Problem Solving by searching

Problem Solving by Searching in AI is a method used by intelligent agents to find solutions to problems by exploring different possible actions or states. It involves systematically navigating through a "search space" to identify a sequence of actions that leads to a desired goal.

Problem formulation

- **Formulating a Problem** in AI involves defining the problem in a way that an intelligent agent can systematically explore and find a solution.
- This requires specifying key components of the problem, including the initial state, goal state, actions, and constraints.

How a problem is formulated ?

1. Define the Initial State

The starting point of the problem.
It specifies the current state of the environment before any actions are taken.
Example: In a maze, the initial state is the starting position of the agent.

2. Specify the Goal State

The desired outcome or condition the agent needs to achieve.
A well-defined goal state makes it easier to evaluate if a solution has been found.
Example: In a puzzle, the goal state is the arrangement of pieces in the correct order.

3**. Define the State Space**

The set of all possible states that can be reached from the initial state by applying a sequence of actions.
Example: In a game of chess, the state space consists of all possible configurations of the chessboard.

4. Define the Actions (Operators)

The possible moves or transitions the agent can take from one state to another.
Each action is a function that transforms the current state into a new state.
Example: In a maze, actions include moving up, down, left, or right.

How a problem is formulated ? Continued…

5. Specify the Transition Model
A description of how the world changes in response to an action.
It maps a state and an action to a new state.
Example: If the robot moves right from position (2,3), the new state becomes (2,4).

6. Define the Path Cost
A numerical cost associated with a sequence of actions or transitions.
The agent aims to minimize the total path cost (for optimization problems).
Example: In a navigation problem, the path cost could be the total distance traveled.

7**. Set Constraints (if any)**
Any restrictions or rules the agent must follow while solving the problem.
Example: In a warehouse, the robot may need to avoid obstacles while moving.

8**. Define the Solution Criteria**
Specify what constitutes a solution, such as a sequence of actions leading to the goal state.
Example: A solution for a maze is a path from the start position to the exit.

Example of Problem Formulation

Problem: A robot in a grid environment needs to find the shortest path to a target location.

1. **Initial State:** Robot starts at position (0,0).
2. **Goal State:** Robot reaches position (5,5).
3. **State Space:** All valid grid positions (e.g., (x, y) where x, y $\geq$ 0 and $\leq$ grid size).
4. **Actions:** Move up, down, left, or right.
5. **Transition Model:** Moving up from (x, y) changes the state to (x, y+1).
6. **Path Cost:** Each move has a cost of 1.
7. **Constraints:** The robot cannot pass through obstacles in the grid.
8. **Solution:** A sequence of moves that takes the robot from (0,0) to (5,5) while minimizing the total path cost.

Key Concepts:

1. **Search Space:**
    - The set of all possible states or configurations of the problem.
    - Example: In a maze, the search space consists of all the possible positions within the maze.
2. **State:**
    - A representation of a specific situation or configuration in the problem.
    - Example: In a chess game, a state can represent the arrangement of pieces on the board.
3. **Initial State:**
    - The starting point of the problem.
    - Example: The starting position in a maze.
4. **Goal State:**
    - The desired final state that solves the problem.
    - Example: Reaching the exit of the maze.
5. **Actions:**
    - The set of possible moves or transitions between states.
    - Example: In a maze, actions might include moving up, down, left, or right.
6. **Path Cost:**
    - A measure of the cost of a particular sequence of actions (e.g., distance, time, or effort).
    - Example: In a navigation problem, the path cost could be the distance traveled.
7. **Search Strategy:**
    - The approach used to explore the search space and find the solution.

Measuring performance of the problem solving algorithm/agent

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

1. **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
2. **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
3. **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
4. **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of Search Strategies

1. Uninformed (Blind) Search:

- No additional information only information about what is the start state & the goal or search space is available.
- Examples:
  - **Breadth-First Search (BFS):** Explores all nodes at the current depth before moving to the next depth.
  - **Depth-First Search (DFS):** Explores as far as possible along a branch before backtracking.

2. Informed (Heuristic) Search:

- Uses additional knowledge (heuristics) to guide the search more efficiently.
- Examples:
  - **Greedy Best-First Search:** Chooses the path that appears to lead most directly to the goal.
  - *A Search:** Combines path cost and heuristic to find the optimal path.

Uninformed/ Blind

The search techniques

- Breadth first search
- Uniform cost search
- Depth first search
- Depth limited search
- Iterative Deepening
- Bi-directional Search

...are all too slow for most real world problems

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches **breadthwise in a tree or graph**, so it is called **breadth-first search.**
- BFS algorithm **starts searching from the root node** of the tree and expands all successor node at the current level before moving to nodes of next level.
- The **breadth-first search algorithm is an example of a general-graph search algorithm.**
- Breadth-first search implemented u**sing FIFO queue data structure.**

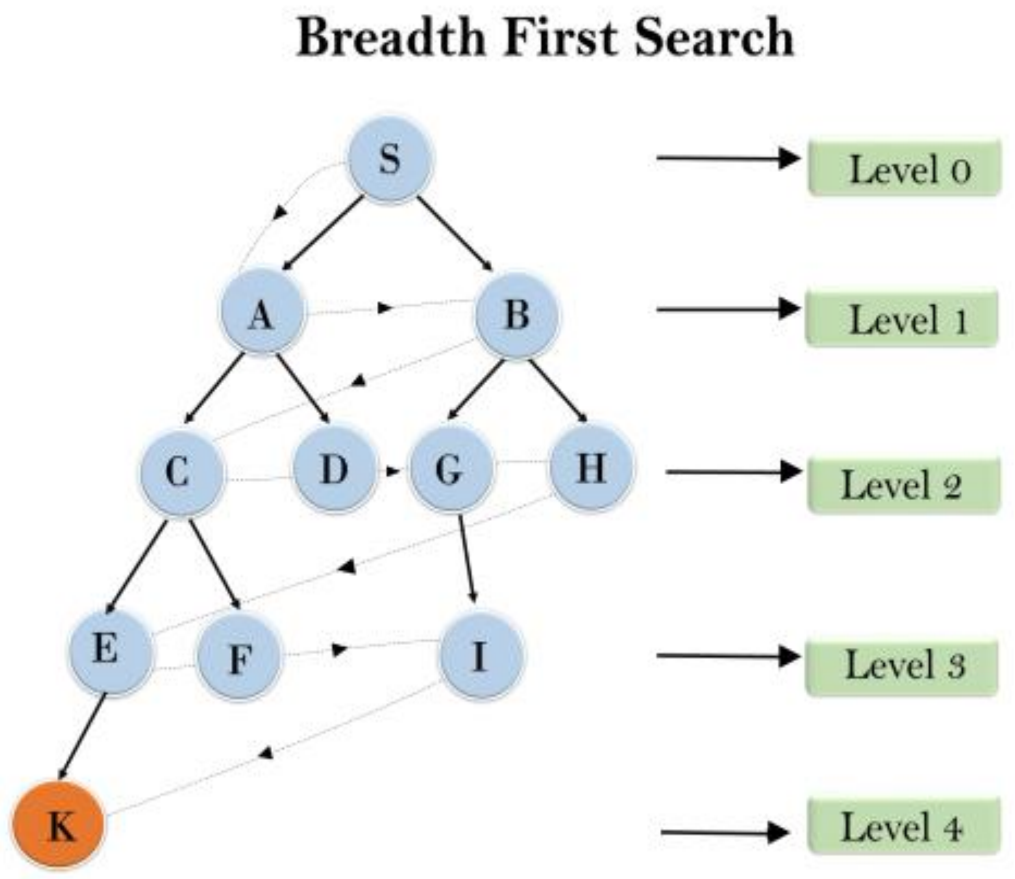
Advantages:

- BFS will provide a solution **if any solution exists.**
- If there **are more than one solutions for a given problem**, then BFS **will provide the minimal solution which requires** the least number of steps.
- It also helps in **finding the shortest path in goal state,** since it needs all nodes at the same hierarchical level before making a move to nodes at lower levels.
- It is also very easy to comprehend with the help of this we can assign the higher rank among path types.

Disadvantages:

- It requires **lots of memory since each level of the tree must be saved** into memory to expand the next level.
- BFS needs l**ots of time if the solution is far away** from the root node.
- It can be very **inefficient approach for searching through deeply layered spaces**, as it needs to thoroughly **explore all nodes at each level before moving on to the next**

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node.

Where the d= depth of shallowest solution and b is a node at every state.

$$T(b) = 1+b^2+b^3+.......+ b^d= O(b^d)$$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Depth-first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- With the help of this we can stores the route which is being tracked in memory to save time as it only needs to keep one at a particular time.

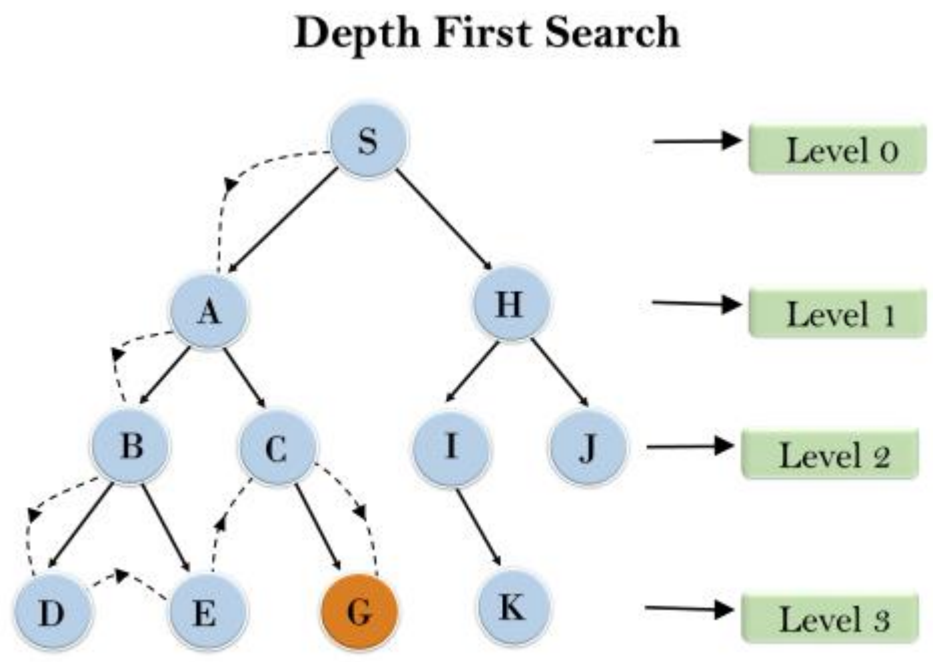
Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.
- The de□pth-first search (DFS) algorithm does not always find the shorte□st path to a solution.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n)= 1+ n^2+ n^3 +.........+ n^m=O(n^m)$$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Difference Between DFS and BFS Algorithm

Feature	DFS (Depth-First Search)	BFS (Breadth-First Search)
Traversal Method	Explores as deep as possible along each branch before backtracking.	Explores all neighbors at the current depth level before moving to the next level.
Data Structure Used	Stack (can be implemented using recursion)	Queue
Order of Exploration	Vertices are explored as far as possible before moving back to explore other branches.	Vertices are explored level by level, starting from the source vertex.
Pathfinding	May not find the shortest path in an unweighted graph.	Always finds the shortest path in an unweighted graph.
Memory Usage	Can be more memory-efficient, especially in graphs with fewer branching factors.	Can require more memory, especially when many vertices are at the same depth level.
Use Cases	Suitable for tasks like topological sorting, detecting cycles, and solving puzzles like mazes.	Ideal for finding the shortest path in unweighted graphs, level-order traversal in trees, and web crawling.
Handling Cycles	Can handle cycles by using a visited array to avoid revisiting nodes.	Also handles cycles with a visited array, preventing infinite loops.
Tree Traversal	Basis for pre-order, in-order, and post-order tree traversals.	Basis for level-order tree traversal.
Complexity (Time)	O(V + E) where V is vertices and E is edges	O(V + E) where V is vertices and E is edges
Complexity (Space)	O(V) due to the recursion stack (in the worst case)	O(V) due to the need to store vertices in the queue

Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. **Depth-limited search can solve the drawback of the infinite path in the Depth-first search**. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

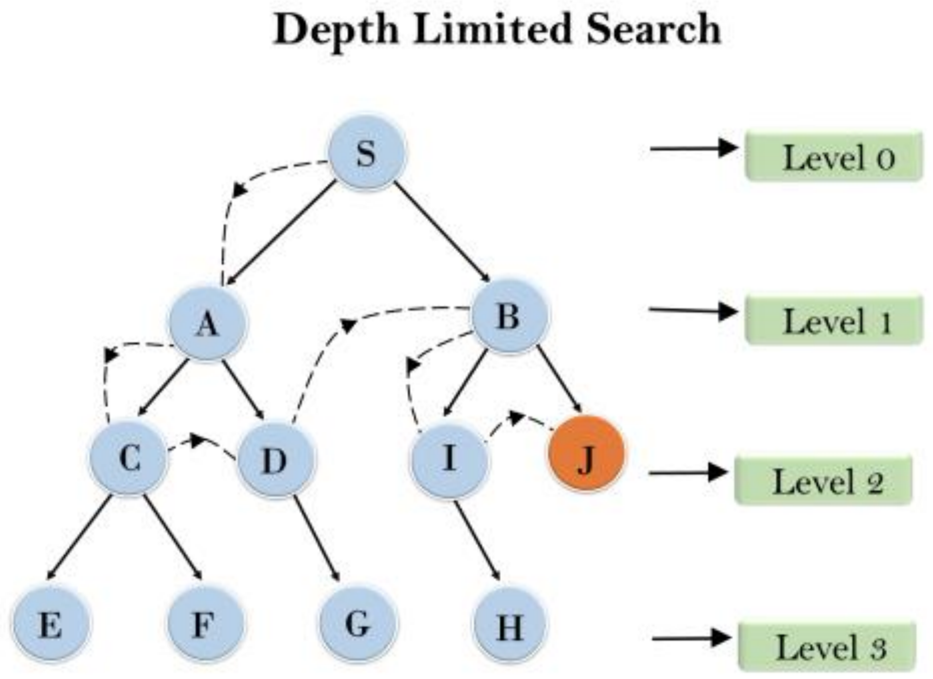
- **Standard failure value**: It indicates that problem does not have any solution.
- **Cutoff failure value**: It defines no solution for the problem within a given depth limit.

Advantages:

- Depth-Limited Search will restrict the **search depth of the tree,** thus, the algorithm **will require fewer memory resource**s than the straight BFS (Breadth-First Search) and IDDFS (Iterative Deepening Depth-First Search).
- When there is a **leaf node depth** which is as large as the highest level allowed, **do not describe its children, and then discard it from the stack.**

Disadvantages:

- Depth-limited search also has a disadvantage of **incompleteness.**
- It **may not be optimal if the problem has more than one solution.**
- The effectiveness of the **Depth-Limited Search (DLS) algorithm is largely dependent on the depth limit specified.** If the depth limit is set too low, the algorithm may fail to find the solution altogether.



Completeness: DLS search algorithm is complete if the **solution is above the depth-limit.**

Time Complexity: Time complexity of DLS algorithm is O(bℓ) where b is the branching factor of the search tree, and l is the depth limit.

Space Complexity: Space complexity of DLS algorithm is O(b×ℓ) where b is the branching factor of the search tree, and l is the depth limit.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

Uniform-cost Search Algorithm:

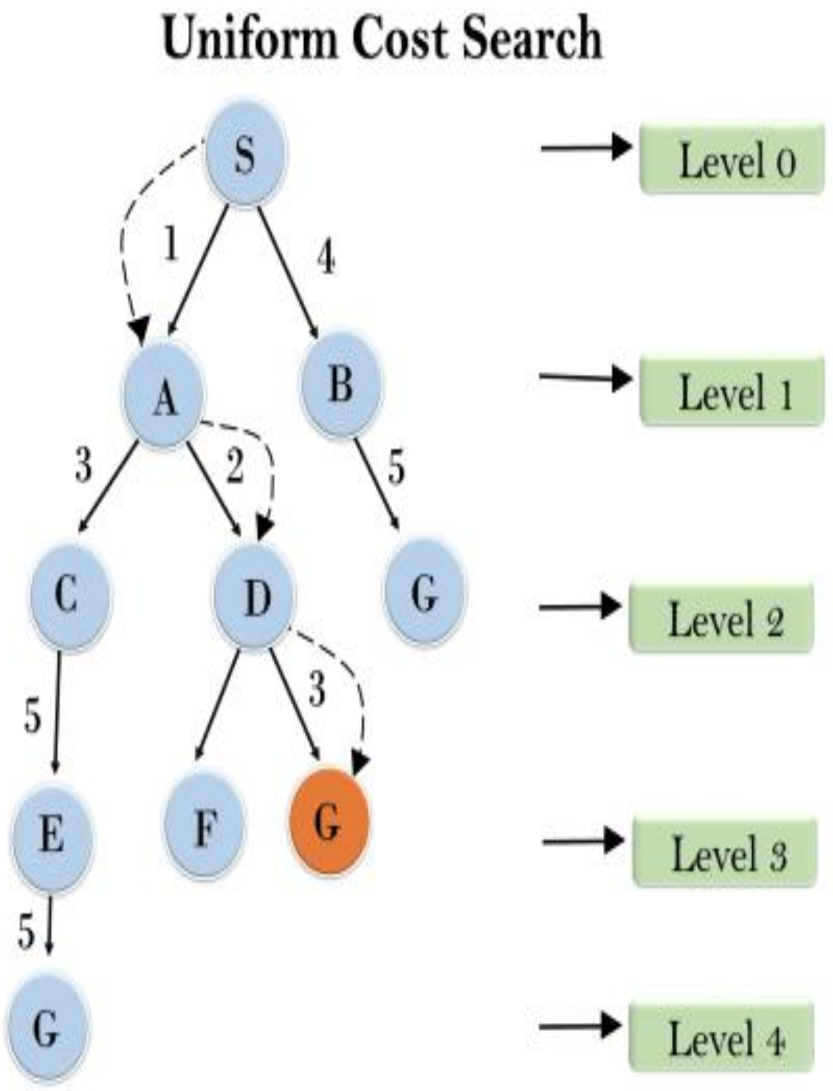
- Uniform-cost search is a searching algorithm used for **traversing a weighted tree or graph**.
- This algorithm comes into play **when a different cost is available for each edge.**
- The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.
- Uniform-cost search expands nodes according to their path costs form the root node.
- It can be used to solve any graph/tree where the optimal cost is in demand.
- A uniform-cost search algorithm is implemented by the **priority queue**. It gives **maximum priority to the lowest cumulative cost.**
- ***Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Advantages:

- Uniform cost search is **optimal because at every state the path with the least cost is chosen.**
- It is an efficient **when the edge weights are small,** as it **explores the paths in an order that ensures that the shortest** path is found early.
- It's a f**undamental search method** that is **not overly complex**, making it accessible for many users.
- It is a **type of comprehensive algorithm that will find a solution if one exists.** This means the algorithm is complete, ensuring it can locate a solution whenever a viable one is available. The **algorithm covers all the necessary steps to arrive at a resolution.**

Disadvantages:

- It does not care about **the number of steps involved in searching and only concerned about path cost.** Due to **which this algorithm may be stuck in an infinite loop.**
- When in operation, **UCS shall know all the edge weights to start off the search.**
- This search holds **constant the list of the nodes that it has already discovered in a priority queue**. Such is a much weightier thing if you have a large graph
- Algorithm **allocates the memory by storing the path sequence of prioritizes**, which can be memory intensive as the graph gets larger.
- With the help **of Uniform cost search we can end up with the problem if the graph has edge's cycles with smaller cost** than that of the shortest path.
- The Uniform cost search will **keep deploying priority queue so that the paths explored can be stored in any case as the graph size** can be even bigger that can eventually result in too much memory being used.



Completeness:
Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:
Let C* is Cost of the optimal solution, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is O(b1 + [C*/ε])/.

Space Complexity:
The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is O(b1 + [C*/ε]).

Optimal:
Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Iterative deepening depth-first Search

- The iterative deepening algorithm is a **combination of DFS and BFS algorithms.** This search algorithm finds out the **best depth limit and does it by gradually increasing the limit until a goal is found.**
- This algorithm performs **depth-first search up to a certain "depth limit"**, and **it keeps increasing the depth limit after each iteration until the goal node is found.**
- This Search algorithm combines the benefits of **Breadth-first search's fast search** and **depth-first search's memory efficiency**.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Here are the steps for Iterative deepening depth first search algorithm:

- Set the depth limit to 0.
- Perform DFS to the depth limit.
- If the goal state is found, return it.
- If the goal state is not found and the maximum depth has not been reached, increment the depth limit and repeat steps 2-4.
- If the goal state is not found and the maximum depth has been reached, terminate the search and return failure.

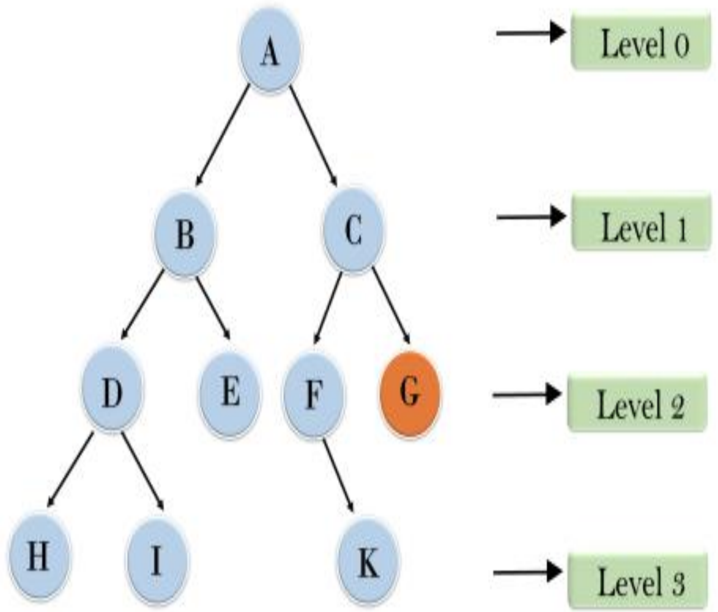
Advantages:

- It combines the **benefits of BFS and DFS search algorithm** in terms of fast search and memory efficiency.
- It is a type of s**traightforward which is used to put into practic**e since it builds upon the **conventional depth-first search algorithm**.
- It is a type of search algorithm which **provides guarantees to find the optimal solution,** as long as the cost of each edge in the search space is the same.
- It is a type of complete algorithm, and the meaning of this is it will always find a solution if one exists.
- The Iterative Deepening Depth-First Search (IDDFS) algorithm **uses less memory compared to Breadth-First Search (BFS)** because it only stores the current path in memory, rather than the entire search tree.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Iterative deepening depth first search



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is if the branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $O(b^d)$.

Space Complexity:

The space complexity of IDDFS will be O(bd).

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

Bidirectional Search Algorithm:

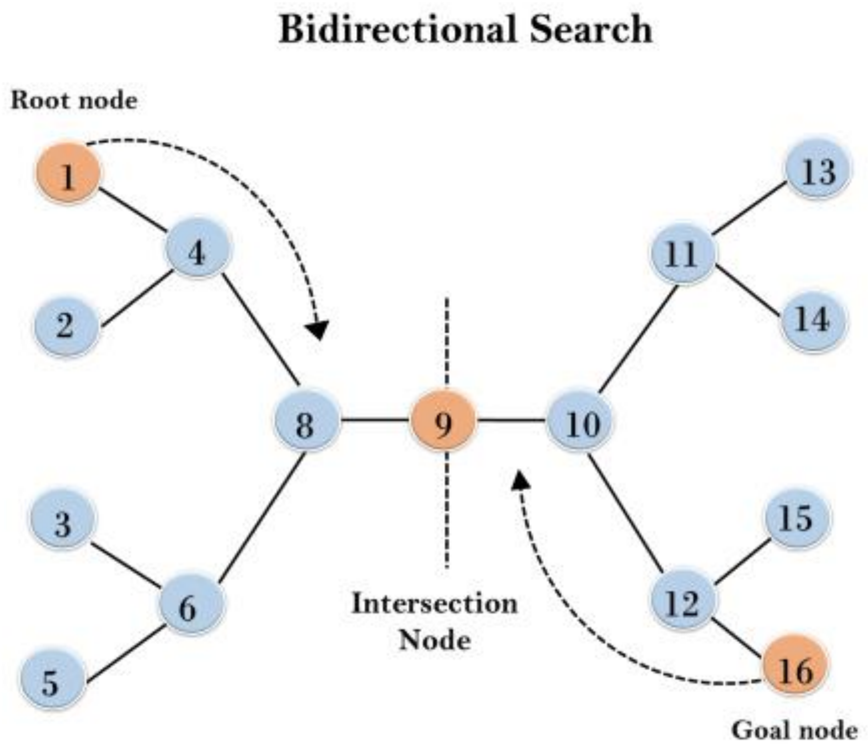
- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory
- The graph can be extremely helpful when it is very large in size and there is no way to make it smaller. In such cases, using this tool becomes particularly useful.
- The cost of expanding nodes can be high in certain cases. In such scenarios, using this approach can help reduce the number of nodes that need to be expanded.

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.
- Finding an efficient way to check if a match exists between search trees can be tricky, which can increase the time it takes to complete the task.



In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.

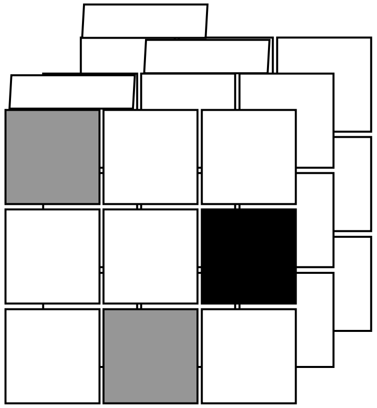
Sometimes we can tell that some states appear better that others...

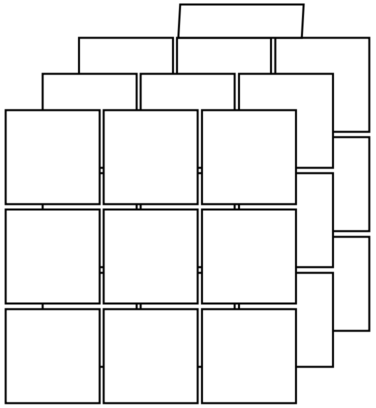
7	**8**	**4**
3	**5**	**1**
6	**2**	

1	**2**	**3**
4	**5**	**6**
7		**8**

$\frac{\mathbf{FWD}}{\mathbf{C}}$

$\frac{\mathbf{D}}{\mathbf{FW\ \ C}}$





...we can use this knowledge of the relative merit of states to guide search

Heuristic Search (informed search)

A **Heuristic** is a function that, when applied to a state, returns a number that is an estimate of the merit of the state, with respect to the goal.

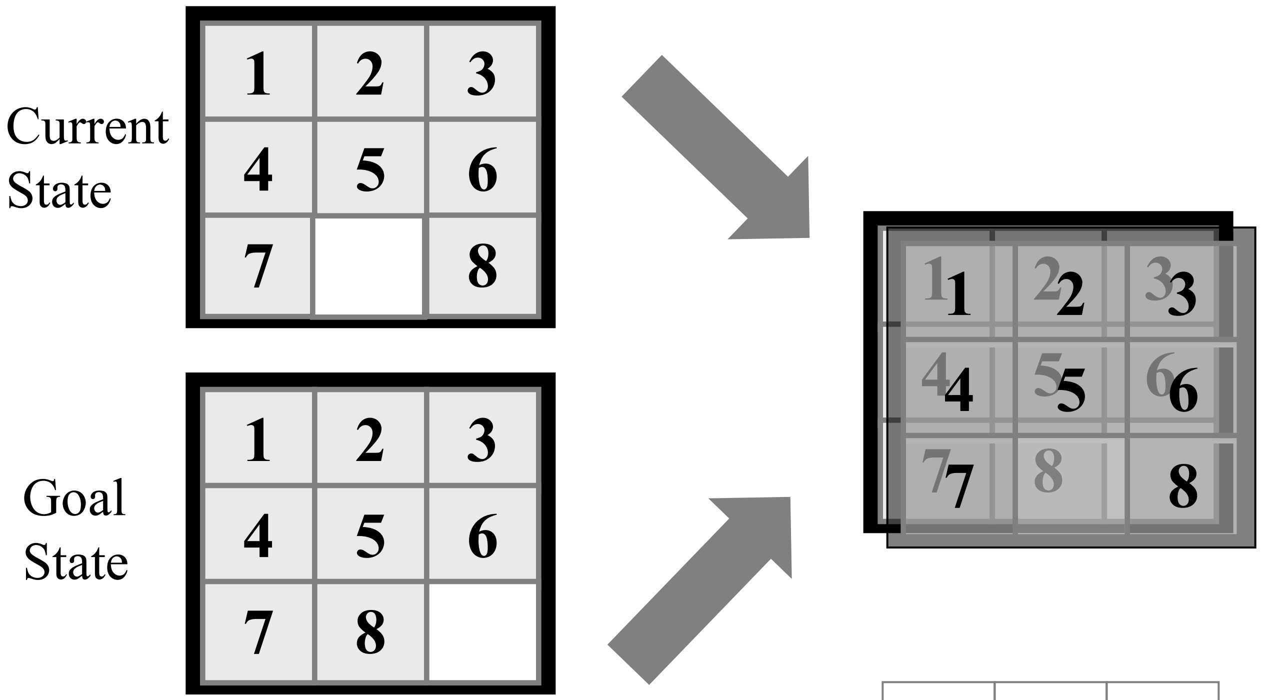
In other words, the heuristic tells us approximately how far the state is from the goal state*.

Note we said "approximately". Heuristics might underestimate or overestimate the merit of a state. But for reasons which we will see, heuristics that *only* underestimate are very desirable, and are called admissible.

*I.e Smaller numbers are better

Heuristics for 8-puzzle I

- The number of **misplaced tiles** (not including the blank)



In this case, only "**8**" is misplaced, so the heuristic function evaluates to 1.

In other words, the heuristic is *telling* us, that it *thinks* a solution might be available in just 1 more move.

N	**N**	**N**
N	**N**	**N**
N	**Y**	

Notation: *h*(n) *h*(current state) = 1

Heuristics for 8-puzzle II

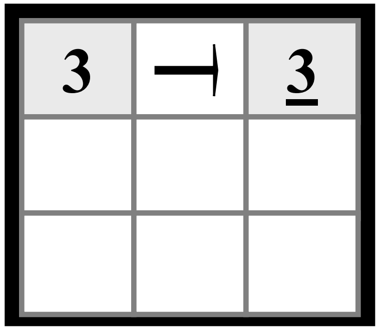
- The **Manhattan Distance** (not including the blank)

Current State

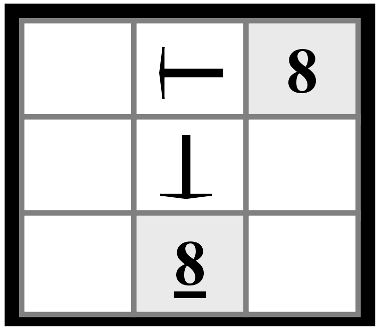
3	**2**	**8**
4	**5**	**6**
7	**1**	

Goal State

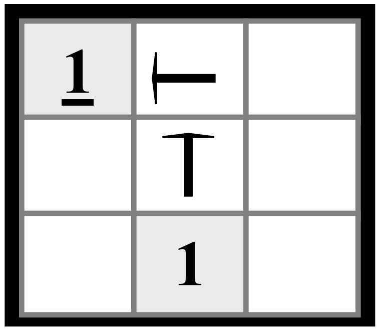
1	**2**	**3**
4	**5**	**6**
7	**8**	



2 spaces



3 spaces



3 spaces

Total 8

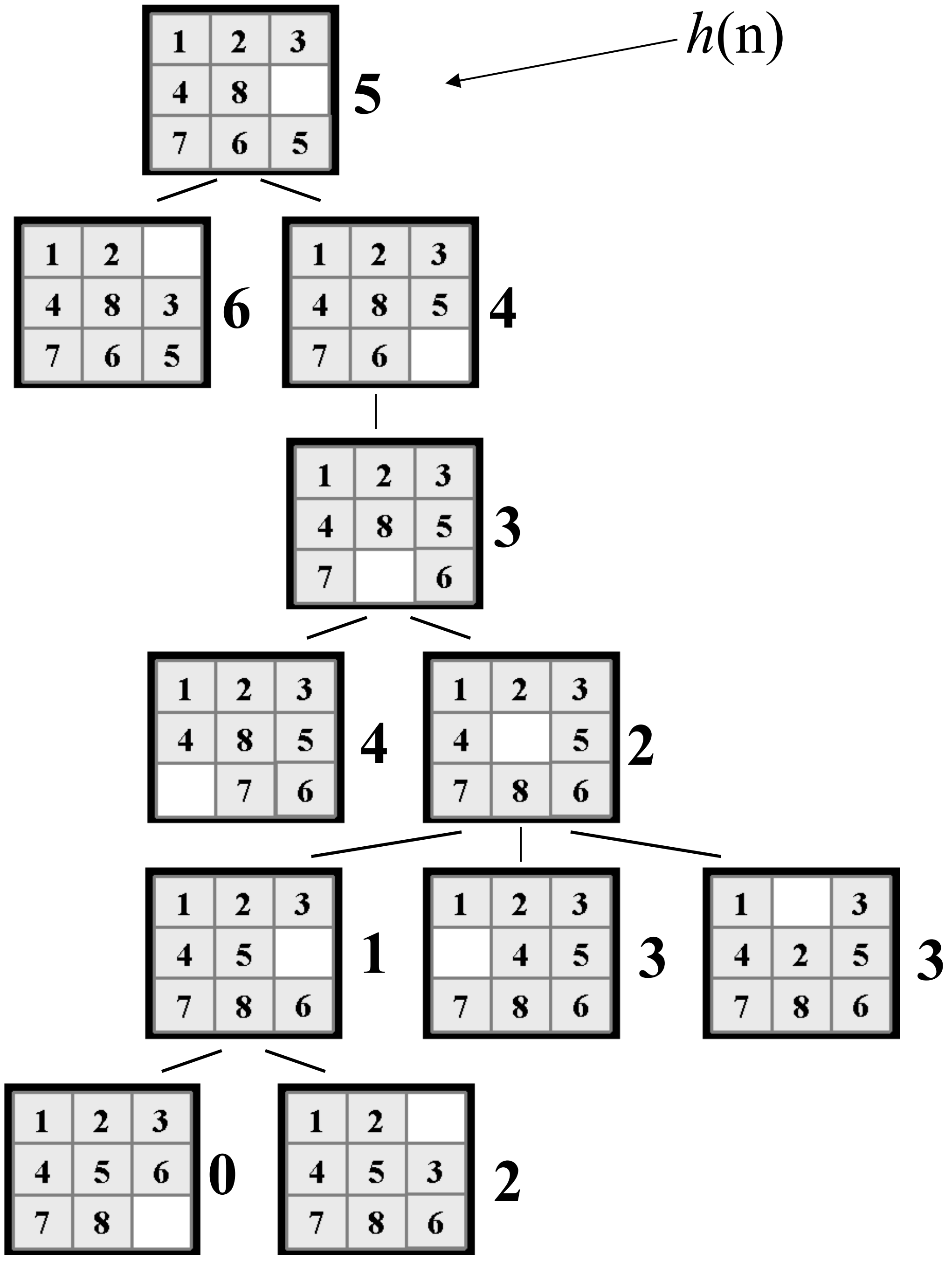
In this case, only the "**3**", "**8**" and "**1**" tiles are misplaced, by 2, 3, and 3 squares respectively, so the heuristic function evaluates to 8.

In other words, the heuristic is *telling* us, that it *thinks* a solution is available in just 8 more moves.

Notation: *h*(n) *h*(current state) = 8

We can use heuristics to guide "hill climbing" search.

In this example, the Manhattan Distance heuristic helps us quickly find a solution to the 8-puzzle.

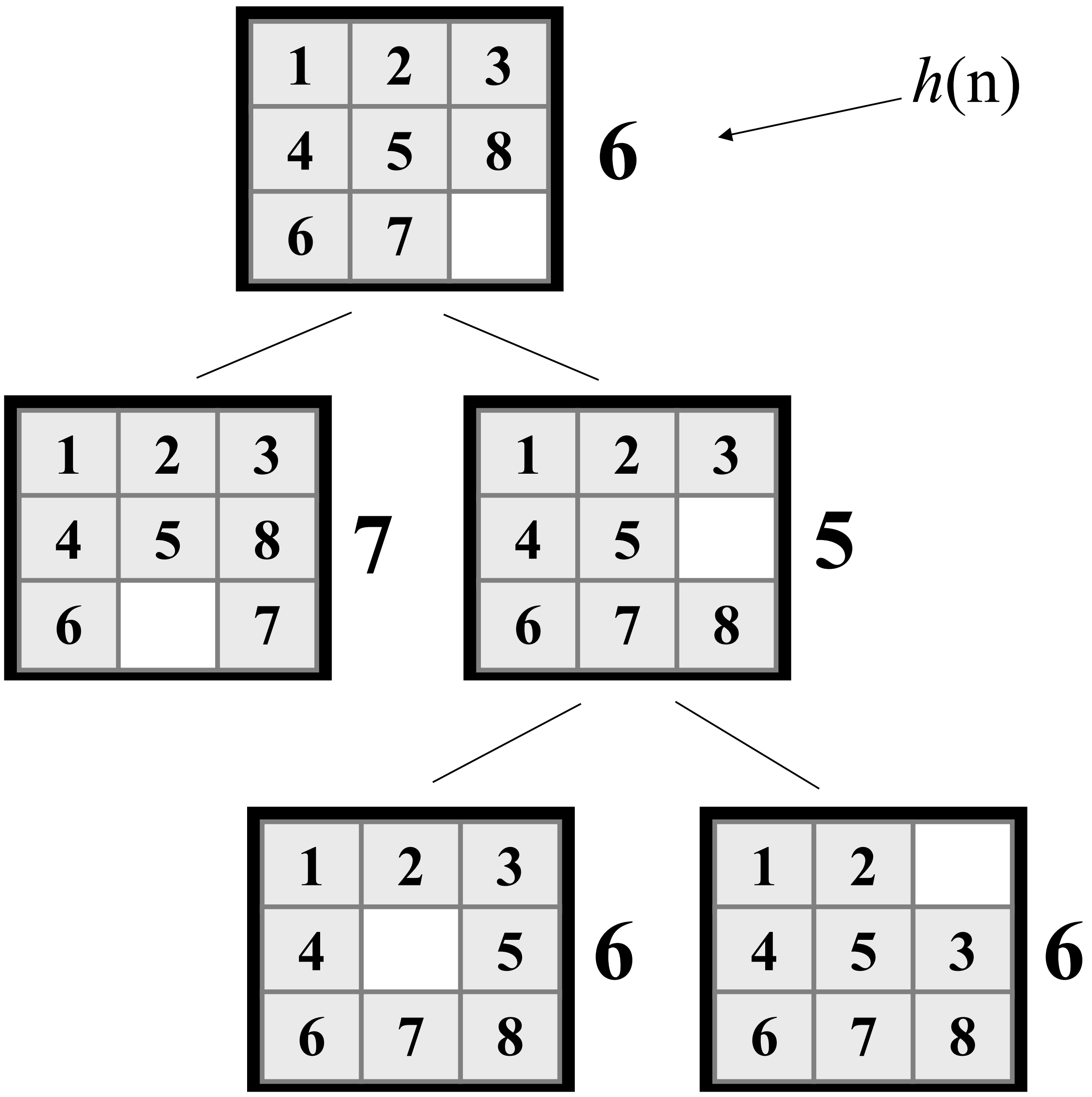


But "hill climbing has a problem..."

In this example, hill climbing does not work!

All the nodes on the fringe are taking a step "backwards" (local minima)

Note that this puzzle *is* solvable in just 12 more steps.



We have seen two interesting algorithms.

Uniform Cost

- Measures the cost to each node.
- Is optimal and complete!
- Can be very slow.

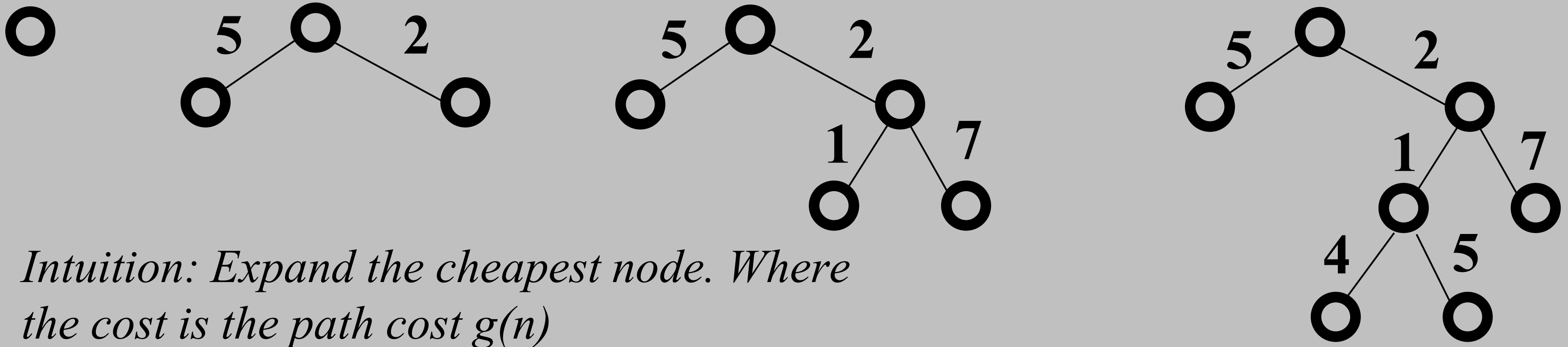
Hill Climbing

- Estimates how far away the goal is.
- Is neither optimal nor complete.
- Can be very fast.

Can we combine them to create an optimal and complete algorithm that is also very fast?

Uniform Cost Search

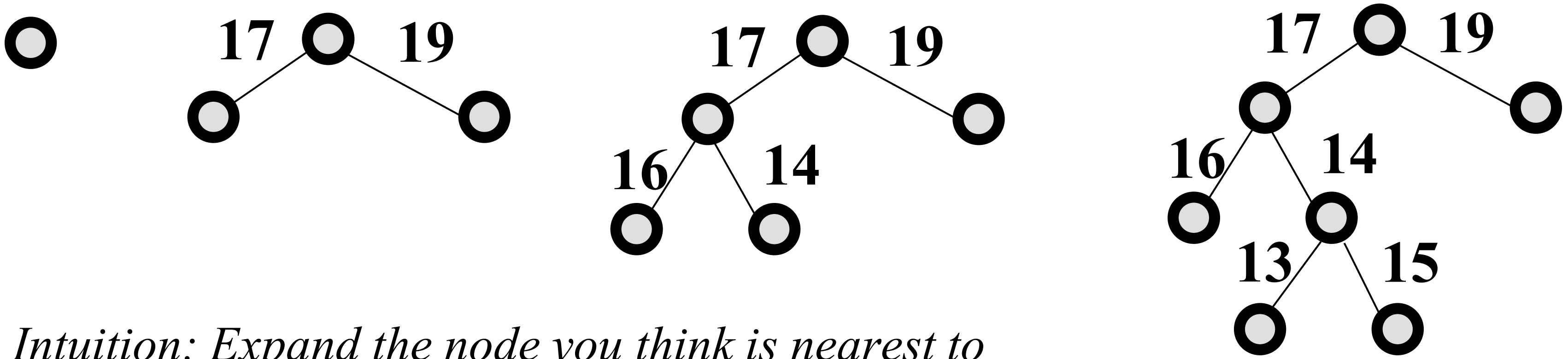
Enqueue nodes in order of cost



Intuition: Expand the cheapest node. Where the cost is the path cost g(n)

Hill Climbing Search

Enqueue nodes in order of estimated distance to goal



Intuition: Expand the node you think is nearest to goal. Where the estimate of distance to goal is h(n)

The A* Algorithm ("A-Star")

Enqueue nodes in order of estimate cost to goal, f(n)

g(n) is the cost to get to a node.
h(n) is the estimated distance to the goal.

$f(n) = g(n) + h(n)$

We can think of f(n) as the estimated cost of the cheapest solution that goes through node n

Note that we can use the general search algorithm we used before. All that we have changed is the queuing strategy.

If the heuristic is optimistic, that is to say, it never overestimates the distance to the goal, then…

A* is optimal and complete!

Informal *proof* outline of A* completeness

- Assume that every operator has some minimum positive cost, *epsilon* .
- Assume that a goal state exists, therefore some finite set of operators lead to it.
- Expanding nodes produces paths whose actual costs increase by at least epsilon each time. Since the algorithm will not terminate until it finds a goal state, it must expand a goal state in finite time.

Informal *proof* outline of A* optimality

- When A* terminates, it has found a goal state
- All remaining nodes have an estimate cost to goal (*f*(n)) greater than or equal to that of goal we have found.
- Since the heuristic function was optimistic, the actual cost to goal for these other paths can be no better than the cost of the one we have already found.

How fast is A*?

A* is the fastest search algorithm. That is, for any given heuristic, no algorithm can expand fewer nodes than A*.

How fast is it? Depends of the quality of the heuristic.

- If the heuristic is useless (ie *h*(n) is hardcoded to equal 0), the algorithm degenerates to uniform cost.

- If the heuristic is perfect, there is no real search, we just march down the tree to the goal.

Generally we are somewhere in between the two situations above. The time taken depends on the quality of the heuristic.

What is A*'s space complexity?

A* has worst case O(b^d) space complexity, but an iterative deepening version is possible (IDA*)

A Worked Example: Maze Traversal

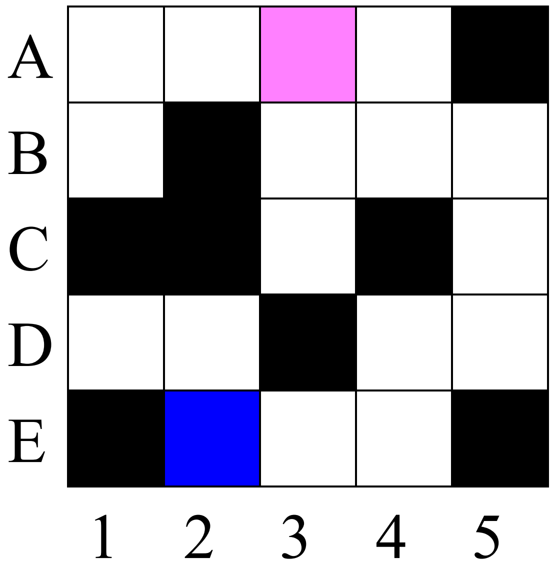
Problem: To get from square **A3** to square **E2**, one step at a time, avoiding obstacles (black squares).

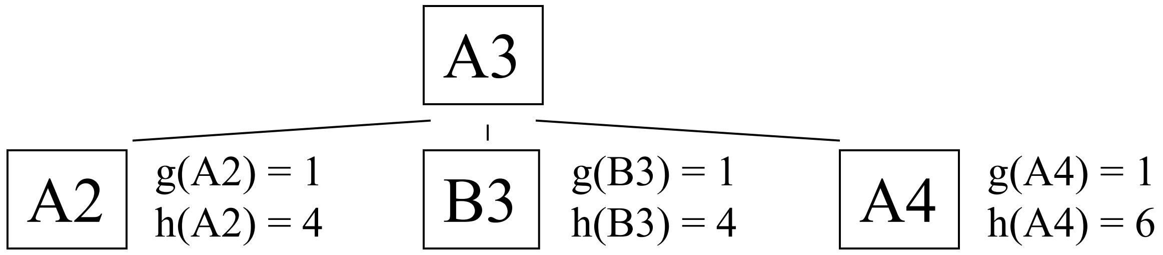
Operators: (in order)

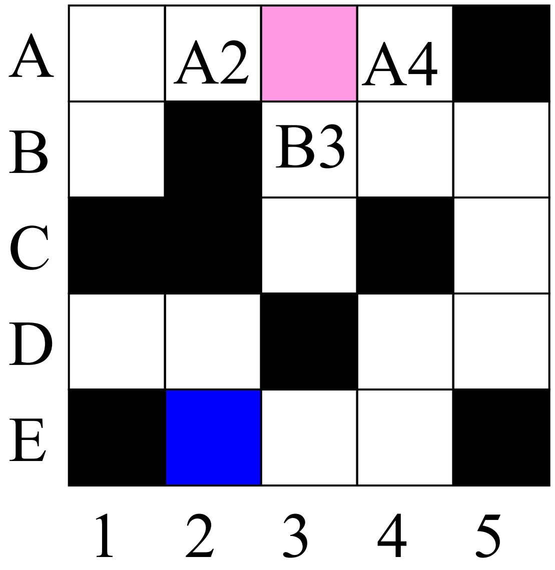
- **`go_left`**(n)
- **`go_down`**(n)
- **`go_right`**(n)

each operator costs 1.

Heuristic: Manhattan distance



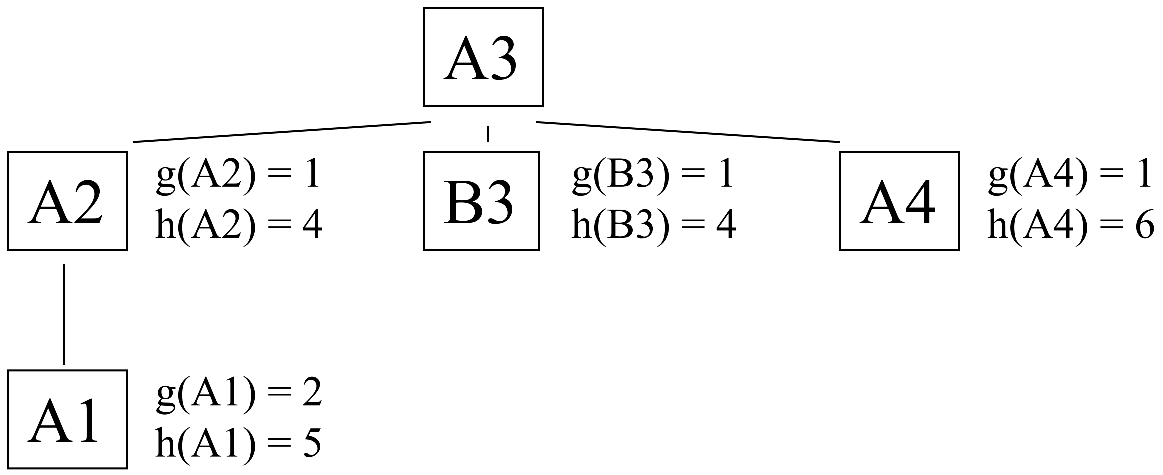


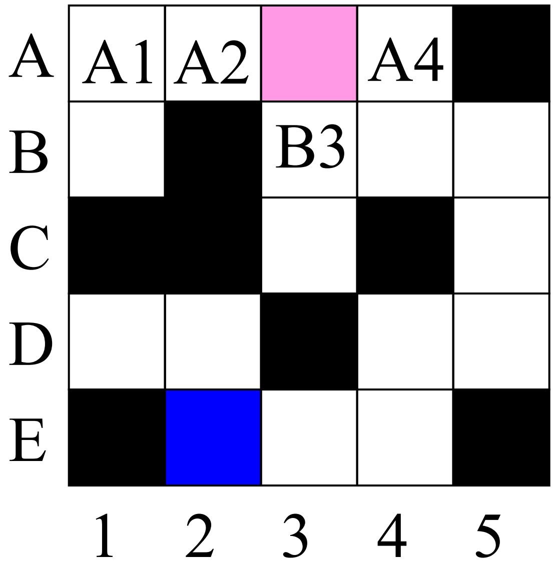


Operators: (in order)

- **`go_left`**(n)
- **`go_down`**(n)
- **`go_right`**(n)

each operator costs 1.

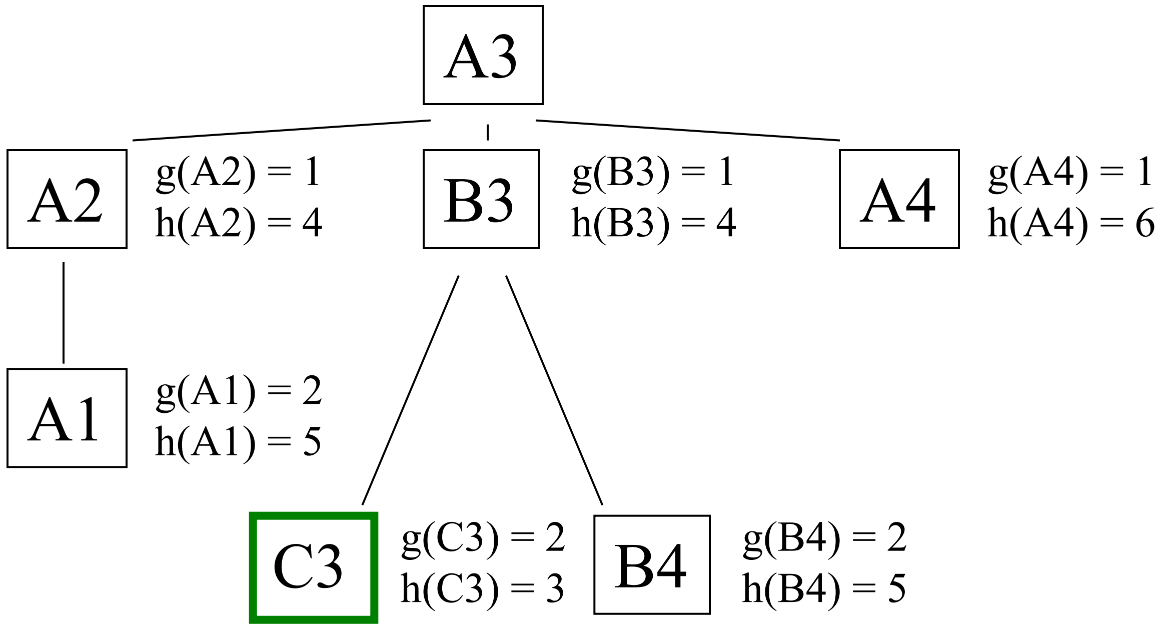


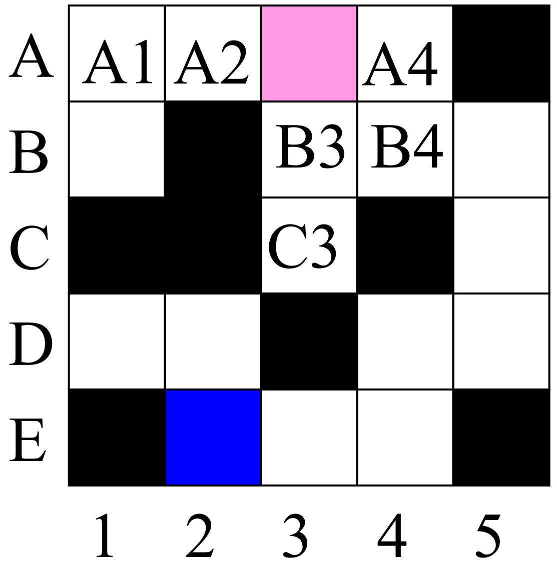


Operators: (in order)

- **`go_left`**(n)
- **`go_down`**(n)
- **`go_right`**(n)

each operator costs 1.

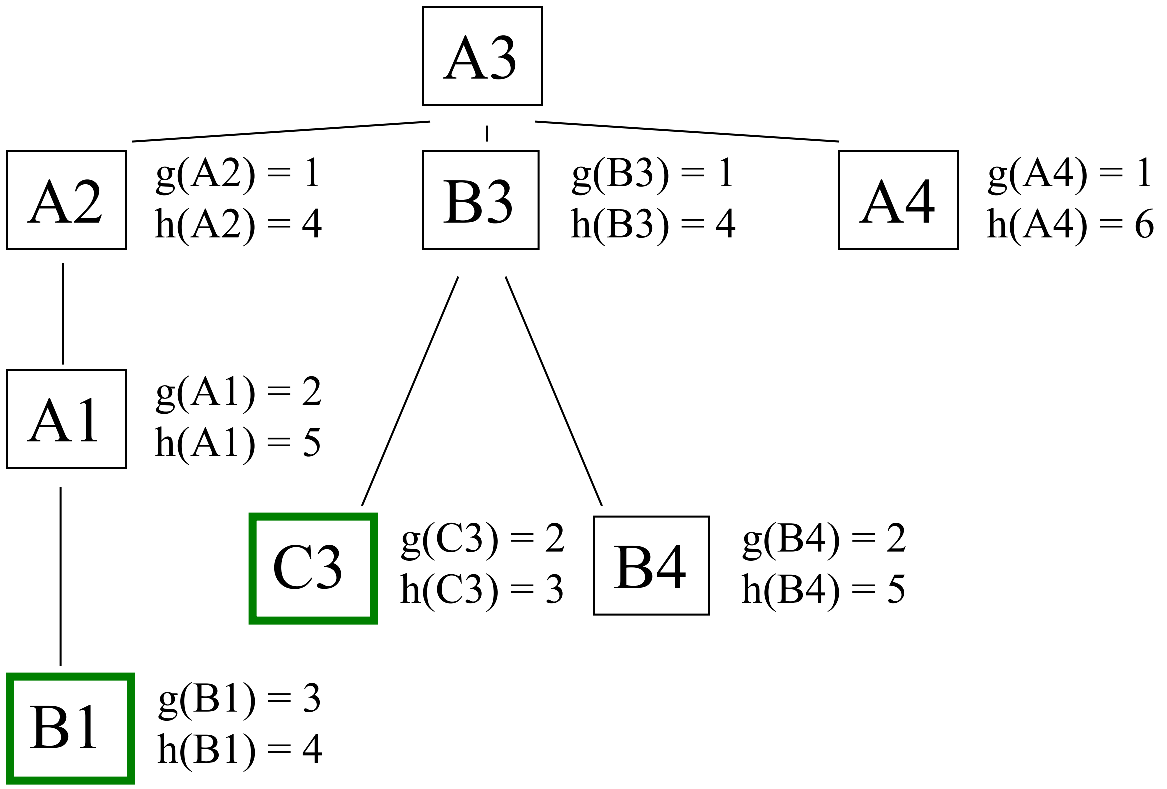


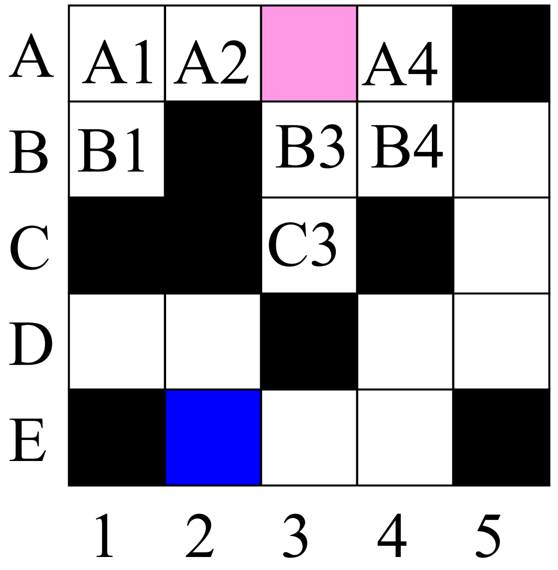


Operators: (in order)

- **`go_left`**(n)
- **`go_down`**(n)
- **`go_right`**(n)

each operator costs 1.

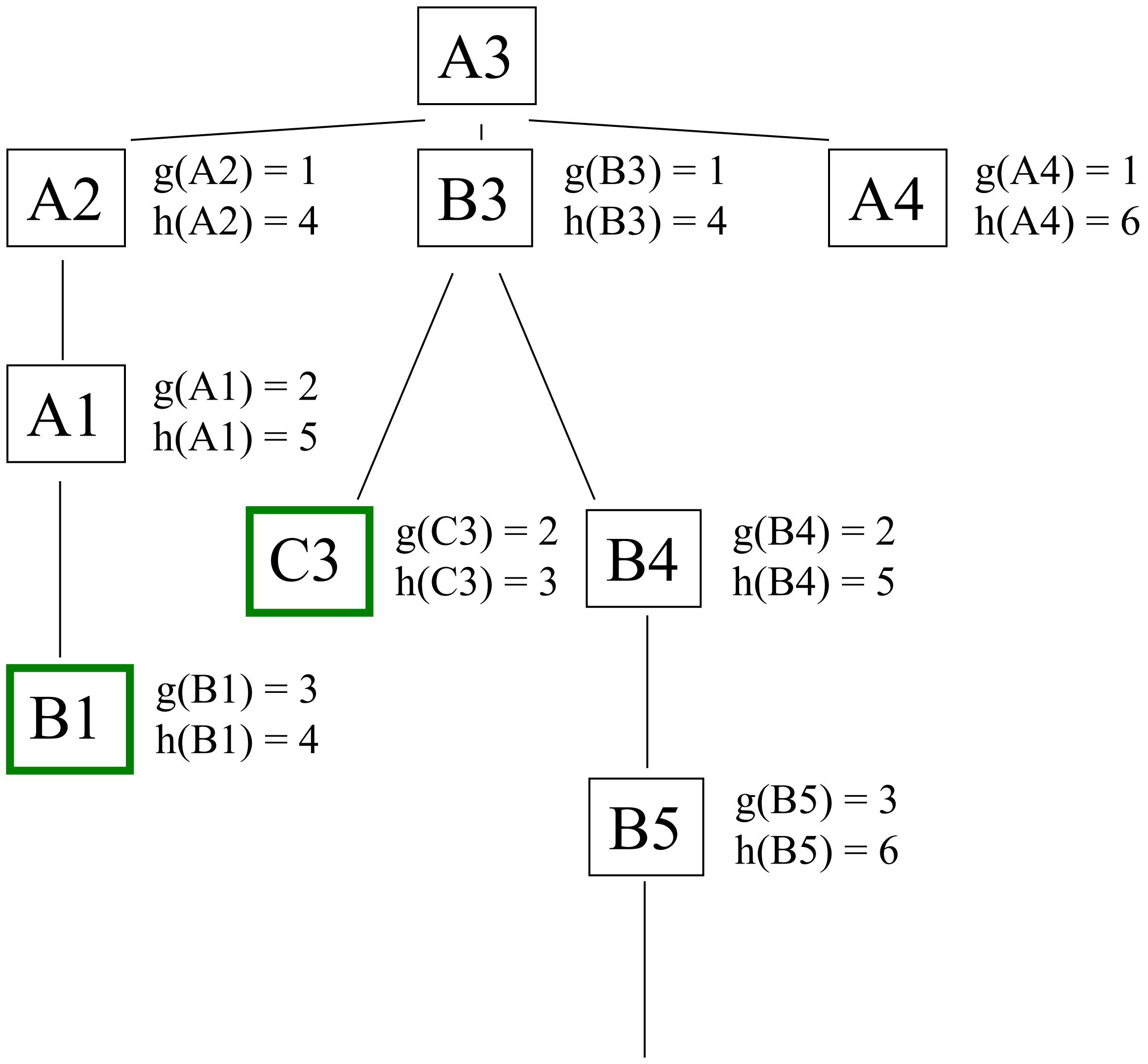


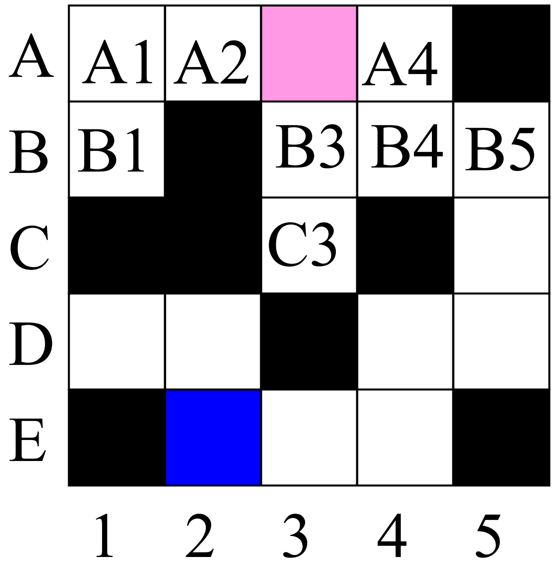


Operators: (in order)

- **`go_left`**(n)
- **`go_down`**(n)
- **`go_right`**(n)

each operator costs 1.





Operators: (in order)
- **`go_left`**(n)
- **`go_down`**(n)
- **`go_right`**(n)

each operator costs 1.