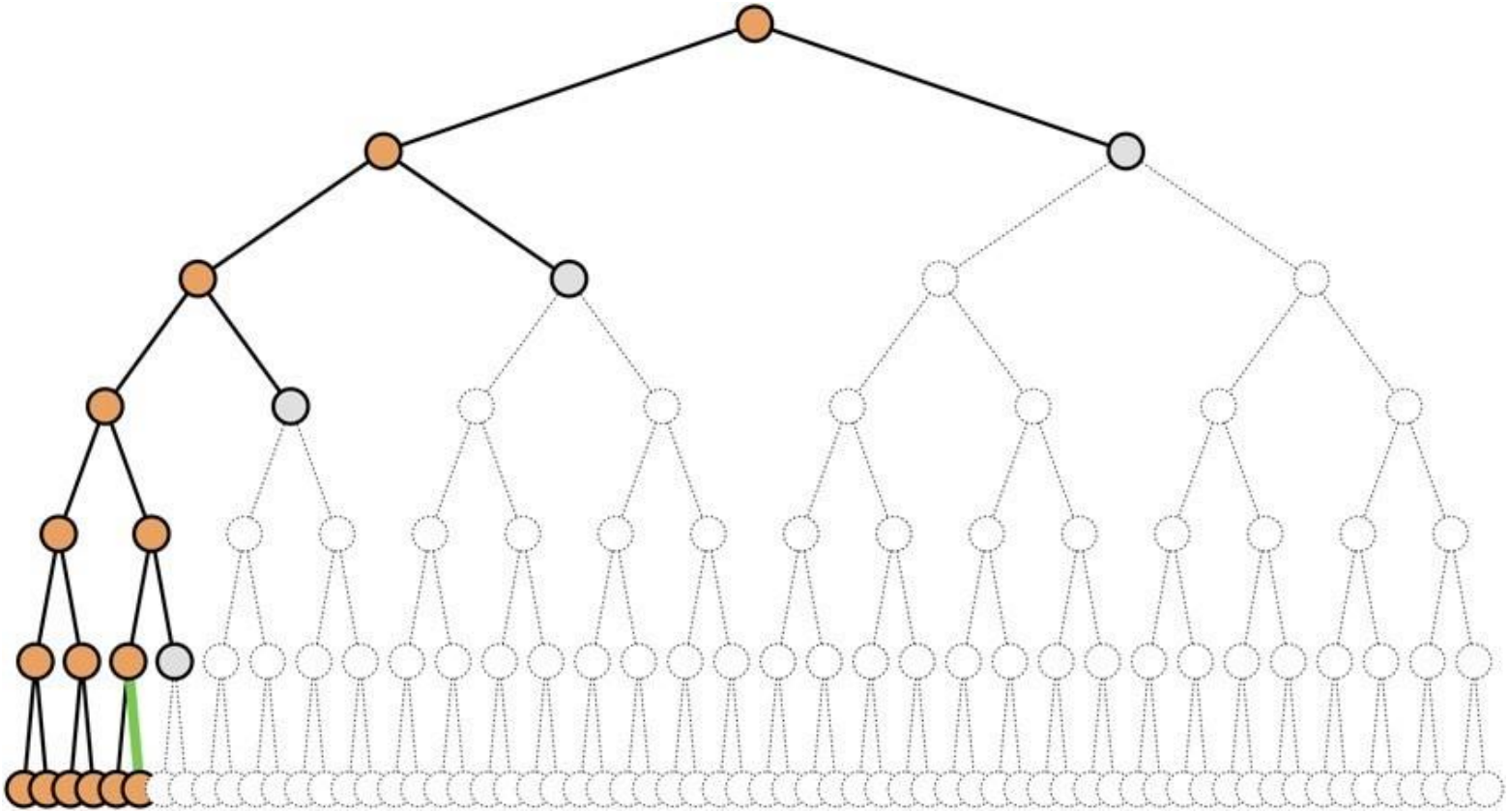
DFS



Searches branch by branch ...

... with each branch pursued to maximum depth.

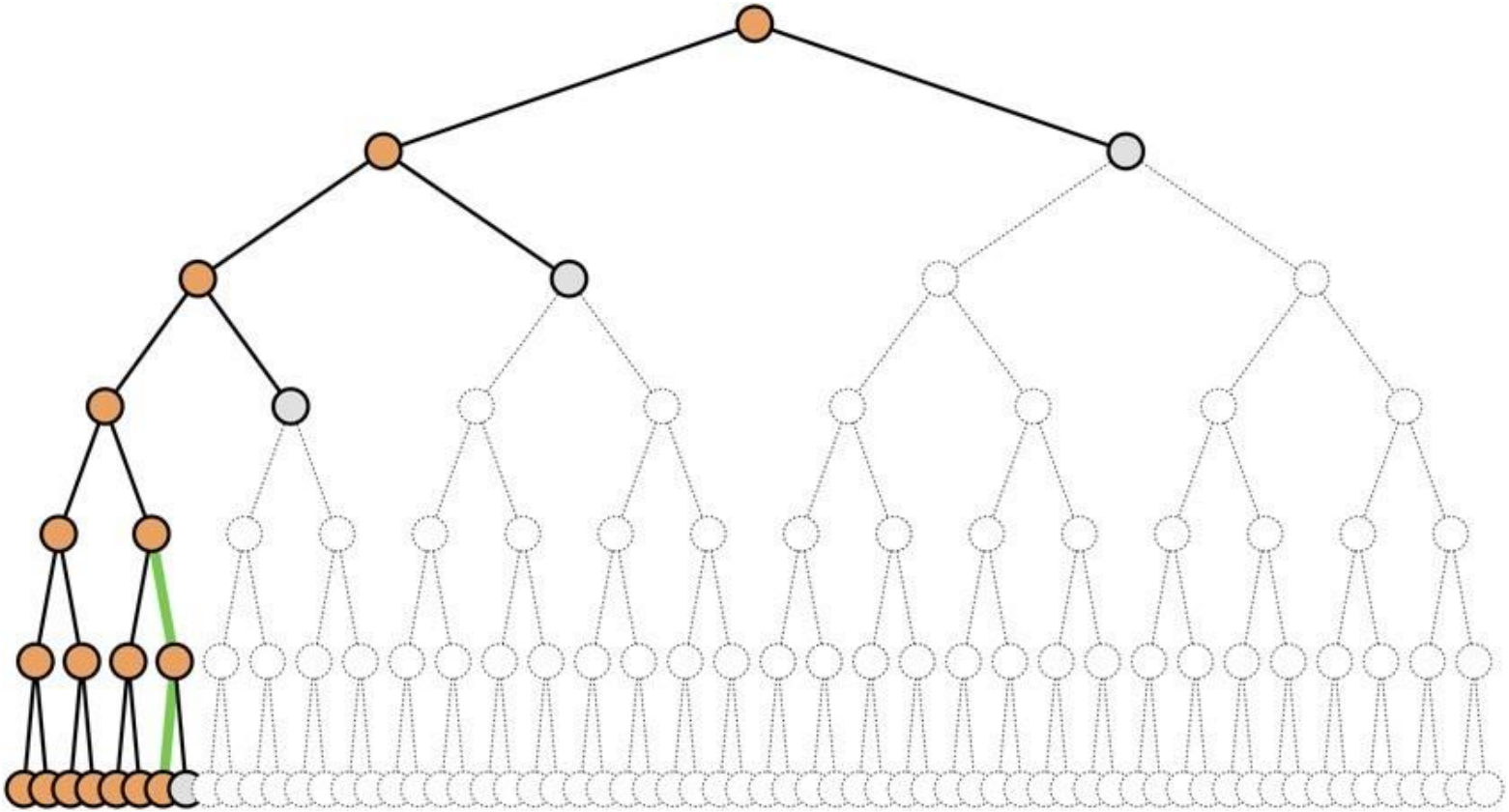
DFS



Searches branch by branch ...

... with each branch pursued to maximum depth.

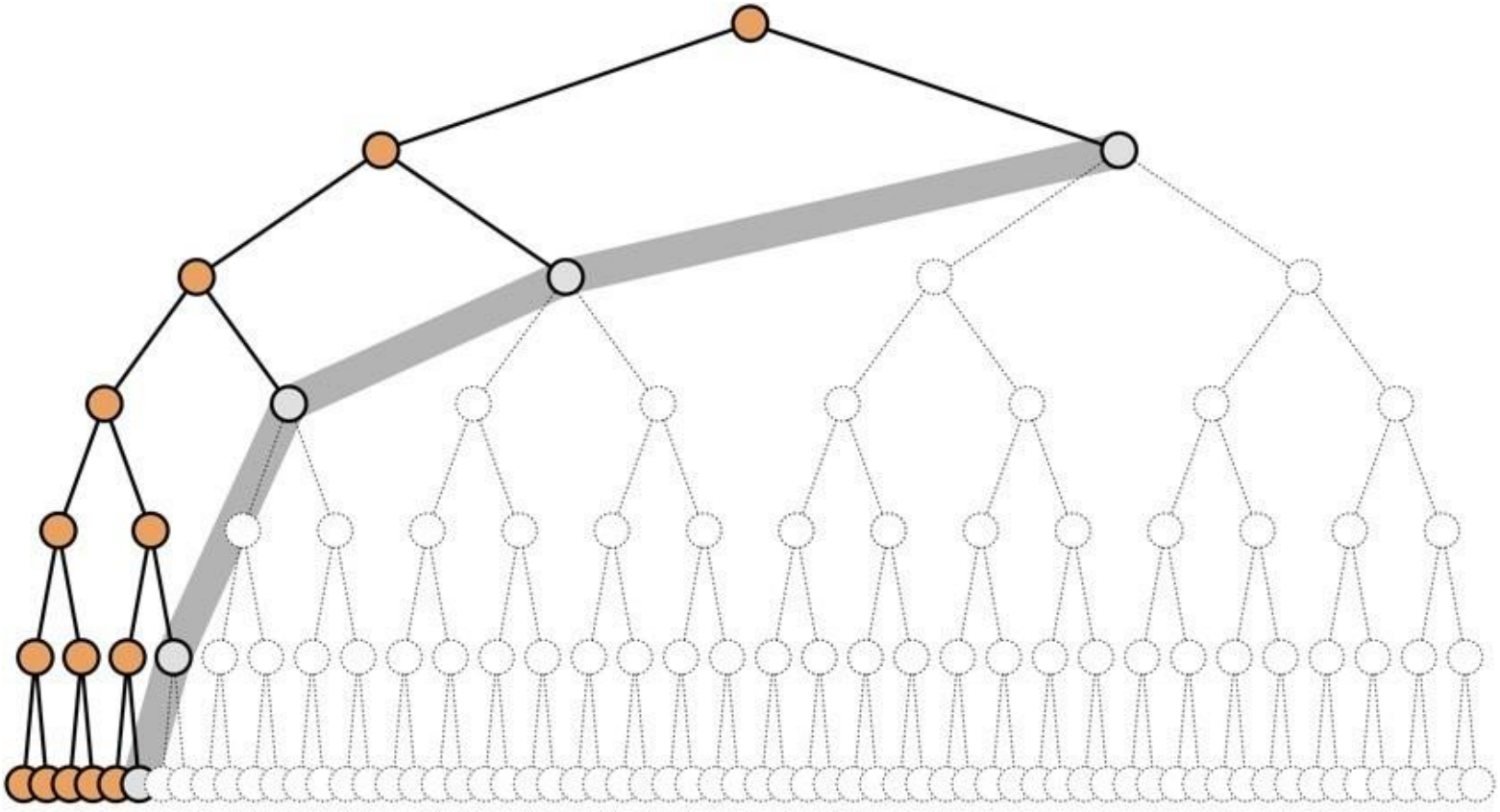
DFS



Searches branch by branch ...

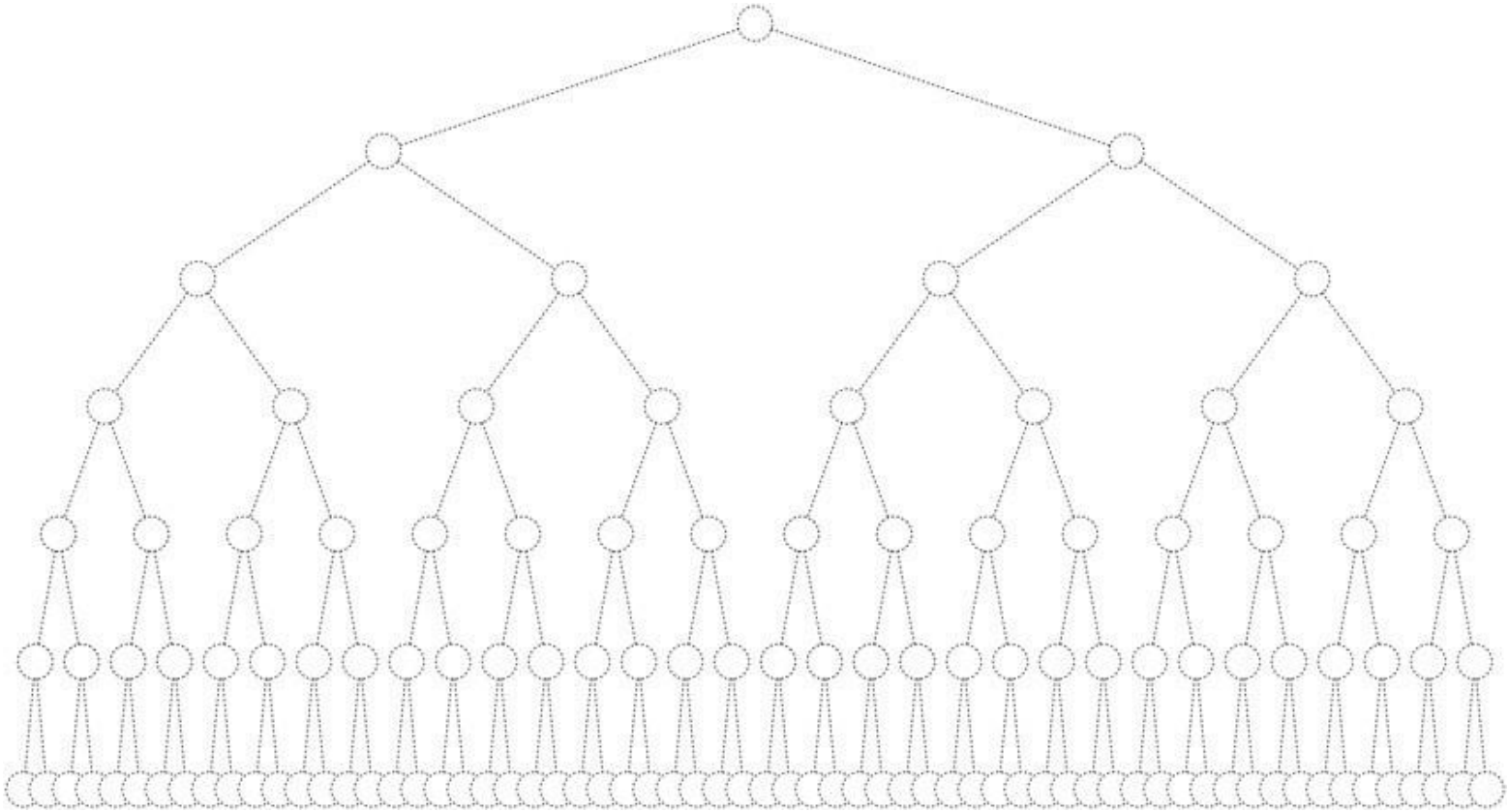
... with each branch pursued to maximum depth.

DFS



The frontier consists of unexplored siblings of all ancestors.
Search proceeds by exhausting one branch at a time.

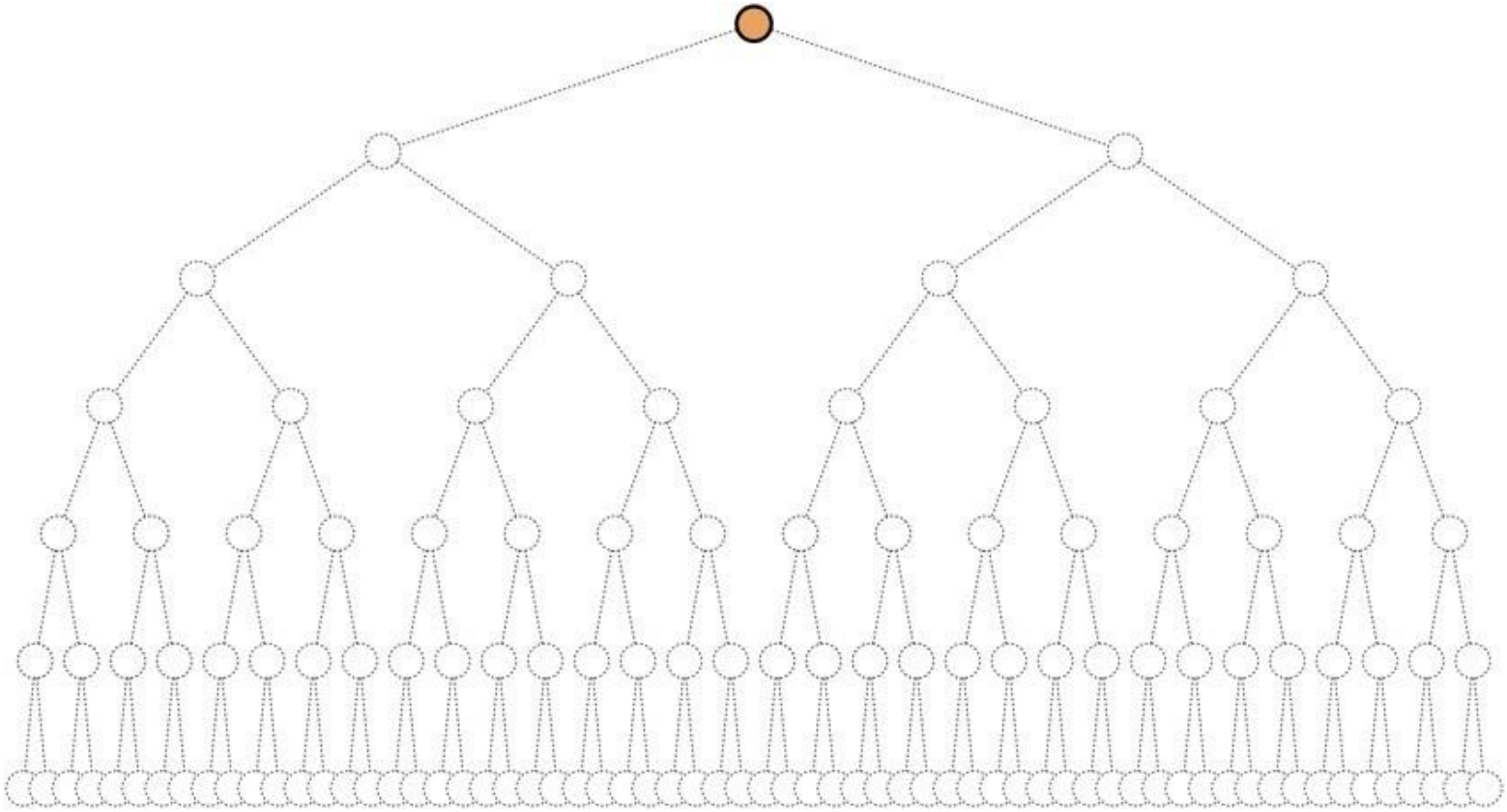
IDS



Searches subtree by subtree …

… with each subtree increasing by depth limit.

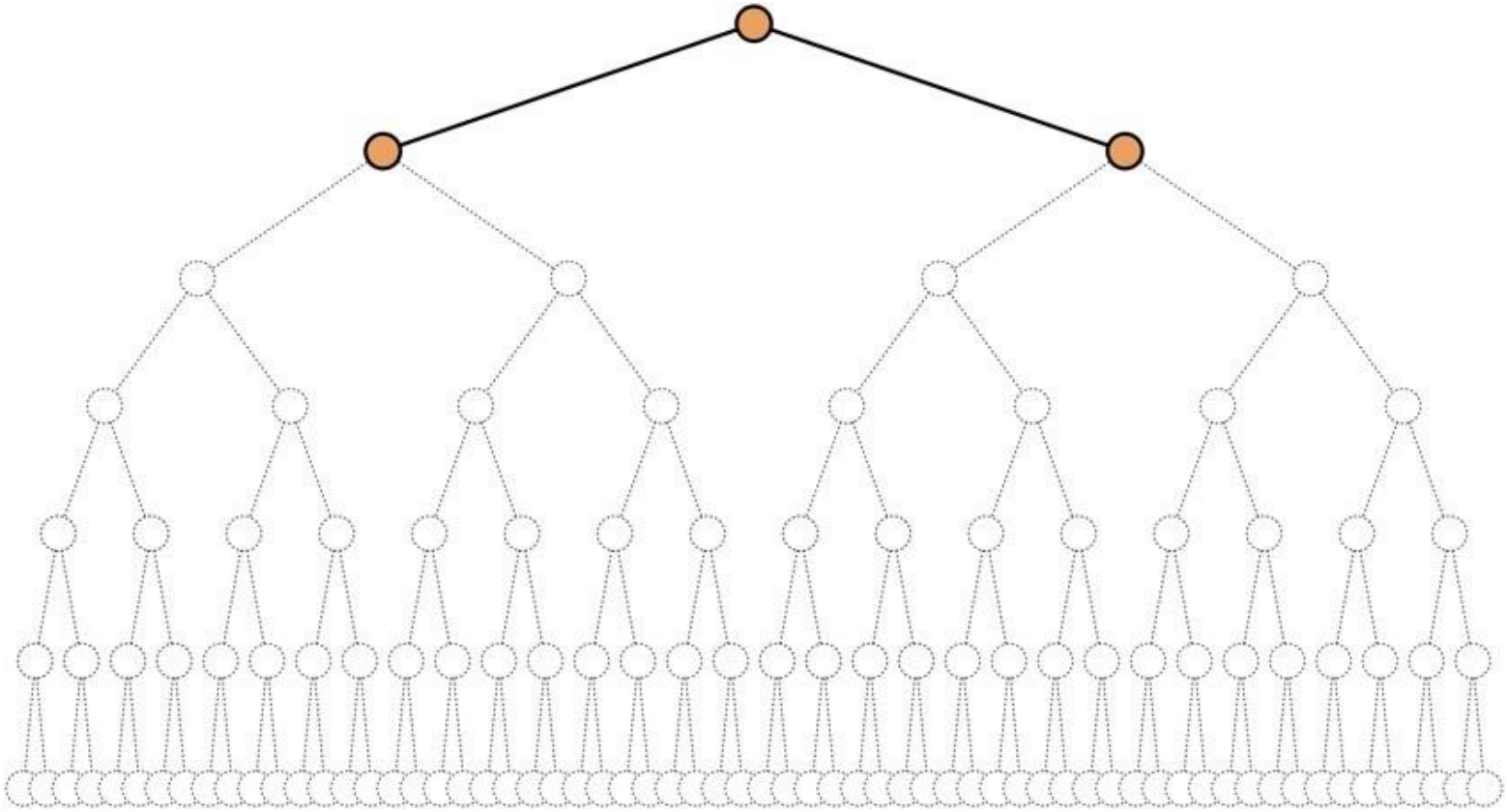
IDS



Searches subtree by subtree …

… with each subtree increasing by depth limit.

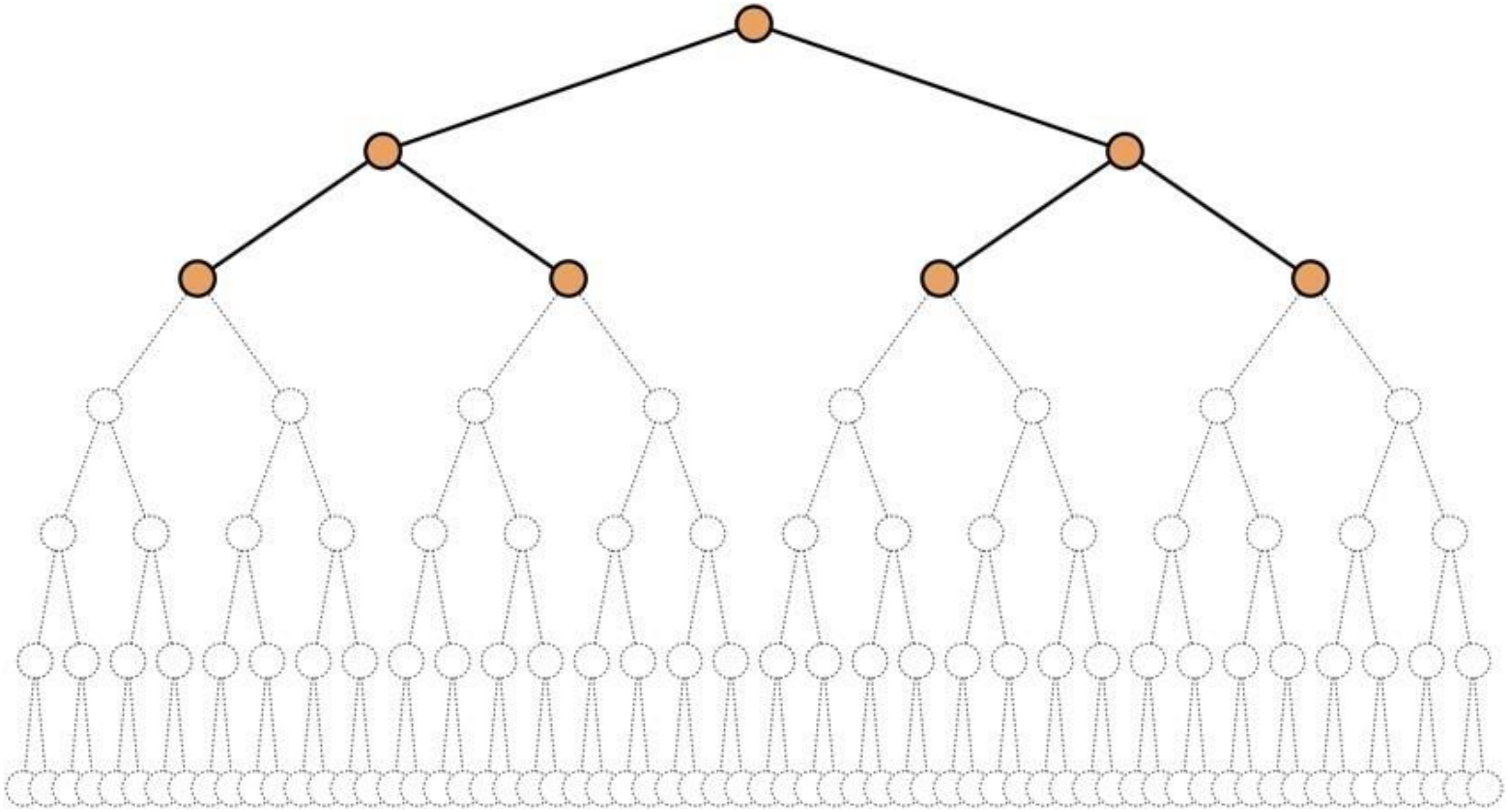
IDS



Searches subtree by subtree …

… with each subtree increasing by depth limit.

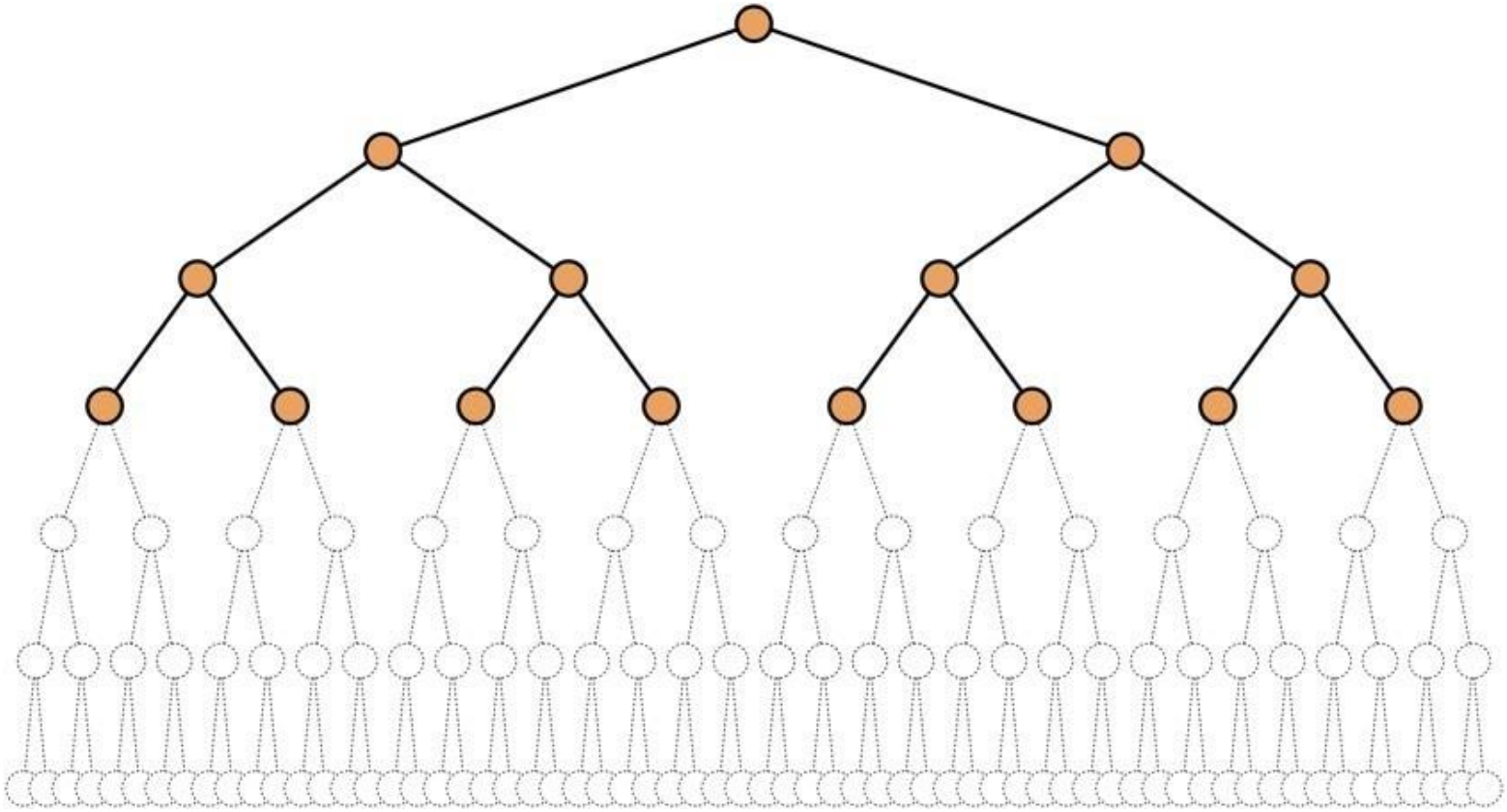
IDS



Searches subtree by subtree ...

... with each subtree increasing by depth limit.

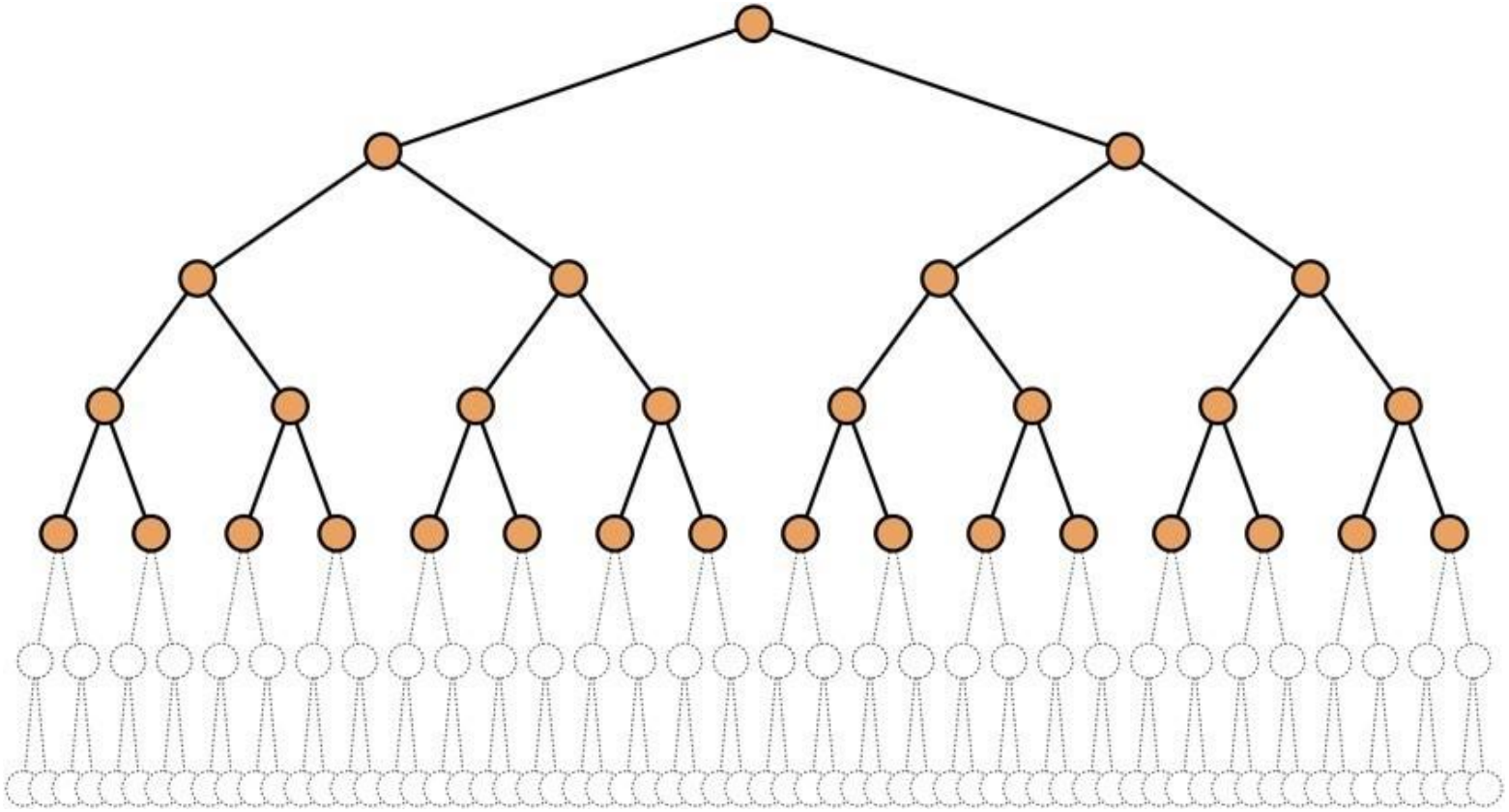
IDS



Searches subtree by subtree ...

... with each subtree increasing by depth limit.

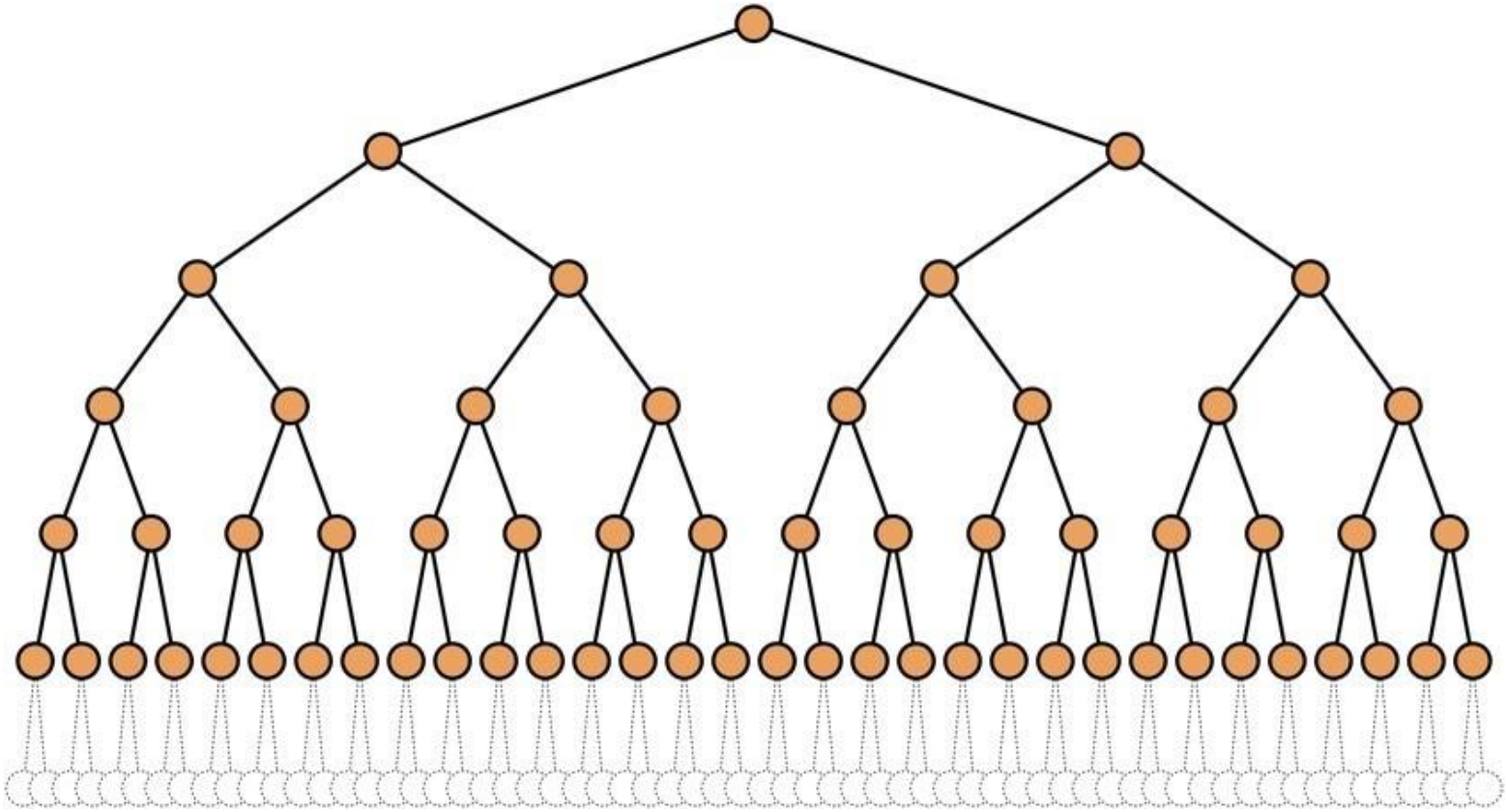
IDS



Searches subtree by subtree ...

... with each subtree increasing by depth limit.

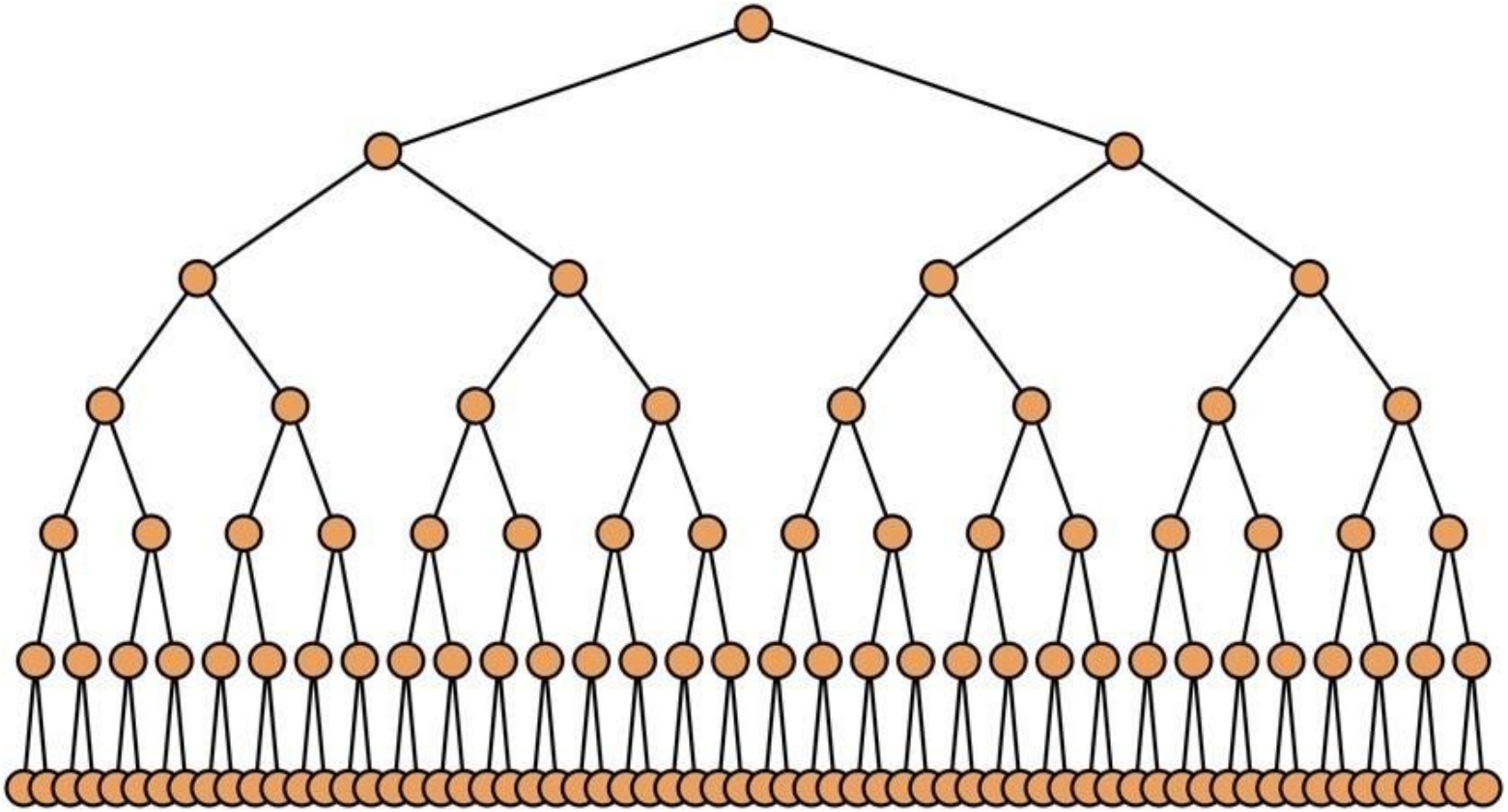
IDS



Searches subtree by subtree ...

... with each subtree increasing by depth limit.

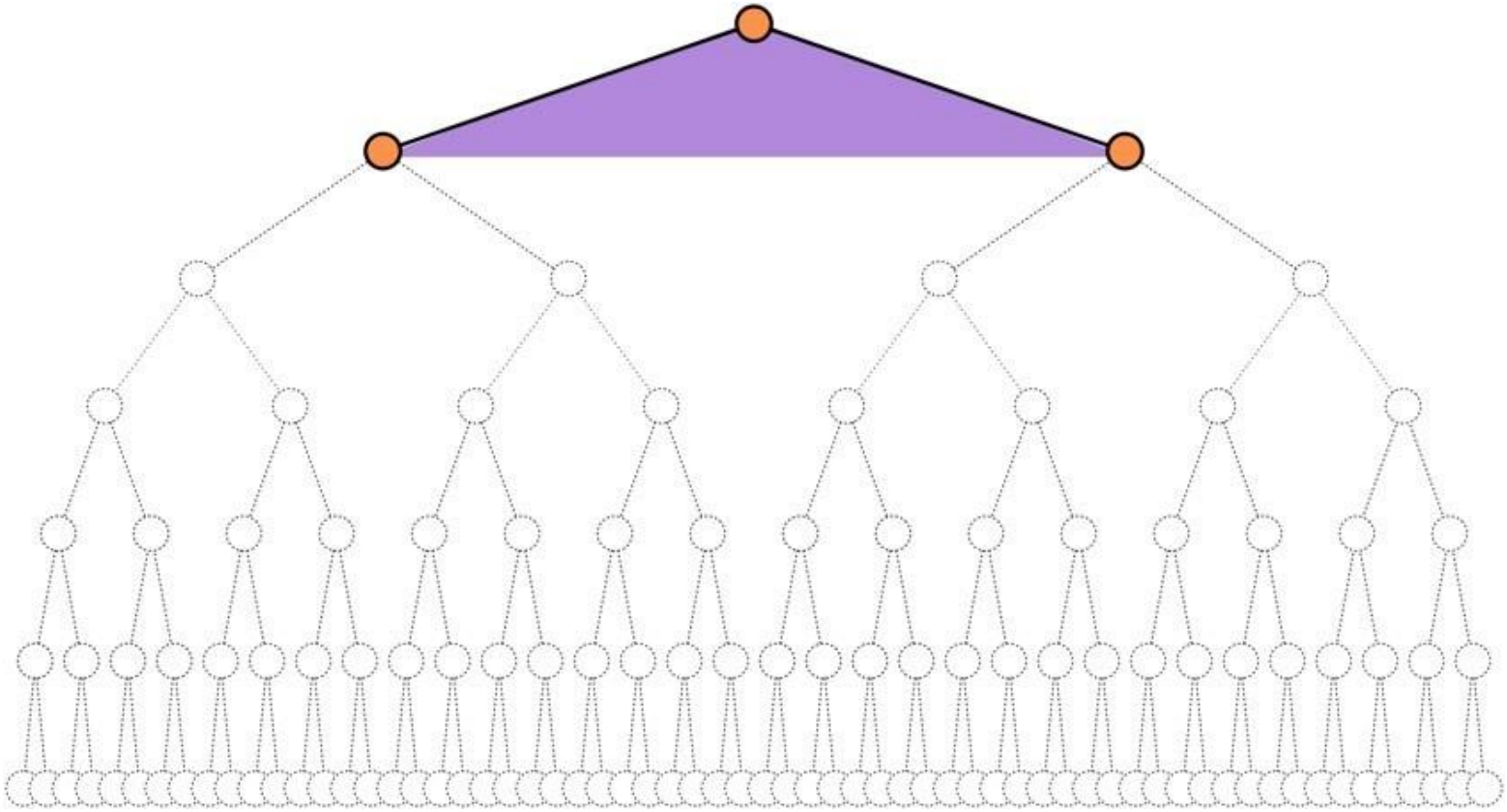
IDS



Searches subtree by subtree …

… with each subtree increasing by depth limit.

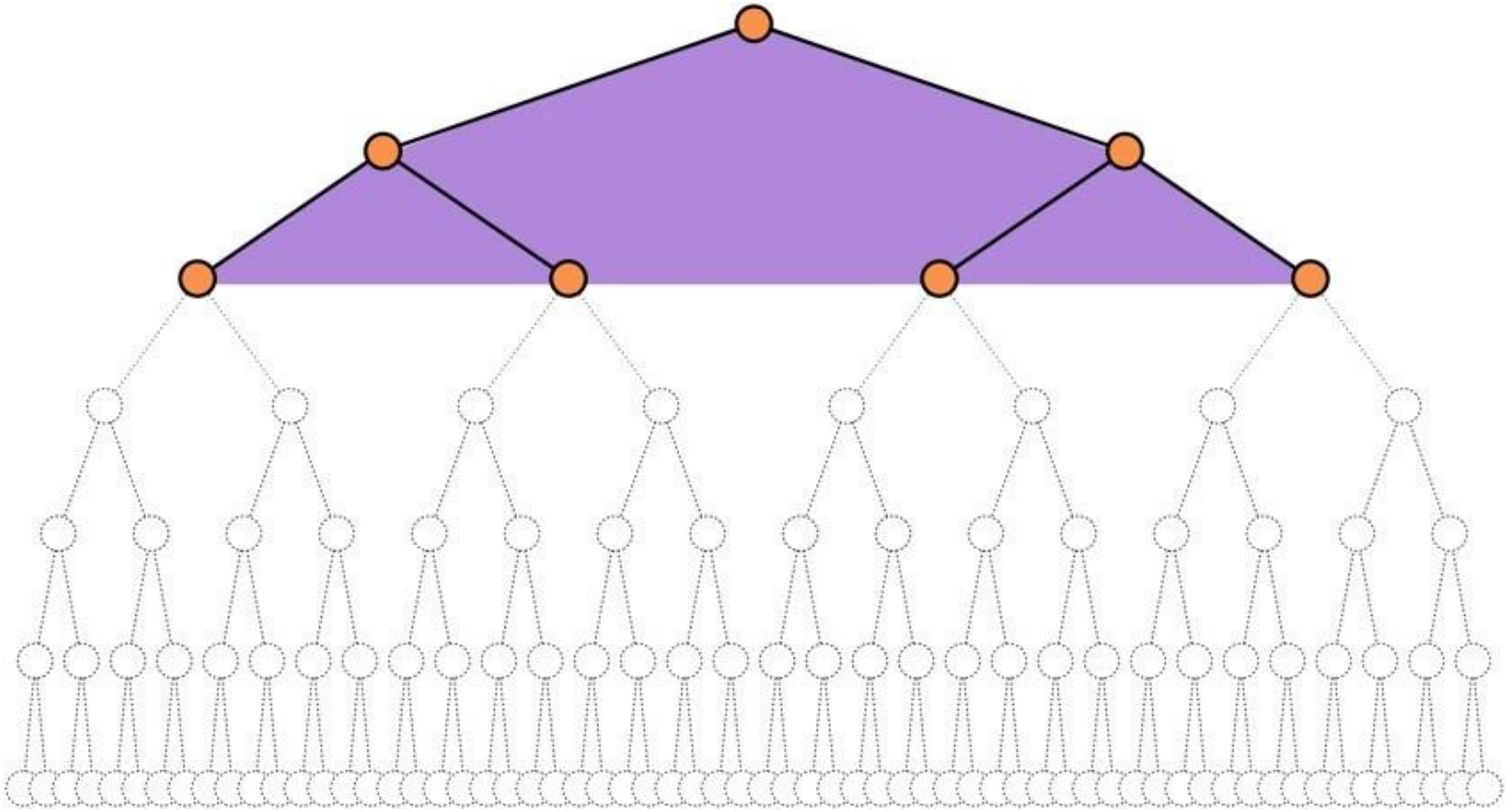
IDS



Each iteration is simply an instance of depth-limited search.

Search proceeds by exhausting larger and larger subtrees.

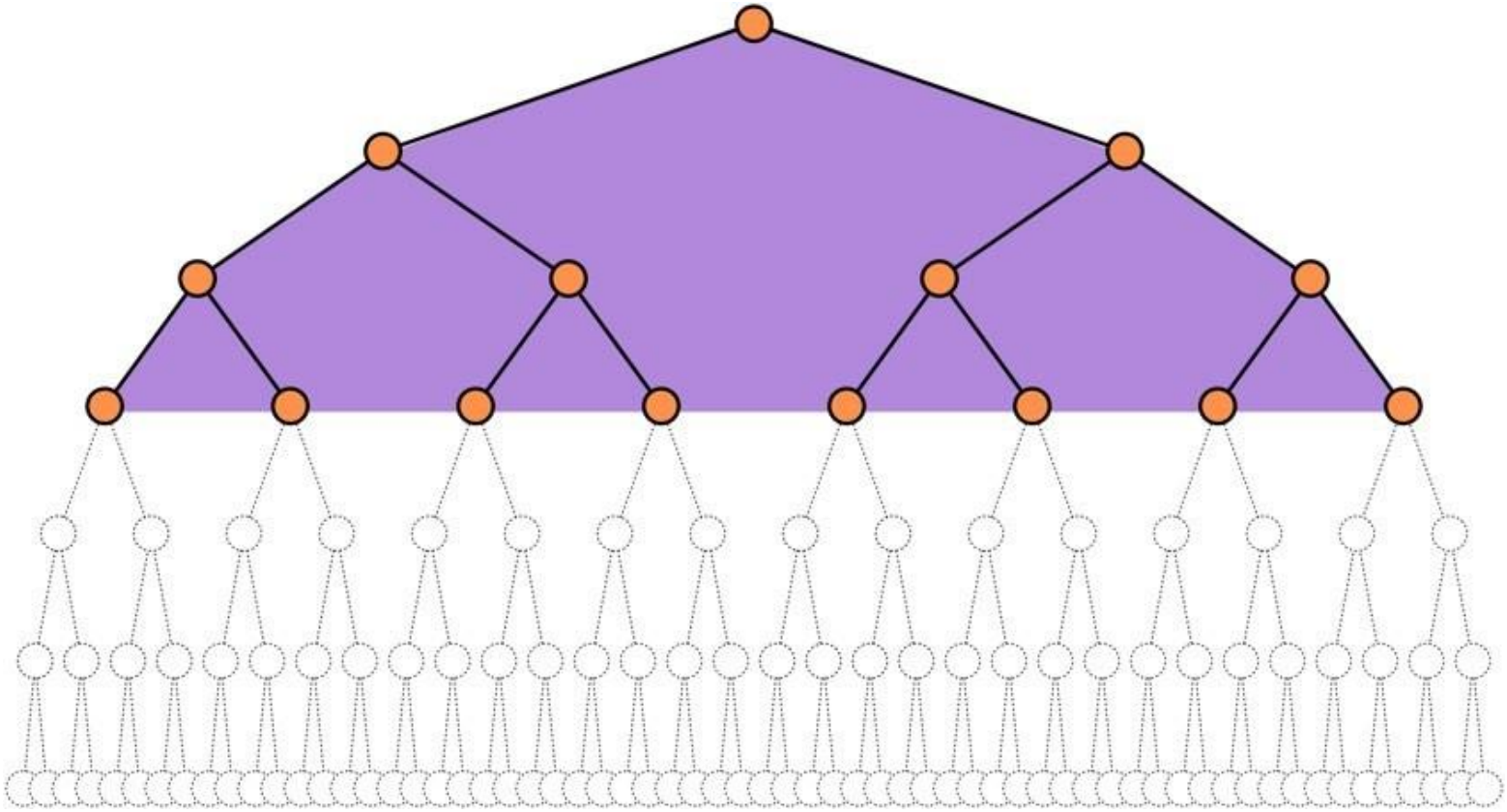
IDS



Each iteration is simply an instance of depth-limited search.

Search proceeds by exhausting larger and larger subtrees.

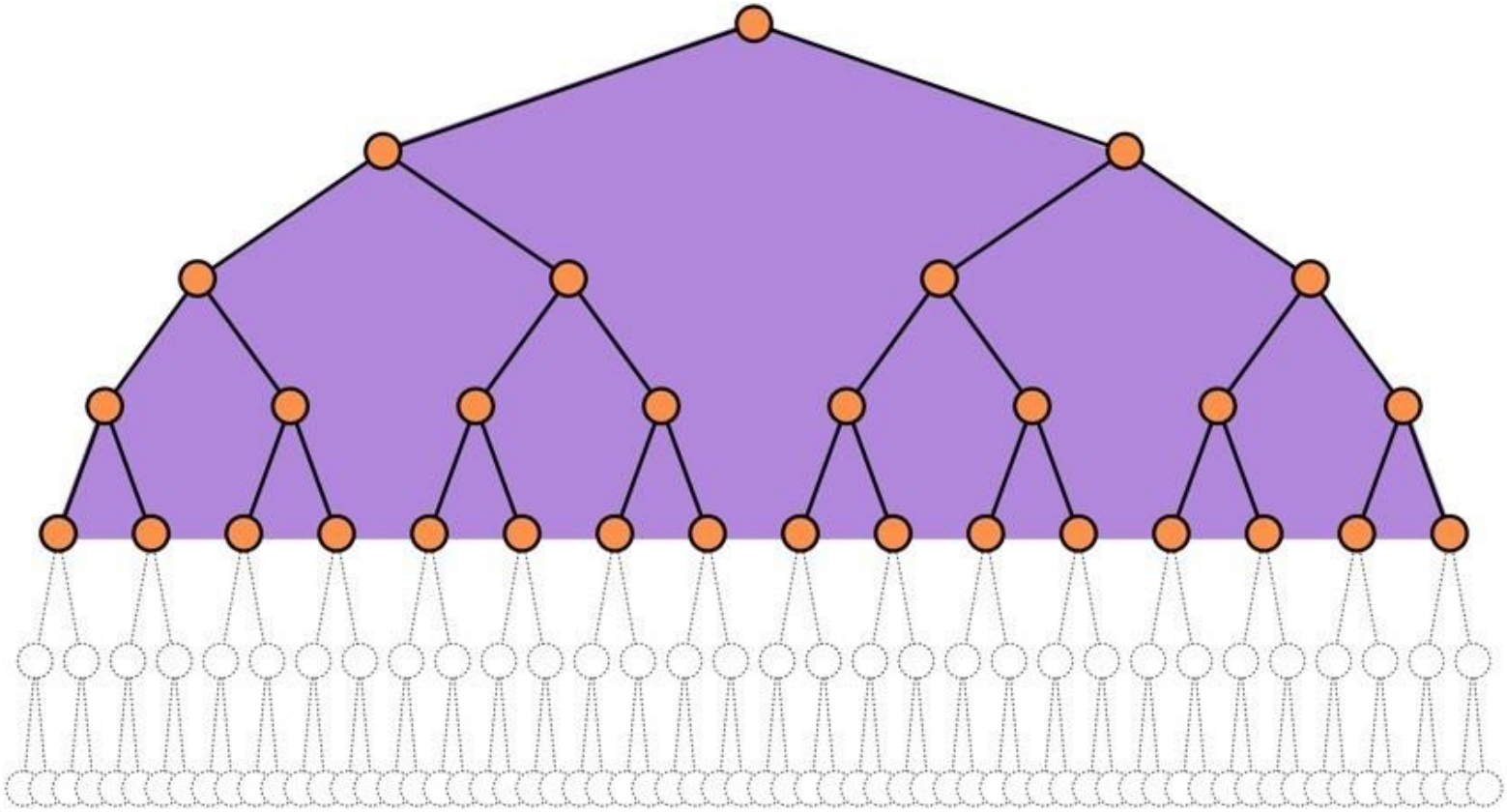
IDS



Each iteration is simply an instance of depth-limited search.

Search proceeds by exhausting larger and larger subtrees.

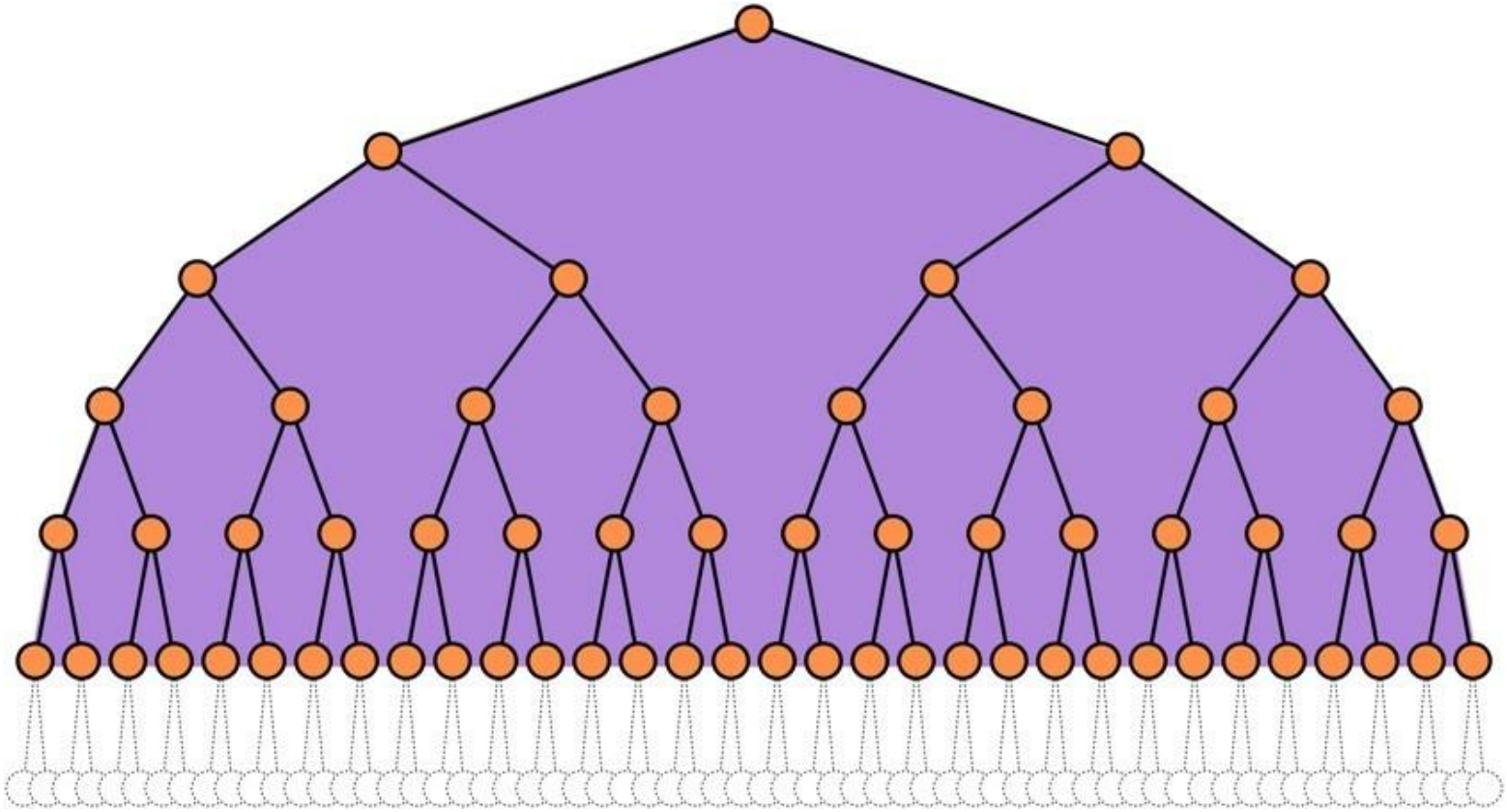
IDS



Each iteration is simply an instance of depth-limited search.

Search proceeds by exhausting larger and larger subtrees.

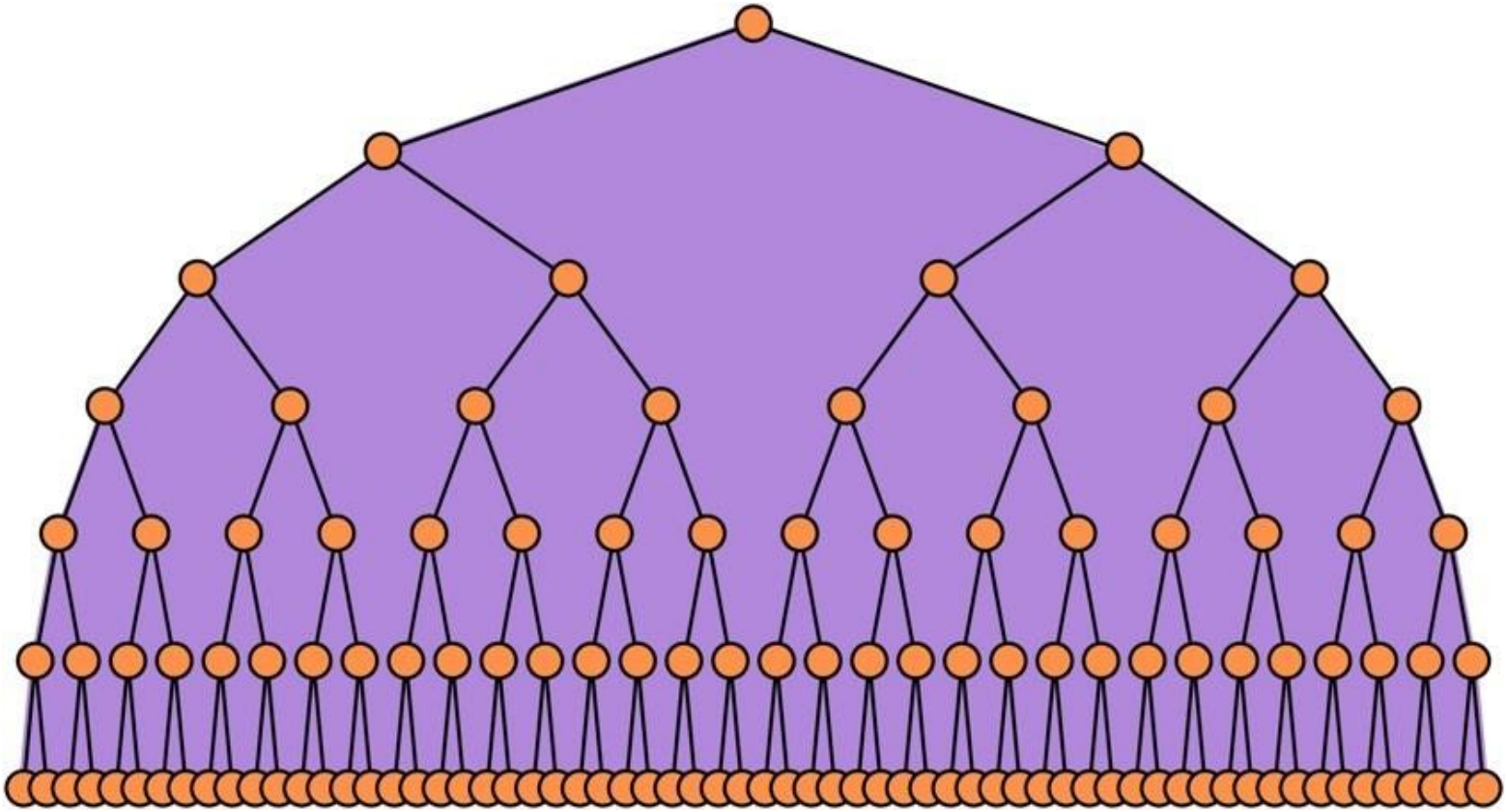
IDS



Each iteration is simply an instance of depth-limited search.

Search proceeds by exhausting larger and larger subtrees.

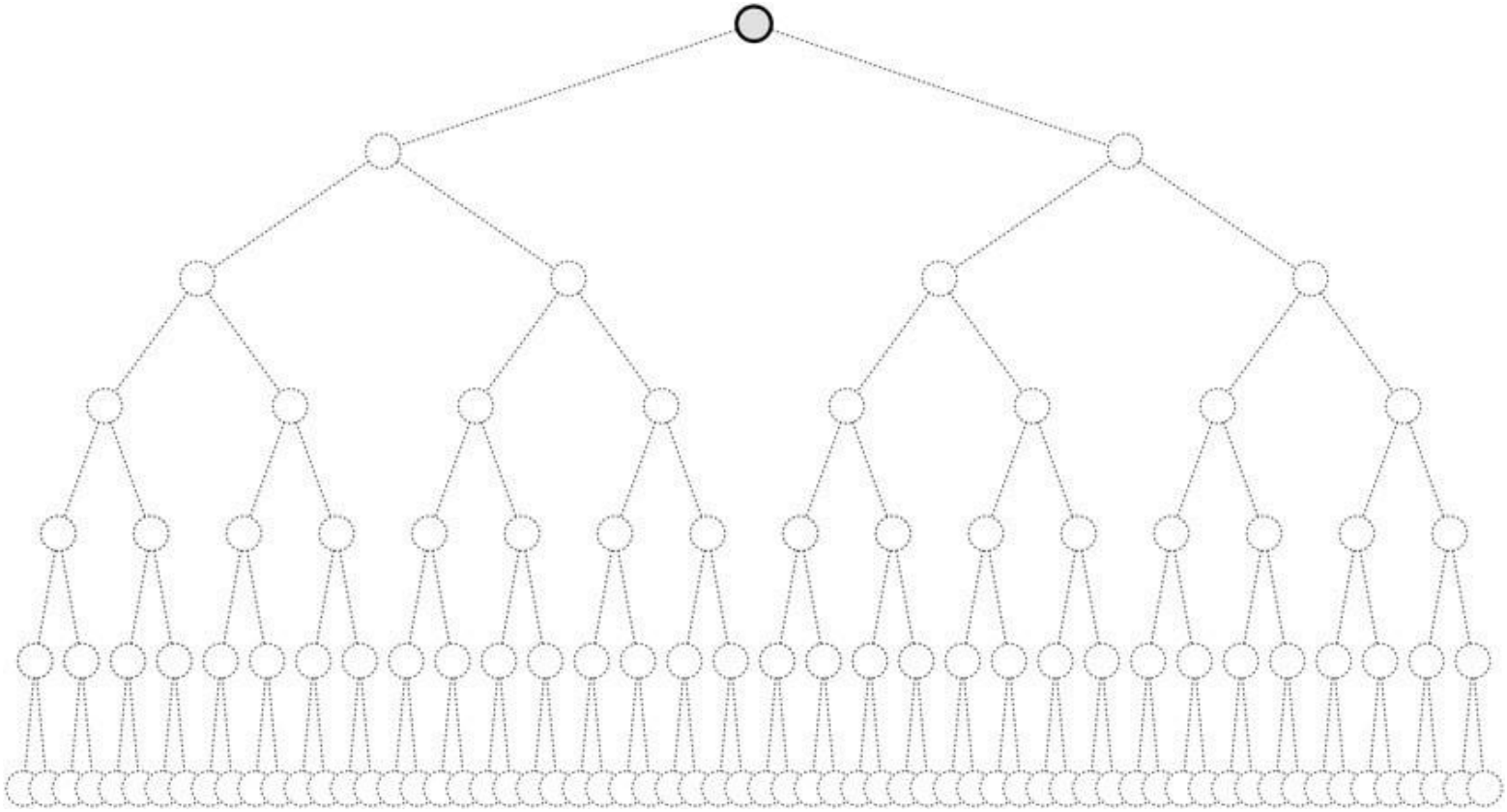
IDS



Each iteration is simply an instance of depth-limited search.

Search proceeds by exhausting larger and larger subtrees.

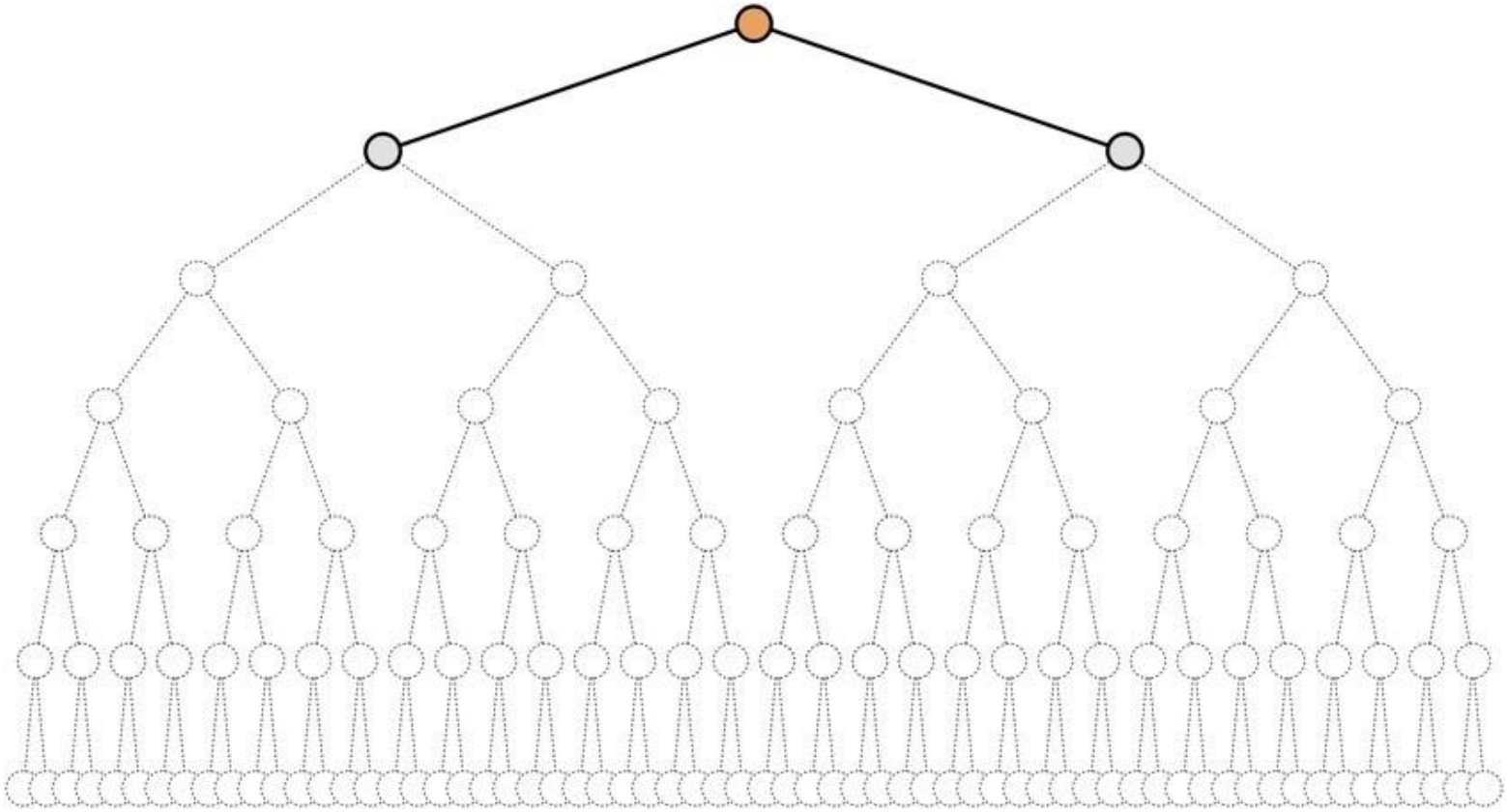
BFS



Searches layer by layer …

… with each layer organized by node depth.

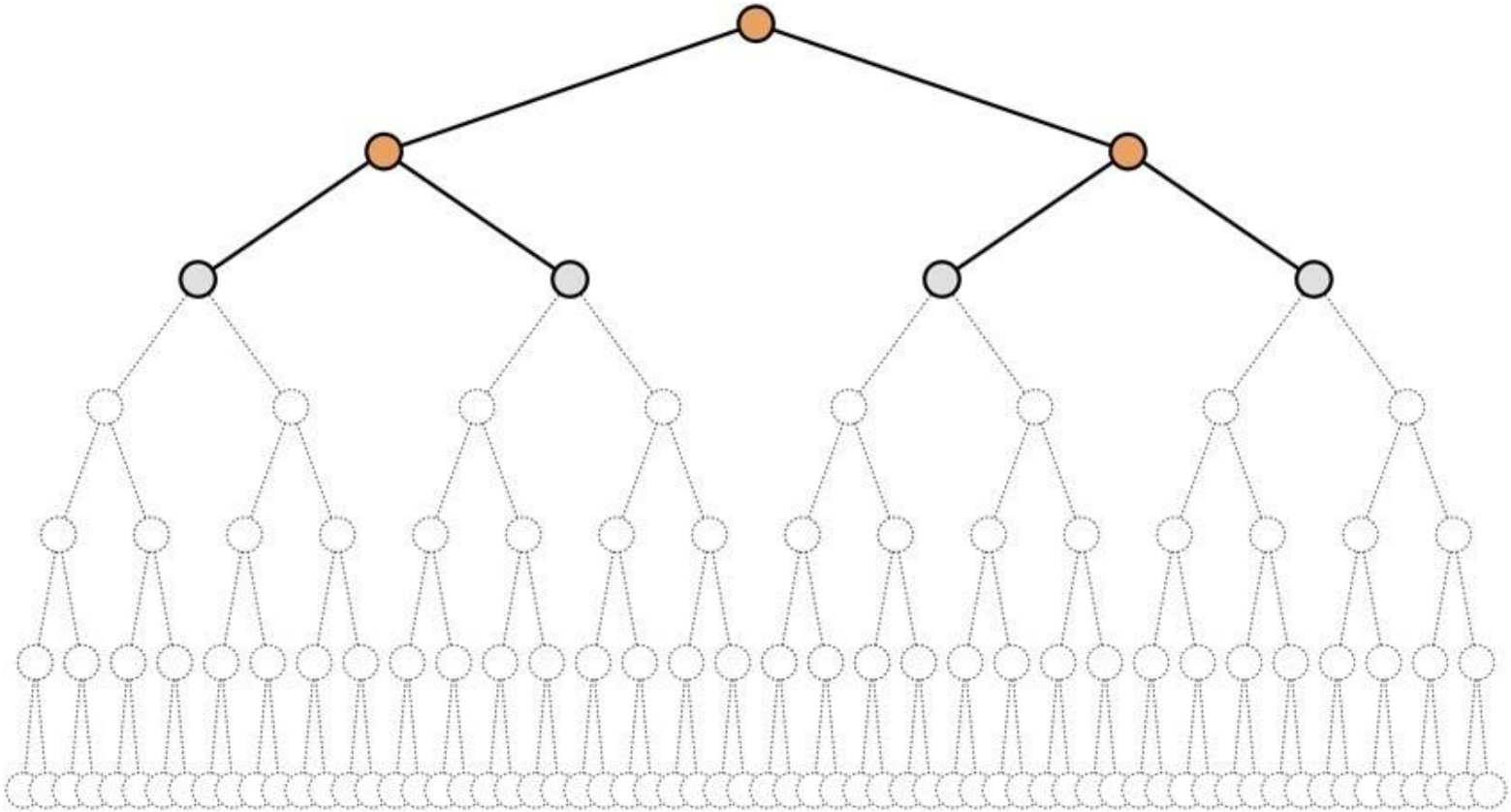
BFS



Searches layer by layer …

… with each layer organized by node depth.

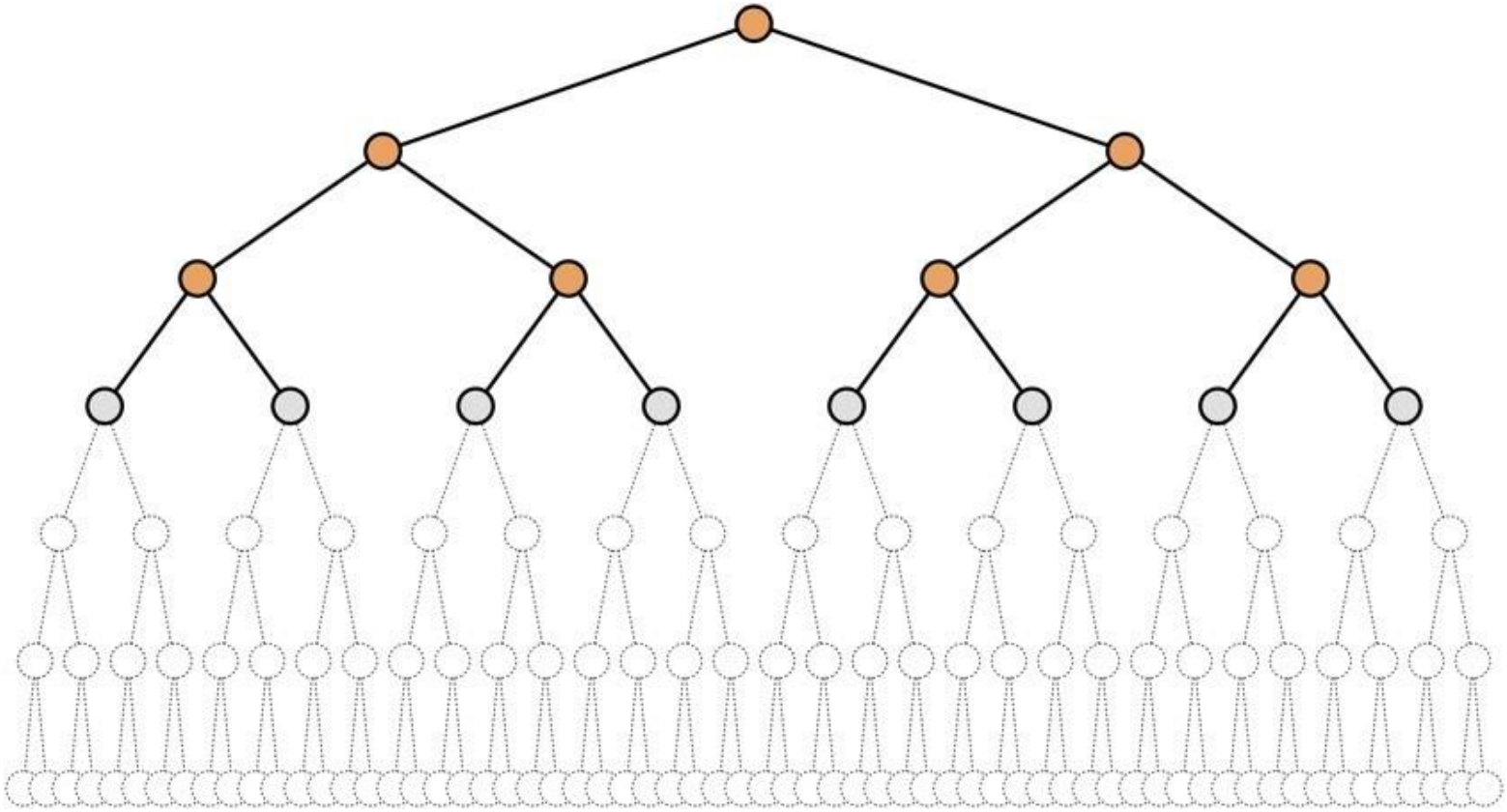
BFS



Searches layer by layer …

… with each layer organized by node depth.

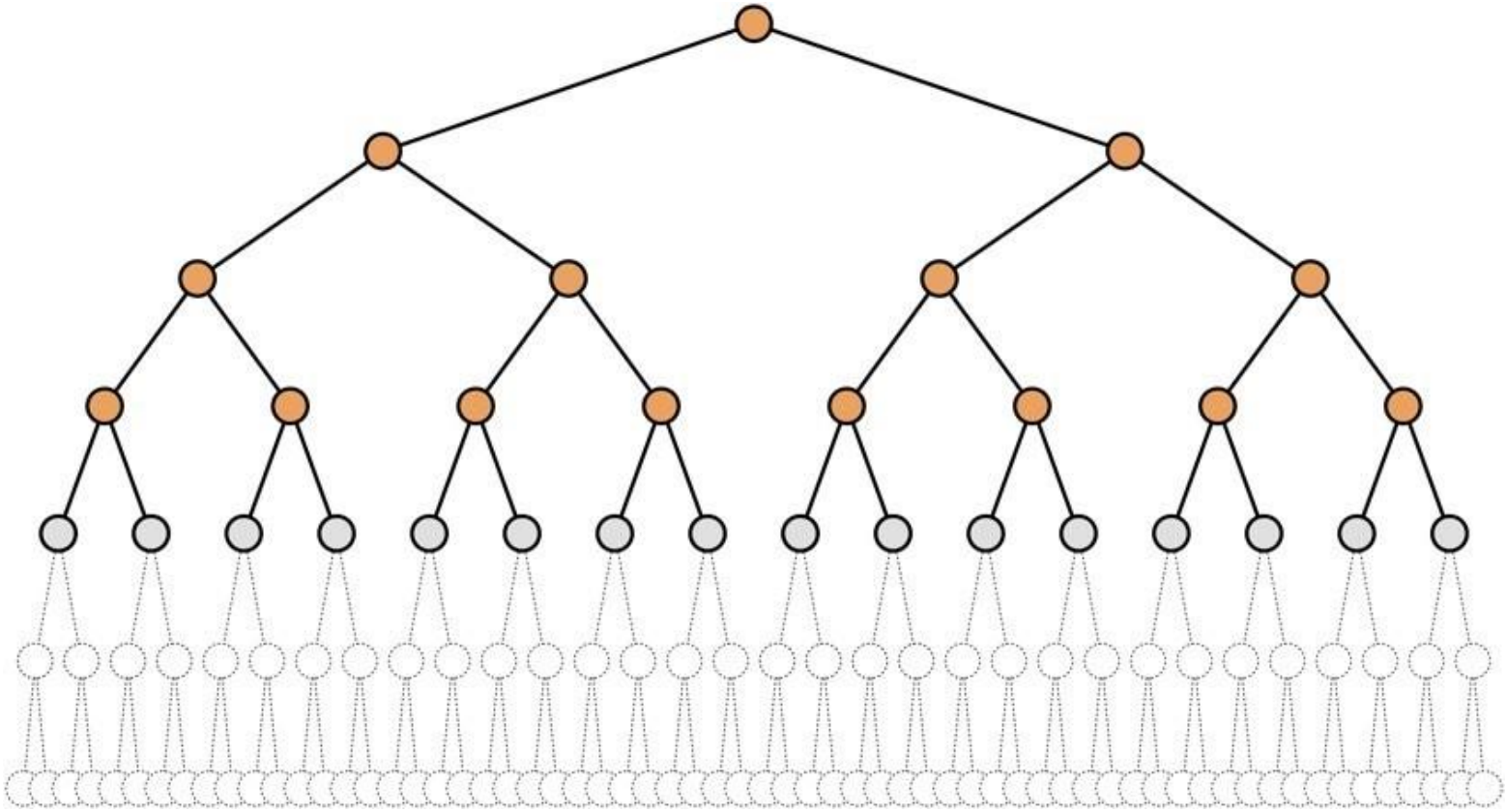
BFS



Searches layer by layer …

… with each layer organized by node depth.

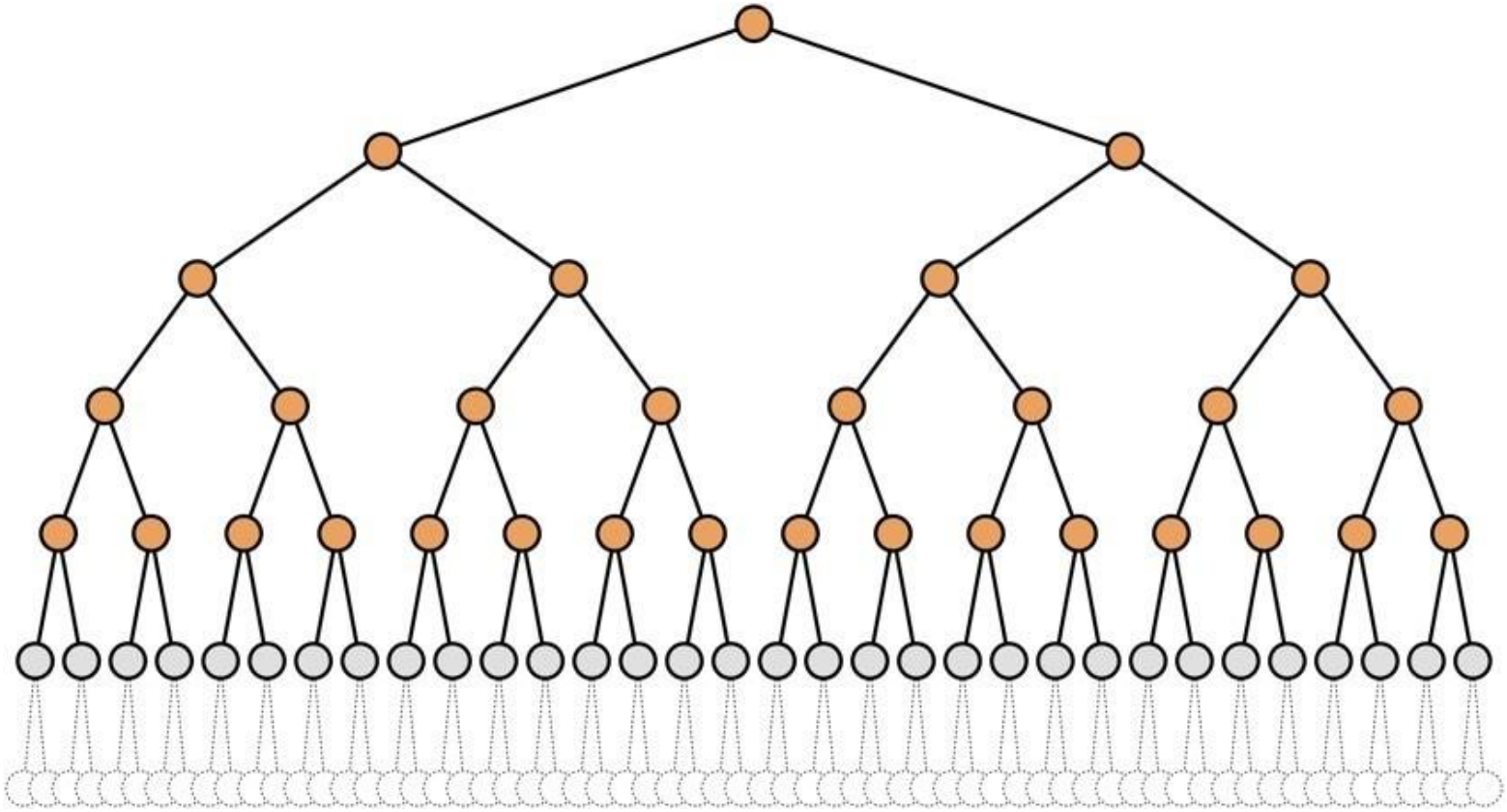
BFS



Searches layer by layer …

… with each layer organized by node depth.

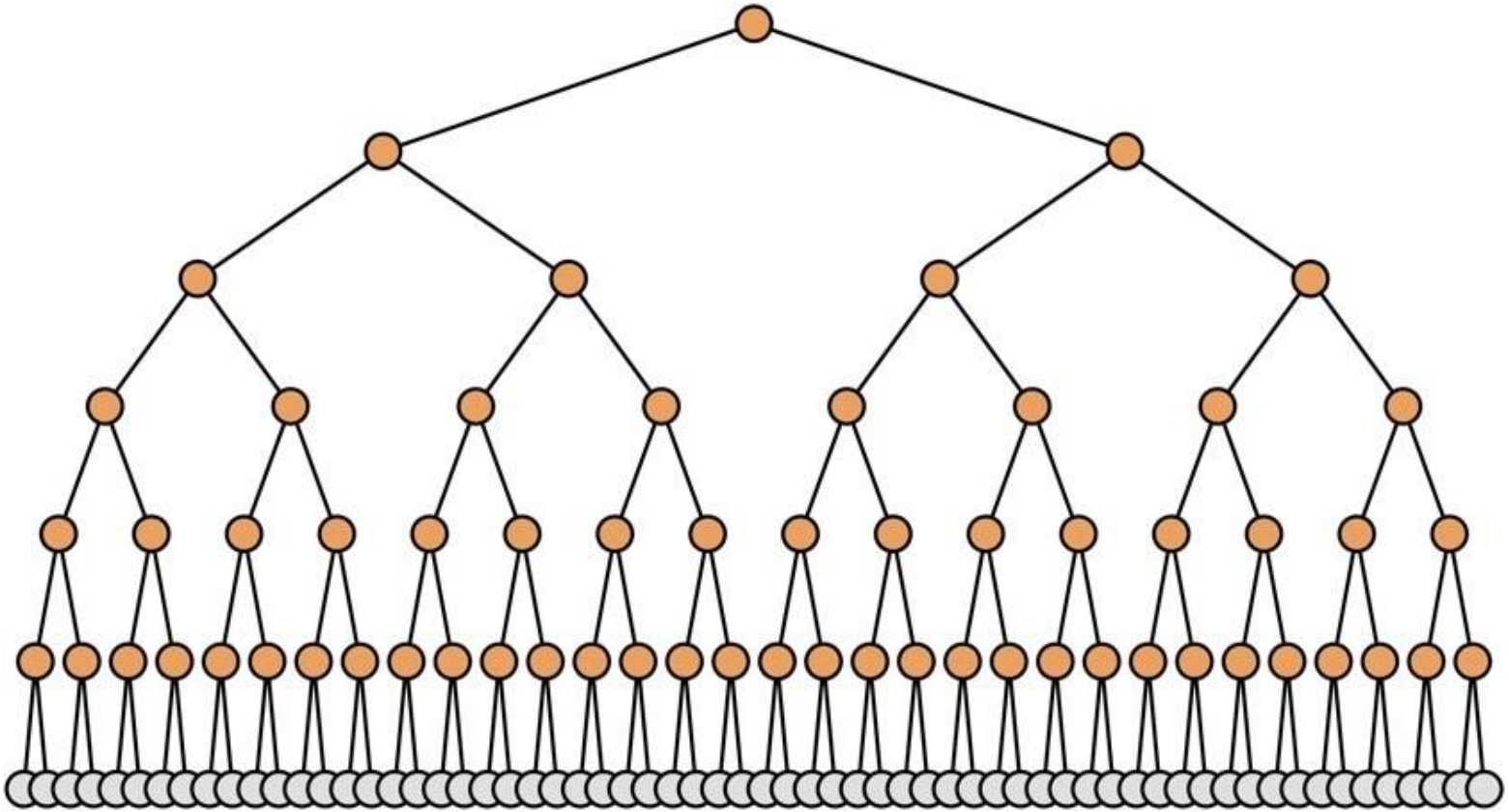
BFS



Searches layer by layer …

… with each layer organized by node depth.

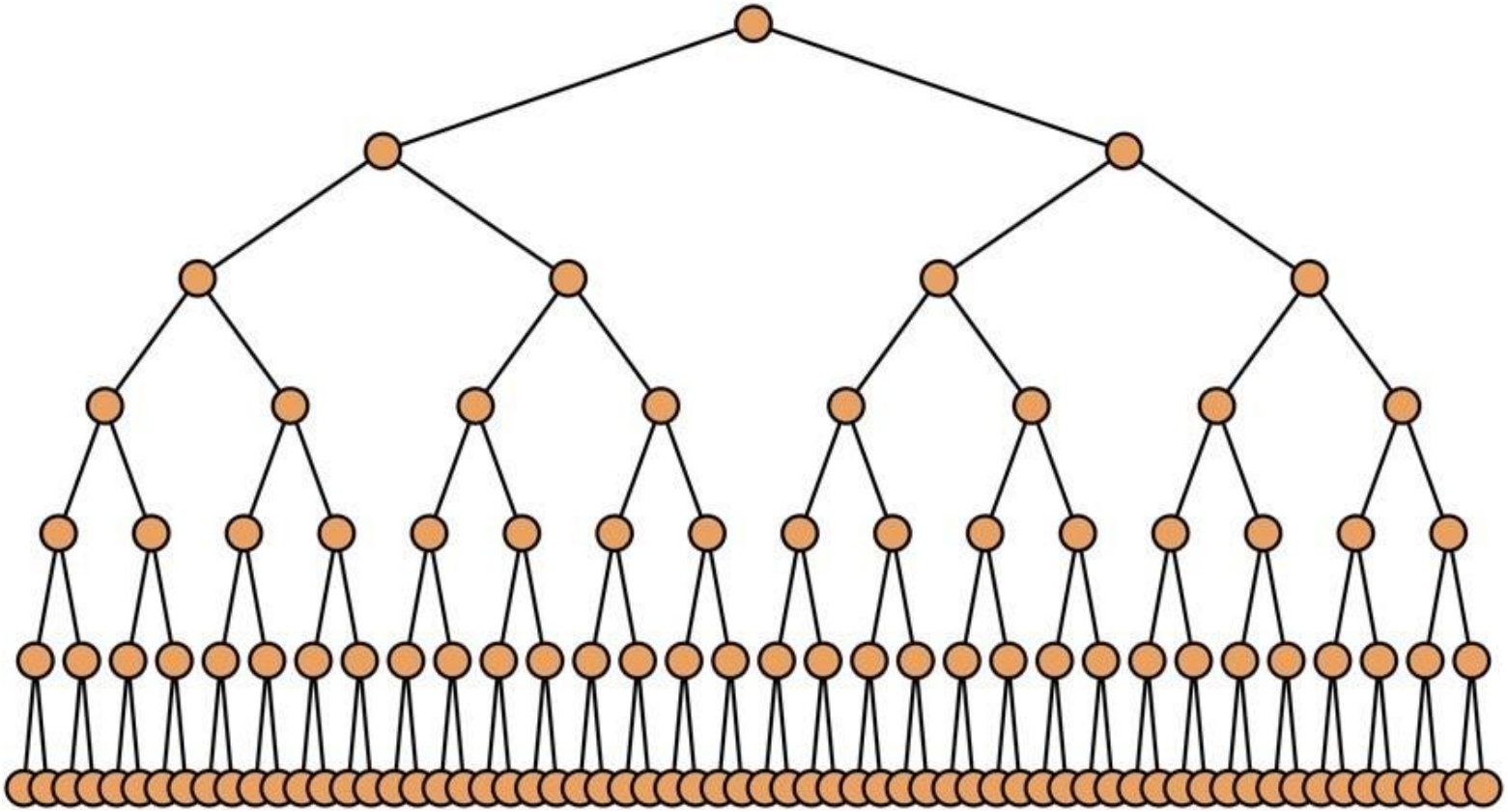
BFS



Searches layer by layer …

… with each layer organized by node depth.

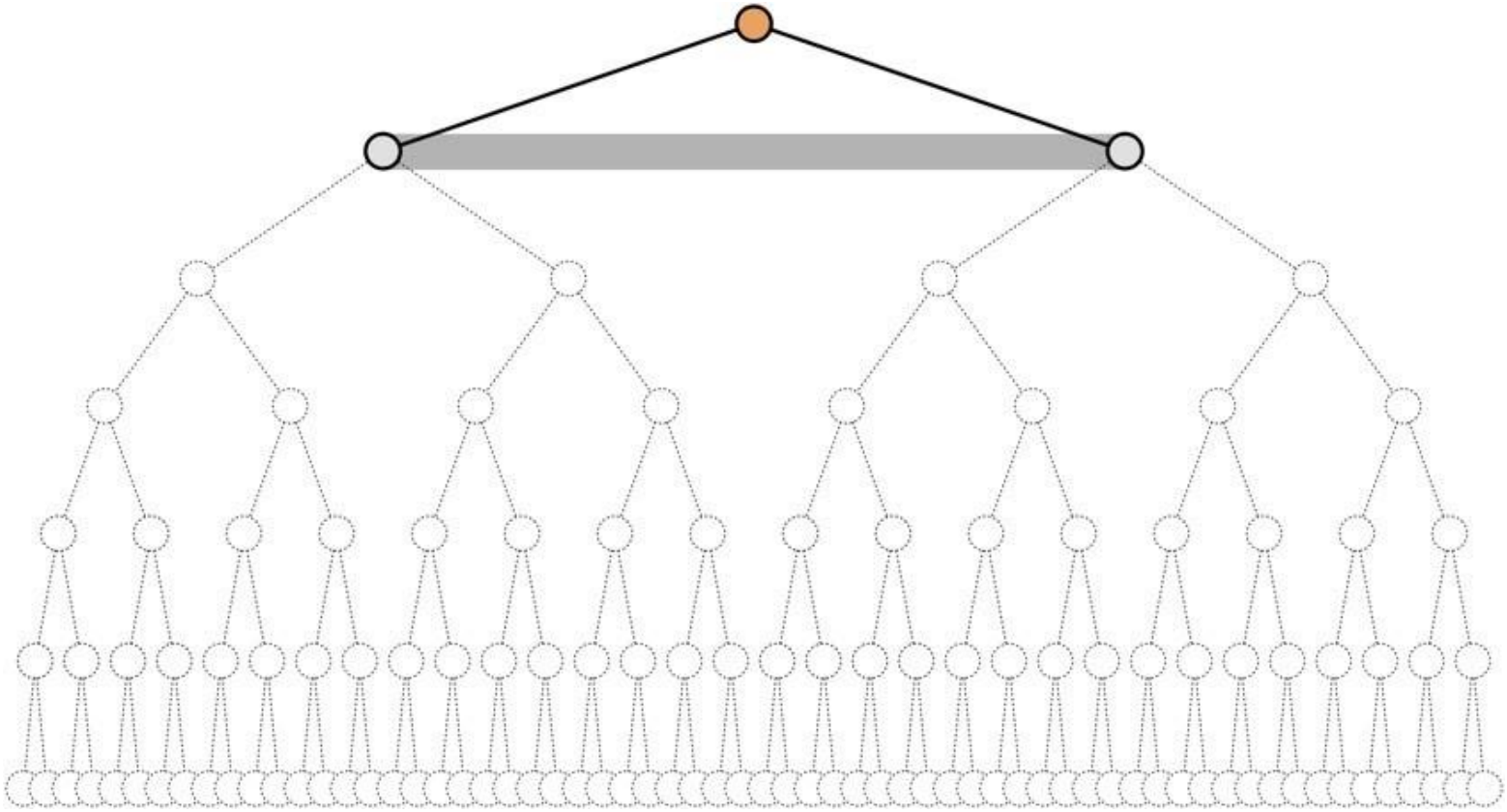
BFS



Searches layer by layer …

… with each layer organized by node depth.

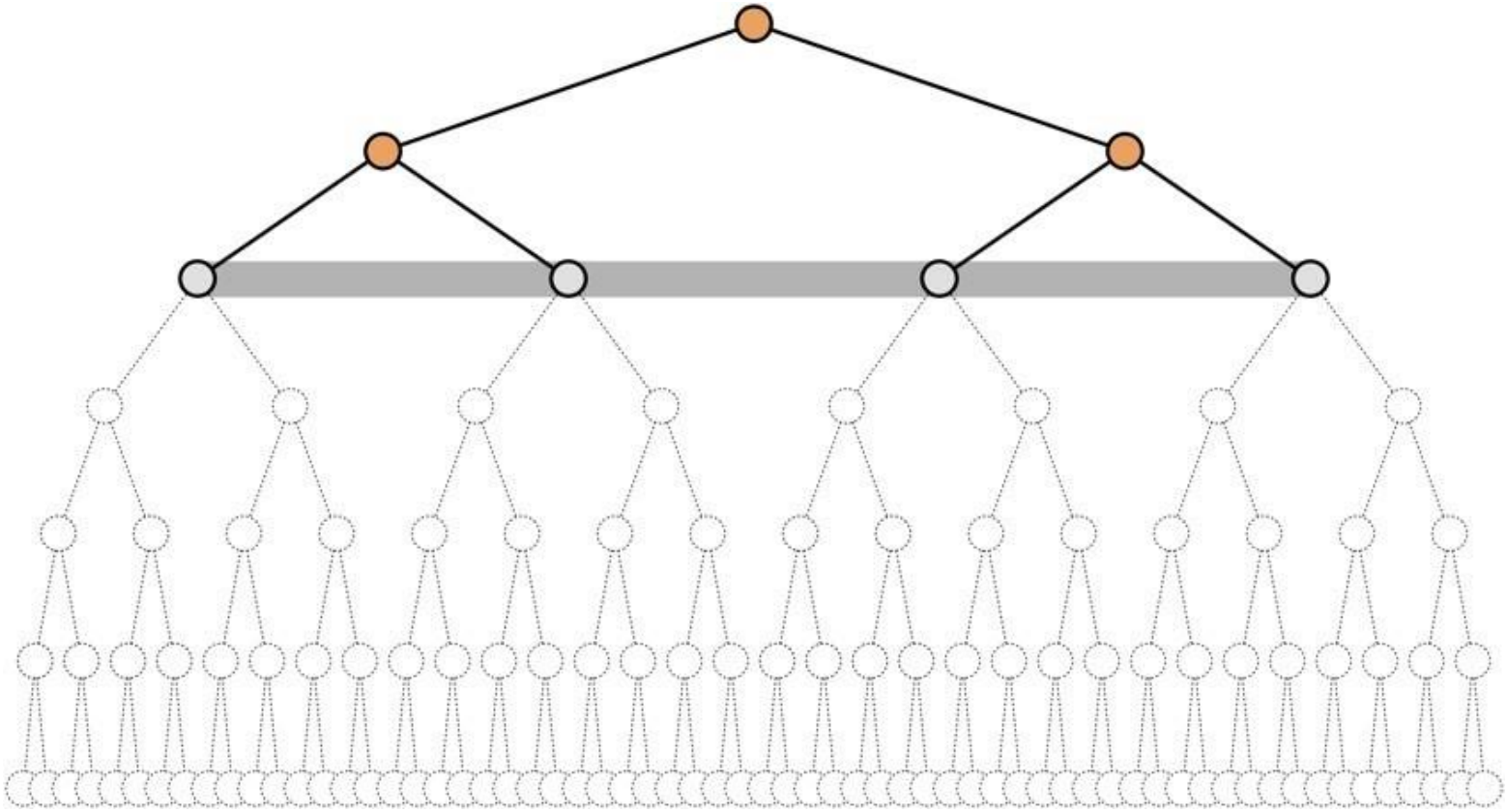
BFS



The frontier consists of nodes of similar depth (horizontal).

Search proceeds by exhausting one layer at a time.

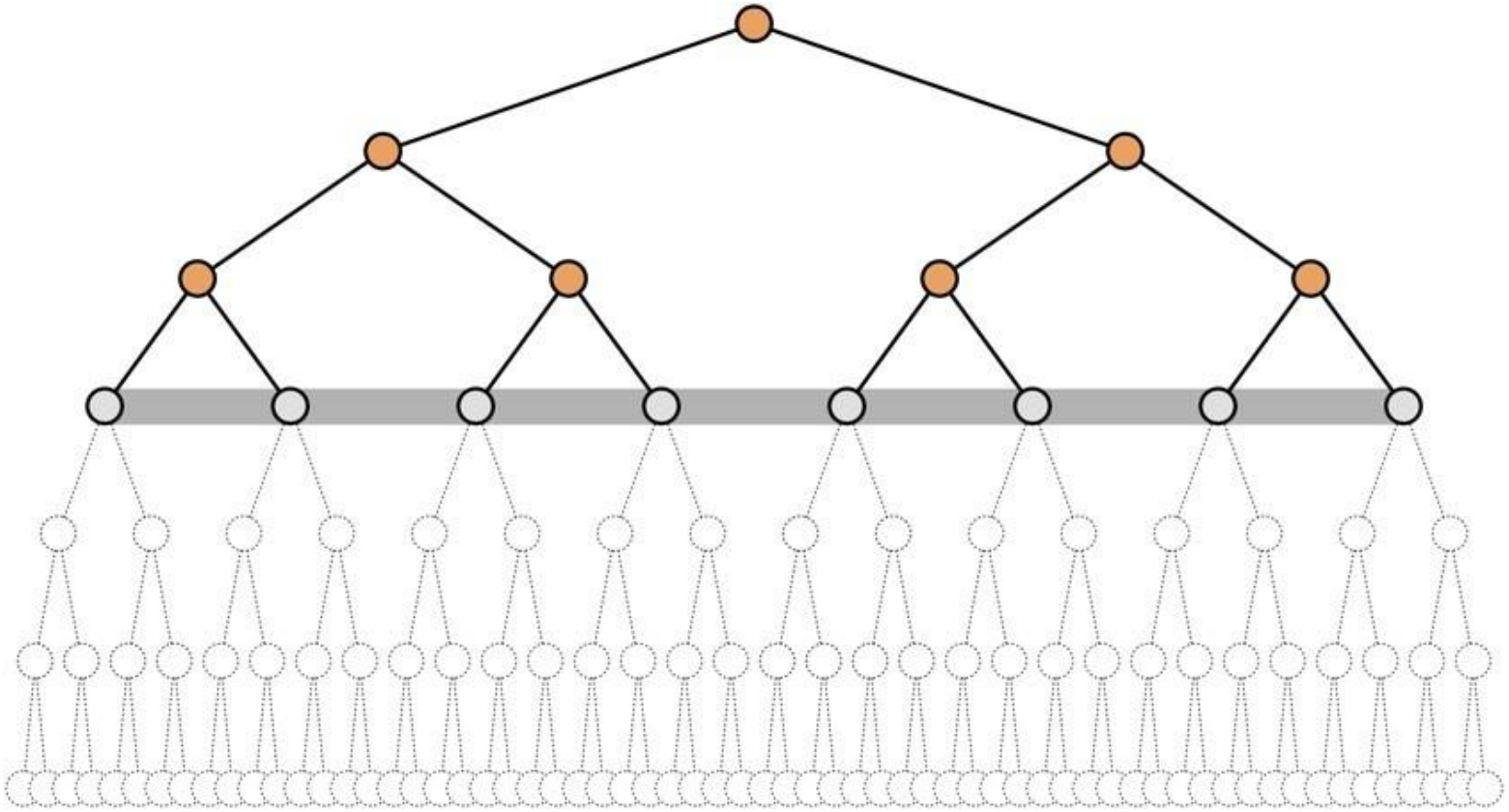
BFS



The frontier consists of nodes of similar depth (horizontal).

Search proceeds by exhausting one layer at a time.

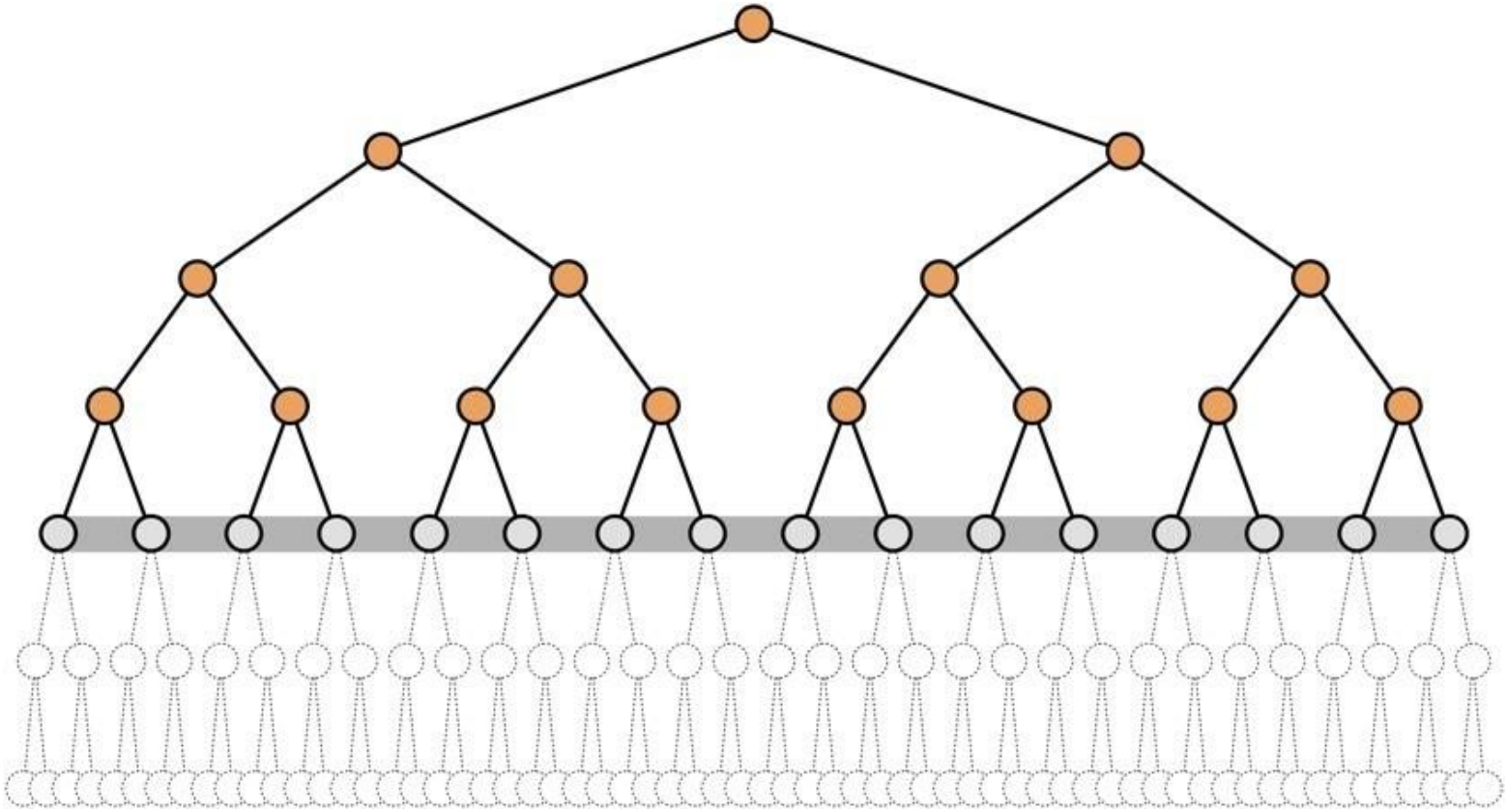
BFS



The frontier consists of nodes of similar depth (horizontal).

Search proceeds by exhausting one layer at a time.

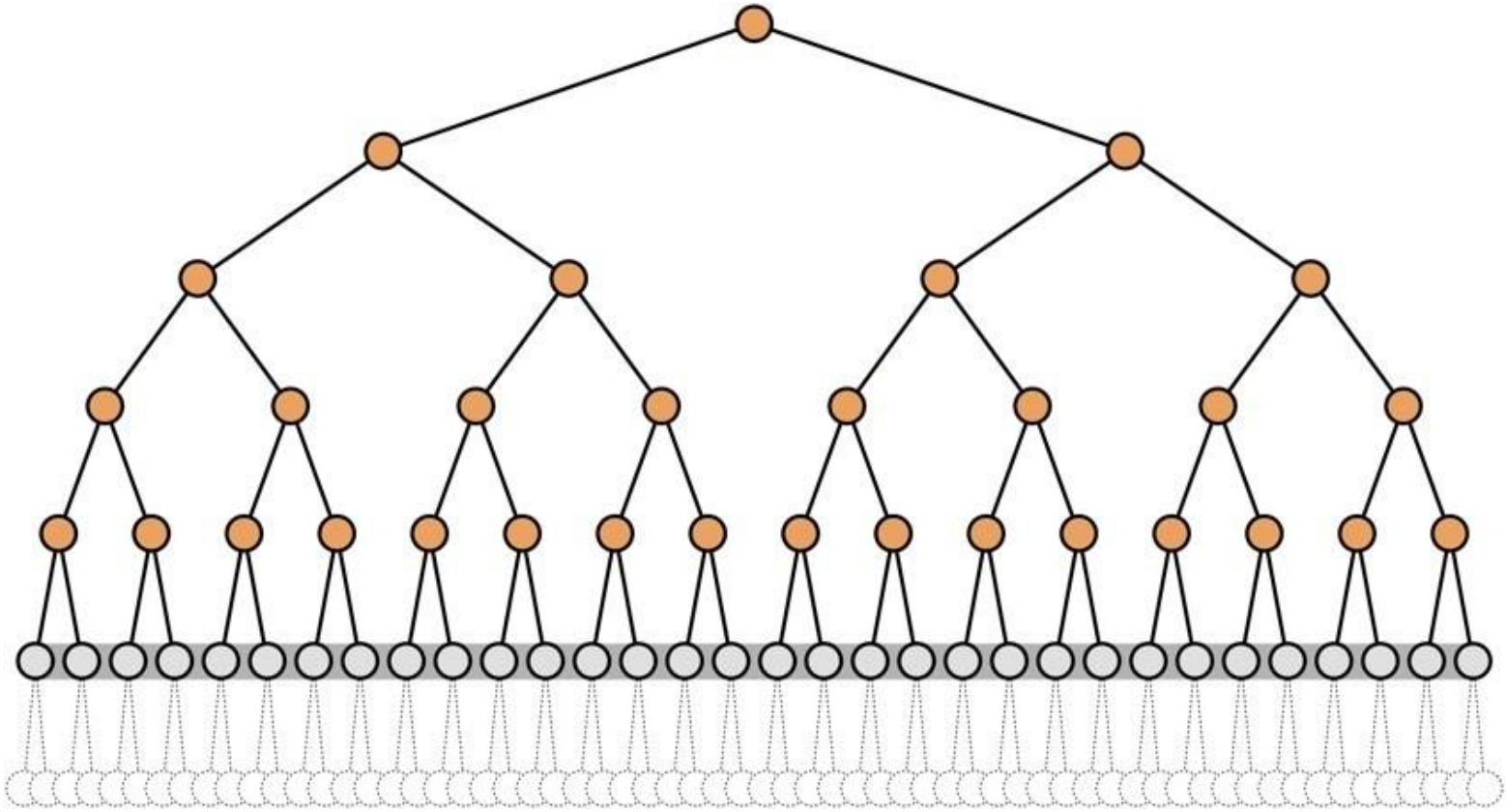
BFS



The frontier consists of nodes of similar depth (horizontal).

Search proceeds by exhausting one layer at a time.

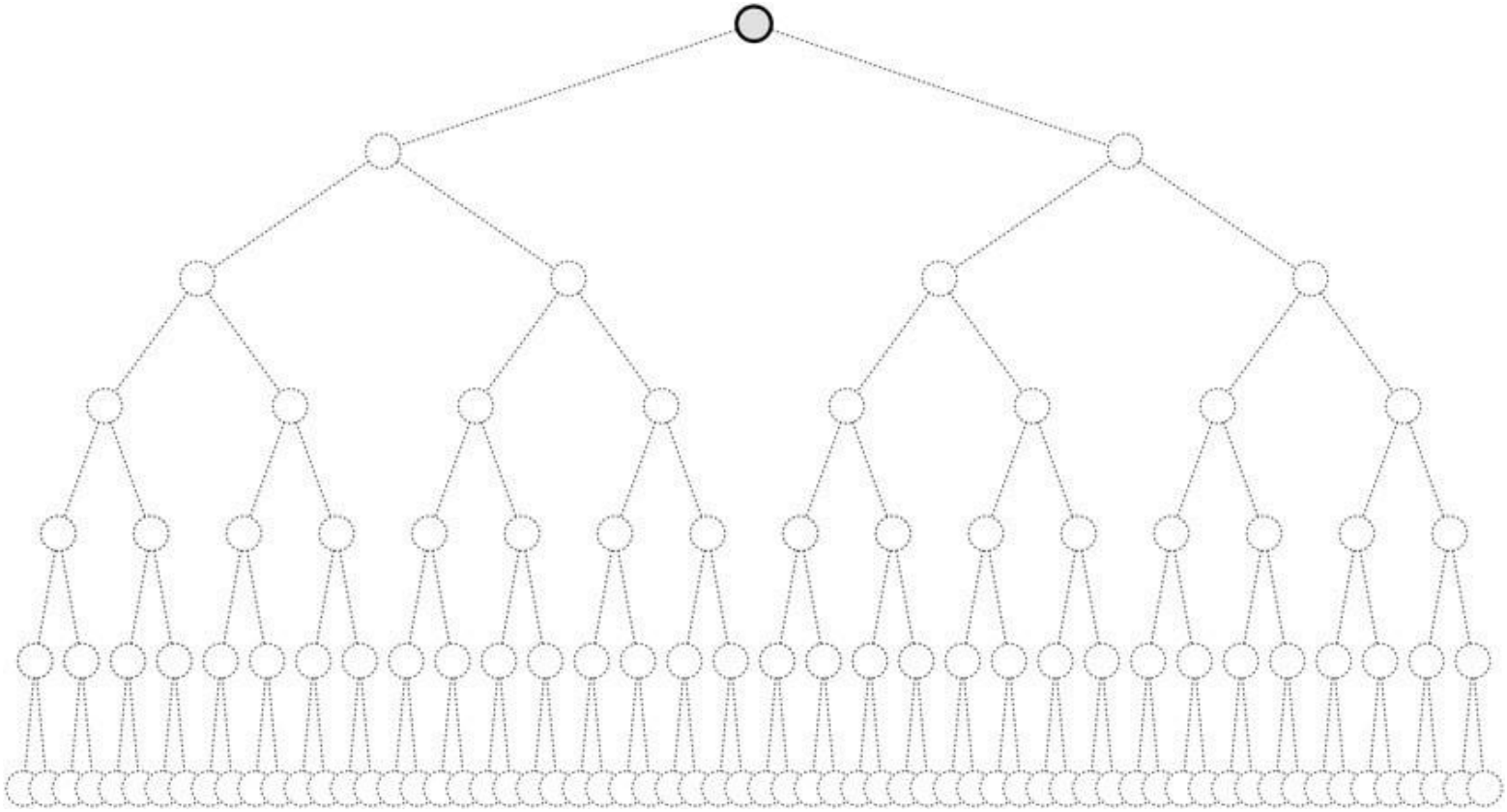
BFS



The frontier consists of nodes of similar depth (horizontal).

Search proceeds by exhausting one layer at a time.

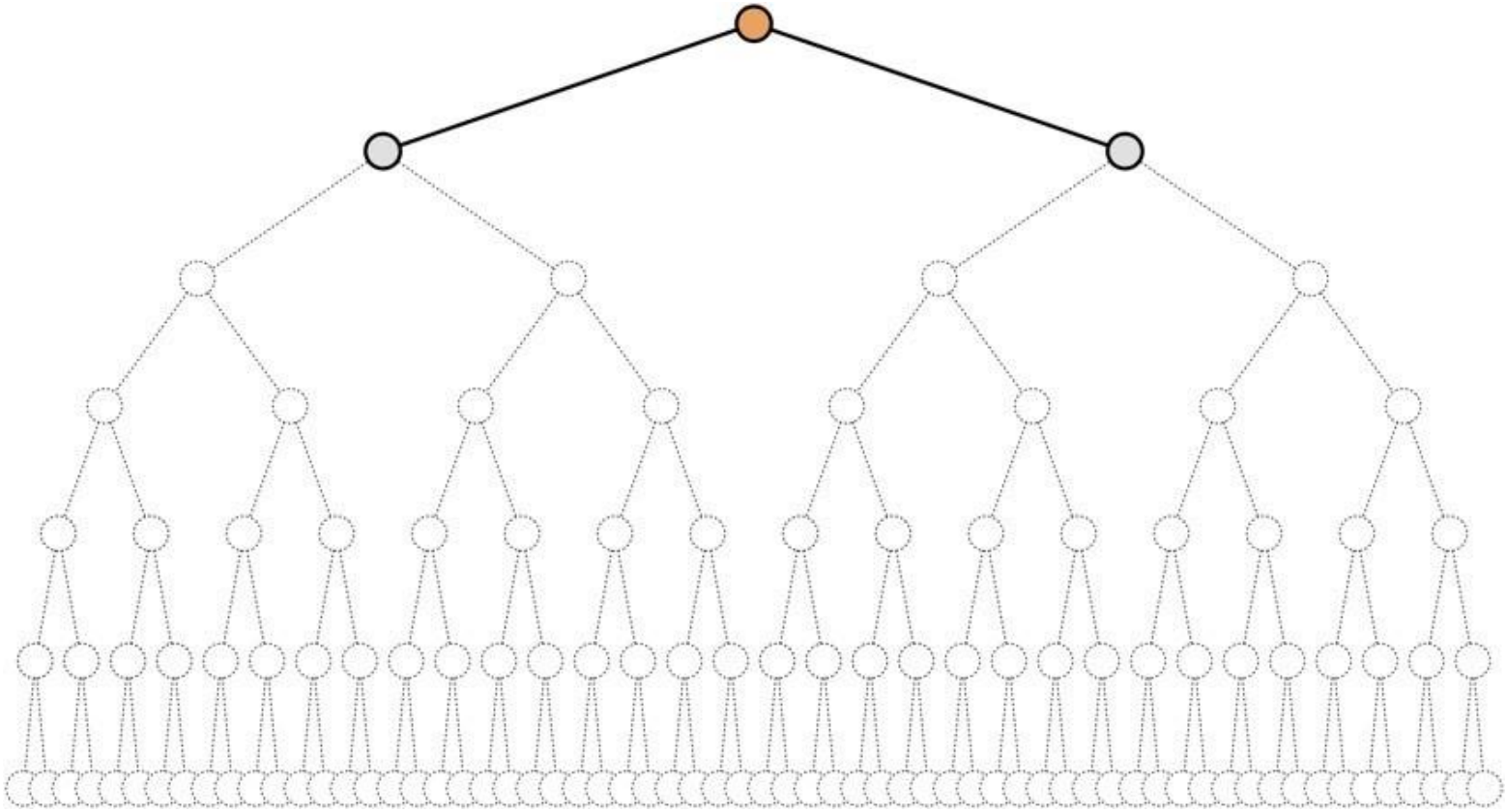
UCS



Searches layer by layer …

… with each layer organized by path cost.

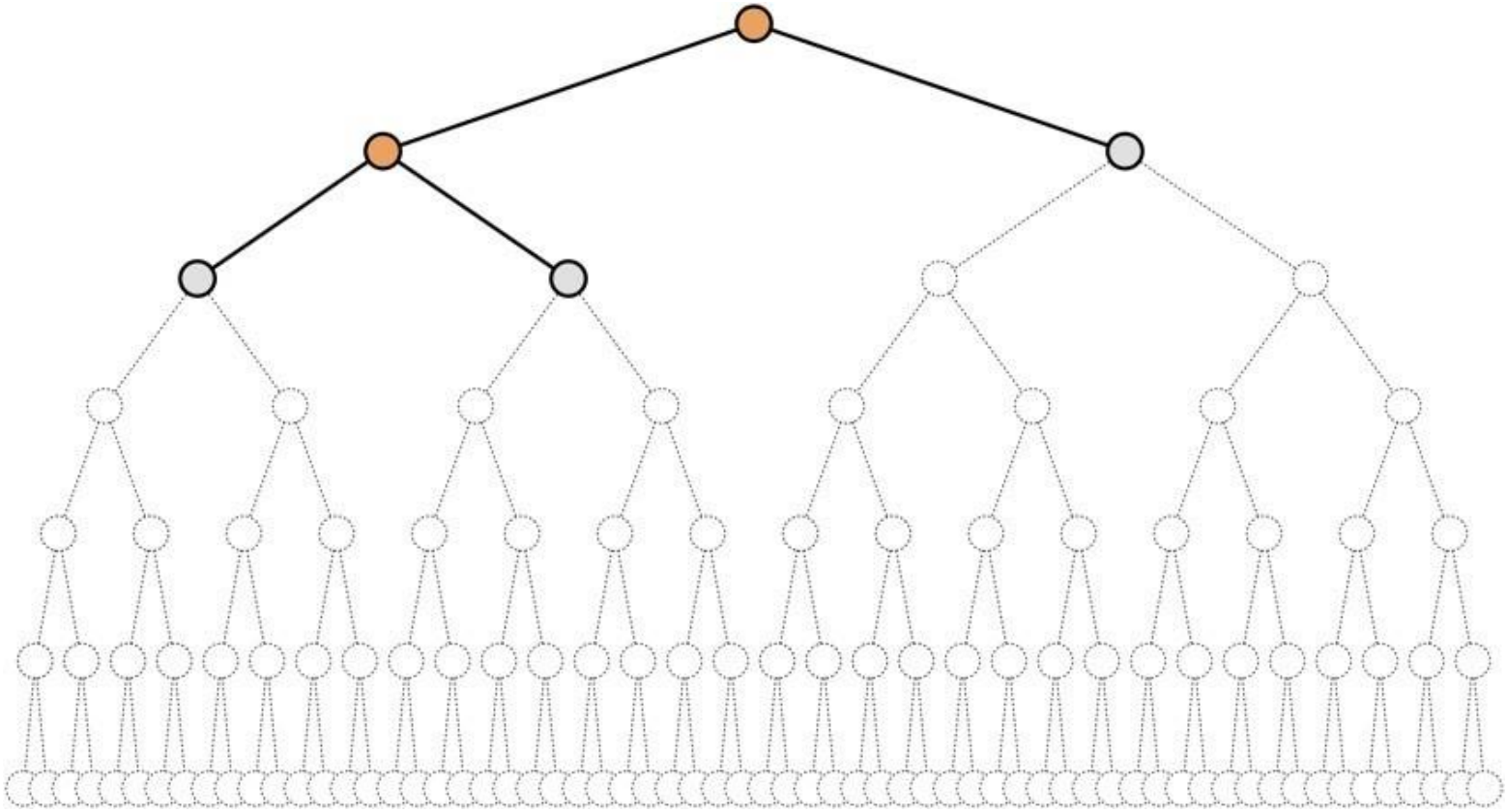
UCS



Searches layer by layer ...

... with each layer organized by path cost.

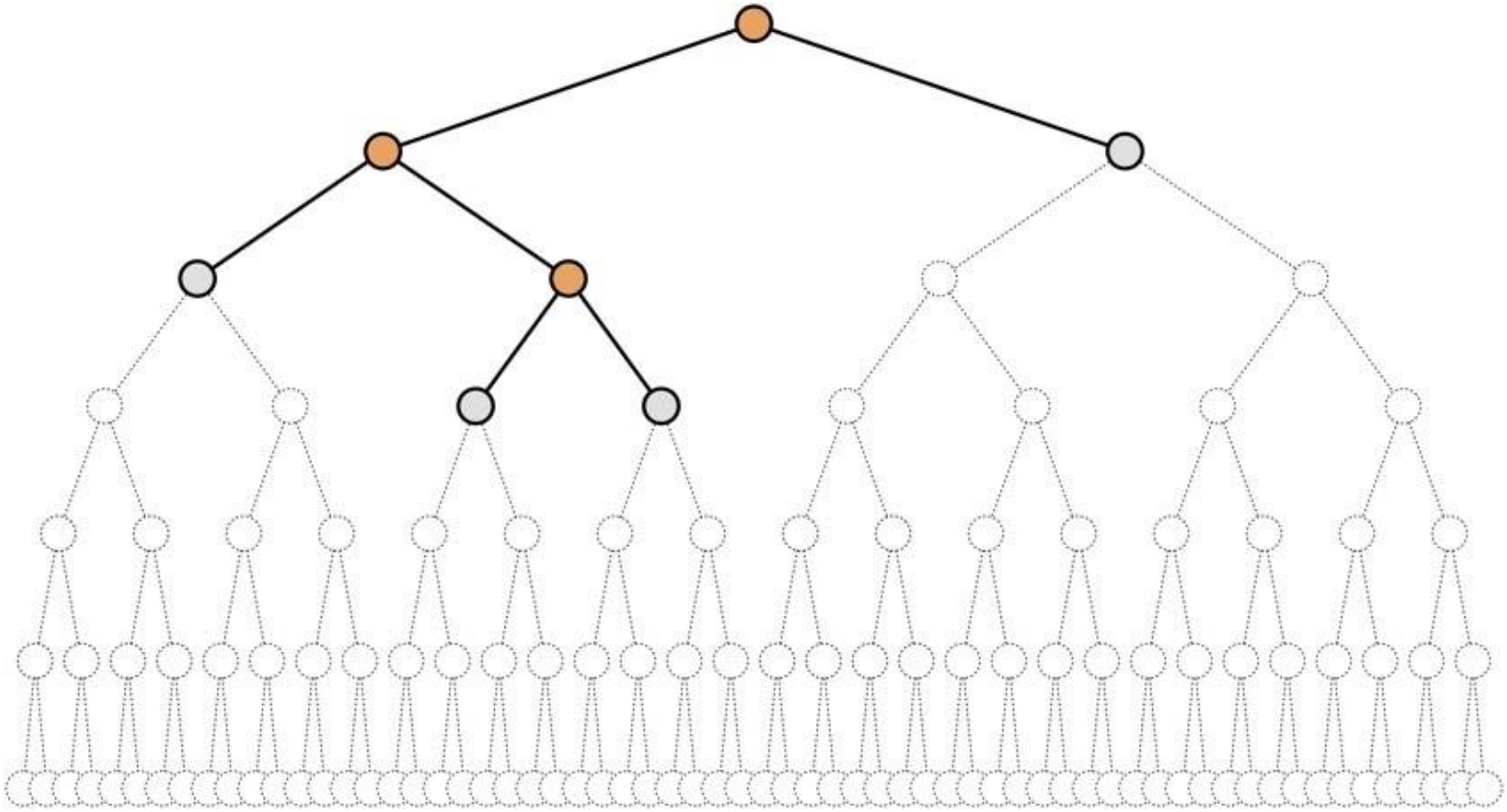
UCS



Searches layer by layer ...

... with each layer organized by path cost.

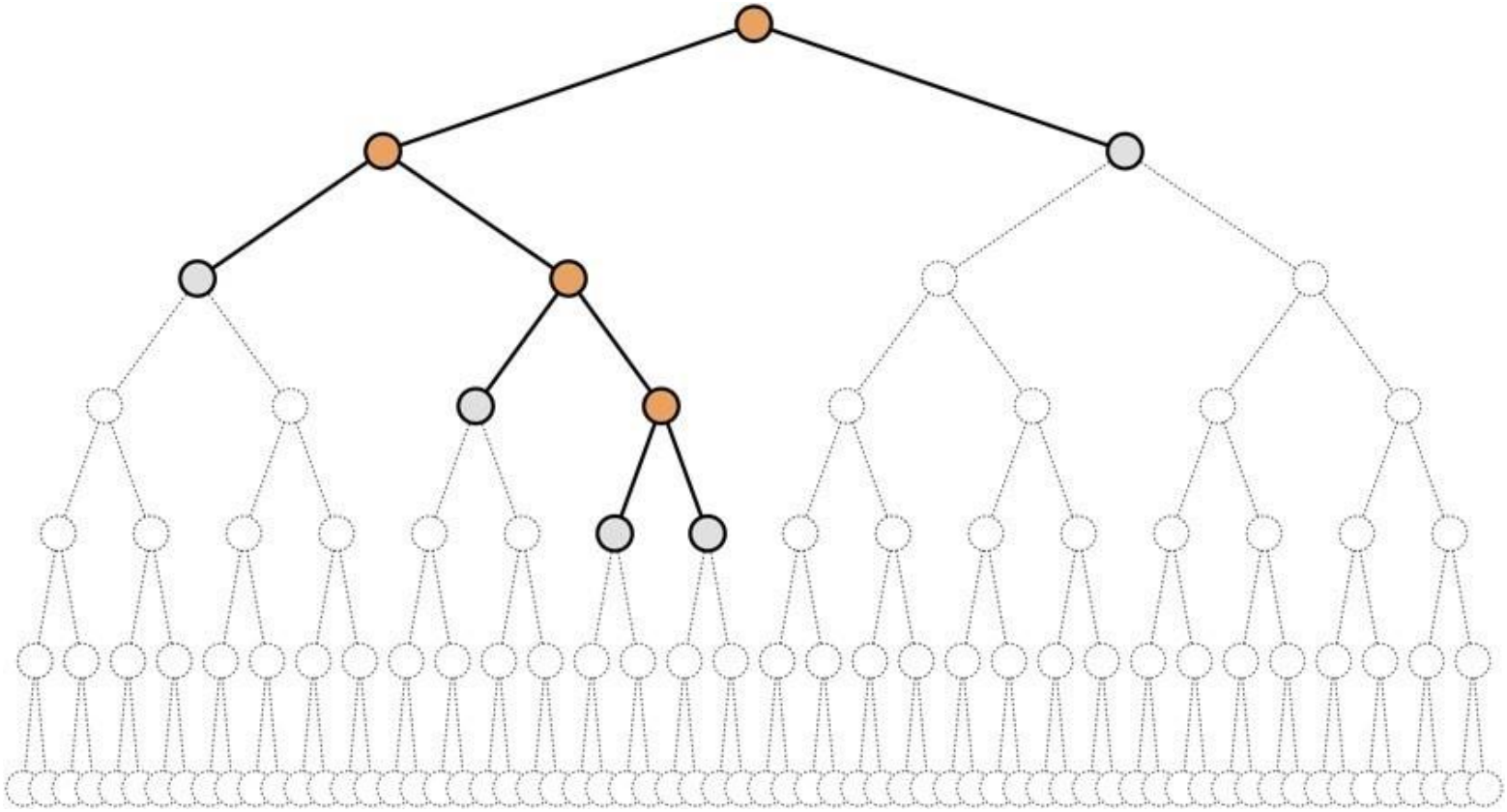
UCS



Searches layer by layer ...

... with each layer organized by path cost.

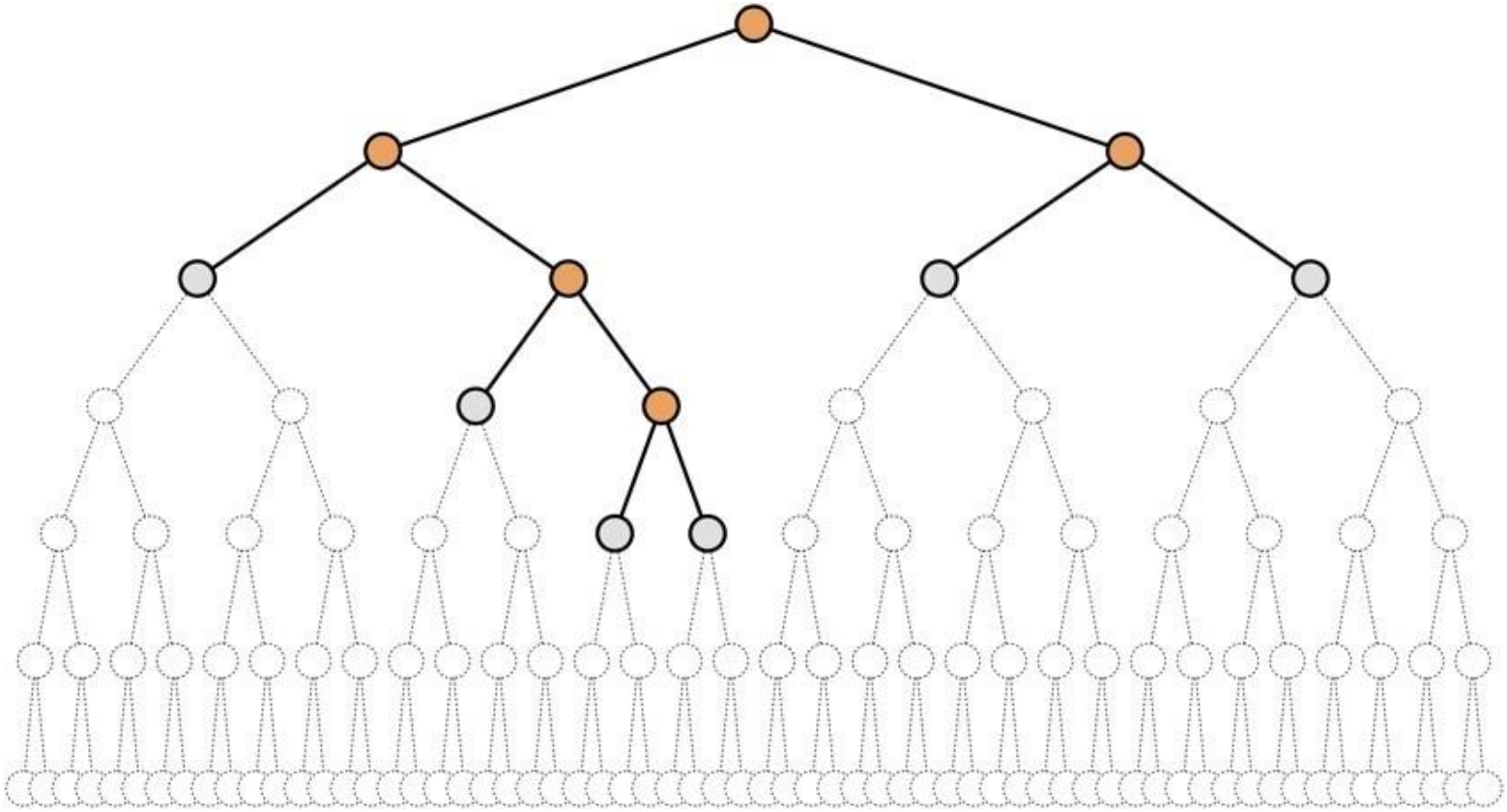
UCS



Searches layer by layer ...

... with each layer organized by path cost.

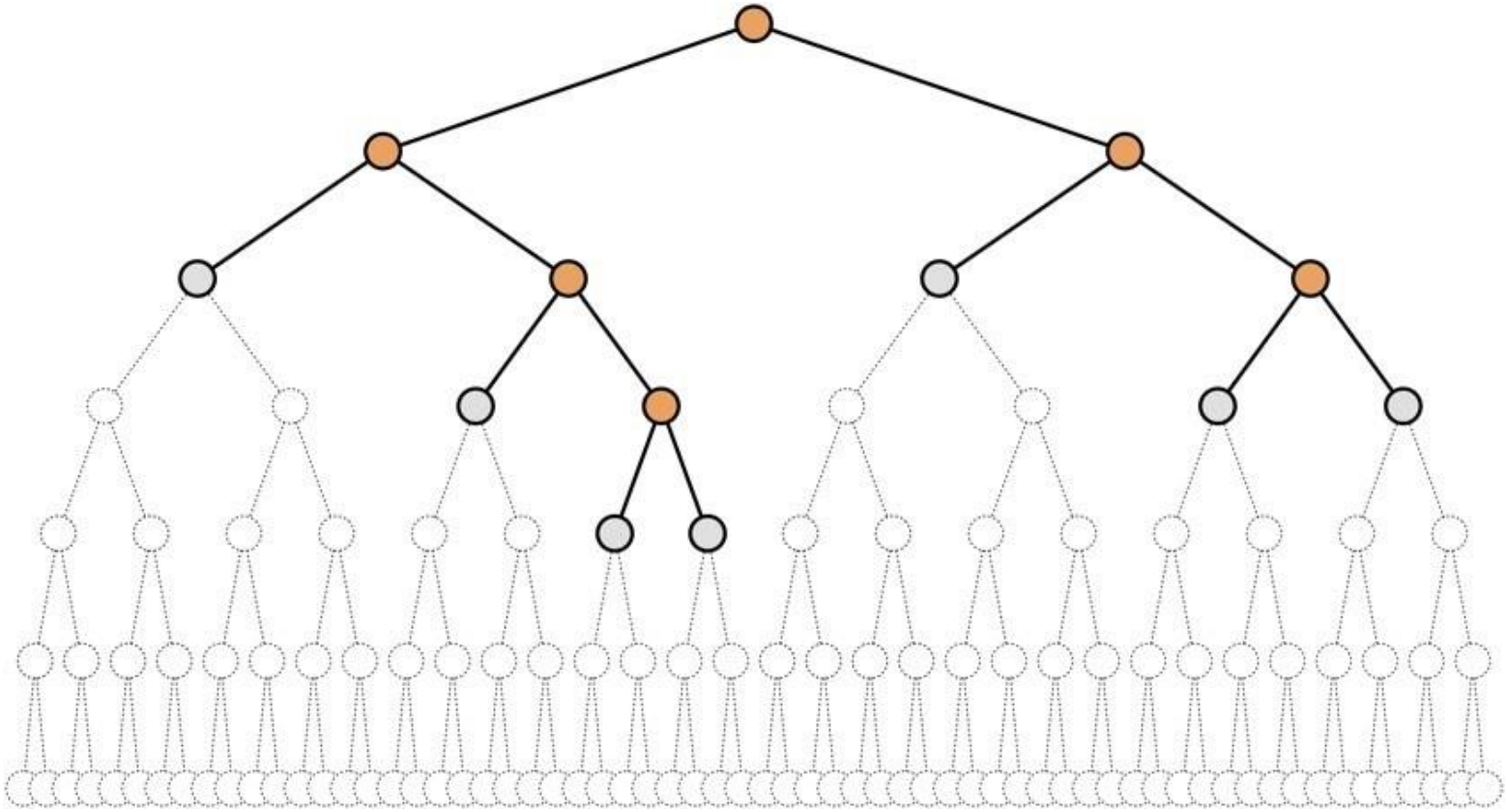
UCS



Searches layer by layer ...

... with each layer organized by path cost.

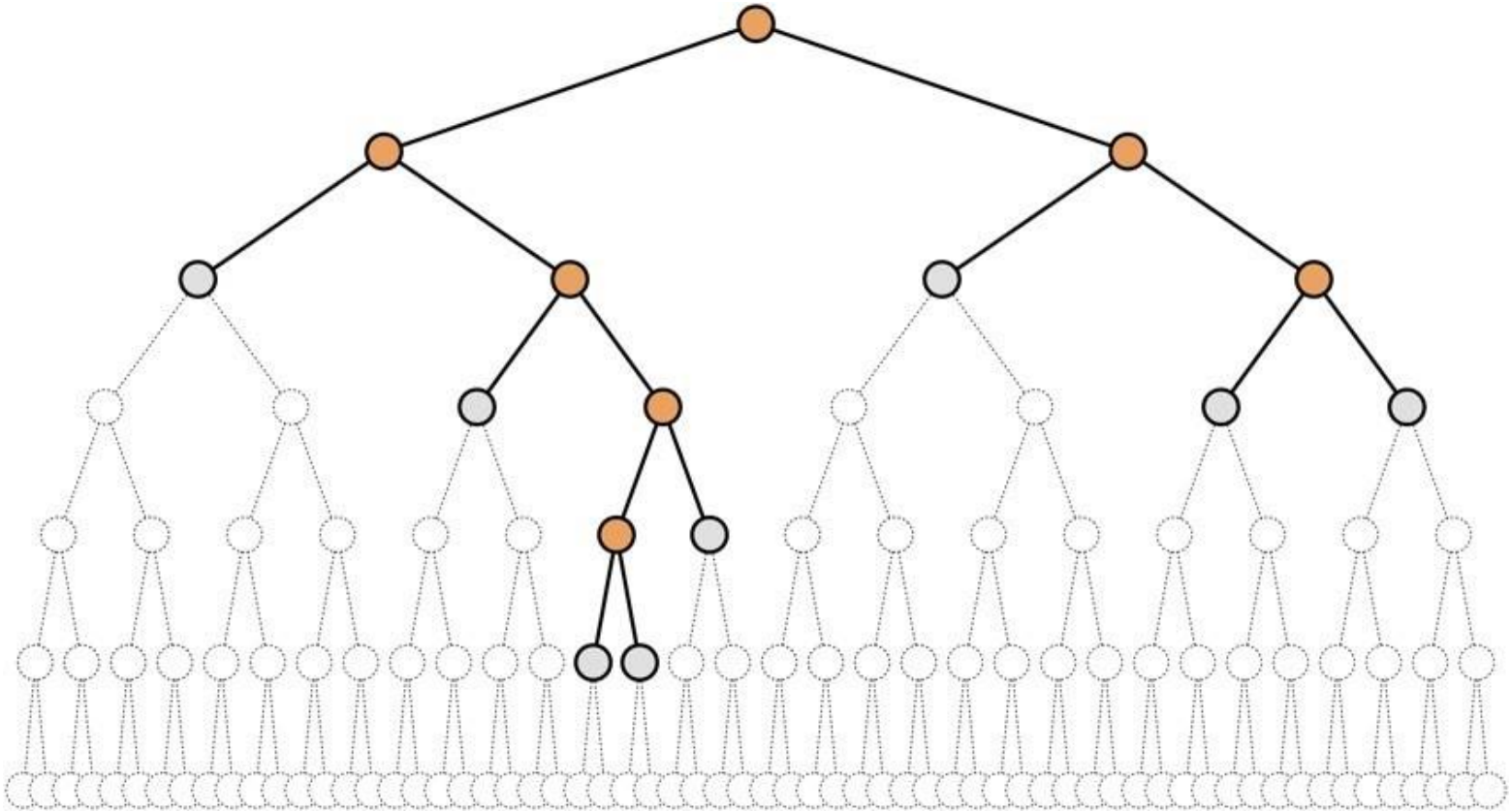
UCS



Searches layer by layer ...

... with each layer organized by path cost.

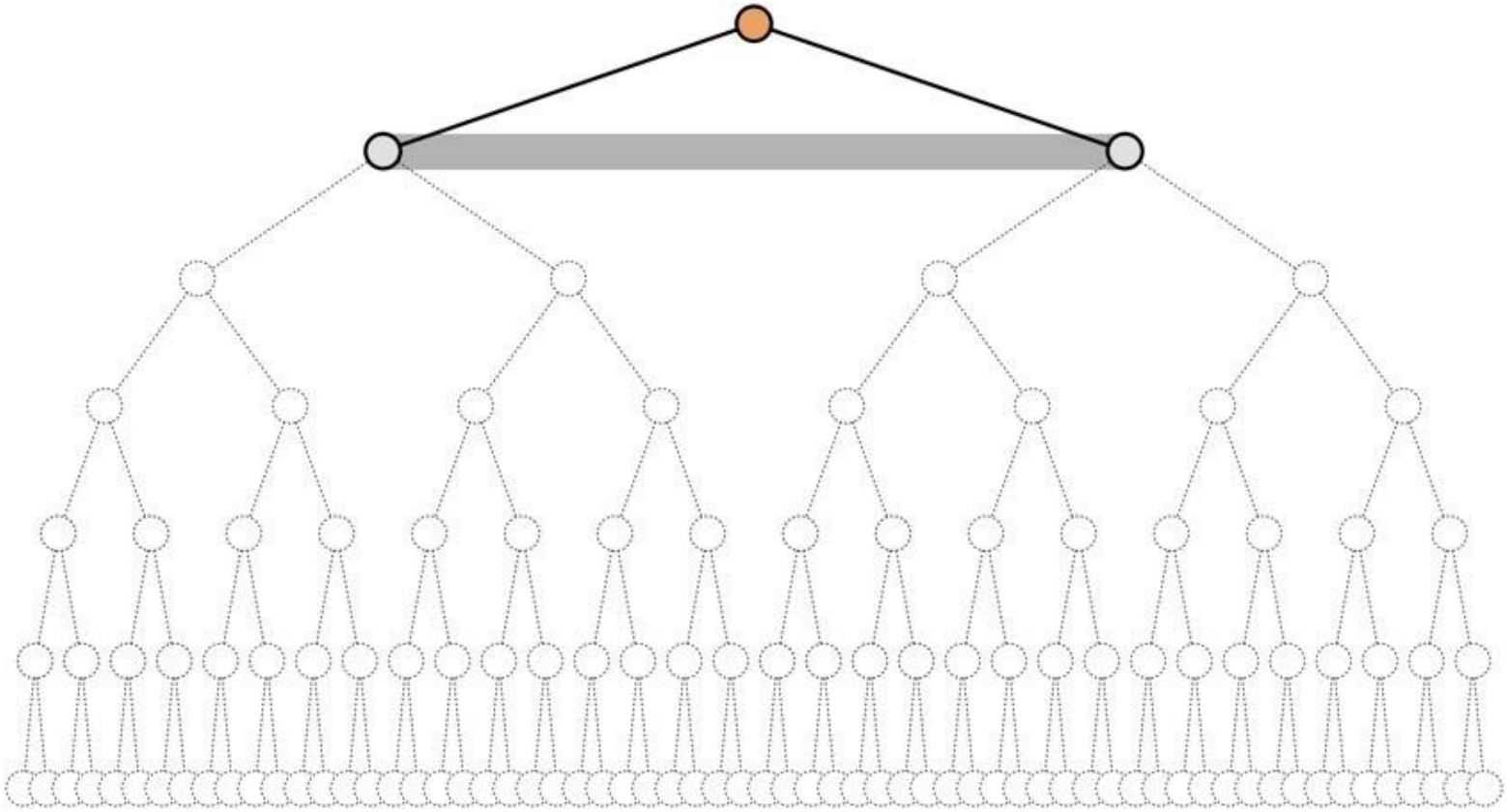
UCS



Searches layer by layer ...

... with each layer organized by path cost.

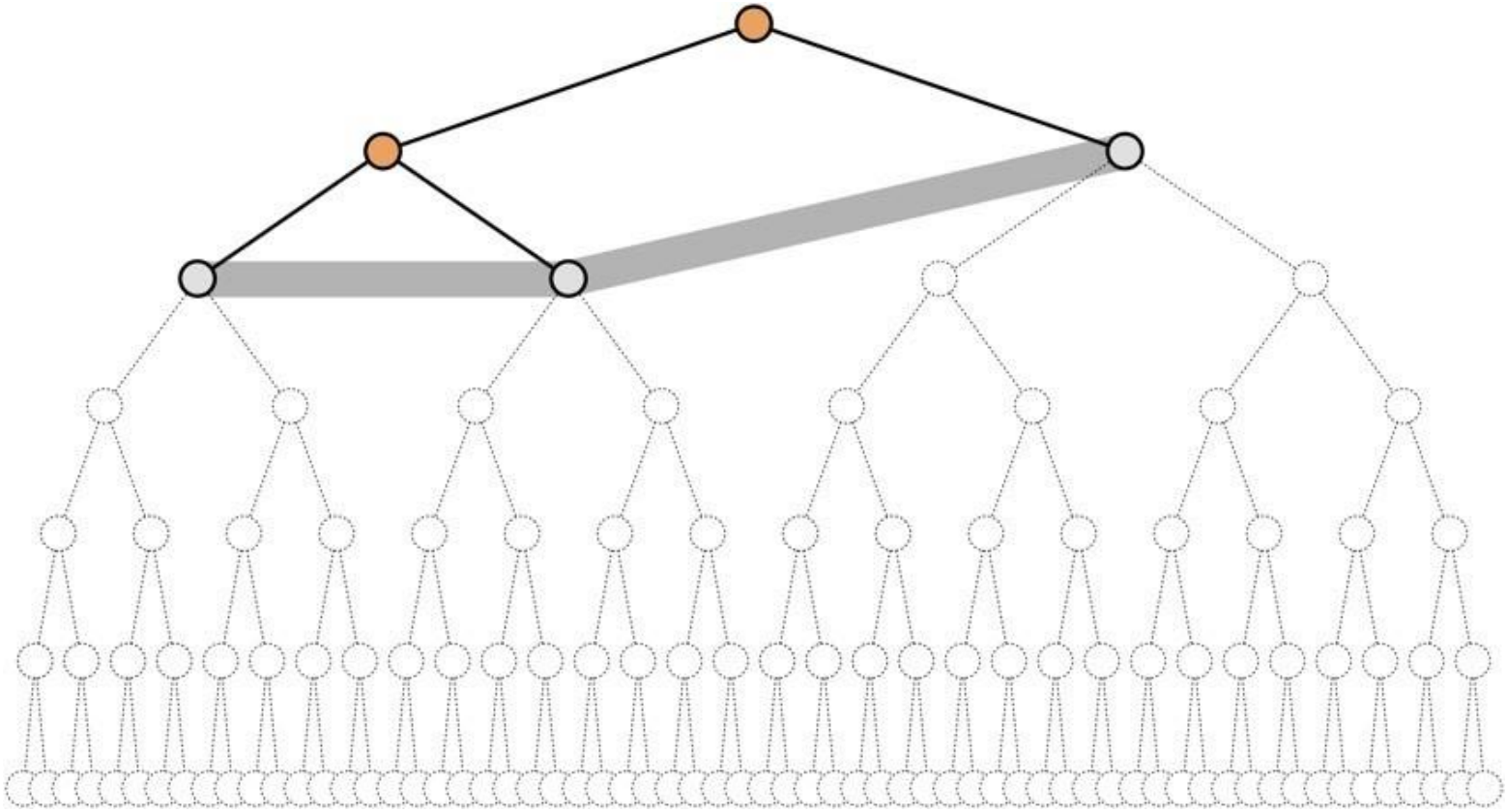
UCS



The frontier consists of nodes of various depths (jagged).

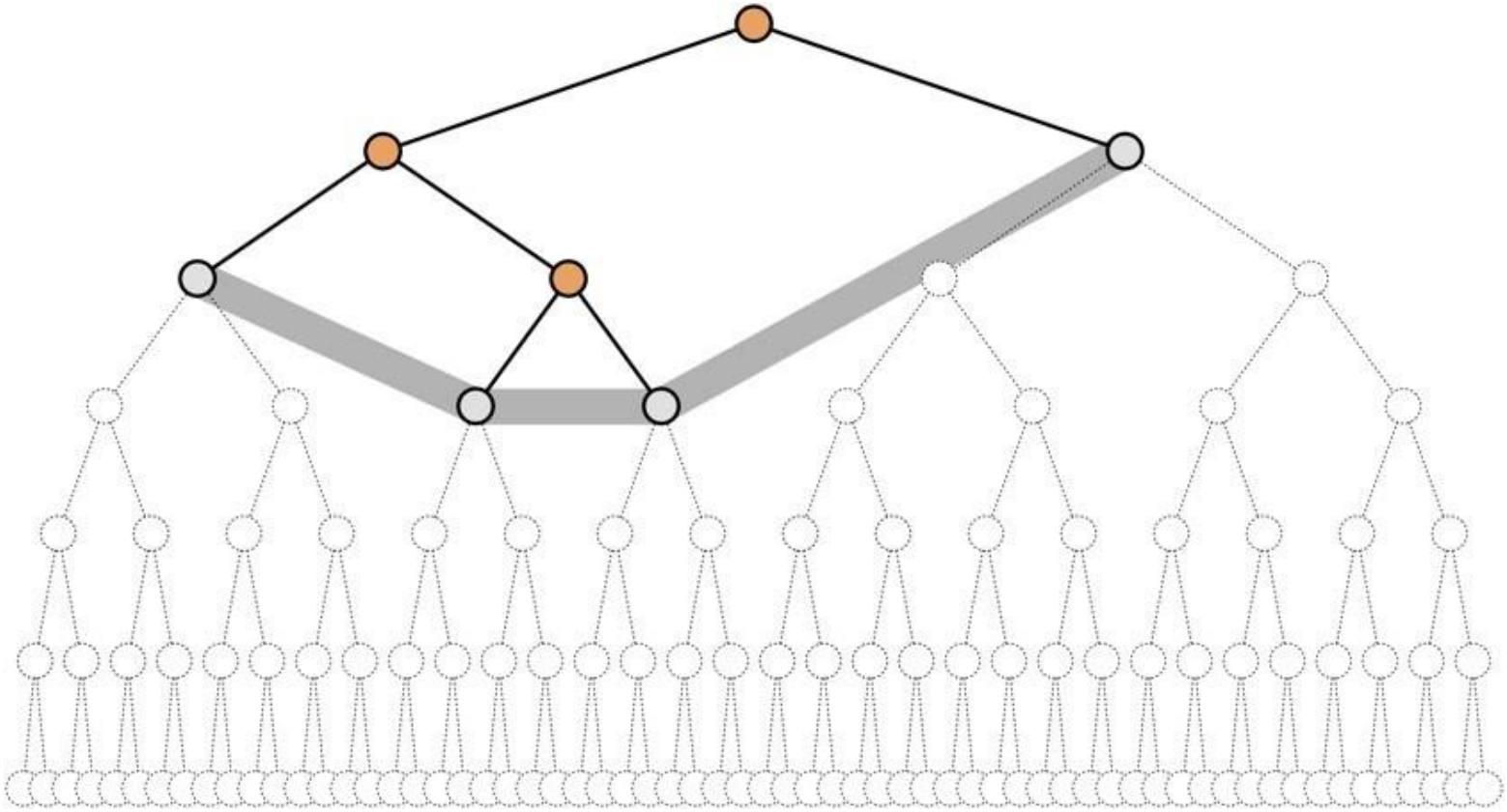
Search proceeds by expanding the lowest-cost nodes.

UCS



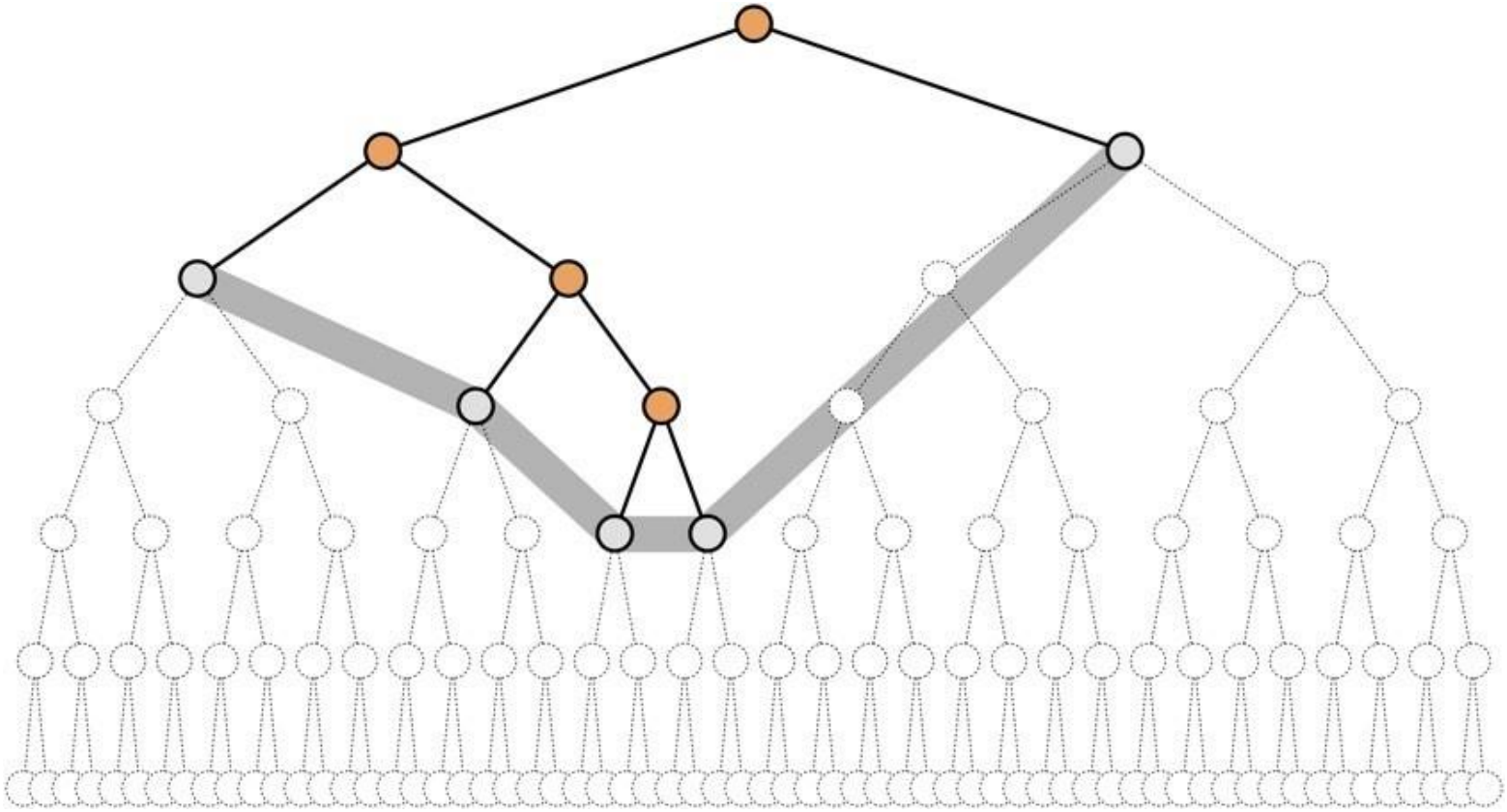
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



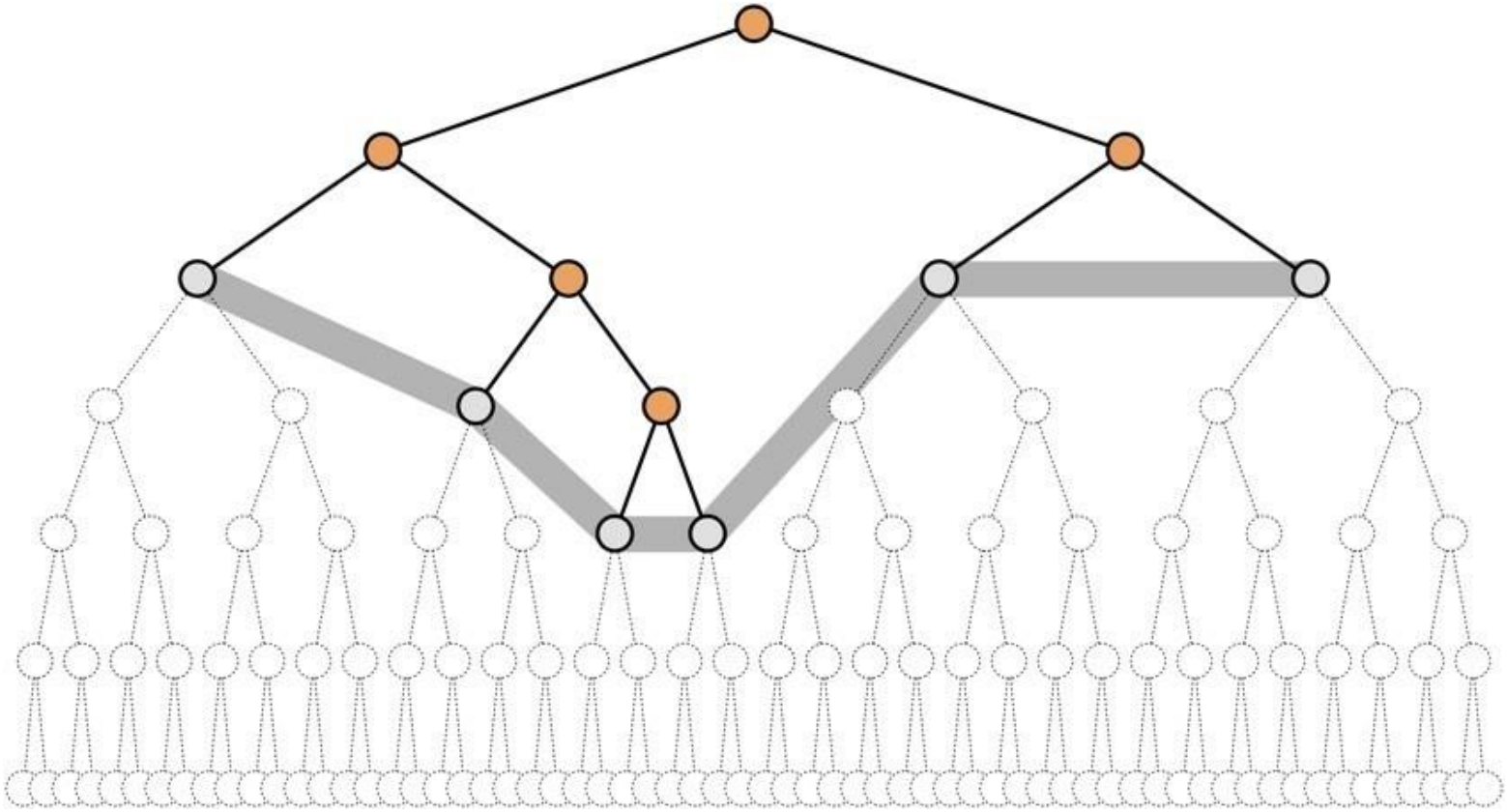
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



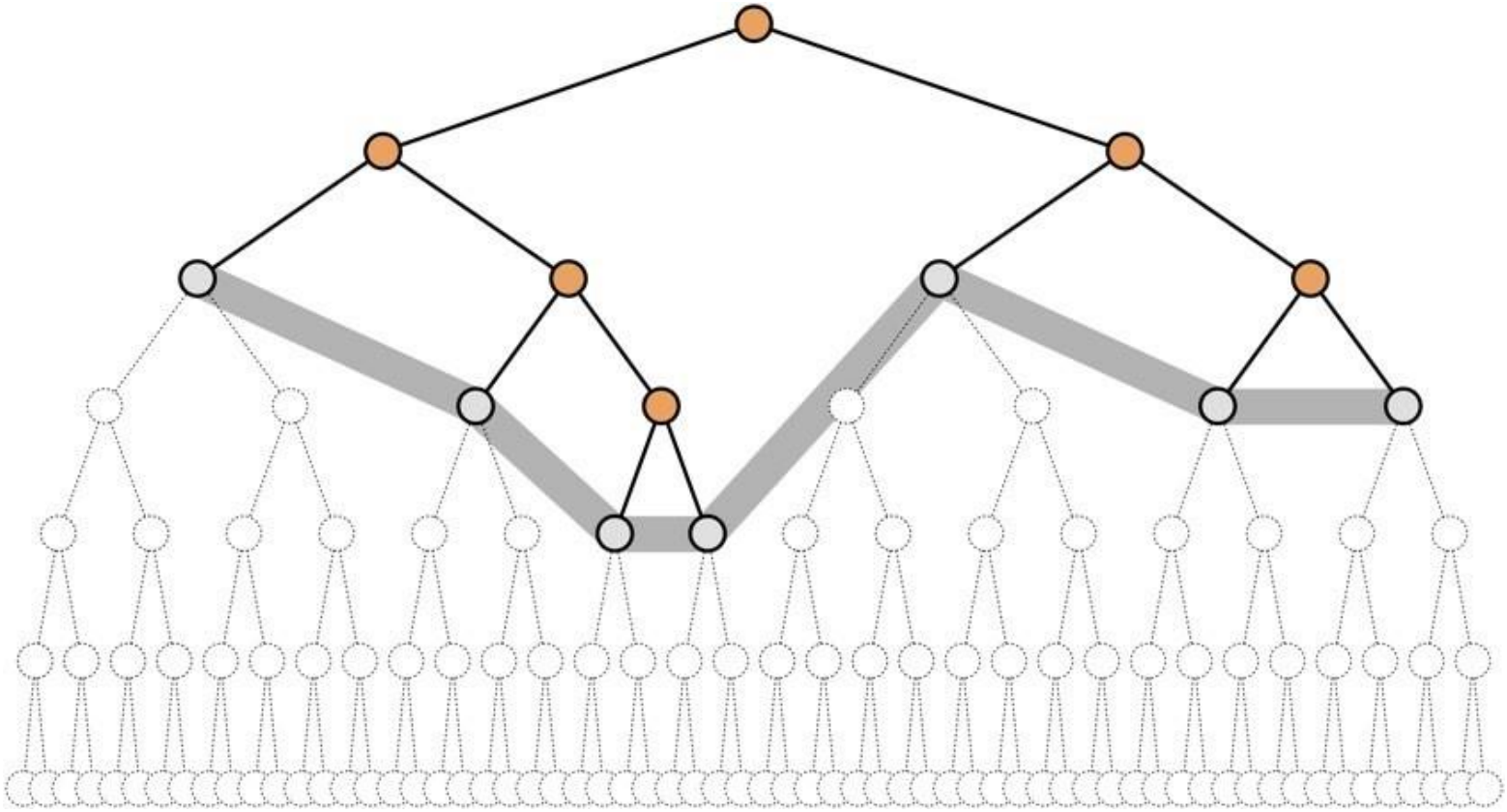
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



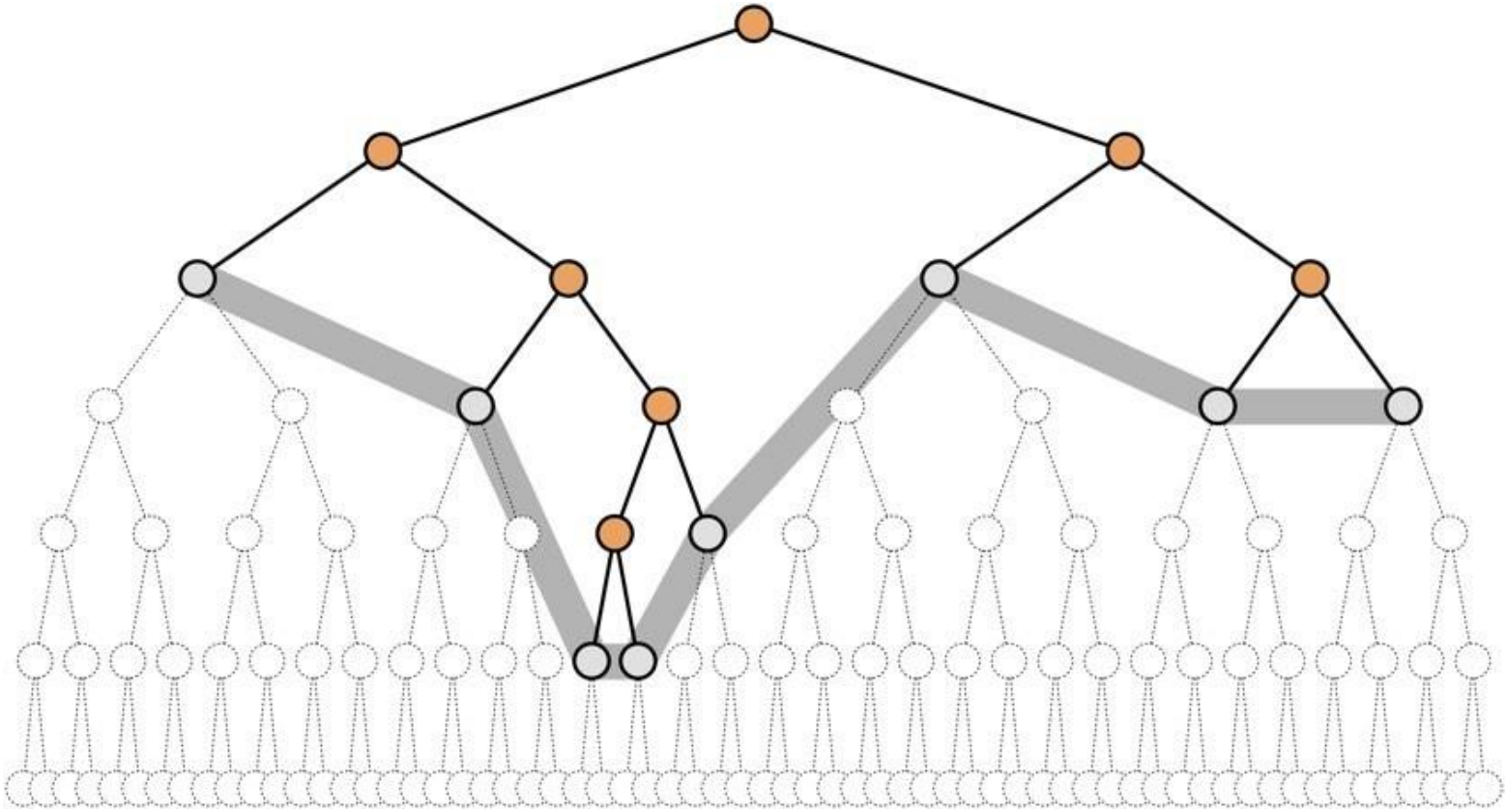
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

Recap

We can organize the algorithms into pairs where the first proceeds by layers, and the other proceeds by subtrees.

(1) Iterate on Node Depth:

- BFS searches layers of increasing node depth.
- IDS searches subtrees of increasing node depth.

Recap

We can organize the algorithms into pairs where the first proceeds by layers, and the other proceeds by subtrees.

(1) Iterate on Node Depth:

- BFS searches layers of increasing node depth.
- IDS searches subtrees of increasing node depth.

(2) Iterate on Path Cost + Heuristic Function:

- A* searches layers of increasing path cost + heuristic function.
- IDA* searches subtrees of increasing path cost + heuristic function.

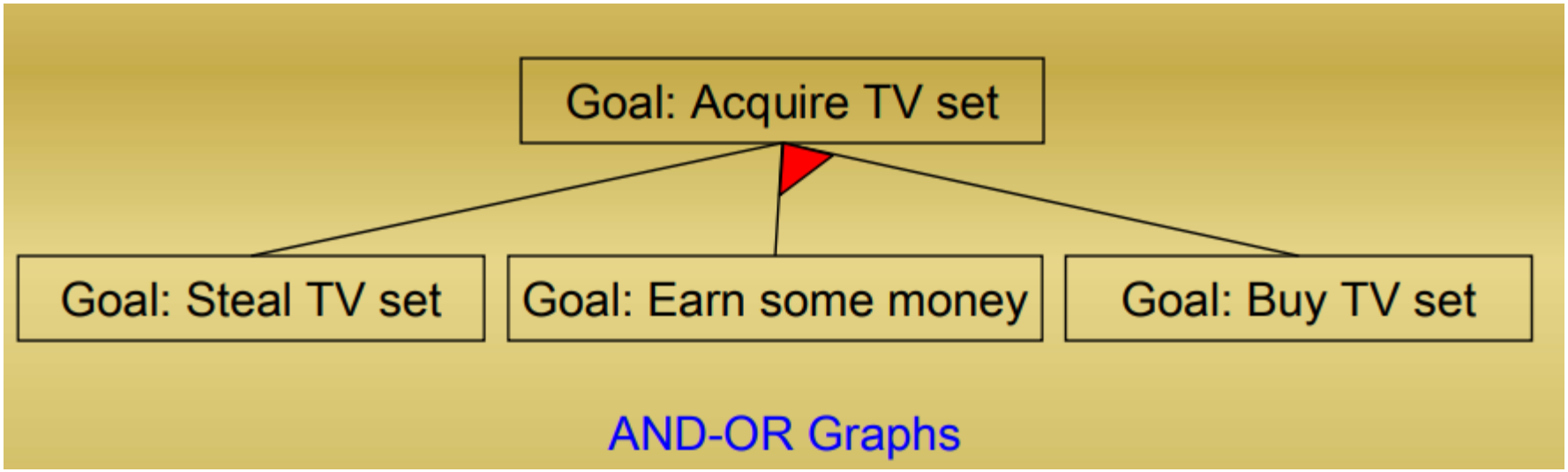
Recap

Which cost function?

- UCS searches layers of increasing path cost.
- Greedy best first search searches layers of increasing heuristic function.
- A* search searches layers of increasing path cost + heuristic function.

AO* Algorithm

AO* is used to perform a heuristic search of an cyclic AND-OR graph.



AND-OR Graph

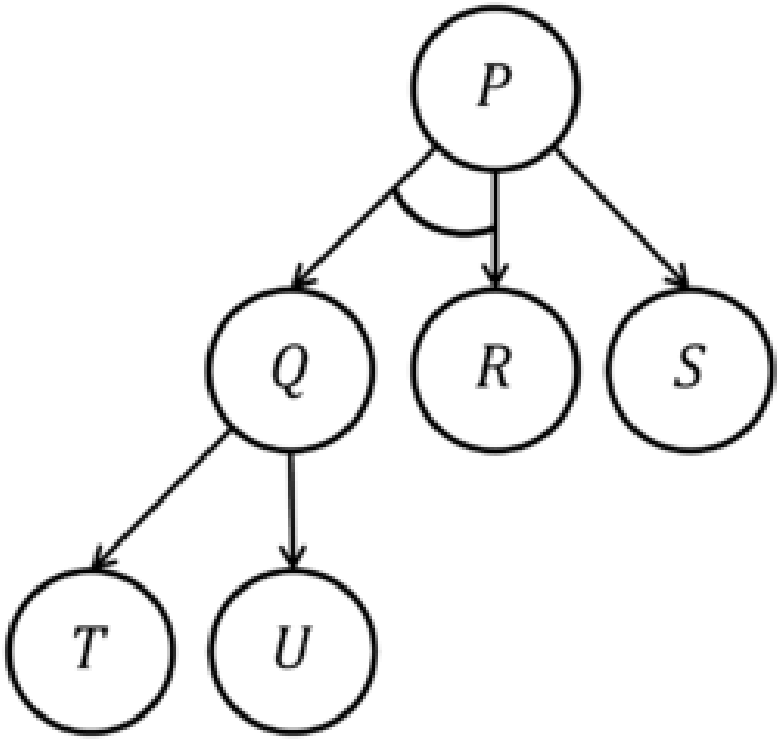
- **Problem decomposition: Some problems are best represented as achieving subgoals, some of which achieved simultaneously and independently (AND).**

- When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND-OR trees are used for representing the solution.

Example

Represents the search space for solving the problem P, using the goal-reduction methods:

P if Q and R
P if S
Q if T
Q if U



Solution to AND-OR Graphs

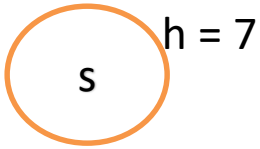
A solution in an AND-OR tree is a *sub tree* whose *leaves* are included in the goal set.

Cost function: sum of costs in AND node $f(n) = f(n_1) + f(n_2) + \ldots + f(n_k)$

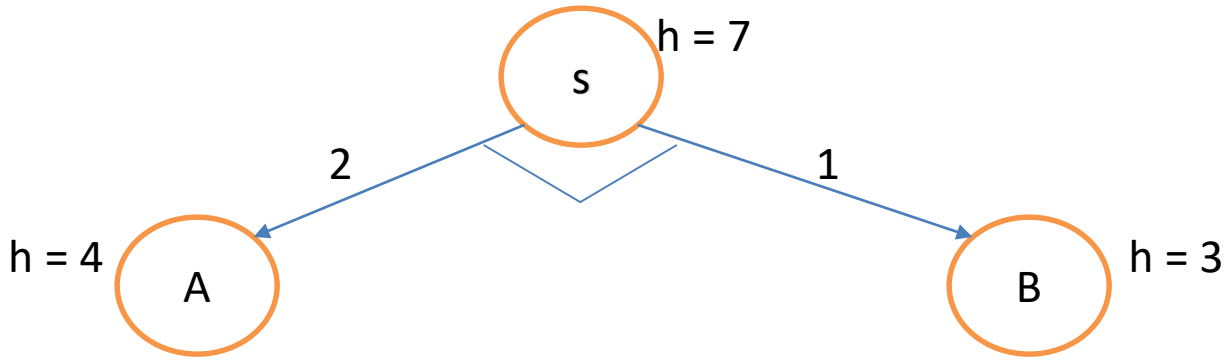
A* can't be used for AND-OR graphs. Why?

How can we extend A* to search AND/OR trees? The AO* algorithm

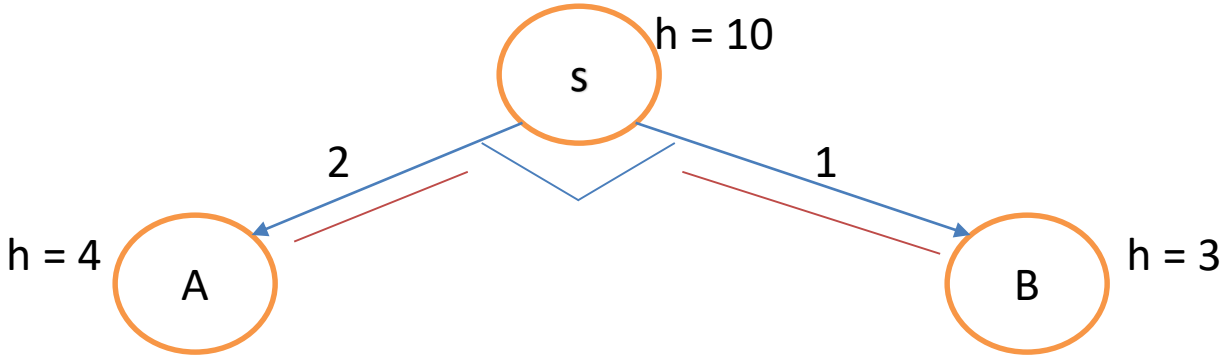
Example



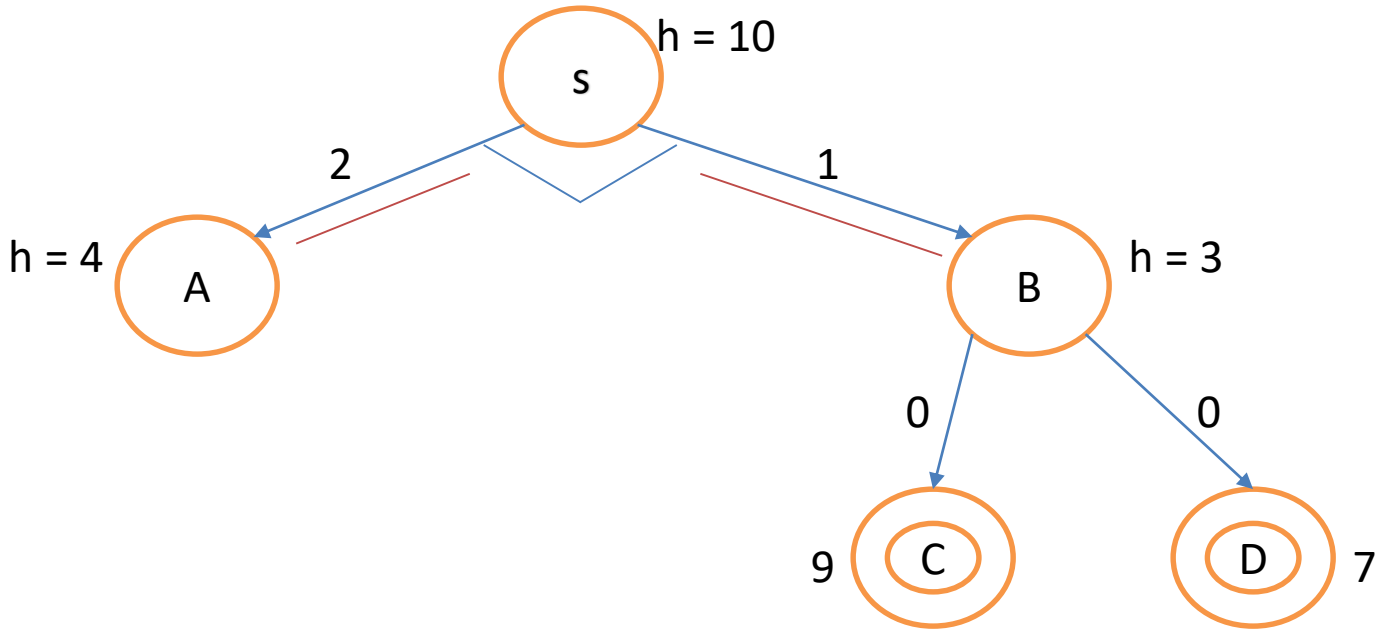
Example



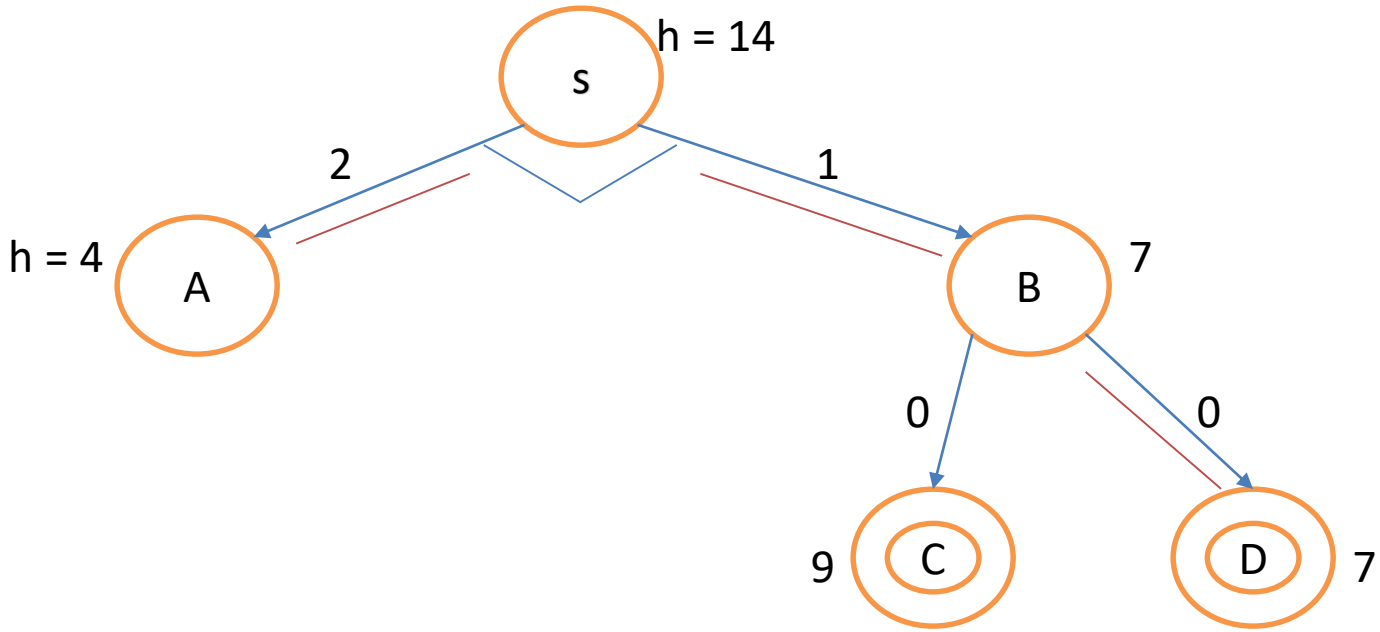
Example



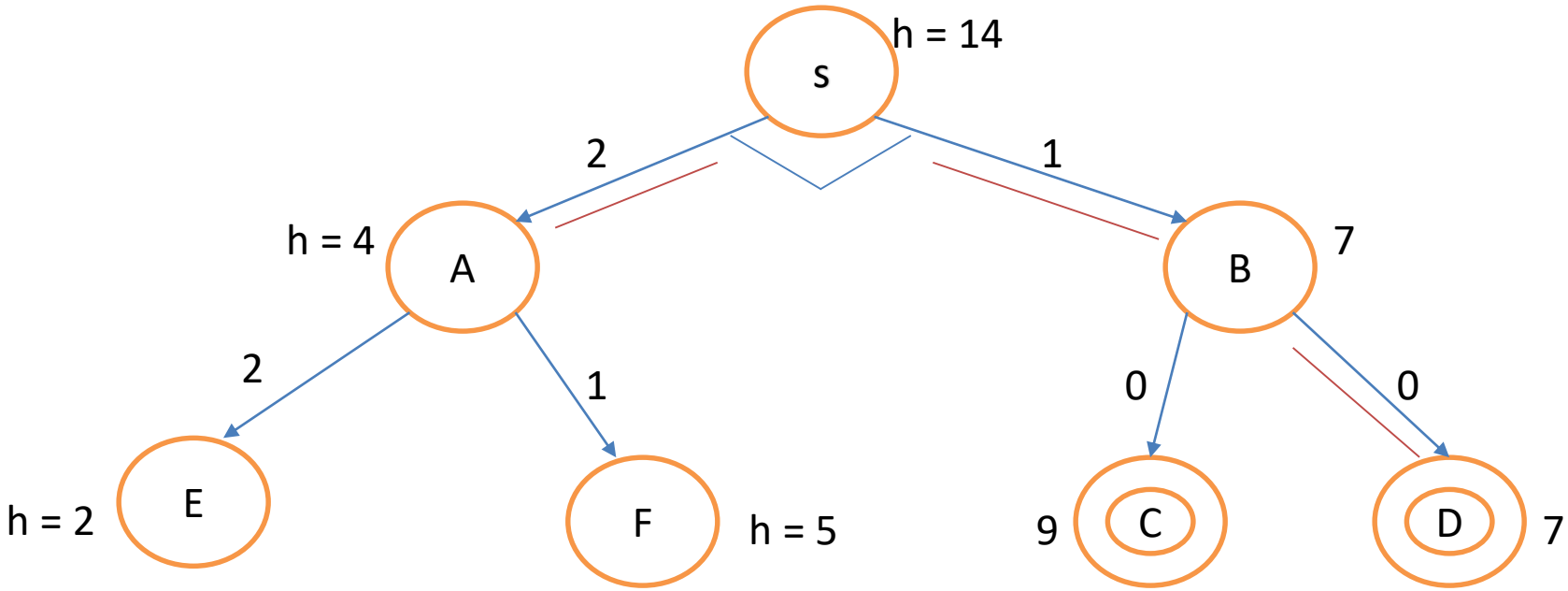
Example



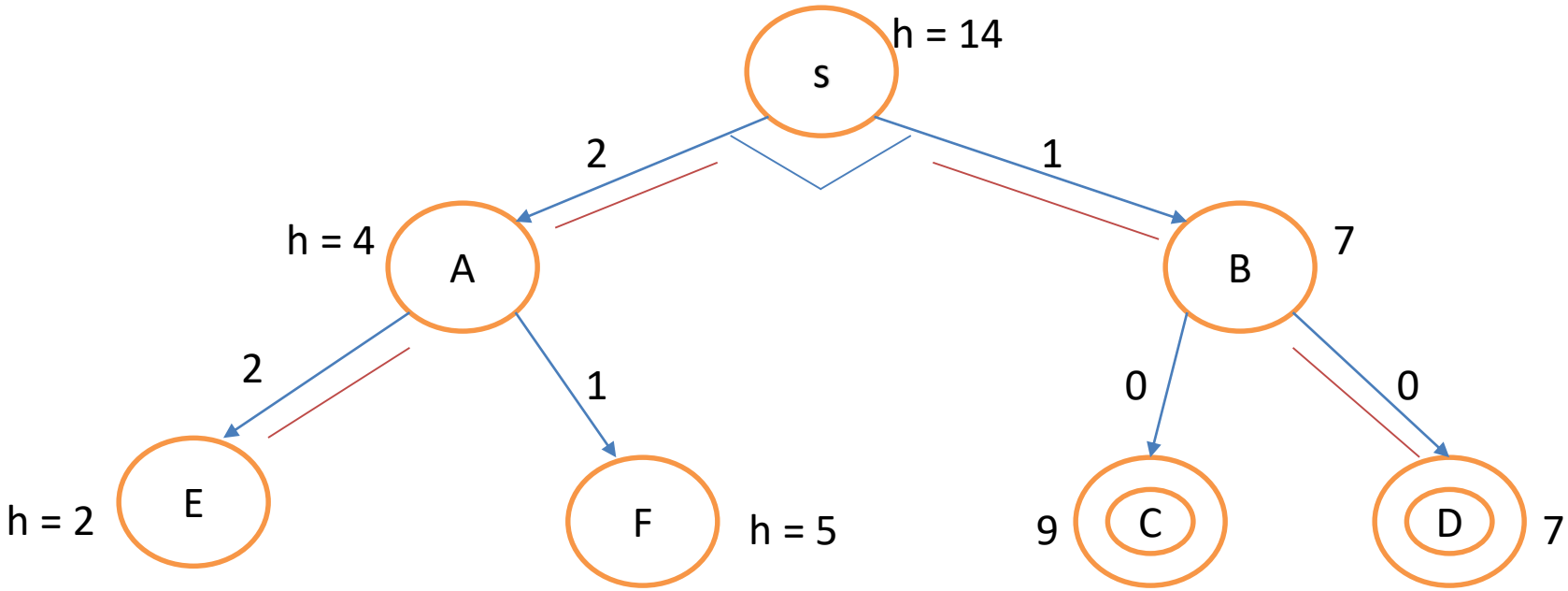
Example



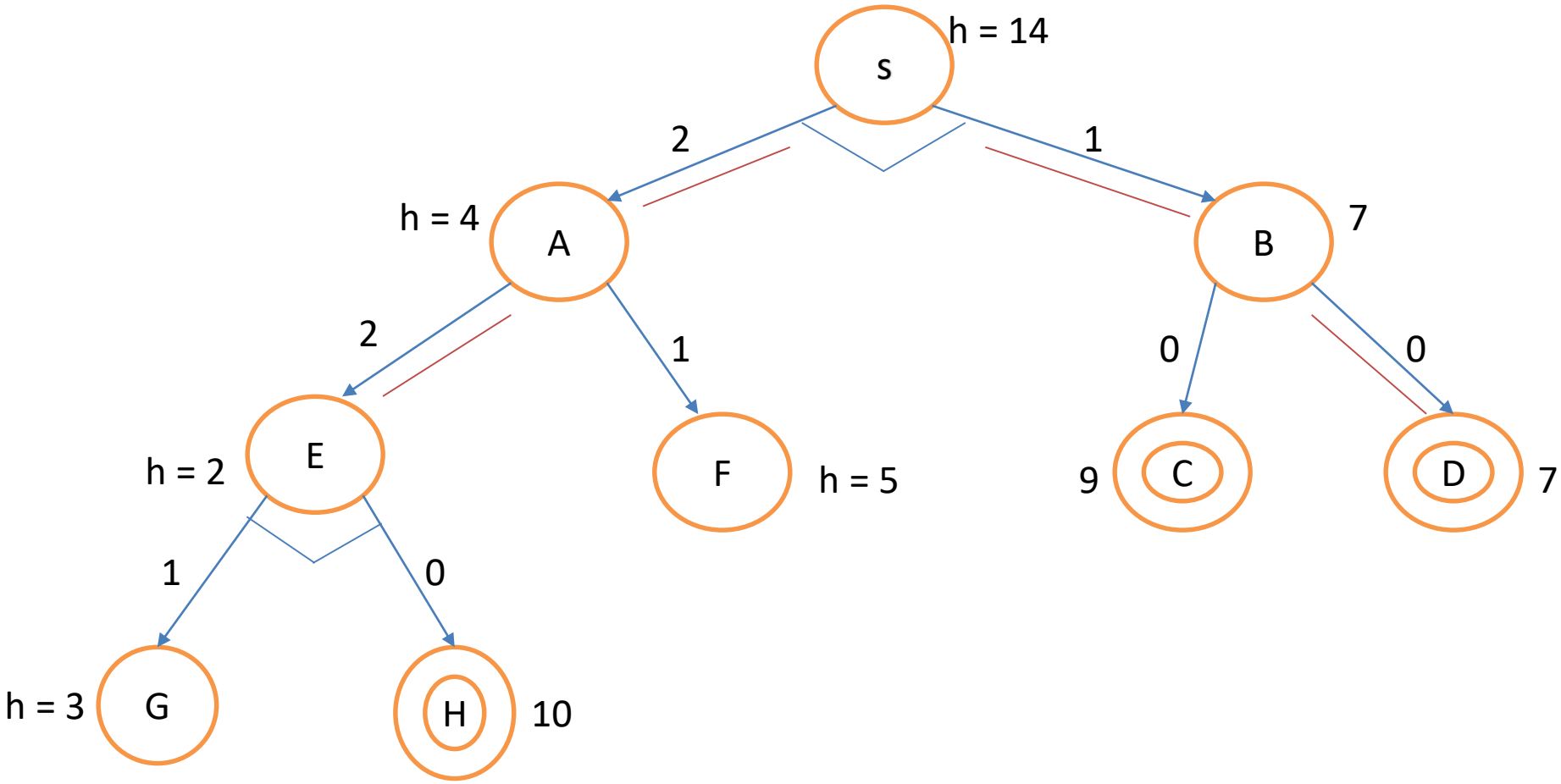
Example



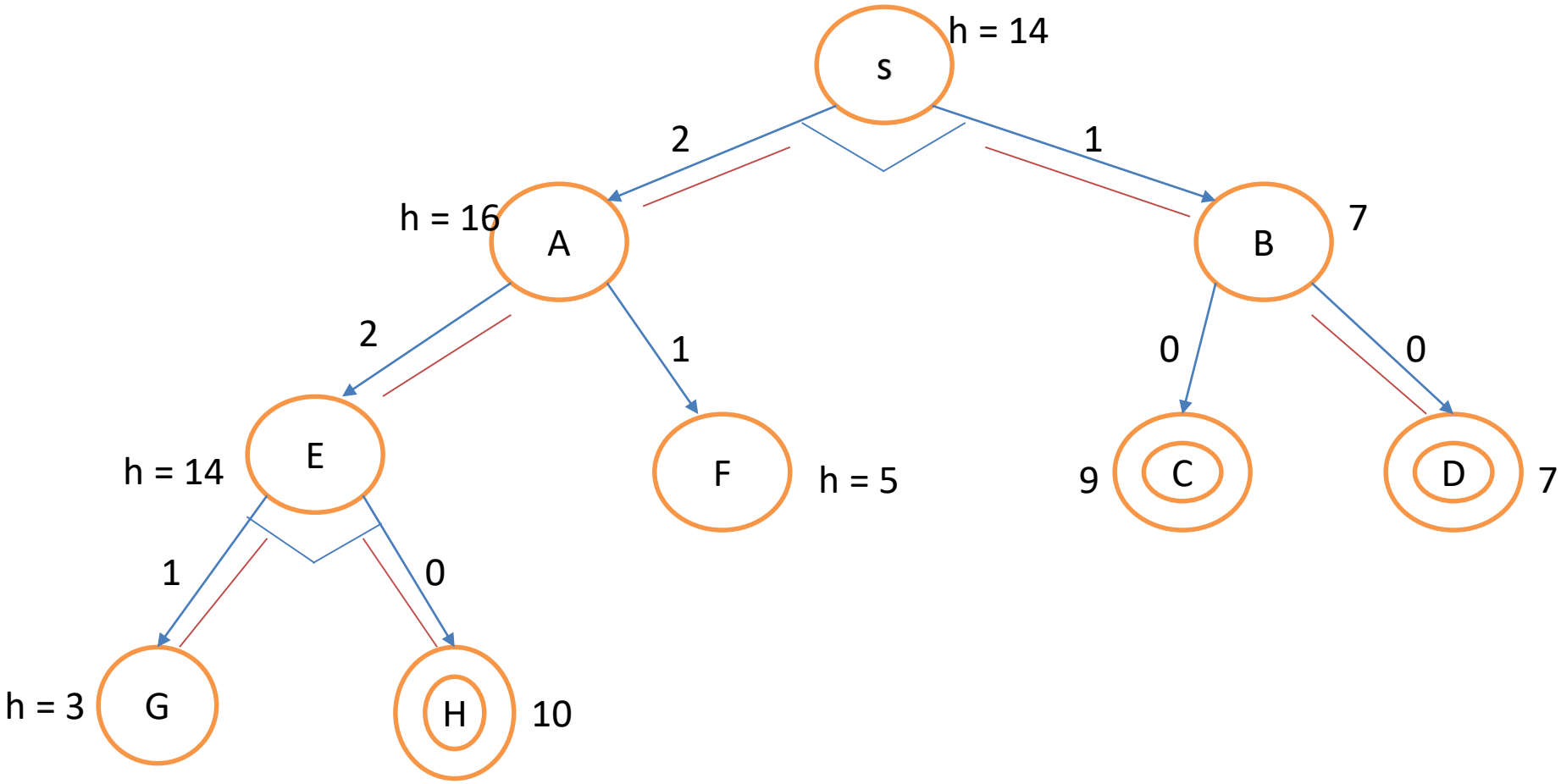
Example



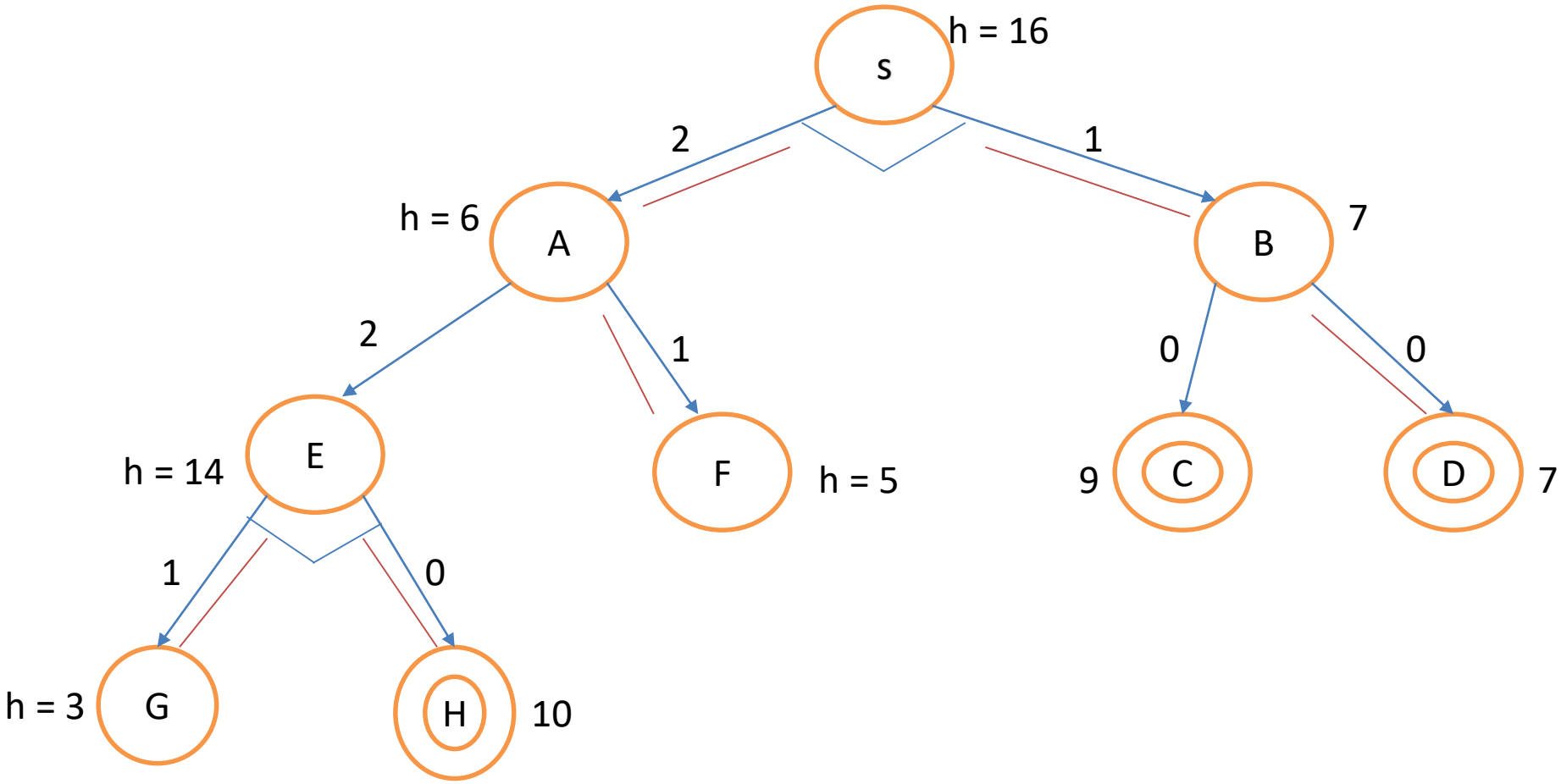
Example



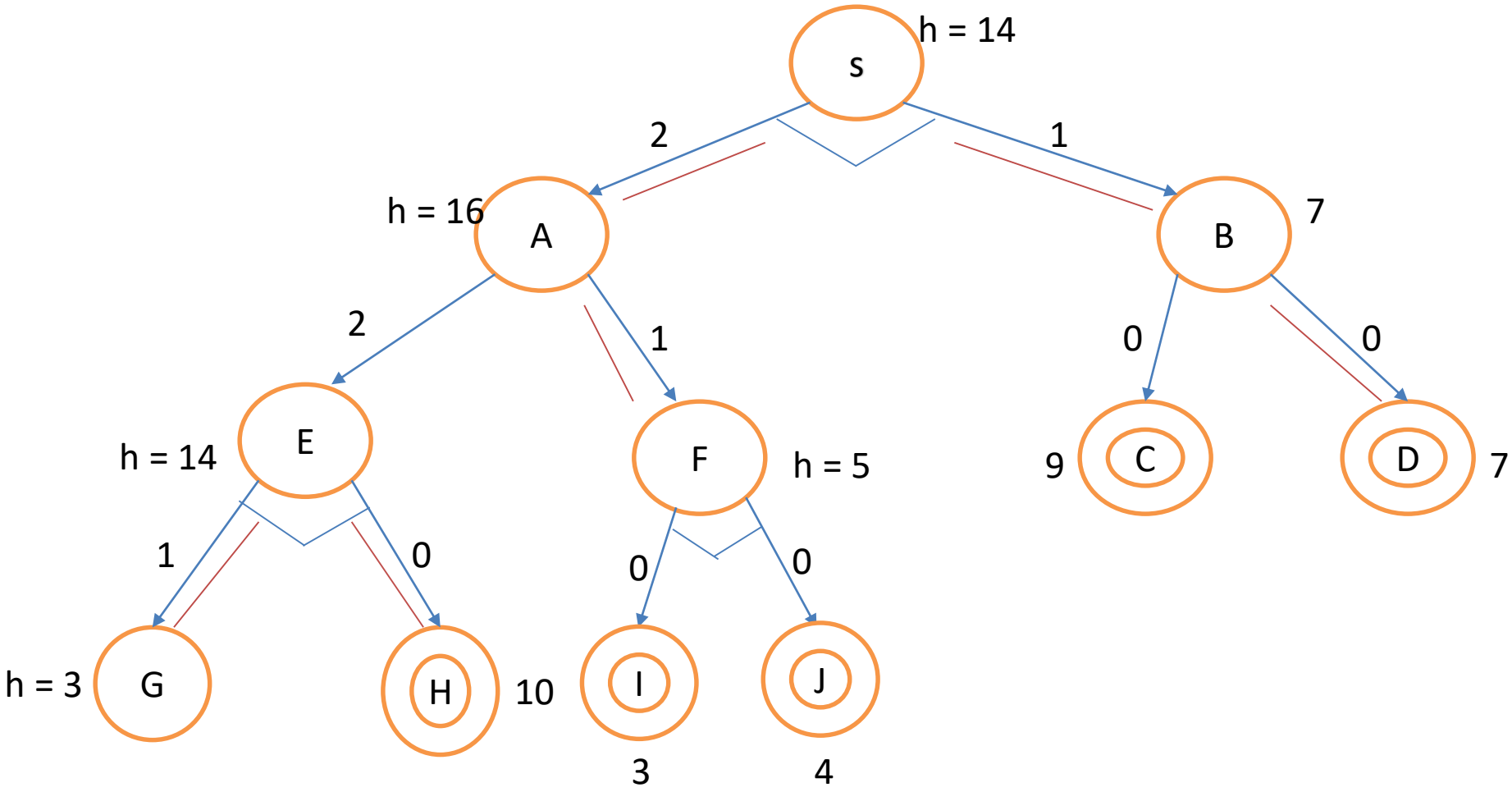
Example



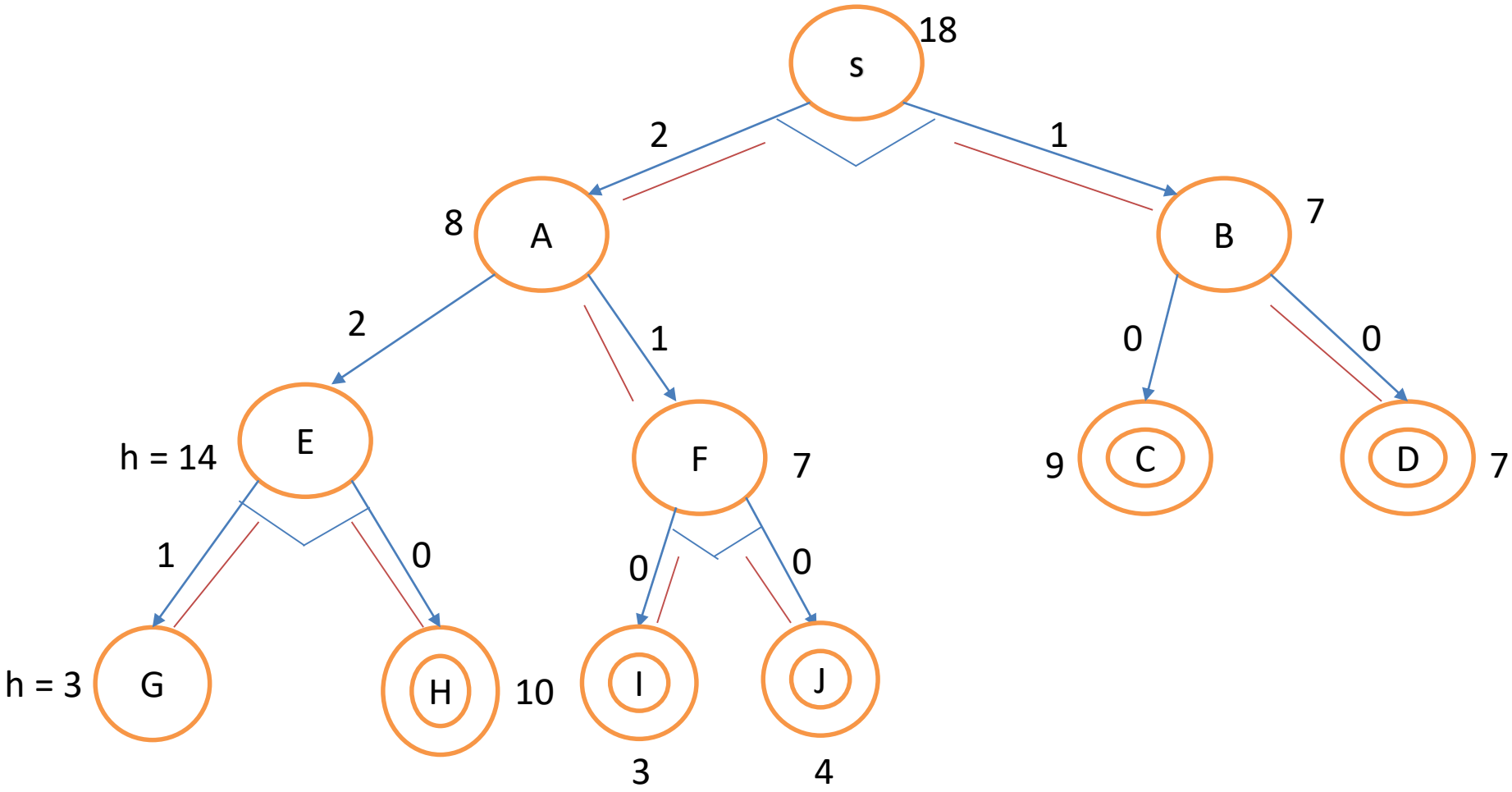
Example



Example



Example



AO* Algorithm

1. Initialize: Set G* = {s}, f(s) = h(s).
If s ϵ T, label s as SOLVED.

2. Terminate: If s is SOLVED, then terminate.

3. Select: Select a non-terminal node n from the marked subtree.

4. Expand: Make explicit the successors of n.
For each new successor m:
Set f(m) = h(m).
If m is terminal, label m as SOLVED.

5. Cost revision: Call cost_revision(n).

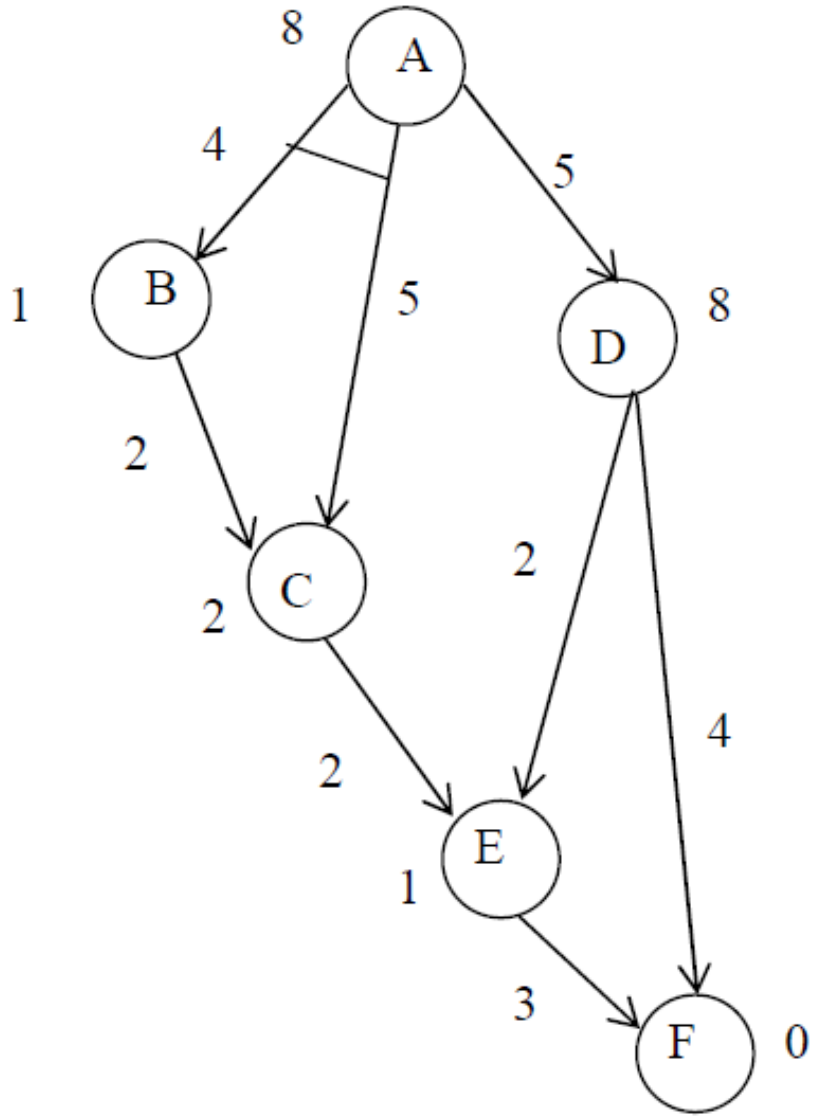
6. Loop: Go to step 2.

AO* Algorithm

cost_revision (n):

1. **Create z = {n}.**
2. **If z = {}, then return.**
3. **Select a node m from z such that m has no descendants in z.**
4. **If m is an AND node with successors r_1, r_2, …, r_k:**
   a) Set f(m) = **Σ** [$f(r_i)$ + $cost(m + r_i)$]
   b) Mark the edge to each successor of m.
   c) If each successor is labelled SOLVED, then label m as SOLVED.
5. **If m is an OR node with successors r_1, r_2, …, r_k:**
   a) Set f(m) = min {$f(r_i)$ + $cost(m + r_i)$}
   b) Mark the edge to the best successor of m.
   c) If the marked successor is labelled SOLVED, then label m as SOLVED.
6. **If the cost/label of m has changed, then insert those parents of m into z for which m is a marked successor.**

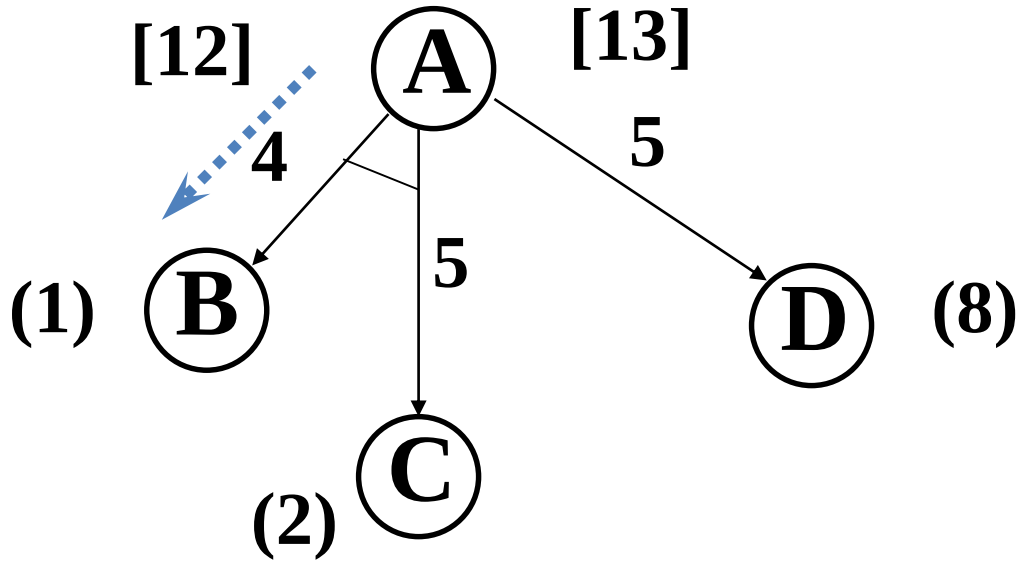
Example



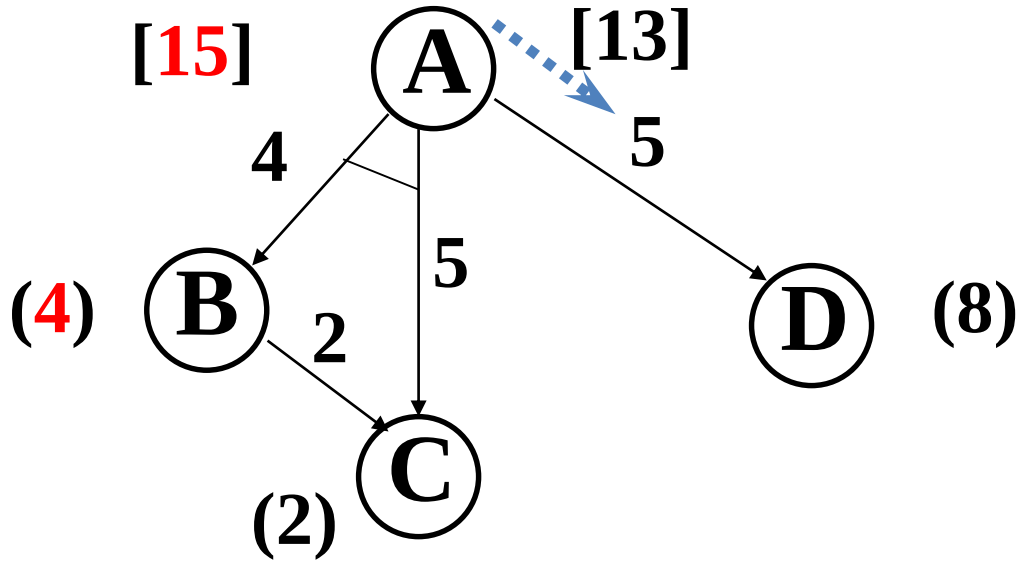
Example

(8) Ⓐ

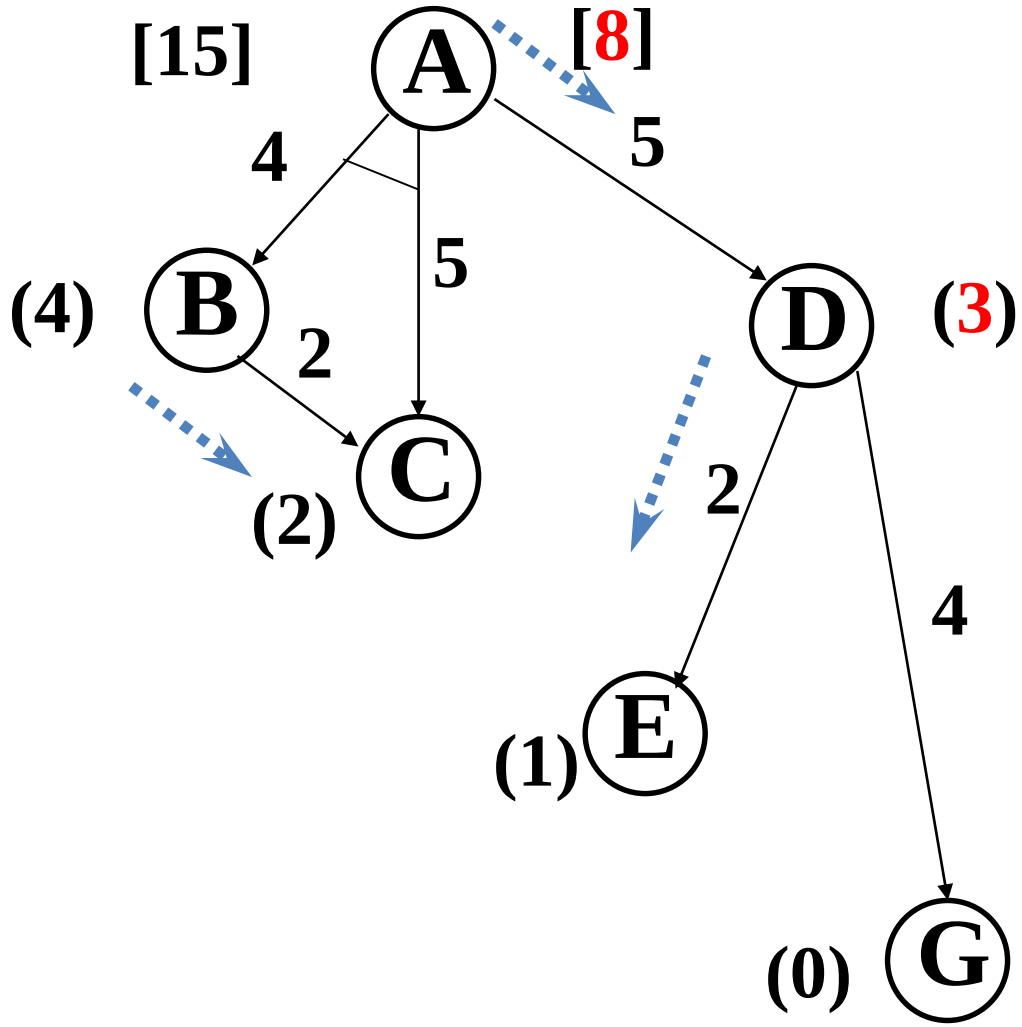
Example



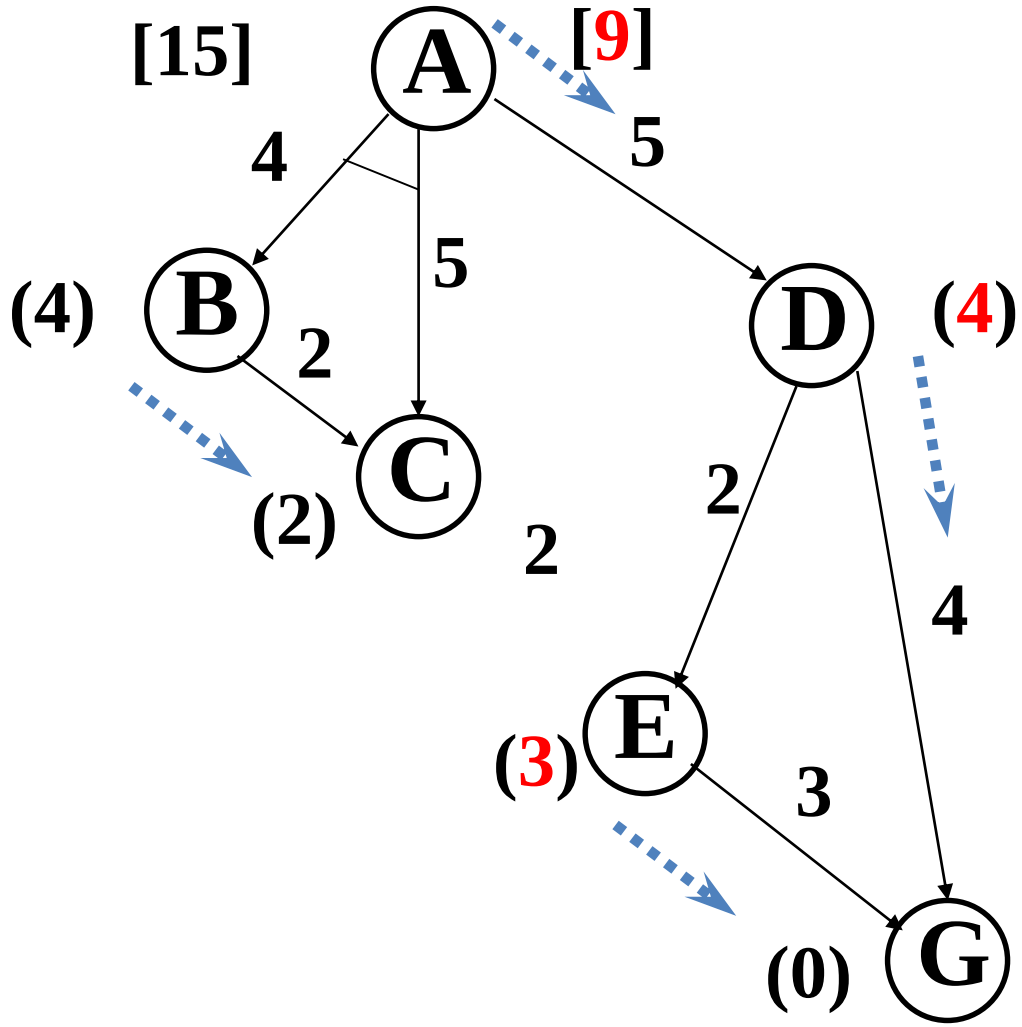
Example



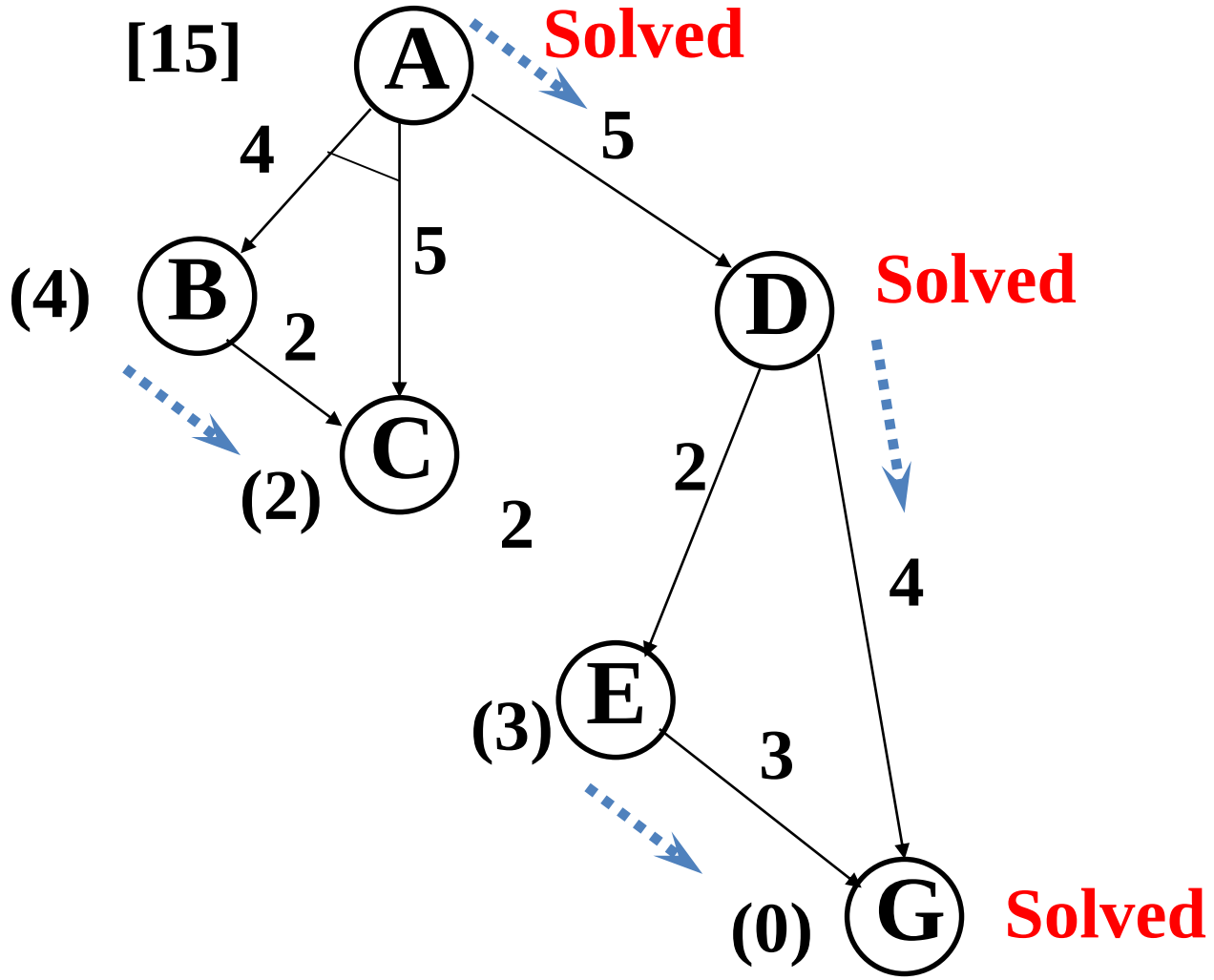
Example



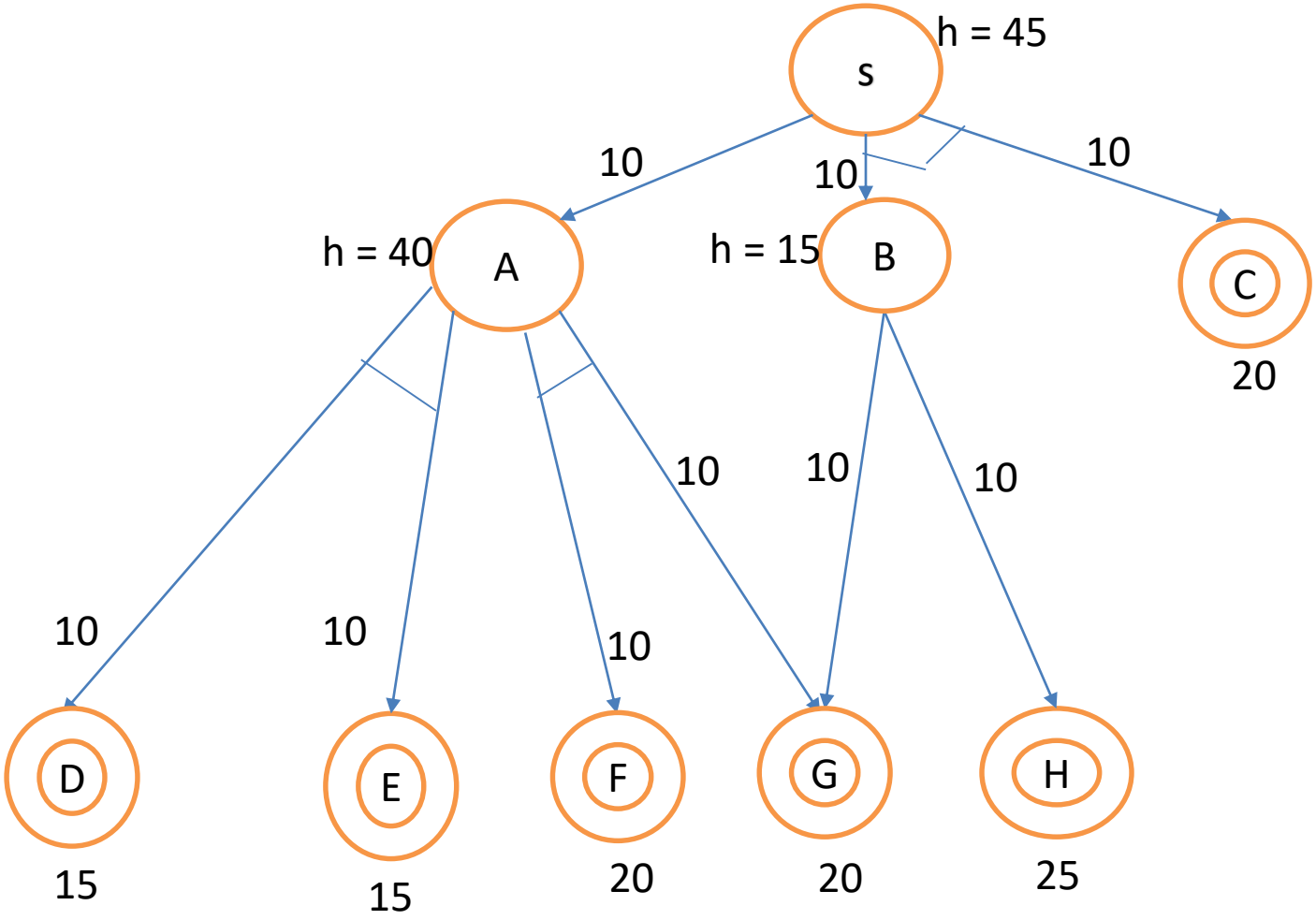
Example



Example



Try it!



Local search

- Search algorithms seen so far are designed to **explore search spaces systematically**.

- Real-World problems are more complex.
- When a goal is found, the path to that goal constitutes a solution to the problem. But, depending on the applications, the path may or may not matter.
- If the path does not matter/systematic search is not possible, then consider another class of algorithms.

Local search

- In such cases, we can use iterative improvement algorithms, **Local search**.
- Also useful in pure **optimization problems** where the goal is to find the best state according to an **optimization function**.
- **Examples:**
  - Integrated circuit design, telecommunications network optimization, etc.
  - N-puzzle or 8-queen: what matters is the final configuration of the puzzle, not the intermediary steps to reach it.

Local search

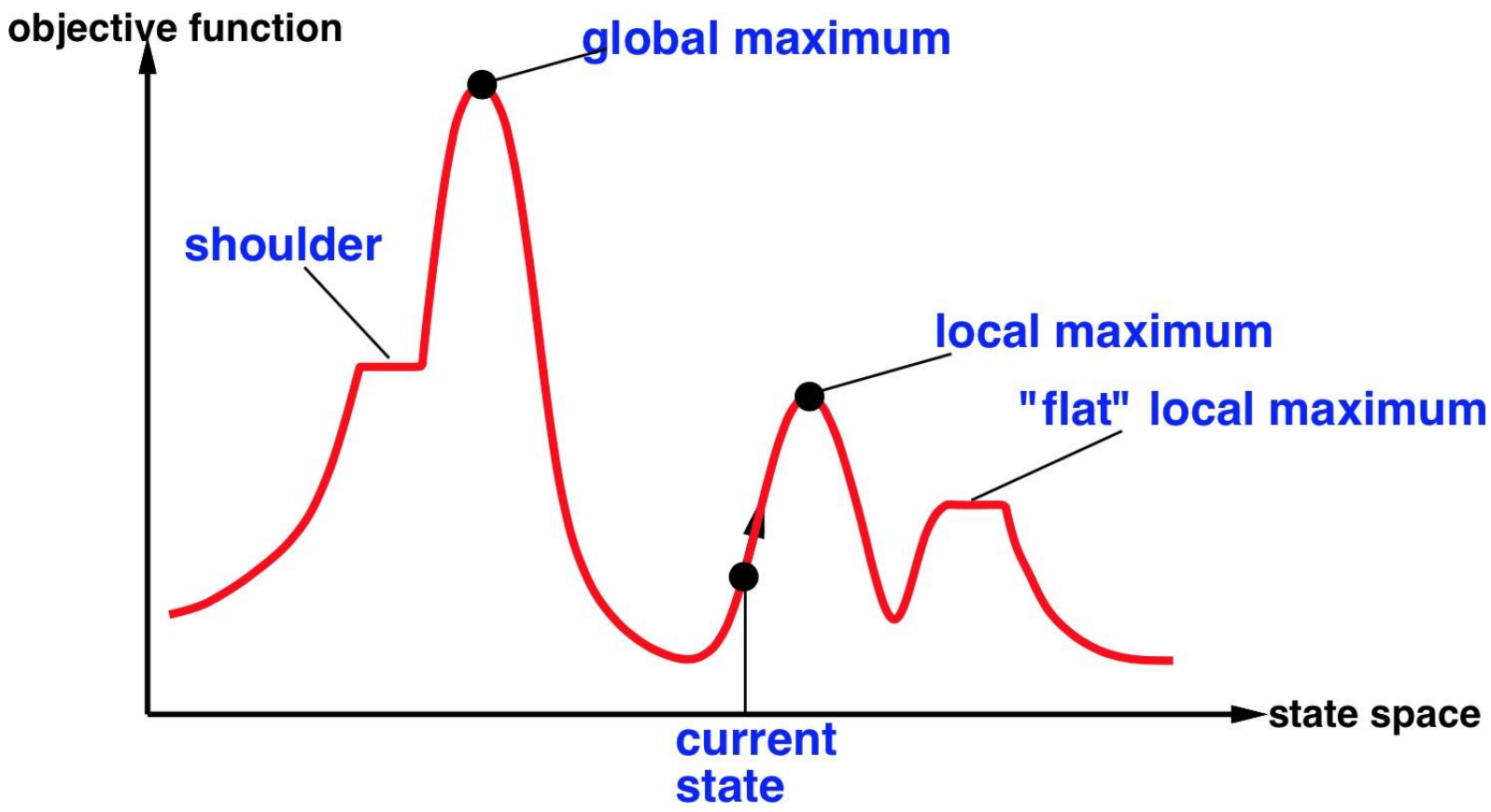
- **Idea**: keep a single "current" state, and try to improve it.
- Move only to neighbors of that node.
- **Advantages**:

  1. No need to maintain a search tree.
  2. Use very little memory.
  3. Can often find good enough solutions in continuous or large state spaces.

Local search

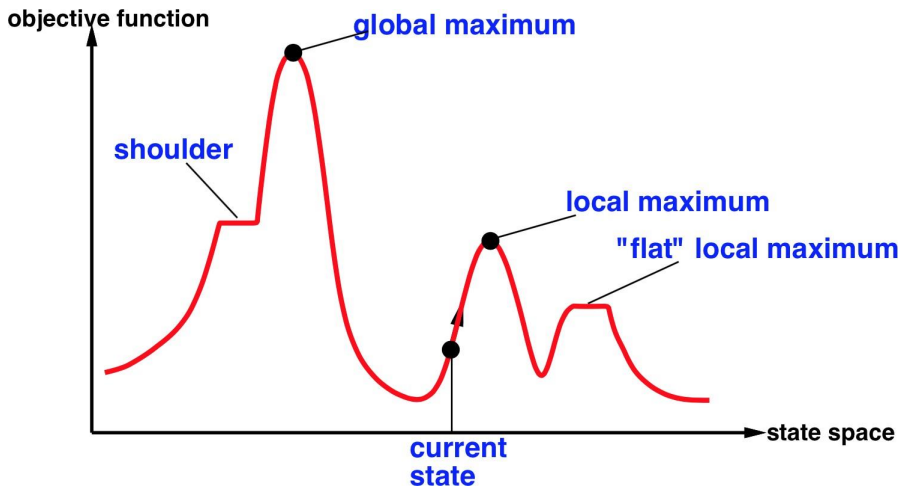
- **Local Search Algorithms:**
  - Hill climbing (simple/steepest ascent).
  - Simulated Annealing: inspired by statistical physics.
  - Local beam search.
  - Genetic algorithms: inspired by evolutionary biology.

Local search



State space landscape

Hill climbing



- Also called **greedy local search**.
- Looks only to immediate good neighbors and not beyond.
- Search moves uphill: moves in the direction of increasing elevation/value to find the top of the mountain.
- Terminates when it reaches a **peak**.
- Can terminate with a local maximum, global maximum or can get stuck and no progress is possible.
- A **node is a state and a value**.

Simple Hill Climbing

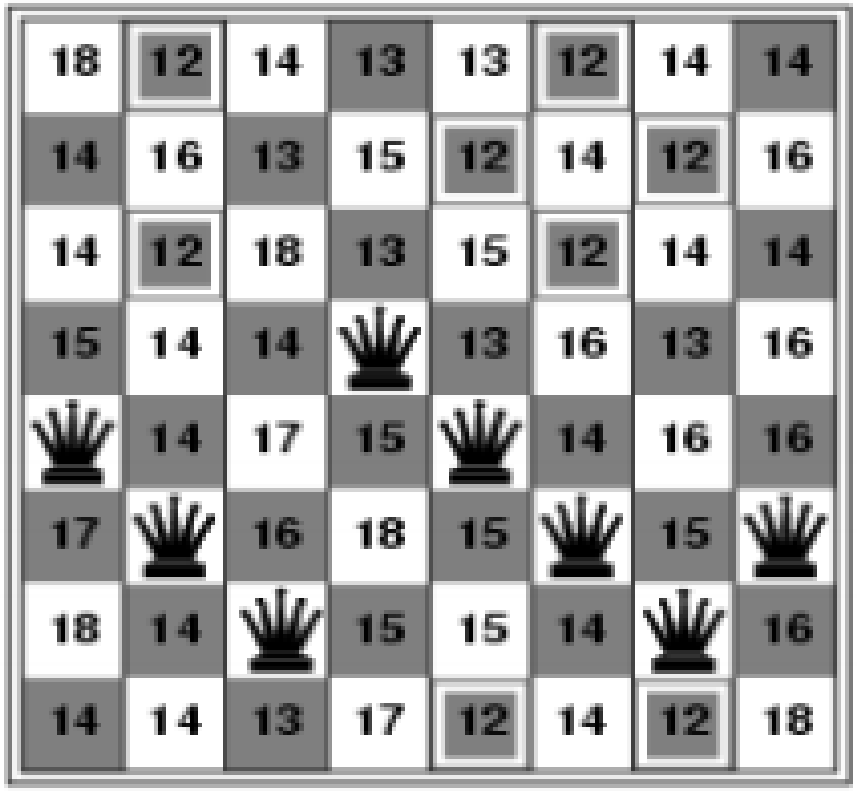
1. Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.
2. Loop until the solution state is found or there are no new operators present which can be applied to current state.
   a) Select a state that has not been yet applied to the current state and apply it to produce a new state.
   b) Perform these to evaluate new state
      i. If the current state is a goal state, then stop and return success.
      ii. If it is better than the current state, then make it current state and proceed further.
      iii. If it is not better than the current state, then continue in the loop until a solution is found.

Steepest Ascent Hill Climbing

1. Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.
2. Loop until the solution state is found or until a complete iteration procedure produces no change to current state.
   a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC.
   b) For each operator that applies to the current state do:
      i. Apply the operator and generate a new state.
      ii. Evaluate the new state.
      iii. If it is a goal state, then stop and return success.
      iv. If not, compare it to SUCC. If it is better than SUCC, then set SUCC to this state. If it is not better, leave SUCC alone.
   c) If SUCC is better than current state, then set current state to SUCC.

8-queens Hill-climbing Search

- **Each state has 8 queens on the board, one per column.**
- **The successors of a state are all possible states generated by moving a single queen to another square in the same column.**
  - So each state has 8 * 7 =56 successors.
- **h = number of pairs of queens that are attacking each other, either directly or indirectly**
- **h = 17 for the below state**



Credit

- Artificial Intelligence, A Modern Approach. Stuart Russell and Peter Norvig. Third Edition. Pearson Education.
- Artificial Intelligence. Elaine Rich, Kevin Knight and Shivashankar B Nair. Third Edition. The McGraw-Hill Companies.