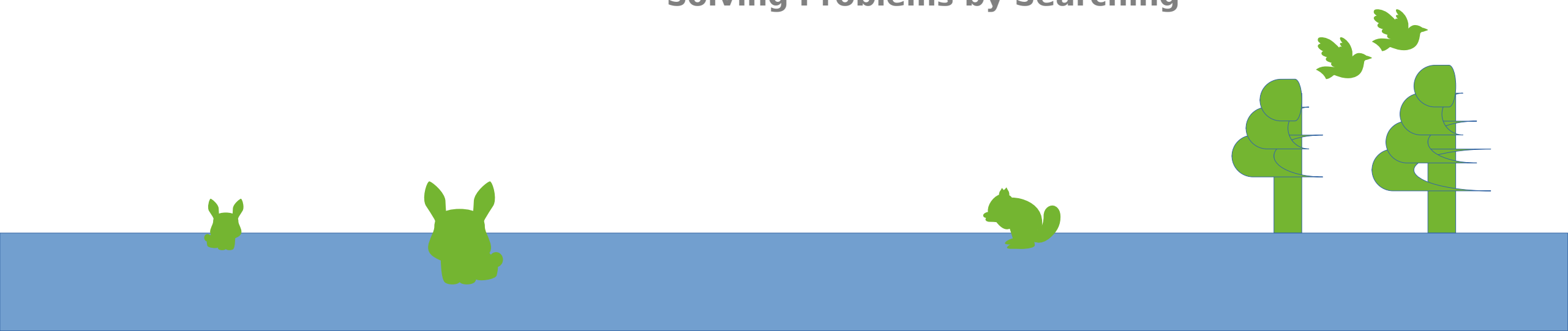
Chapter three

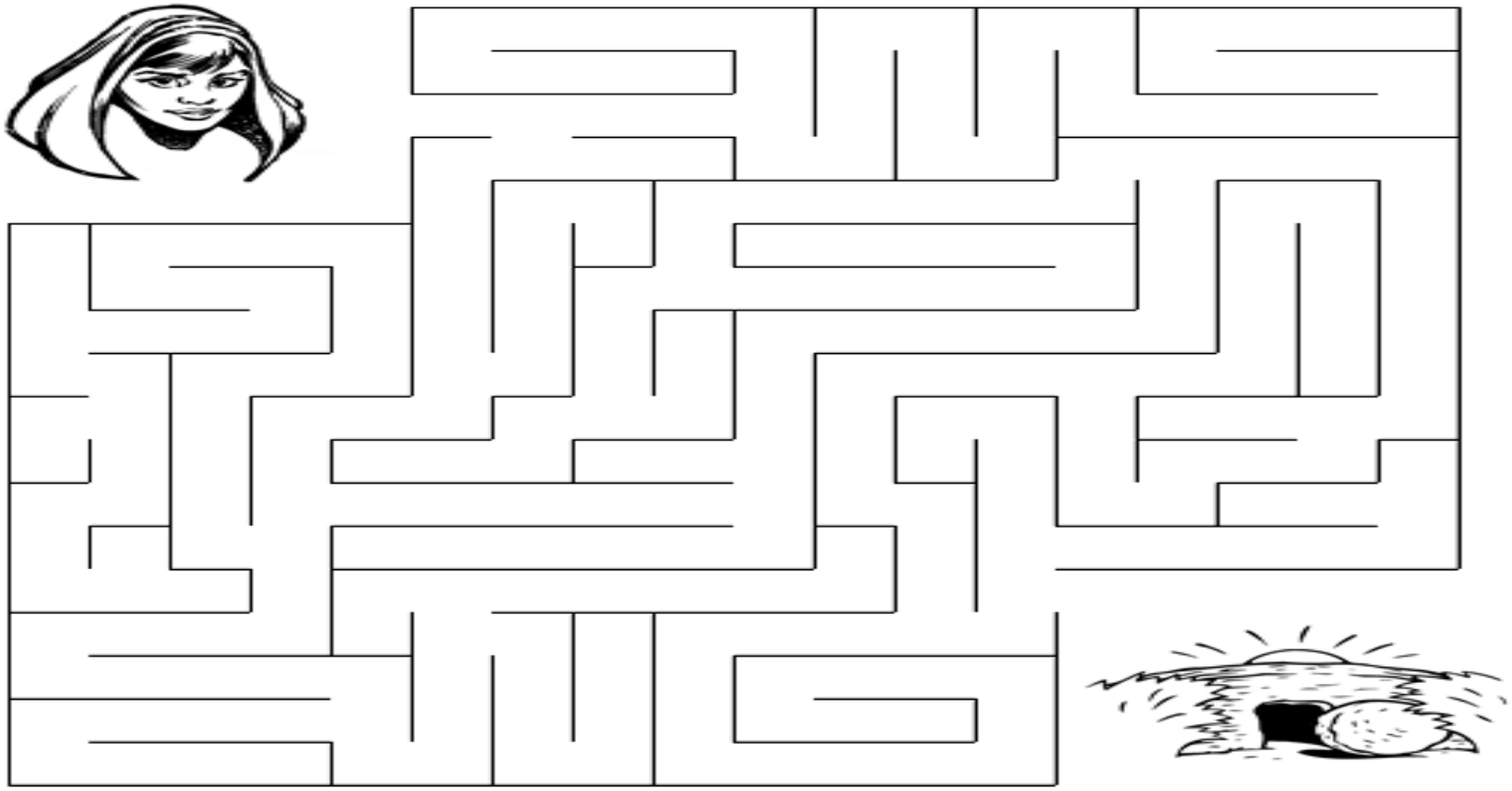
Solving Problems by Searching



Problem Solving by Searching

- Problem is a **goal** and a means for **achieving** the goal.
- The process of exploring what the means can do is **search**.
- **Search** is the process of considering various possible sequences of operators applied to the **initial state**, and finding out a sequence which culminates in a **goal sate.**
- Solution will be a sequence of operations (actions) leading from initial state to goal state (plan)

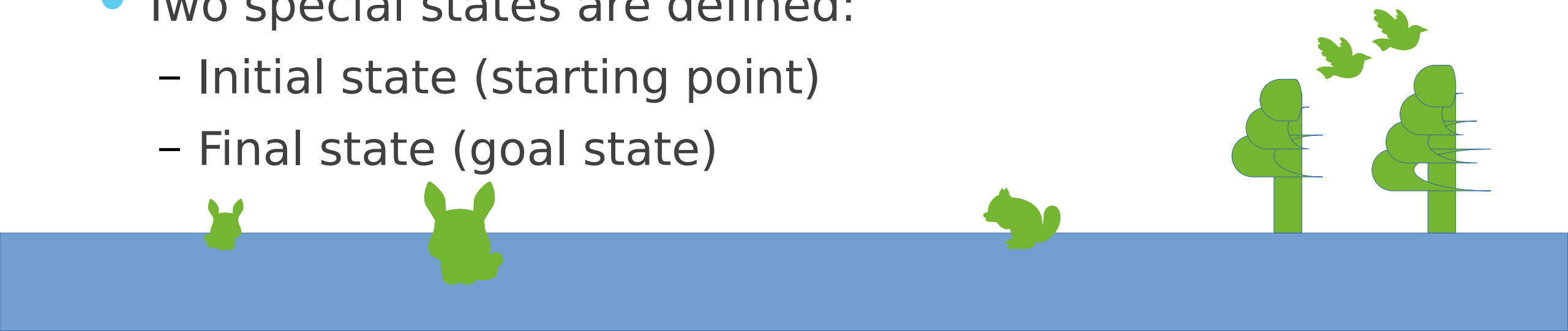
Example



The states in the space can correspond to any kind of configuration For example, the possible settings of a device, positions in a game or (more abstract still) a set of assignments to variables. Paths in state space correspond to possible sequences of transitions between states

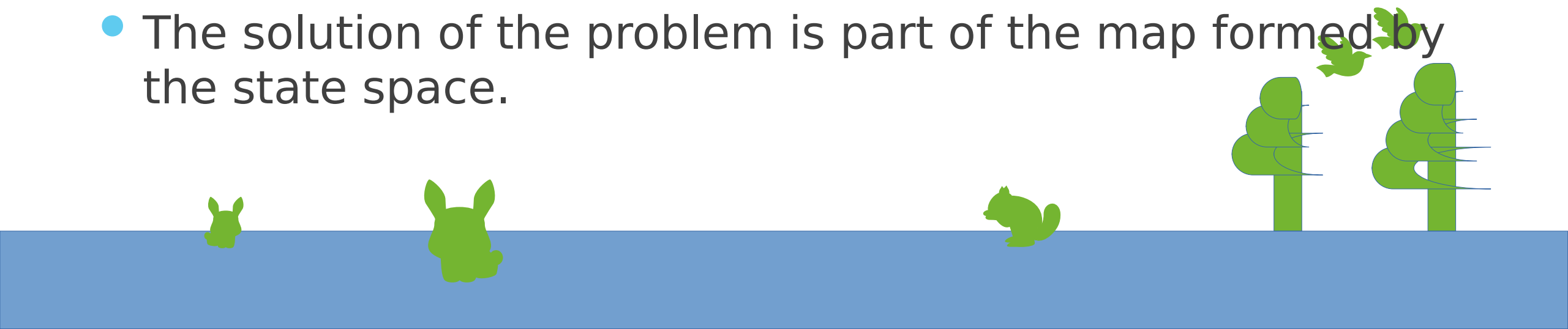
States

- A problem is defined by its **elements** and their **relations.**
- In each instant of the resolution of a problem, those elements have specific descriptors (How to select them?) and relations.
- A **state** is a representation of those elements in a **given moment**.
- Two special states are defined:
  - Initial state (starting point)
  - Final state (goal state)



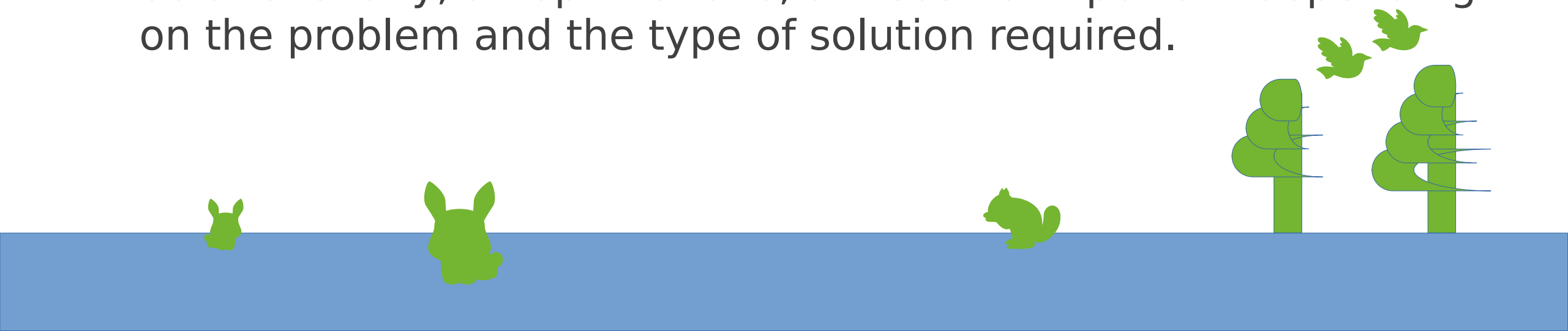
State space

- The **state space** is the set of all states reachable from the **initial state**.
- It forms a graph (or map) in which the nodes are states and the arcs between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The solution of the problem is part of the map formed by the state space.



Problem solution

- A solution in the state space is a path from the initial state to a goal state or, sometimes, just a goal state.
- **Path/solution cost**: function that assigns a numeric cost to each path, the cost of applying the operators to the states
- **Solution quality** is measured by the **path cost function**, and an optimal solution has the lowest path cost among all solutions.
- Solutions: any, an optimal one, all. Cost is important depending on the problem and the type of solution required.

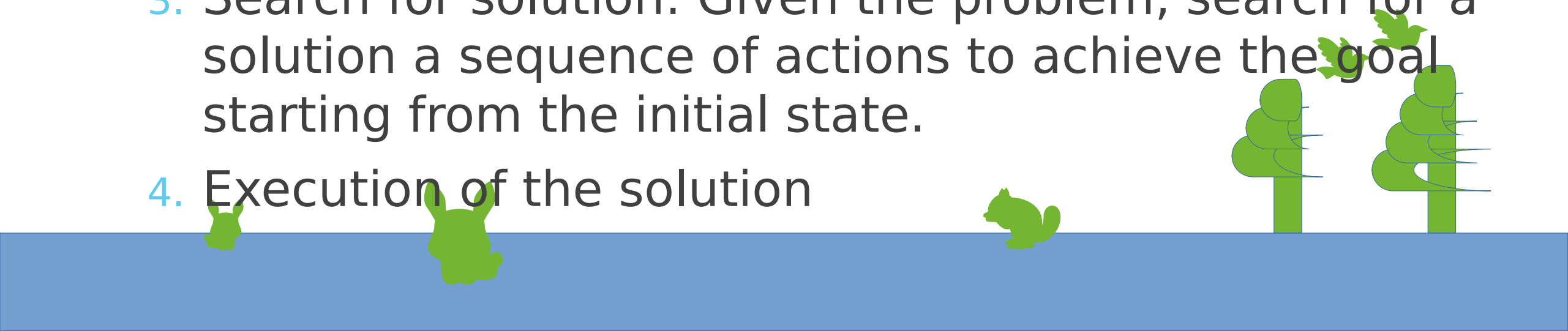


Problem-solving agents

- In Artificial Intelligence, Search techniques are universal problem-solving methods.
- Problem-solving agents are the goal-based agents and use **atomic representation.**

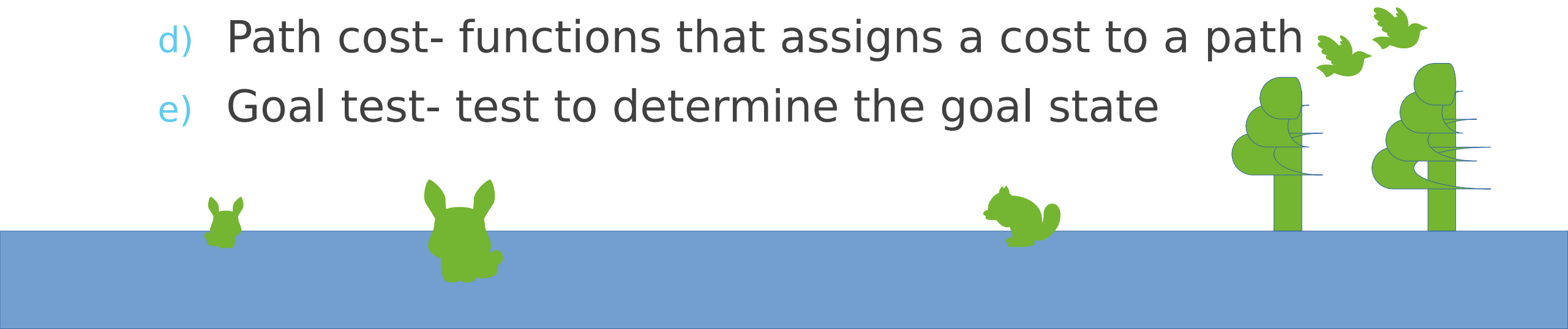
Cont..

- Problem solving agents are goal-directed agents:
  1. Goal Formulation: Set of one or more (desirable) world states (e.g. checkmate in chess).
  2. Problem formulation: What actions and states to consider given a goal and an initial state.
  3. Search for solution: Given the problem, search for a solution a sequence of actions to achieve the goal starting from the initial state.
  4. Execution of the solution



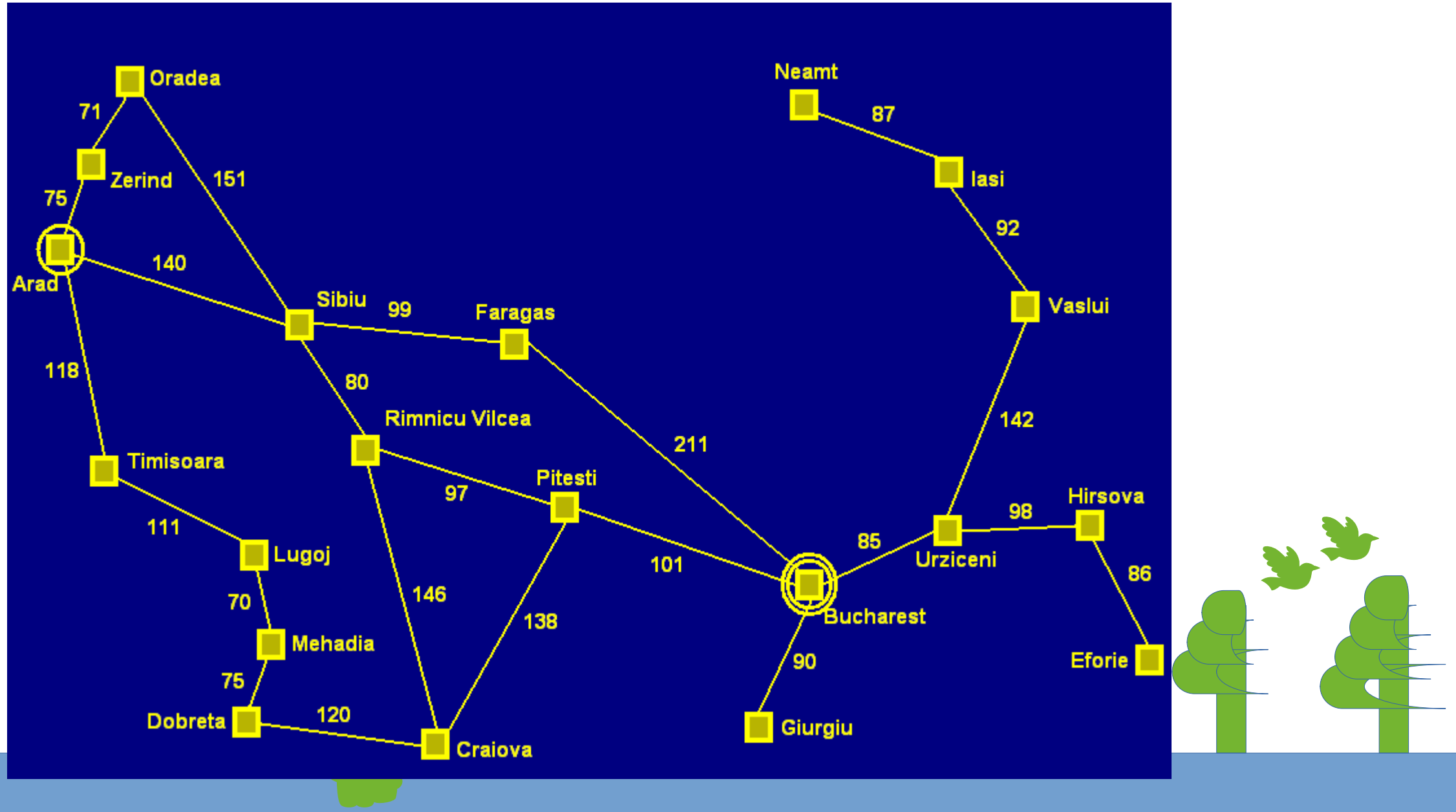
Problem formulation

- A well-defined problem can be described by:
  a) Initial state: The initial state that the agent starts in.
  b) Action: A description of the possible actions available to the agent.
  c) Transition model :A description of what each action does; the formal name for this is the transition model.
  d) Path cost- functions that assigns a cost to a path
  e) Goal test- test to determine the goal state



Example A

- The **initial state is** In(Arad)
- From the state In(Arad), the applicable **actions** are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.
- Transition Model: RESULT(In(Arad), Go(Zerind)) = In(Zerind)
- **Goal test** :{In(Bucharest)}
- **Path cost:** length in kilometers


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Faragas
118
80
Rimnicu Vilcea
211
Timisoara
97
Pitesti
111
Lugoj
101
70
146
138
Mehadia
75
Dobreta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
Hirsova
98
85
Urziceni
86
Bucharest
90
Eforie
Giurgiu

Example B : The 8-puzzle

S: start state

2	8	3
1	6	4
7		5

G: Goal state

1	2	3
8		4
7	6	5

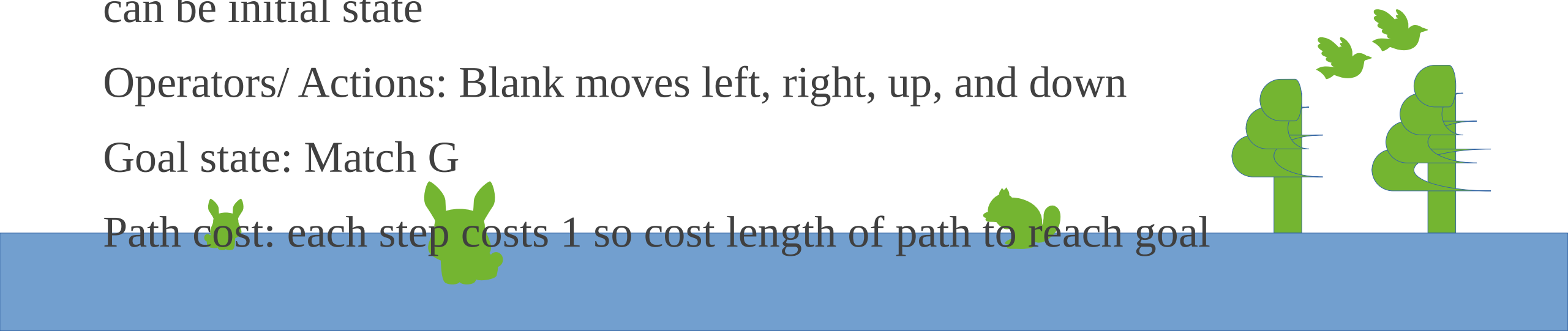
State:

The boards, i.e., Location of blank, integer location of tile Initial sate: any sate can be initial state

Operators/ Actions: Blank moves left, right, up, and down

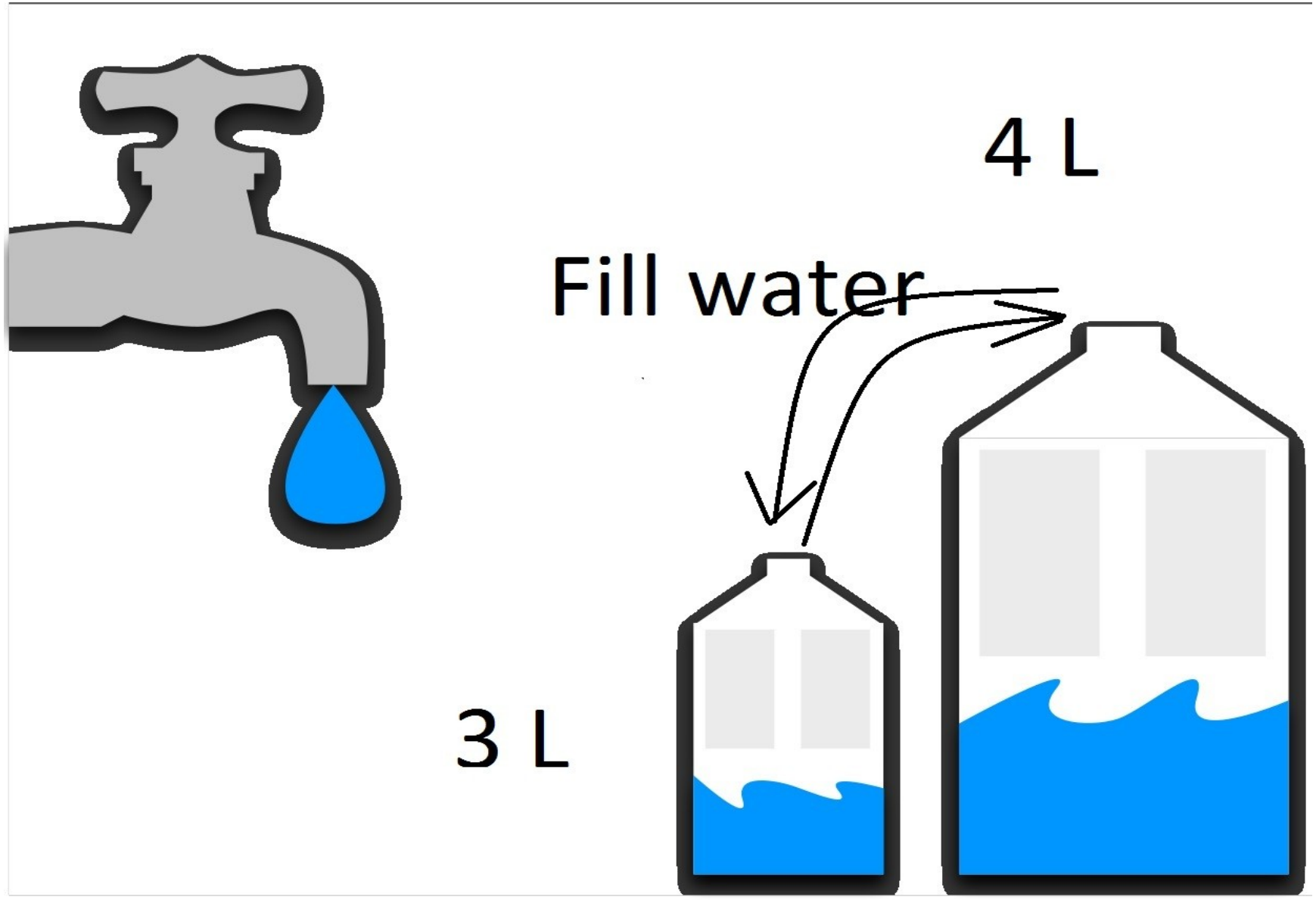
Goal state: Match G

Path cost: each step costs 1 so cost length of path to reach goal



Example

- You are given two jugs, a 4-gallon and a 3-gallon one. Neither has any measuring marks on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jugs.
- Specify the initial state, the goal state , all the possible operators to reach from the start state to the goal state.
- Solution:


4 L
Fill water
3 L

Steps to solve the problem

- There are many possible ways to formulate the problem as search.
- 1st step:
  - State description the integers (x,y) {x:0,1,2,3,4}, {y:0,1,2,3,4}
- 2nd step:
  - Describe the initial and goal state
    - The initial state is {0,0}
    - The goal state is {2,x} where x can take any value.
- 3rd step:
  - List all the action in the production rule(as rules or operators as follows)

Possible answers

- **Option 1**

  (0,3)

  (3,0)

  (3,3)

  (4,2)

  (0,2)

  (2,0)

- **Option2**

  (4,0)

  (1,3)

  (1,0)

  (0,1)

  (4,1)

  (2,3)

  (2,0)

Example C: The wolf-goat-cabbage p

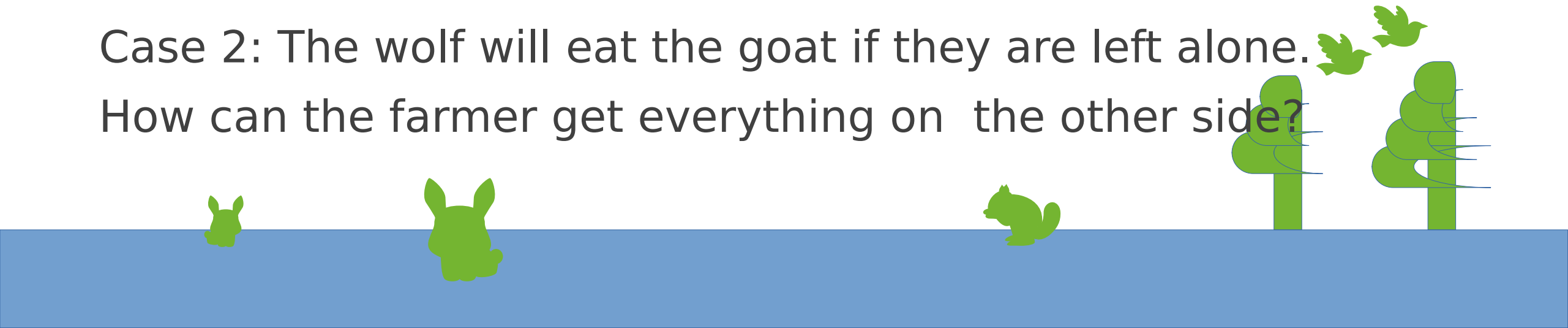
Cont..

A farmer has a goat a wolf and a cabbage on the west side of the river. He wants to get all of his animals and his cabbage across the river onto the cost side. The farmer has a boat but he only has enough room for himself and one other thing.

Case 1: The goat will eat the cabbage if they are left together alone.

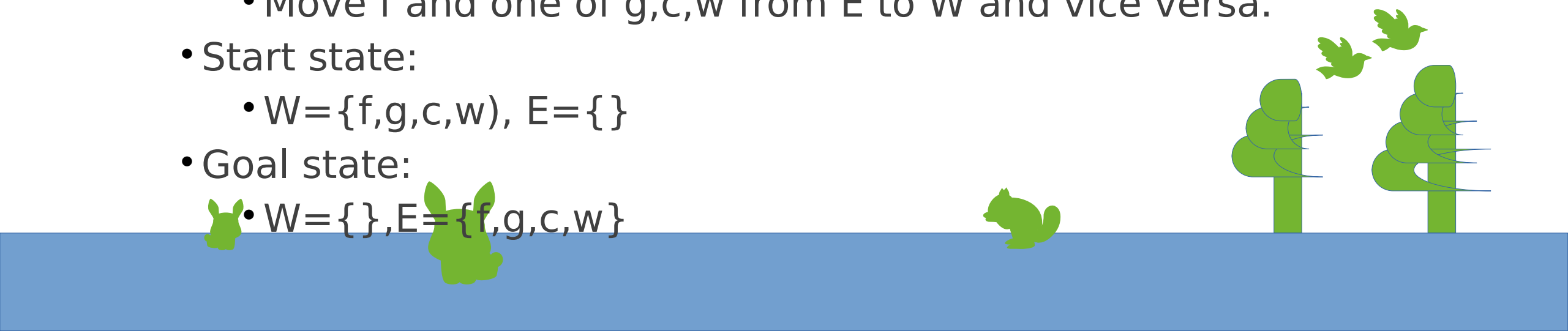
Case 2: The wolf will eat the goat if they are left alone.

How can the farmer get everything on the other side?



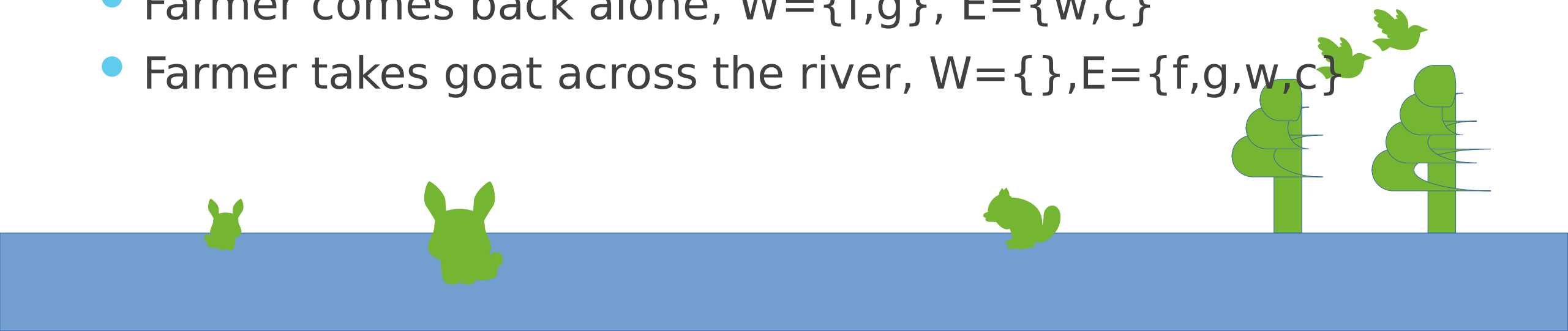
Possible solution

- State Space Representation:
  - We can represent the states of the problem with tow sets W and E. We can also can have representations for the elements in the two sets as f,g,w,c representing the farmer, goat, wolf, and cabbage.
  - Actions :
    - Move f from the E to W and vice versa
    - Move f and one of g,c,w from E to W and vice versa.
  - Start state:
    - W={f,g,c,w), E={}
  - Goal state:
    - W={},E={f,g,c,w}



One possible Solution:

- Farmer takes goat across the river, W={w,c},E={f,g}
- Farmer comes back alone,W={f,c,w,},E={g}
- Farmer takes wolf across the river,W={c},E=f,g,w}
- Farmer comes back with goat, W=={f,g,c},E={w}
- Farmer takes cabbage across the river,W={g},E={f,w,c}
- Farmer comes back alone, W={f,g}, E={w,c}
- Farmer takes goat across the river, W={},E={f,g,w,c}

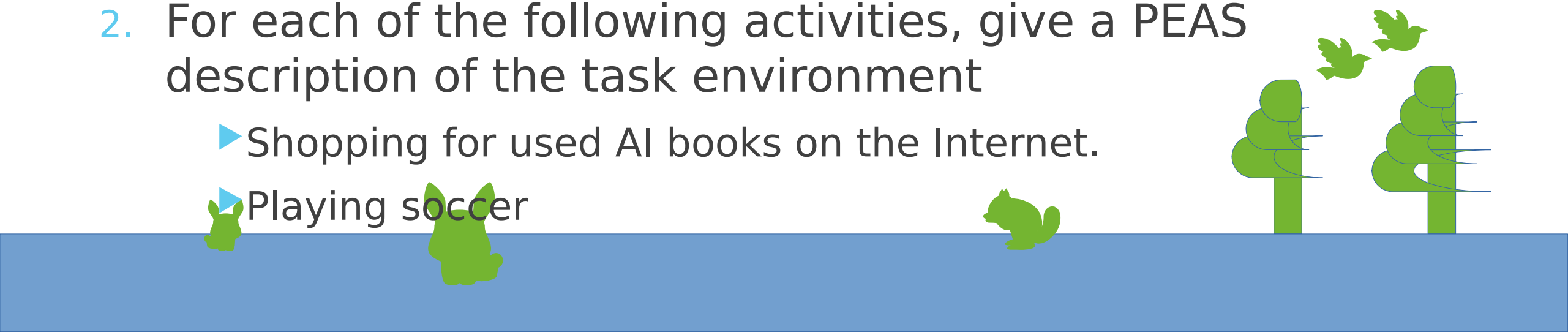


Quiz

1. The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

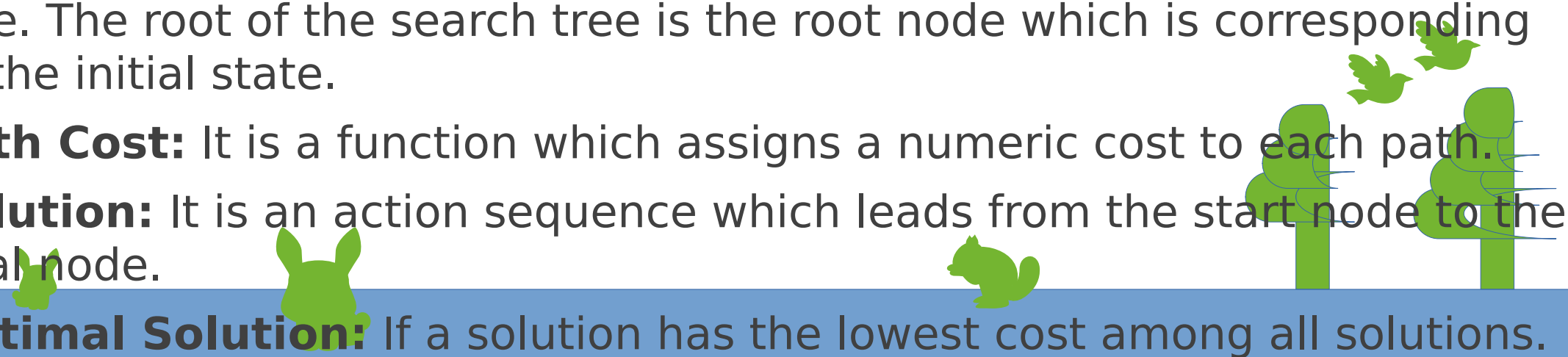
   A. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution.

2. For each of the following activities, give a PEAS description of the task environment
   - Shopping for used AI books on the Internet.
   - Playing soccer



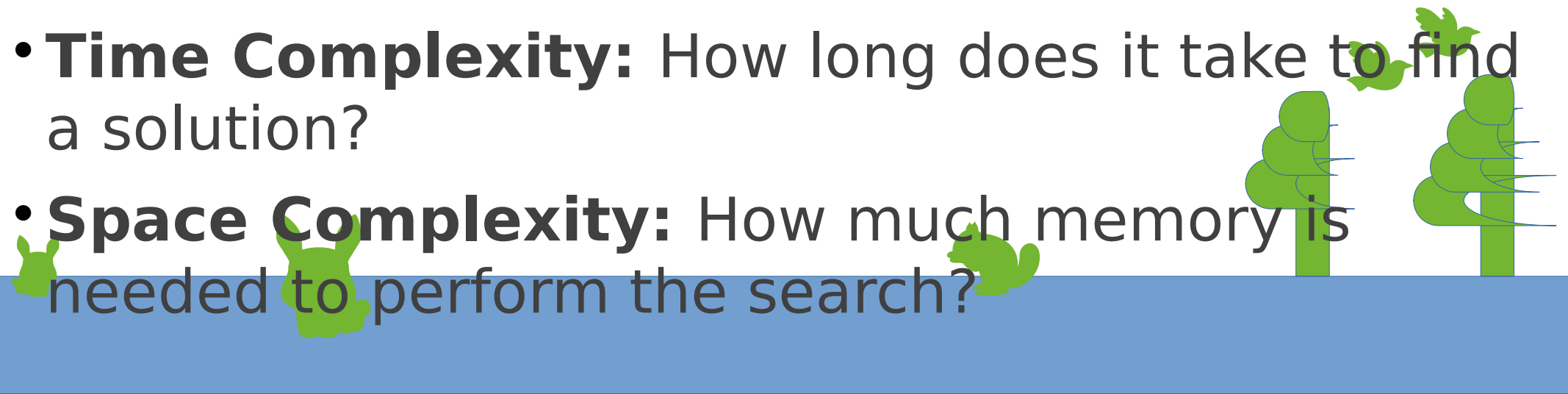
Search Algorithm Terminologies

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
  - **Search Space:** Search space represents a set of possible solutions, which a system may have.
  - **Start State:** It is a state from where agent begins **the search**.
  - **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.



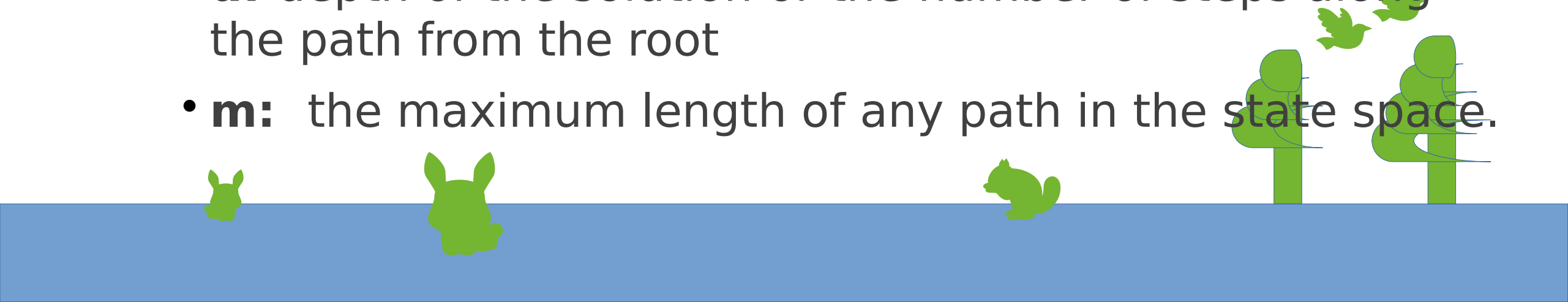
Properties of Search Algorithms

- Following are the four essential properties of search algorithms to compare the efficiency of these algorithms
  - **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
  - **Optimality:** Does the strategy find the optimal solution
  - **Time Complexity:** How long does it take to find a solution?
  - **Space Complexity:** How much memory is needed to perform the search?



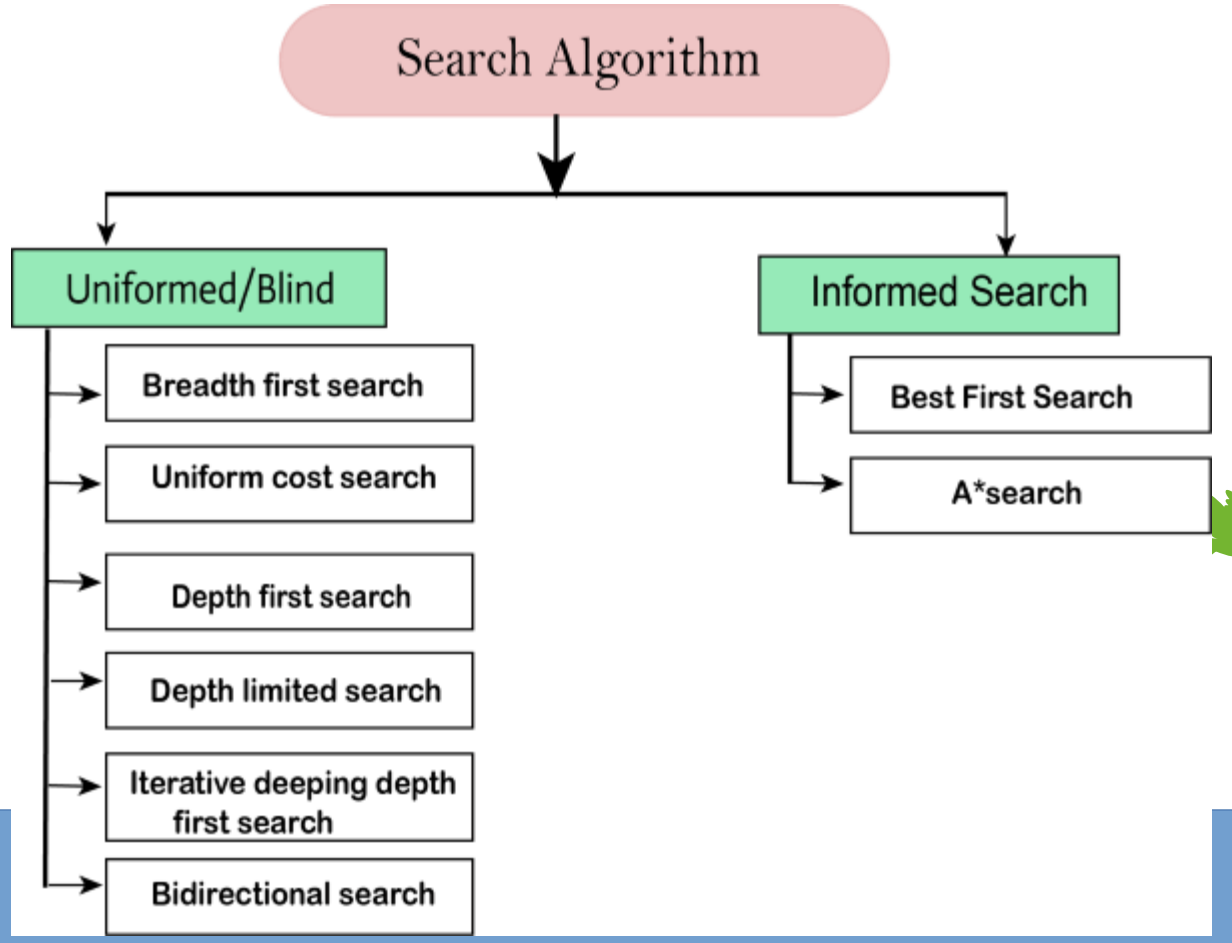
Cont..

- Different search strategies are evaluated along completeness, time complexity, space complexity and optimality.
- The **time** and **space** complexity are measured in terms of:
  - **b:** the branching factor or maximum number of successors of any node.
  - **d:** depth of the solution or the number of steps along the path from the root
  - **m:** the maximum length of any path in the state space.



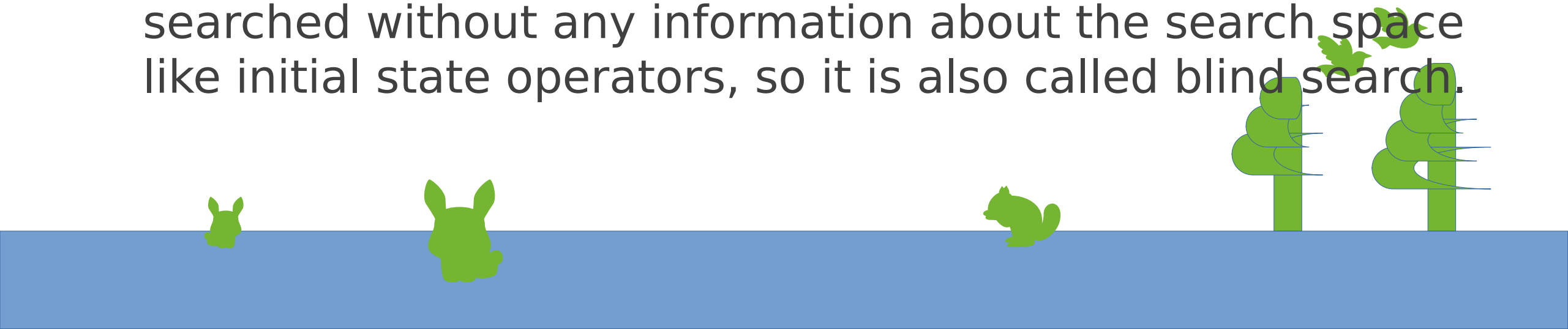
Types of search algorithms

- Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



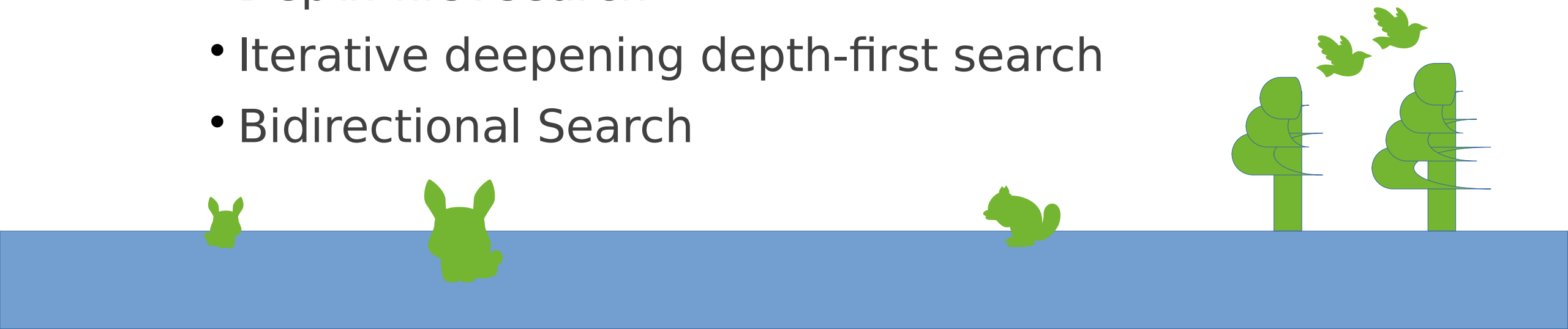
Uninformed/Blind Search

- The uninformed search does not contain any **domain knowledge** such as closeness, the location of the goal.
- It operates in a **brute-force** way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.
- Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators, so it is also called blind search.



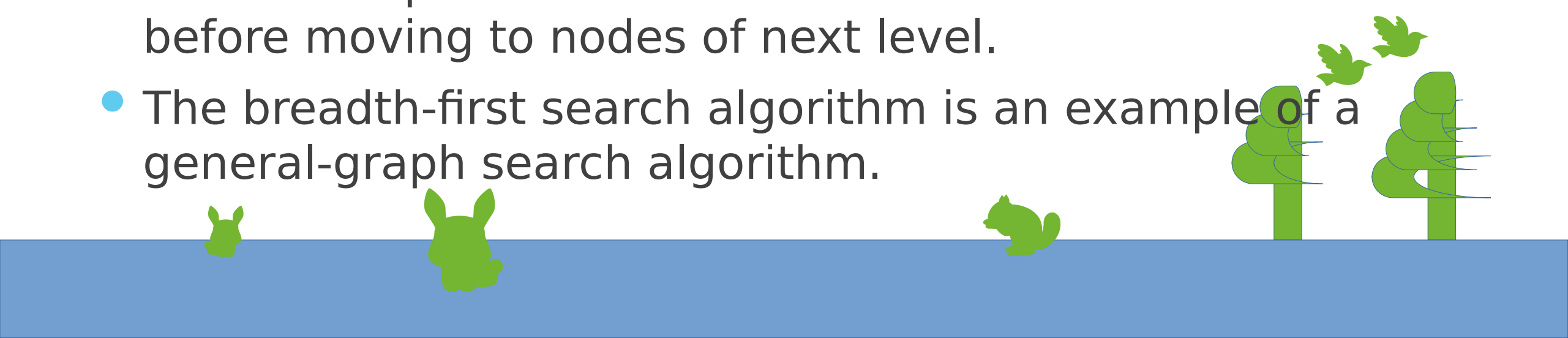
Cont..

- It examines each node of the tree until it achieves the goal node.
- **It can be divided into five main types:**
  - Breadth-first search
  - Uniform cost search
  - Depth-first search
  - Iterative deepening depth-first search
  - Bidirectional Search



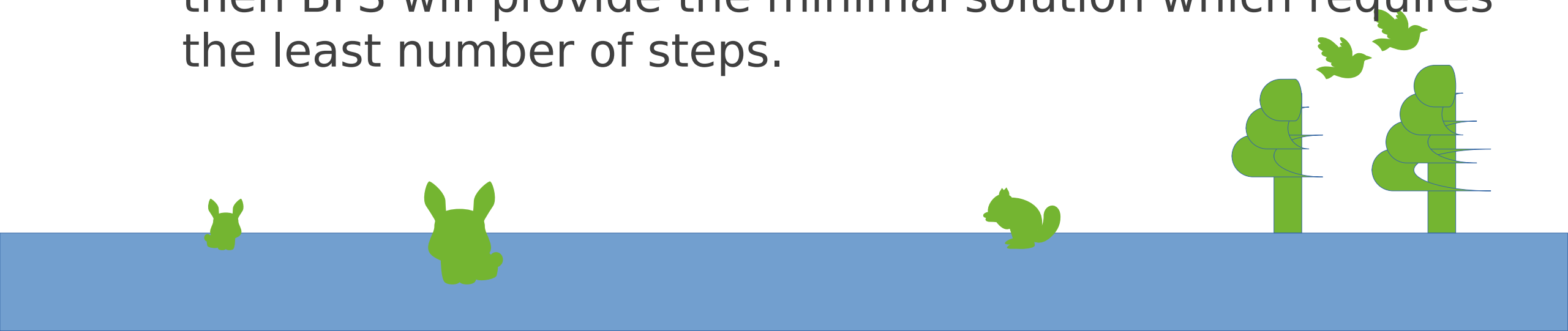
Breadth-first Search

- 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.



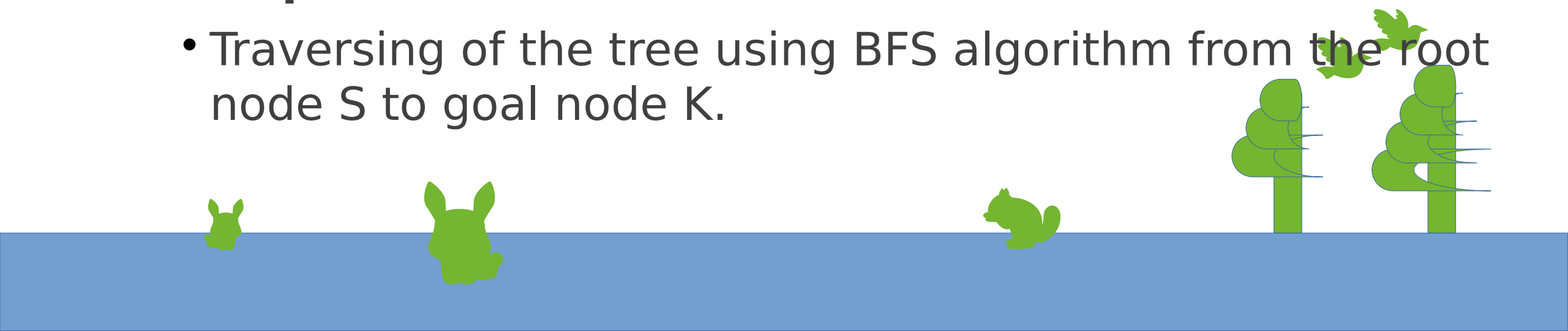
Cont..

- Breadth-first search implemented using **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.

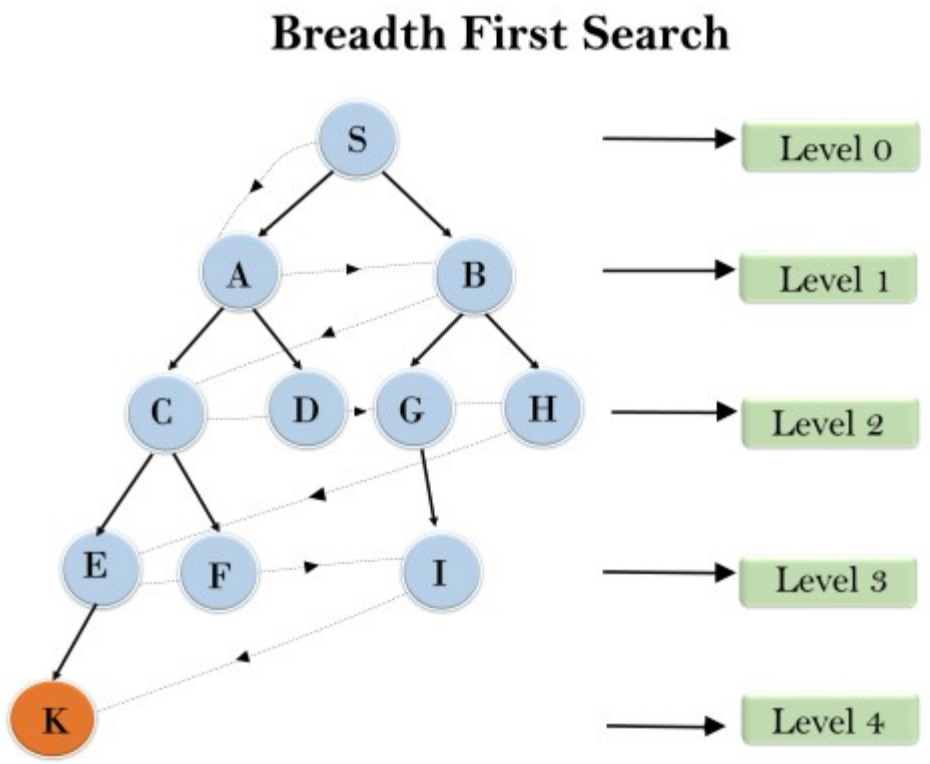


Cont..

- **Disadvantages:**
  - It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
  - BFS needs lots of time if the solution is far away from the root node.
- **Example:**
  - Traversing of the tree using BFS algorithm from the root node S to goal node K.

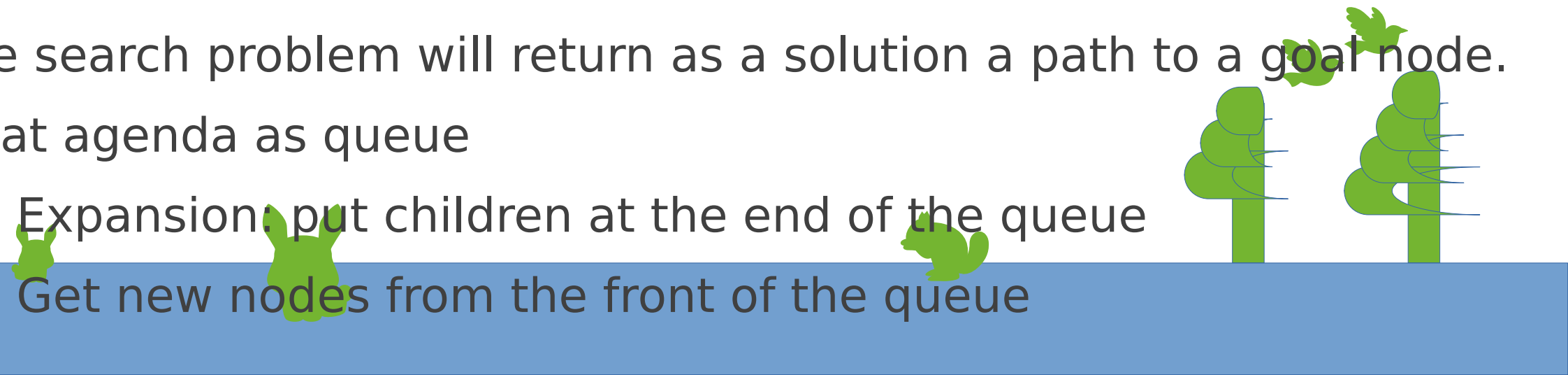


BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:
S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

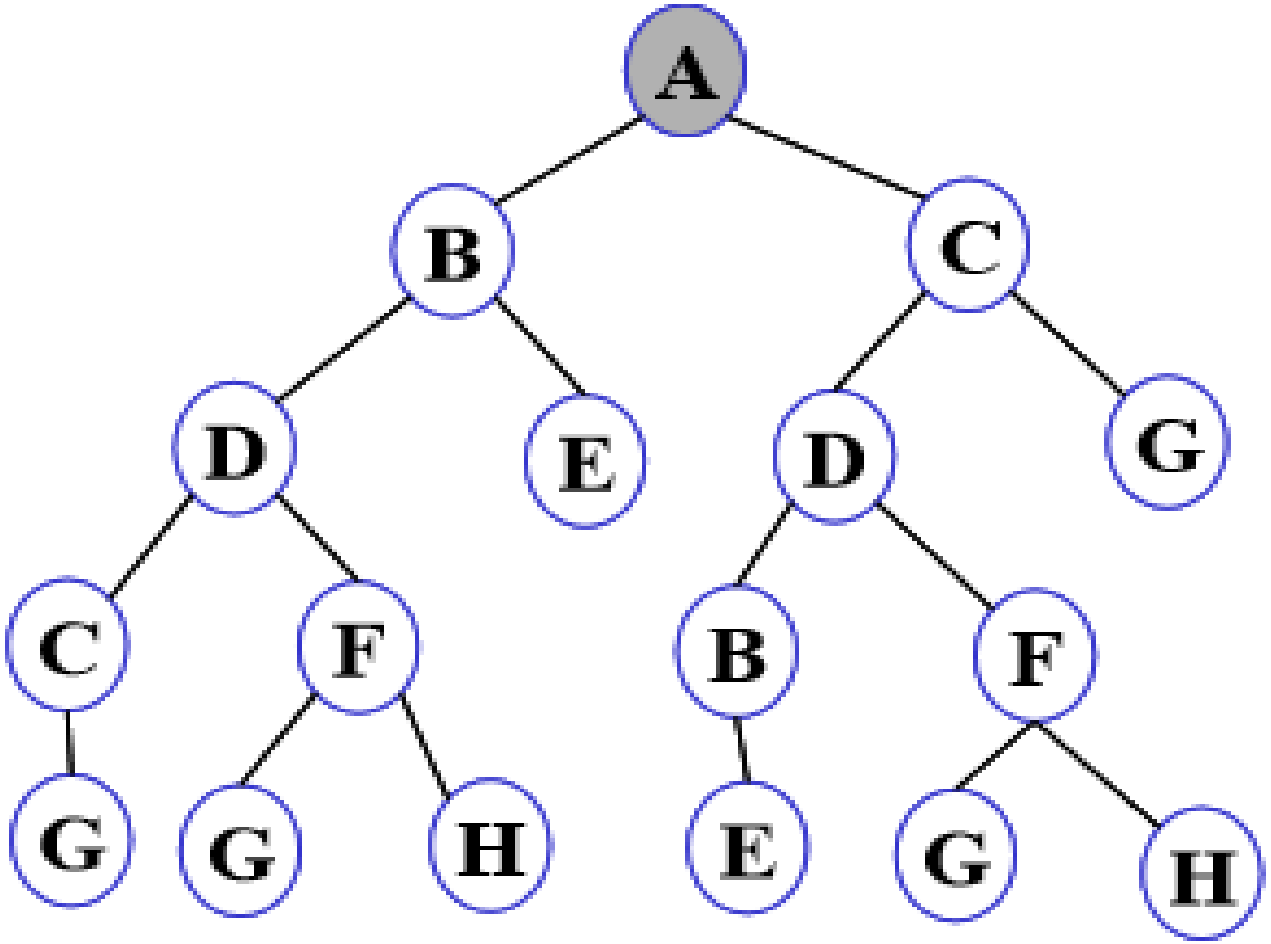


Cont..

- The process constructs a search tree is as follows, where
  - root is the initial state and
  - leaf nodes are nodes
  - not yet expanded (i.e., in fringe/agenda)
  - having no successors (i.e., “dead-ends”)
- Search tree may be infinite because of loops even if state space is small
- The search problem will return as a solution a path to a goal node.
- Treat agenda as queue
  - Expansion: put children at the end of the queue
  - Get new nodes from the front of the queue

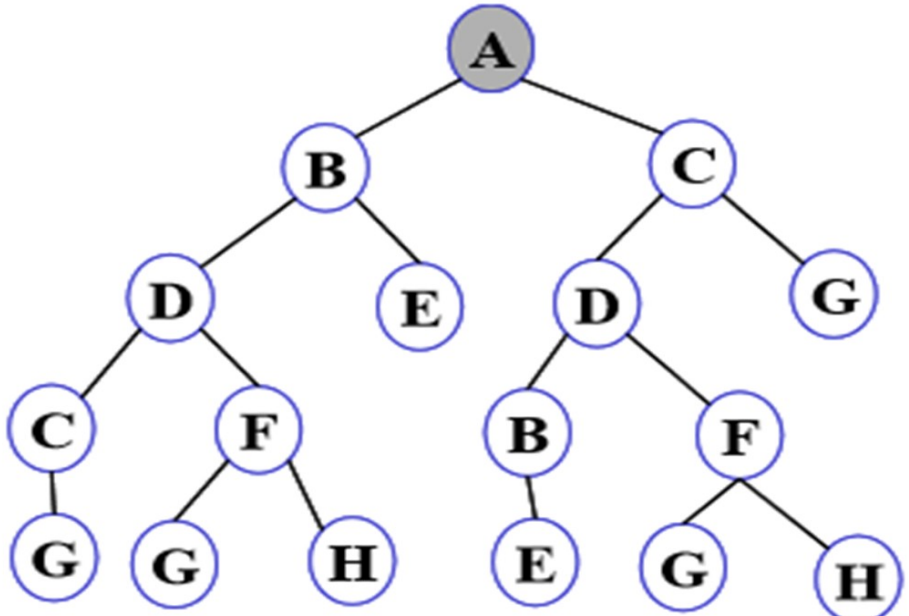


Search tree



BFS illustrated

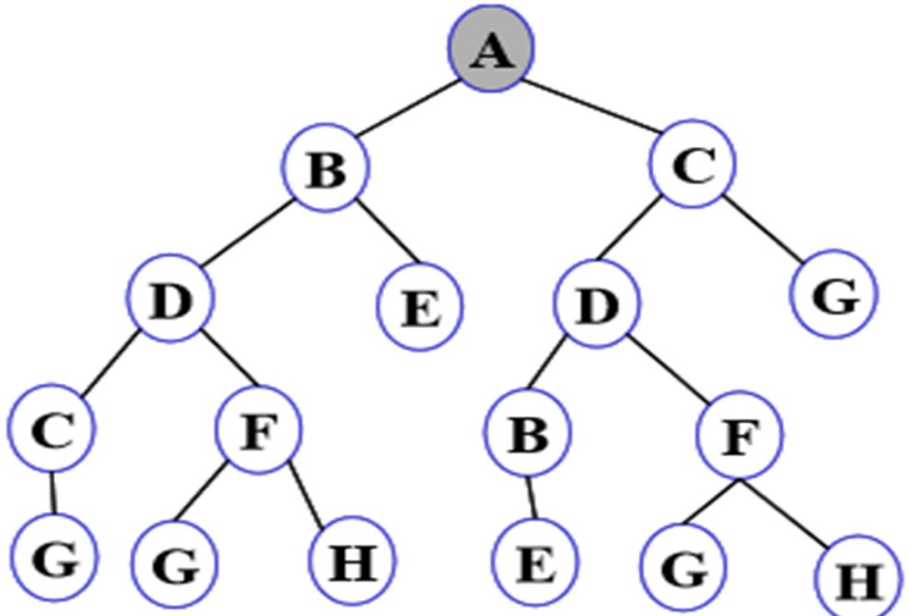
Step 1: Initially fringe contains only one node corresponding to the source state A.



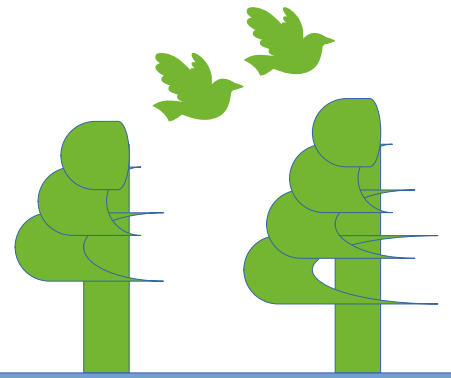
Fringe: A

BFS illustrated

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

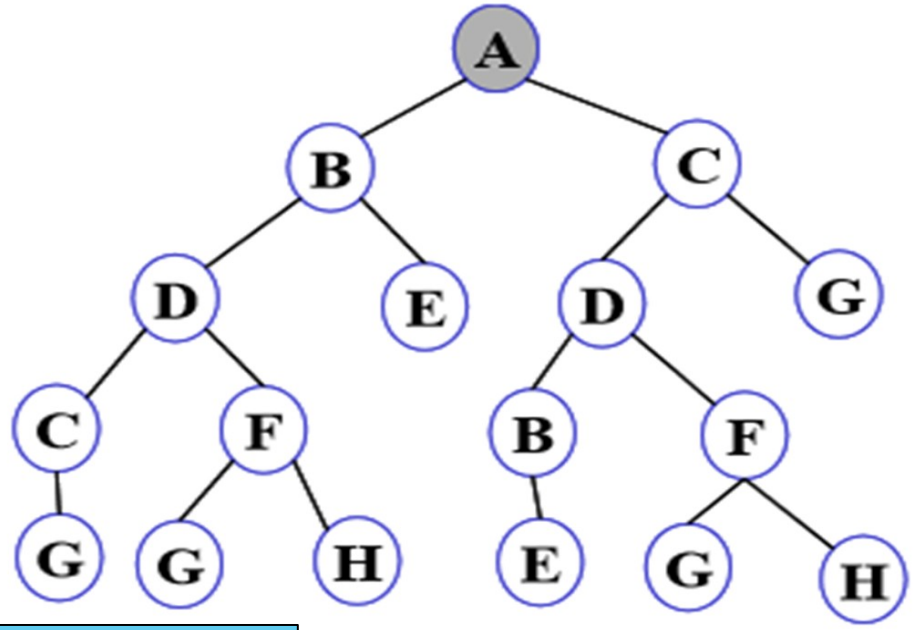


Fringe: BC



BFS illustrated

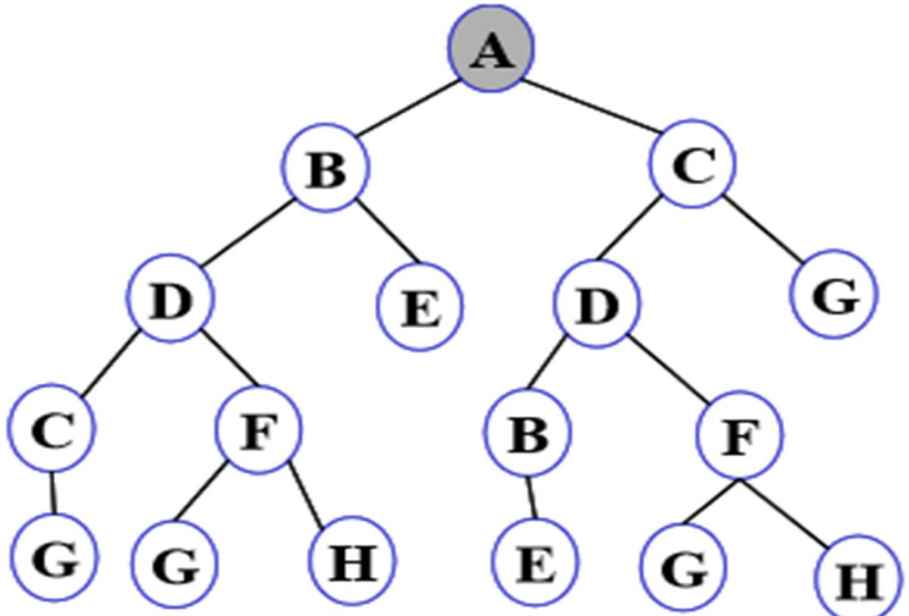
Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.



Fringe: CDE

BFS illustrated

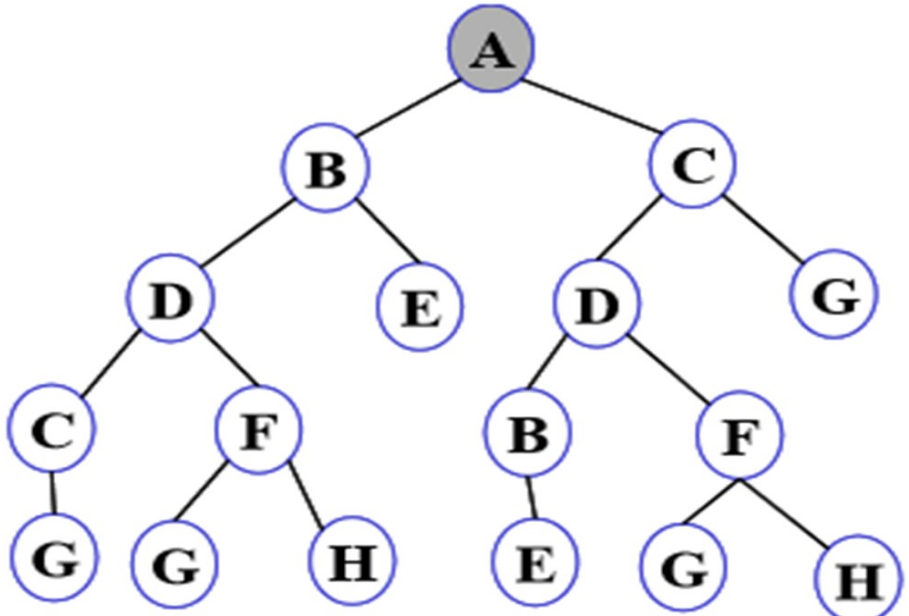
Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.



Fringe: DEDG

BFS illustrated

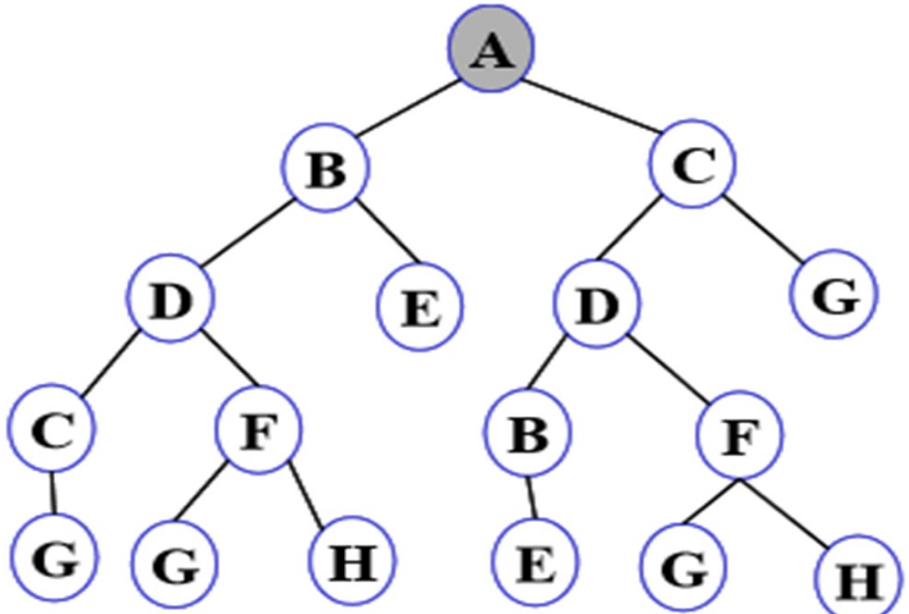
Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.



Fringe: EDGCF

BFS illustrated

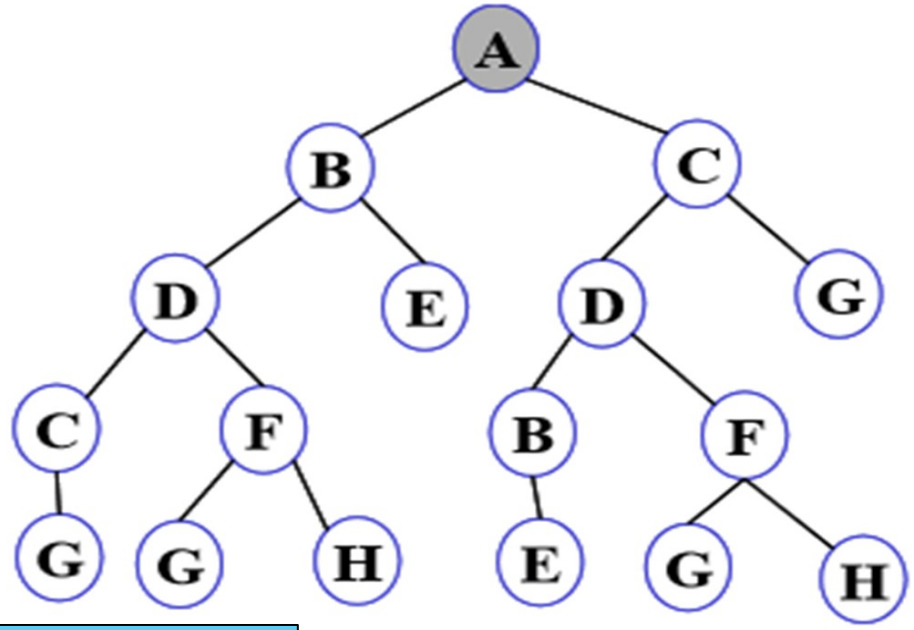
Step 6: Node E is removed from fringe. It has no children.



Fringe: DGCF

BFS illustrated

Step 7: D is expanded, B and F are put in OPEN.



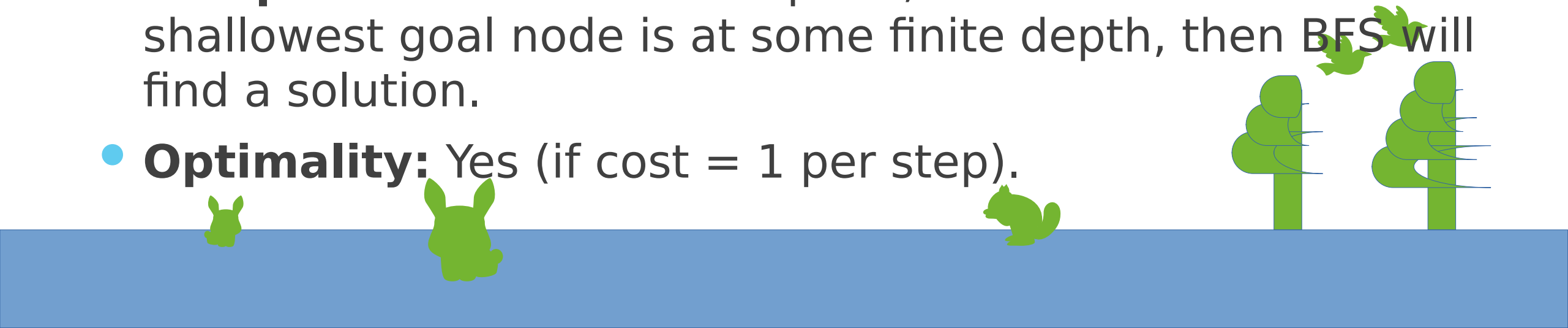
Fringe: GCFBF

cont..

Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

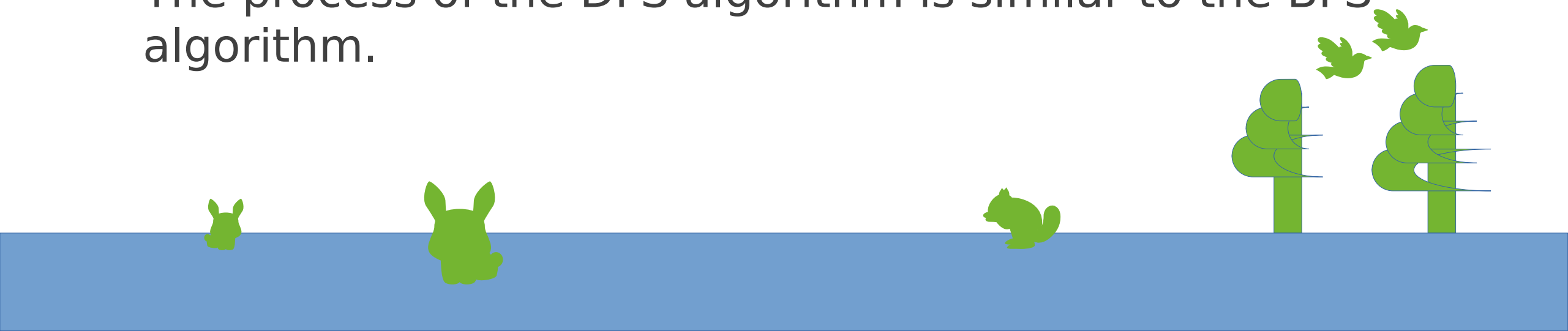
Cont..

- **Time Complexity:** 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:** Yes (if cost = 1 per step).



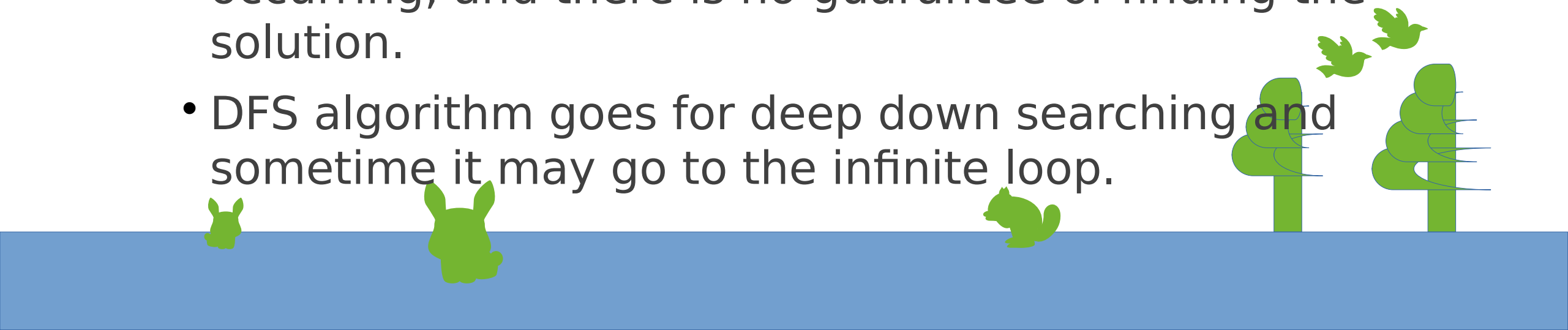
Depth-first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- 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.



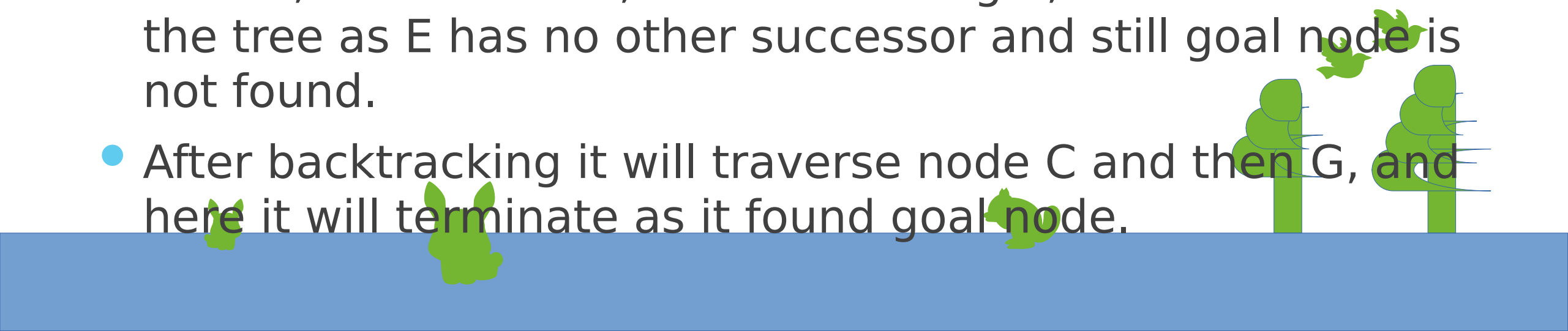
Cont..

- **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.
- **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.

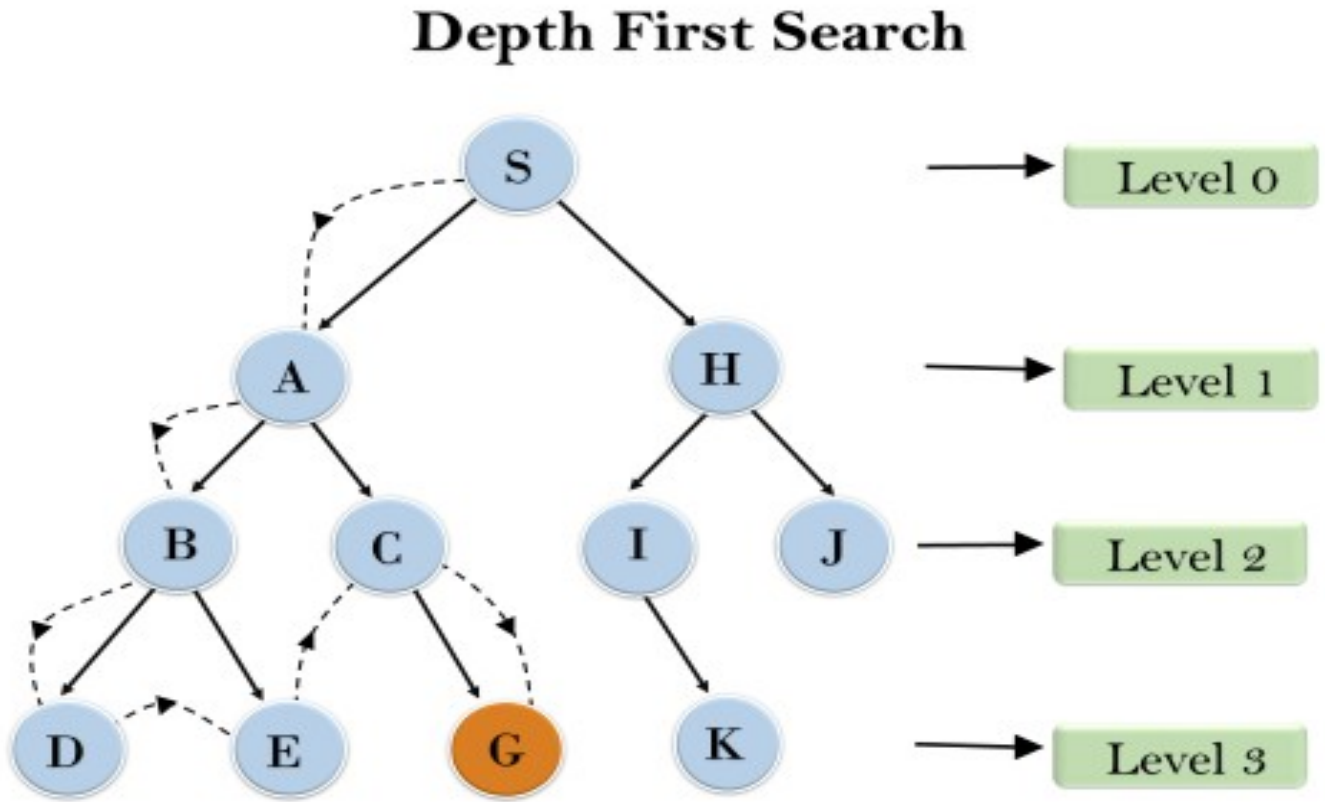


Cont..

- **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.
- It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found.
- After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

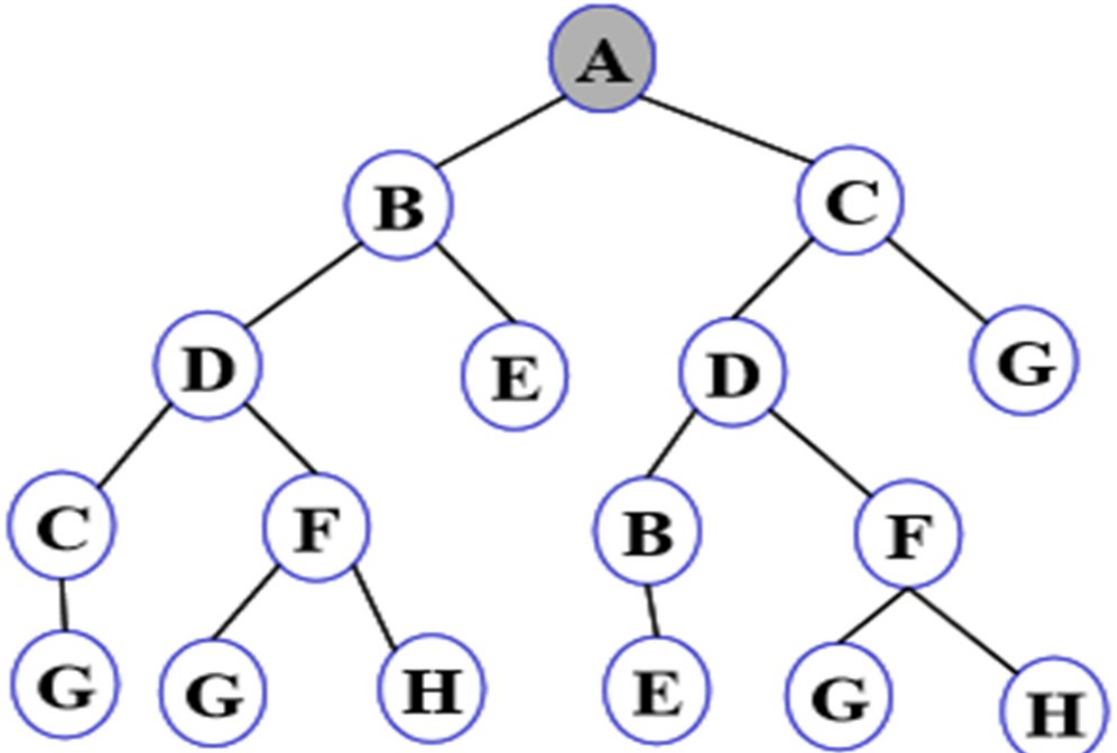


cont..


Depth First Search
S
A
H
B
C
I
J
D
E
G
K
Level 0
Level 1
Level 2
Level 3

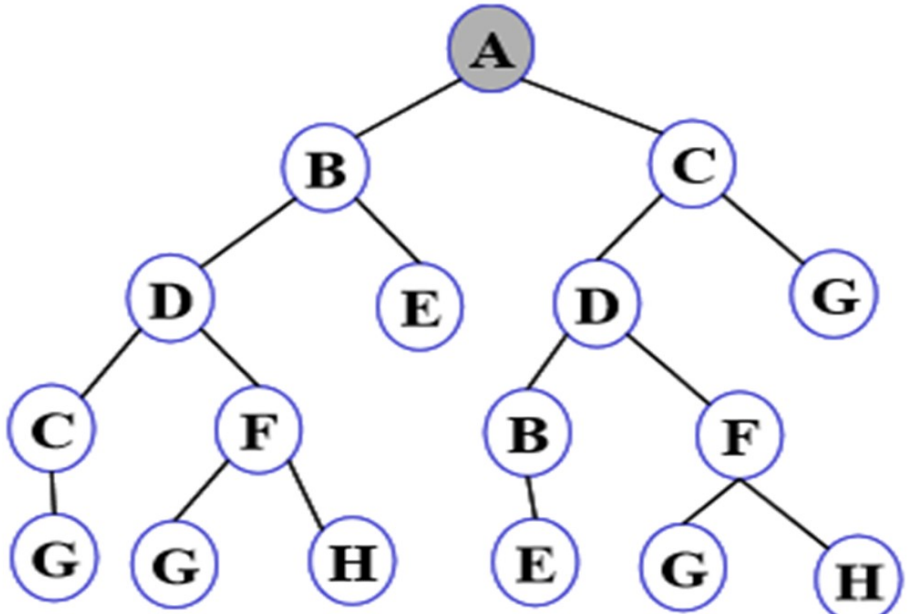
Search tree for the state space

Let us now run Depth First Search on the search space given in Figure below, and trace its progress



DFS illustrated

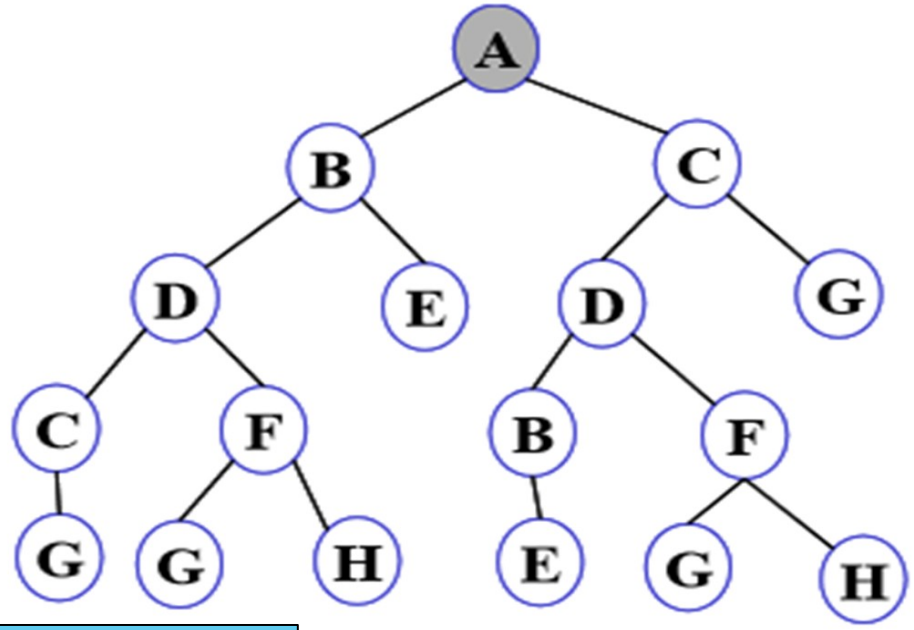
Step 1: Initially fringe contains only the node for A.



Fringe: A

DFS illustrated

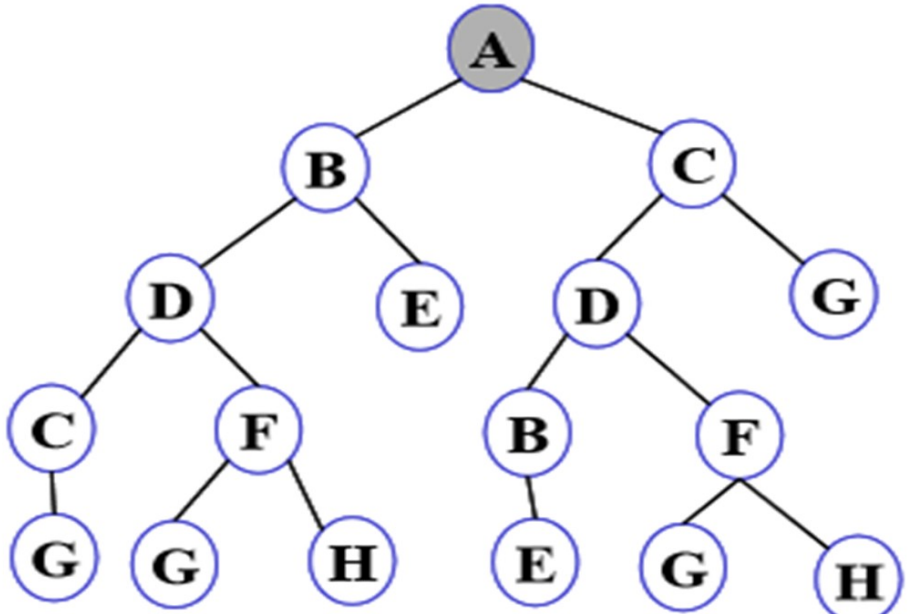
Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe



Fringe: BC

DFS illustrated

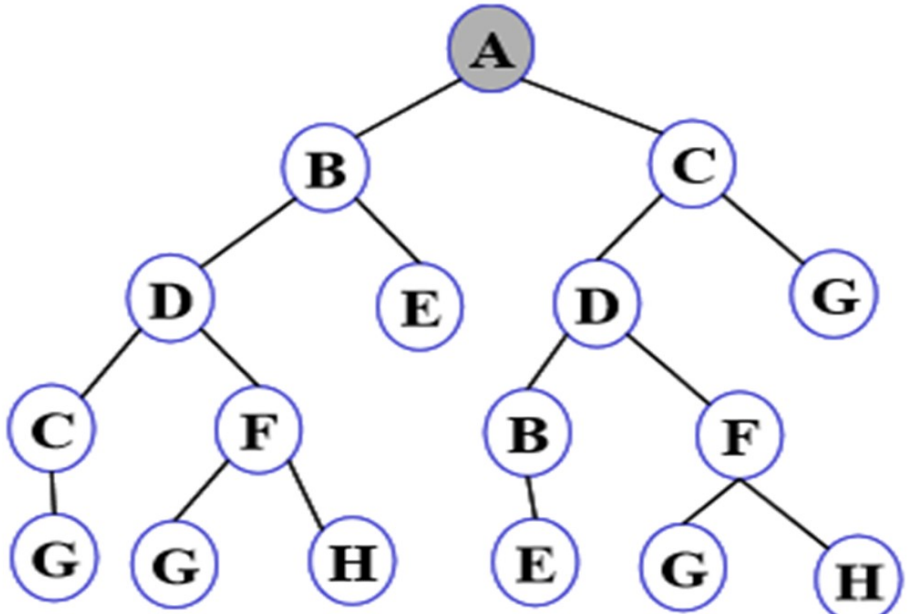
Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



Fringe: DEC

DFS illustrated

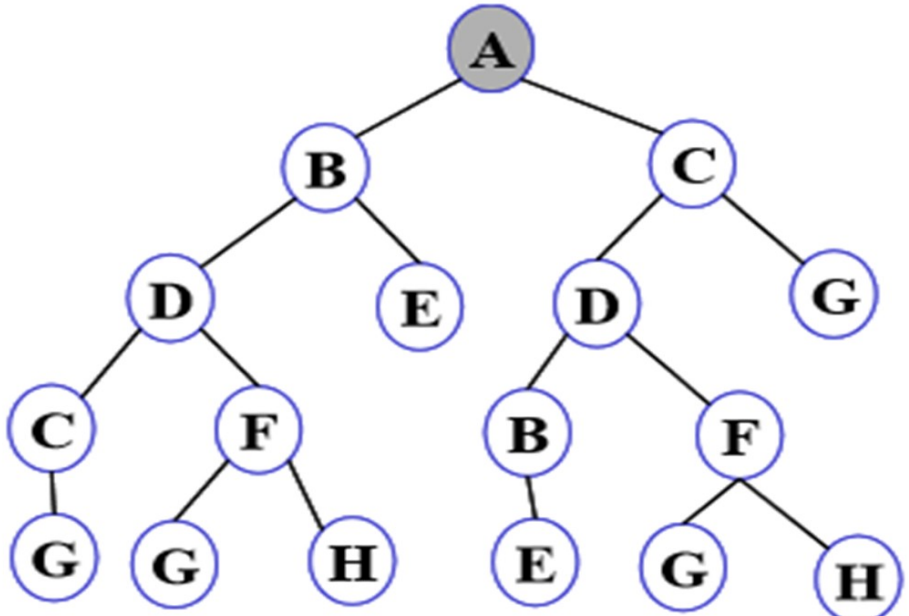
Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



Fringe: CFEC

DFS illustrated

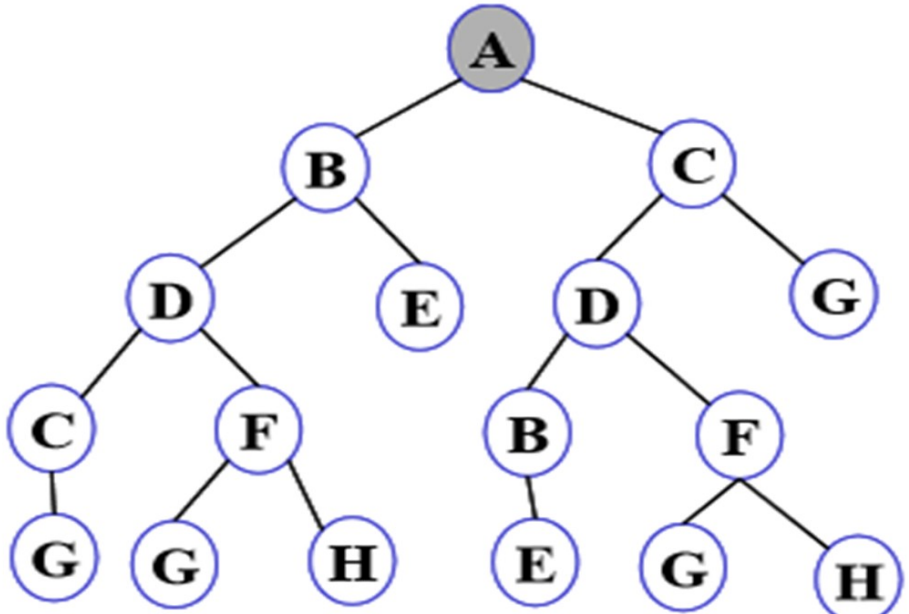
Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



Fringe: GFEC

DFS illustrated

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.

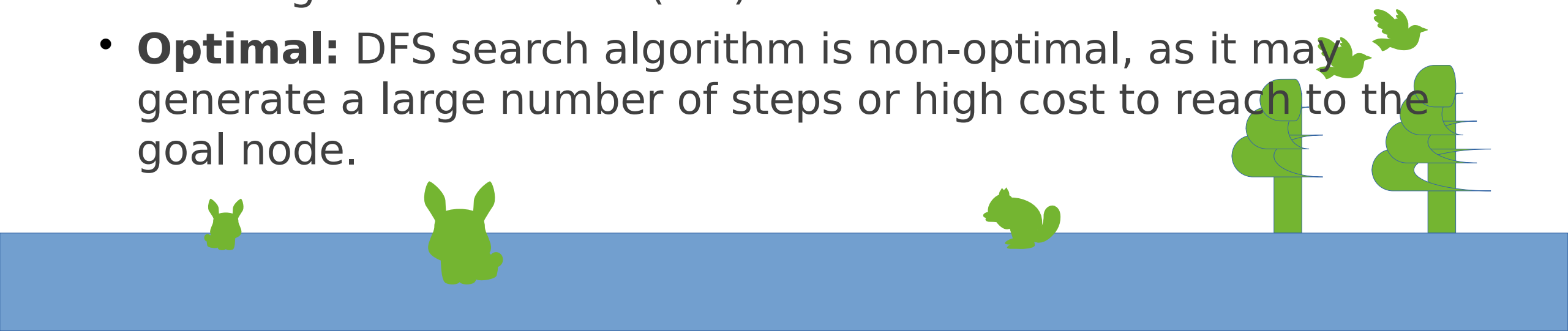


Goal.

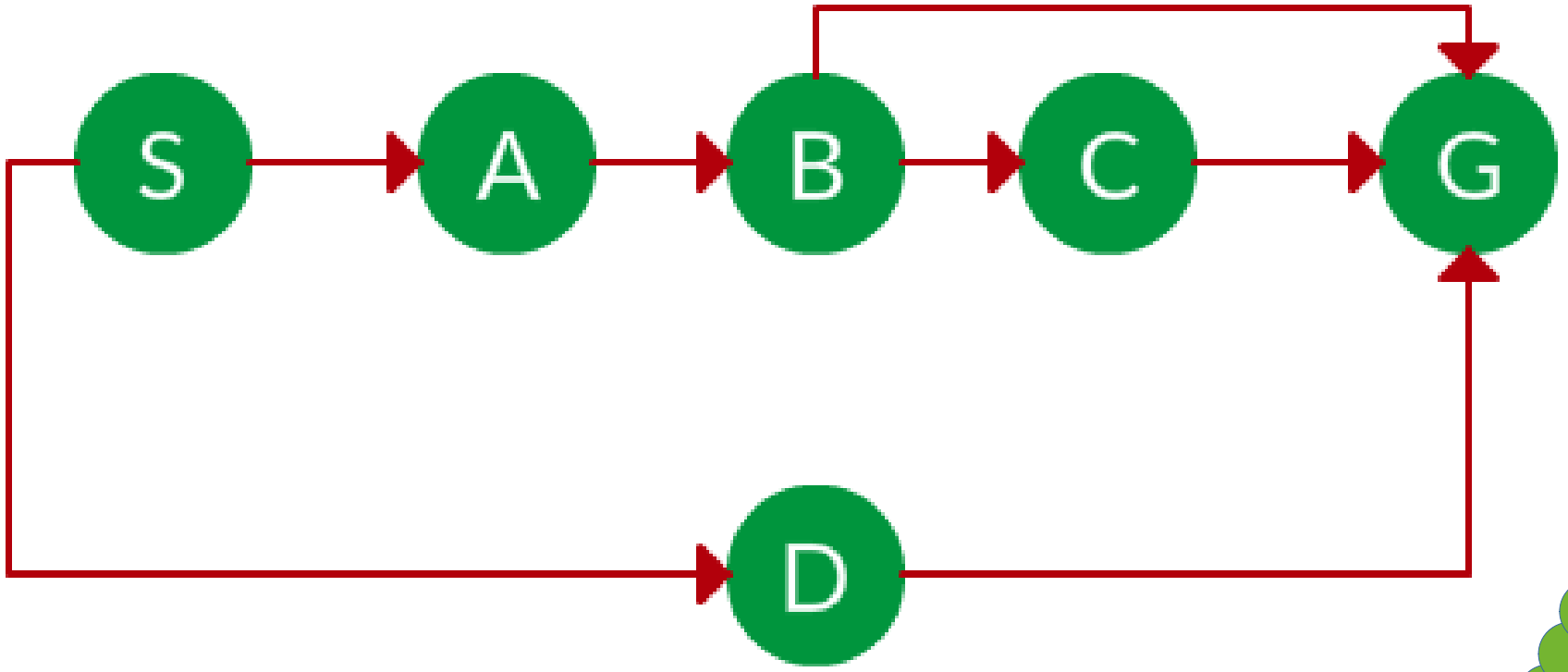
Fringe: GFEC

Cont..

- **Completeness:** No: fails in infinite-depth spaces, spaces with loops.
  - complete in finite spaces
- **Time Complexity:** O(bm): 1 + b + b2 + b3 + ... + bm = O(bm)
  - bad if m is much larger than d
  - but if solutions are dense, may be much faster than BFS.
- **Space Complexity:** DFS algorithm 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.

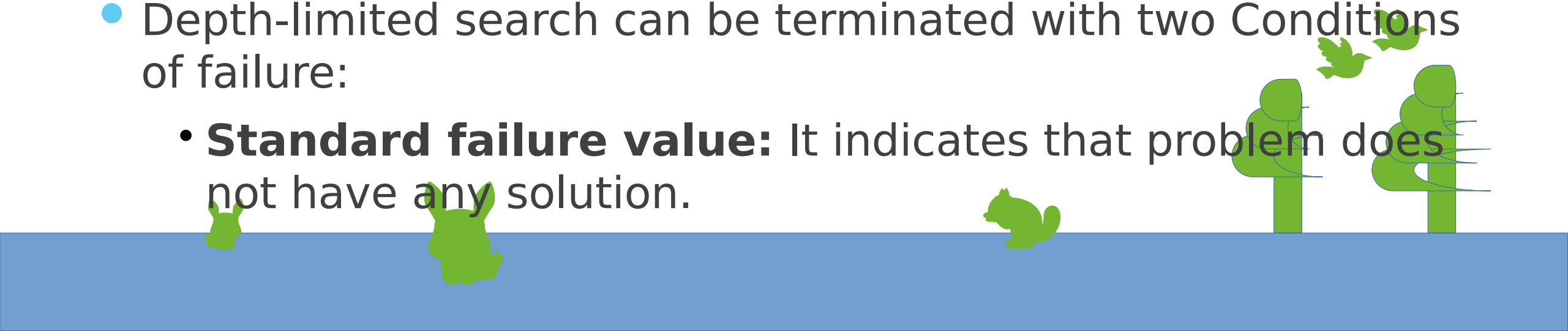


Exercise: Consider the following graph representing the state space and operators of a navigation problem: What is the order that BFS and DFS will expand the nodes



Depth-Limited Search Algorithm

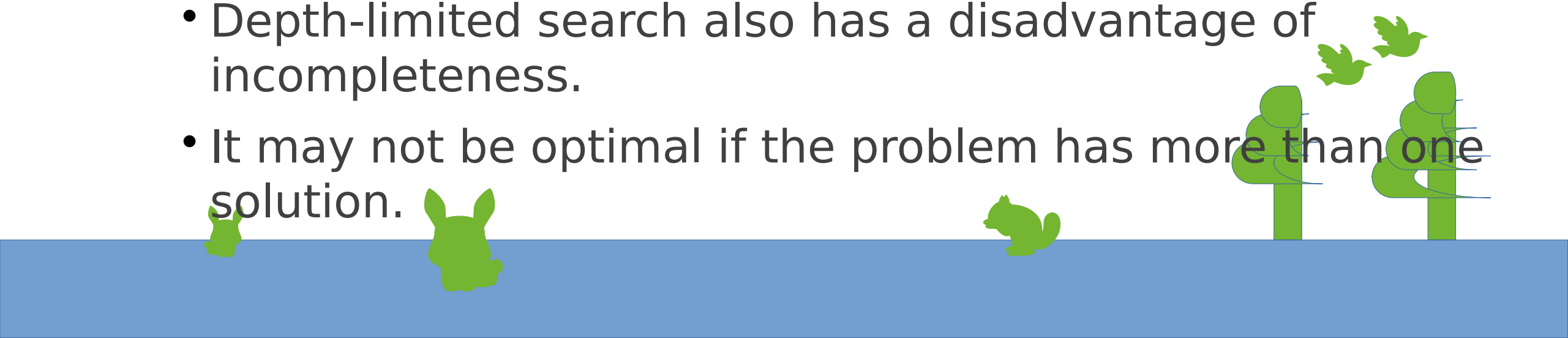
- A depth-limited search algorithm is similar to depth-first search with a **predetermined limit**.
- Depth-limited search can solve the drawback of the **infinite path** in the Depth-first search.
- In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of failure:
  - **Standard failure value:** It indicates that problem does not have any solution.



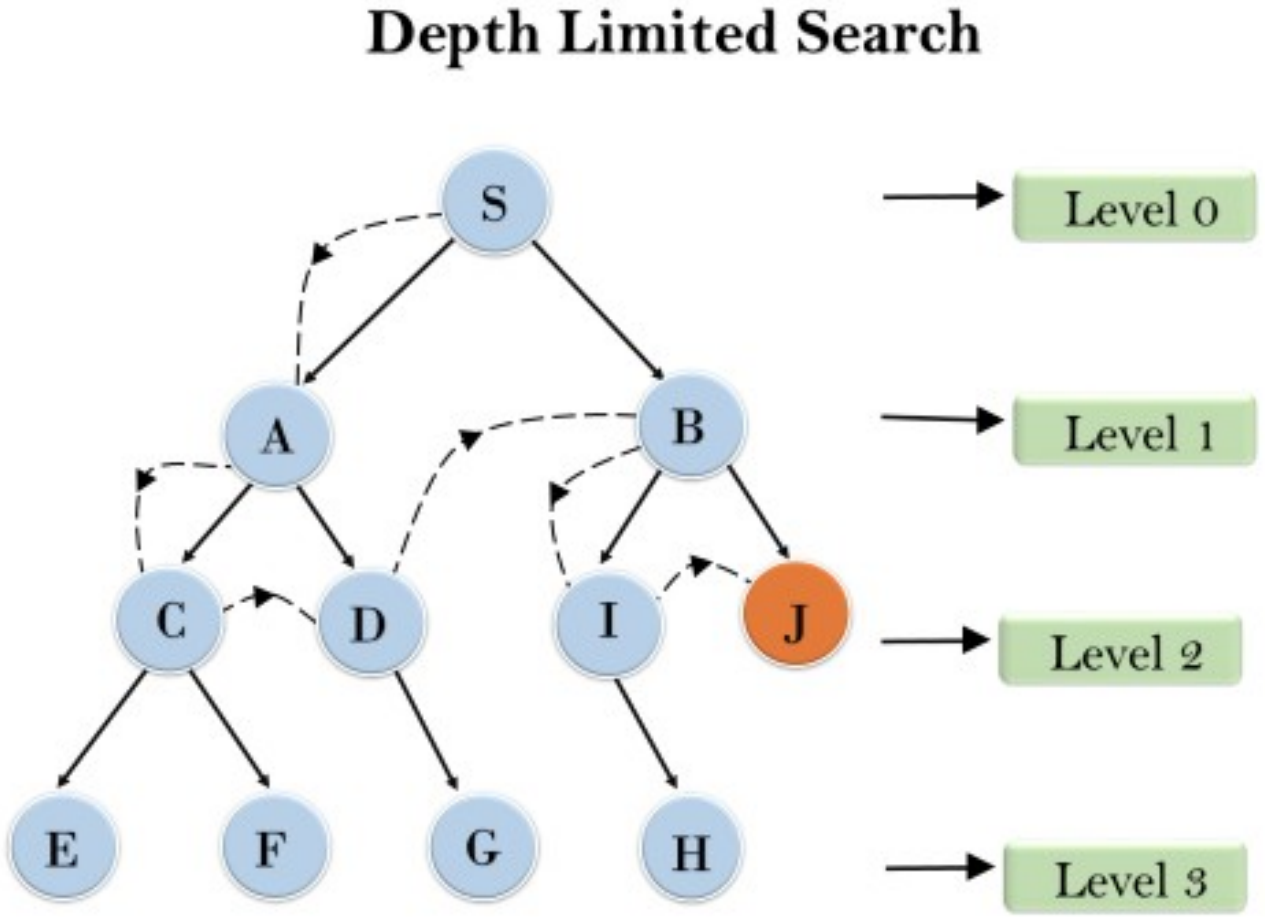
Cont..

- **Cutoff failure value:** It defines no solution for the problem within a given depth limit.

- **Advantages:**
  - Depth-limited search is Memory efficient.
- **Disadvantages:**
  - Depth-limited search also has a disadvantage of incompleteness.
  - It may not be optimal if the problem has more than one solution.

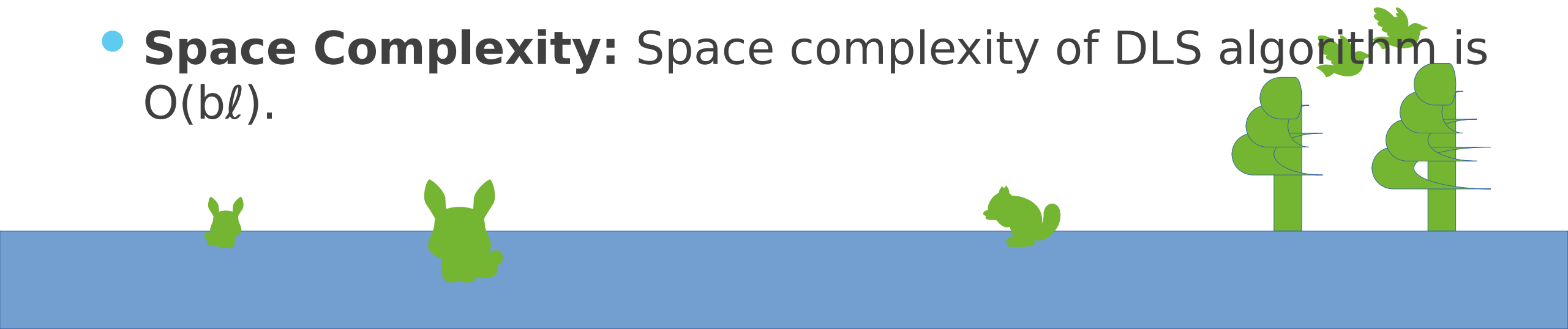


Cont..


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

Cont..

- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal.
- **Time Complexity:** Time complexity of DLS algorithm is $O(b^{\ell})$.
- **Space Complexity:** Space complexity of DLS algorithm is $O(b\ell)$.



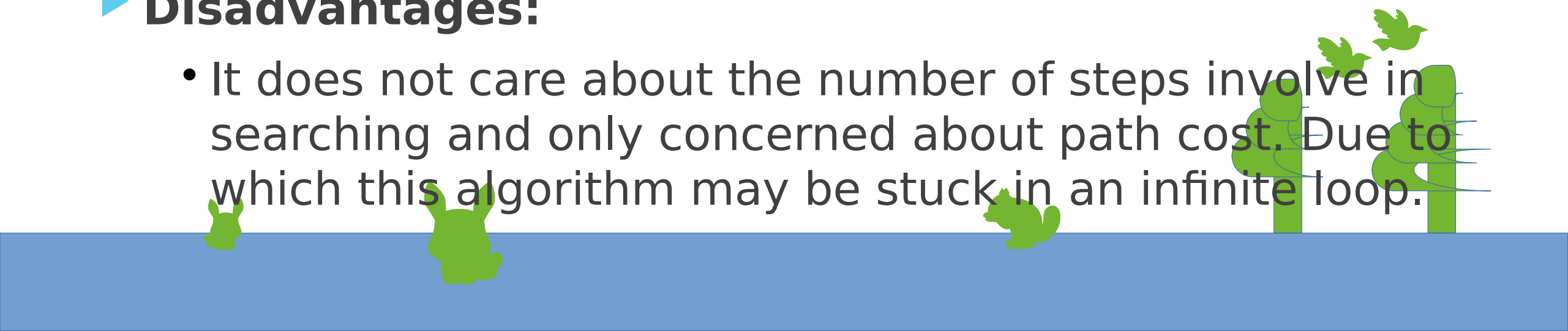
Uniform-cost Search Algorithm

The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.

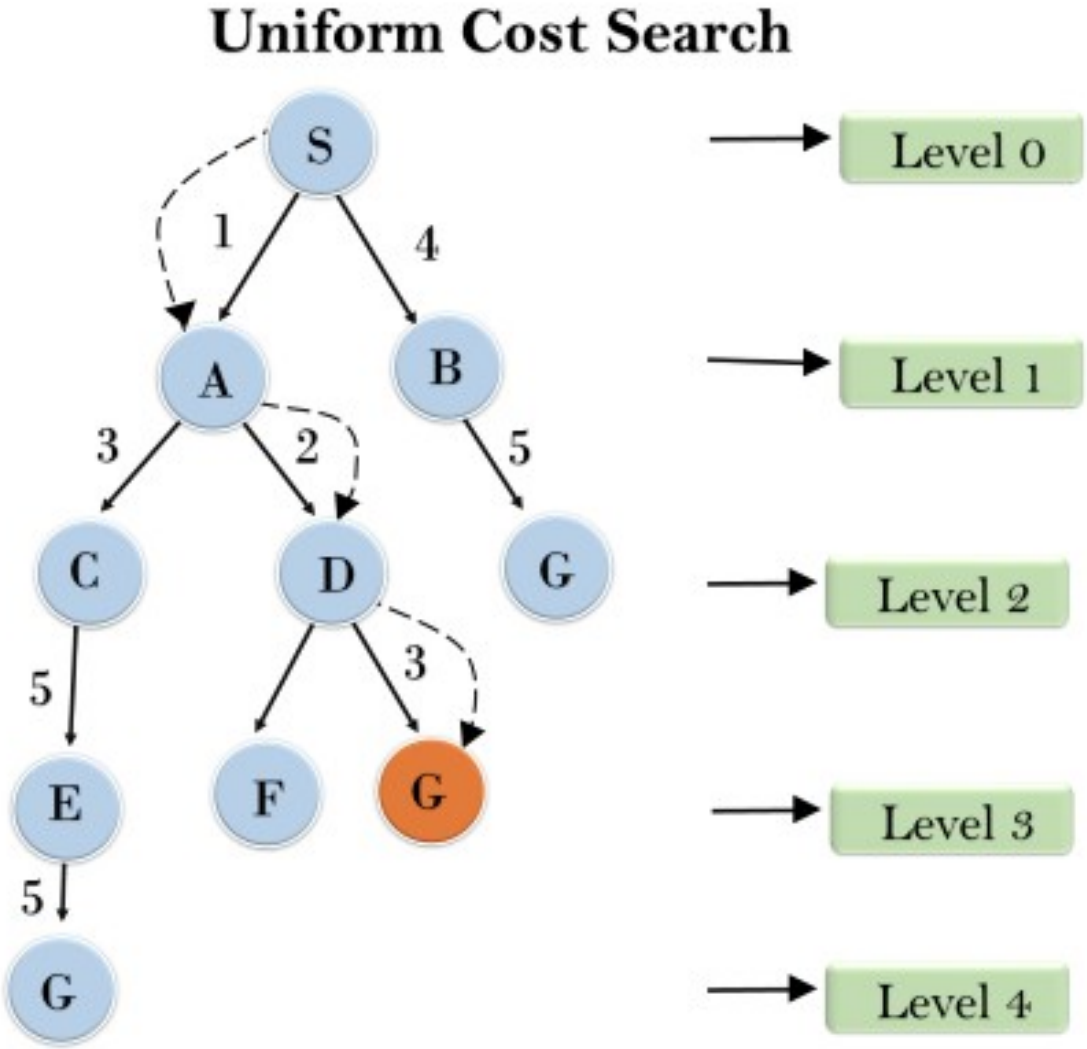
- Uniform-cost search expands nodes according to their path costs form the root node.
- A uniform-cost search algorithm is implemented by the **priority queue**.

Cont..

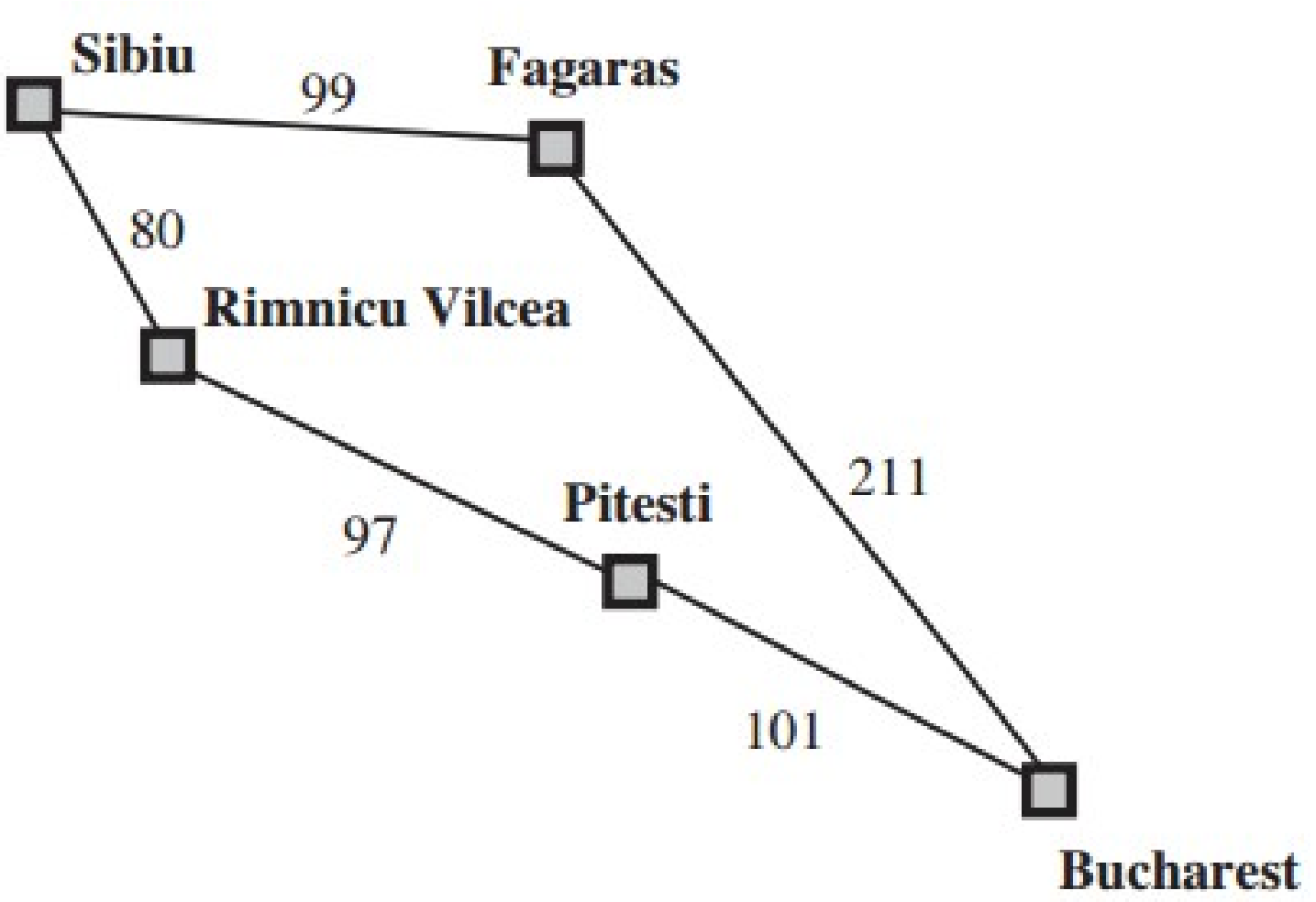
- It gives maximum priority to the lowest cumulative cost.
- Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.
- **Advantages:**
  - Uniform cost search is optimal because at every state the path with the least cost is chosen.
- **Disadvantages:**
  - It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.



Cont..

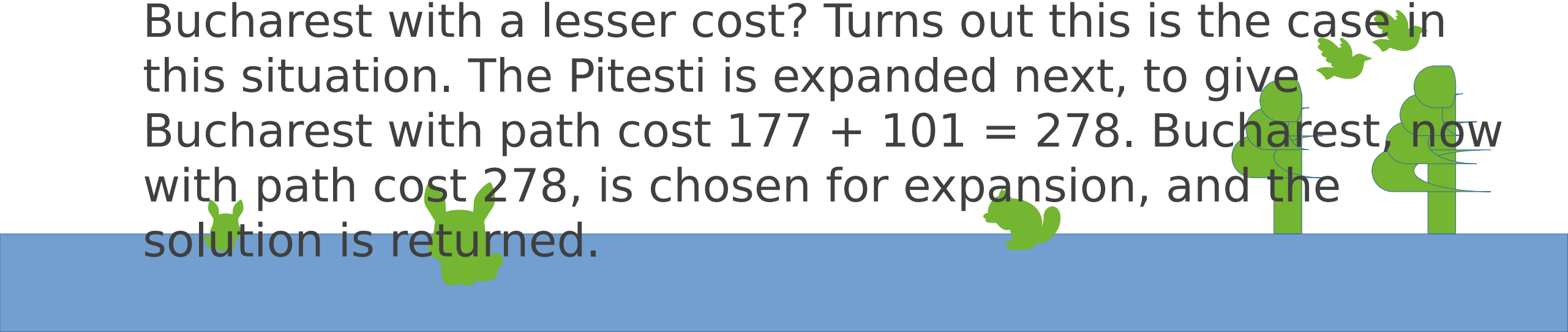

Uniform Cost Search
S
A
B
C
D
G
E
F
G
G
1
4
3
2
5
5
3
5
Level 0
Level 1
Level 2
Level 3
Level 4

Cont..


Sibiu
99
Fagaras
80
Rimnicu Vilcea
211
Pitesti
97
101
Bucharest

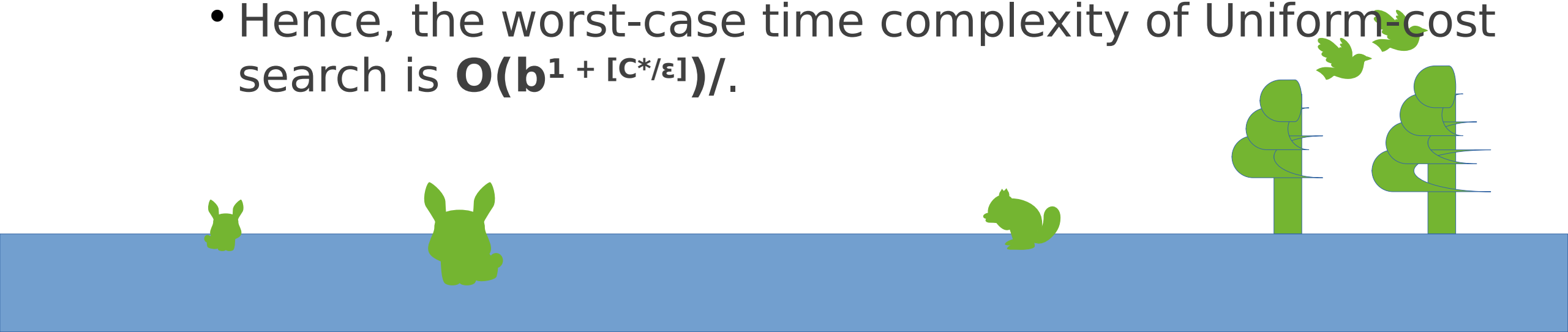
Cont..

- The successors of Sibiu are Riminculus Vilcea (with cost of 80) and Fagaras (cost = 99). The least-cost node is Riminculus Vilcea, which is expanded next to get Pitesti whose path cost from Sibiu is now 80 + 97 =177. The least-cost node is now Fagaras, which is then expanded to get Bucharest with path cost 99 + 211 = 310.
- Now, we have generated the goal node, but the search still continues. What if the path through Pitesti reaches Bucharest with a lesser cost? Turns out this is the case in this situation. The Pitesti is expanded next, to give Bucharest with path cost 177 + 101 = 278. Bucharest, now with path cost 278, is chosen for expansion, and the solution is returned.



Cont..

- **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 $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$/.

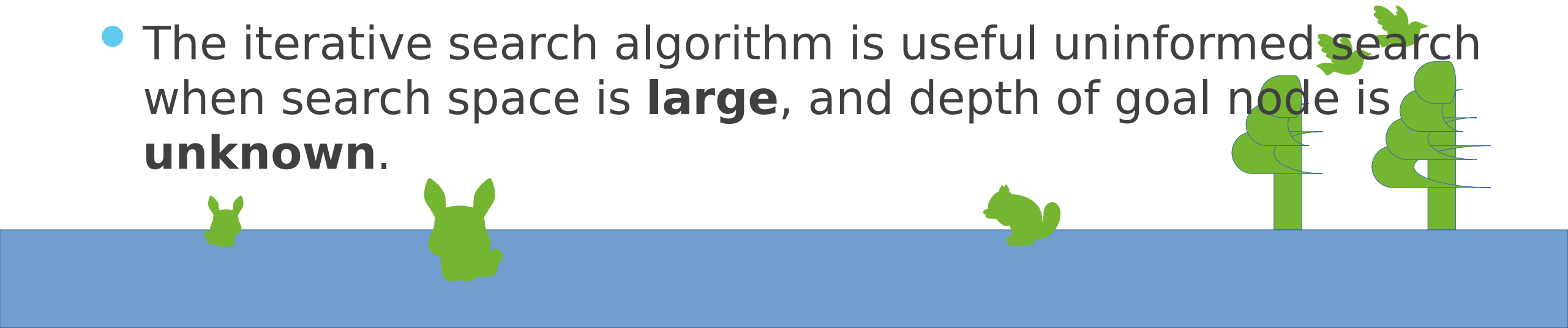


Cont..

- **Space Complexity:** The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.
- **Optimal:** Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Iterative deepening depth-first Search

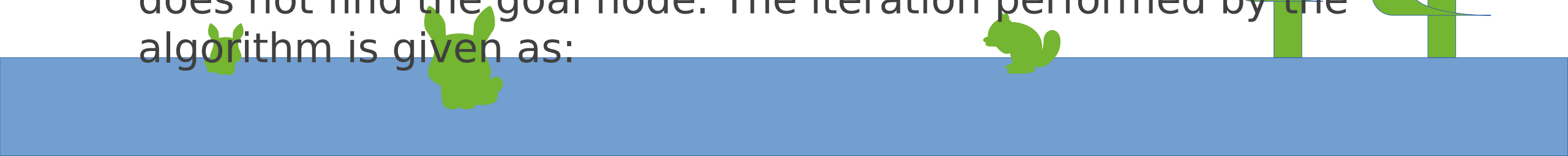
- The iterative deepening algorithm is a combination of DFS and BFS algorithms.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of BFS fast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



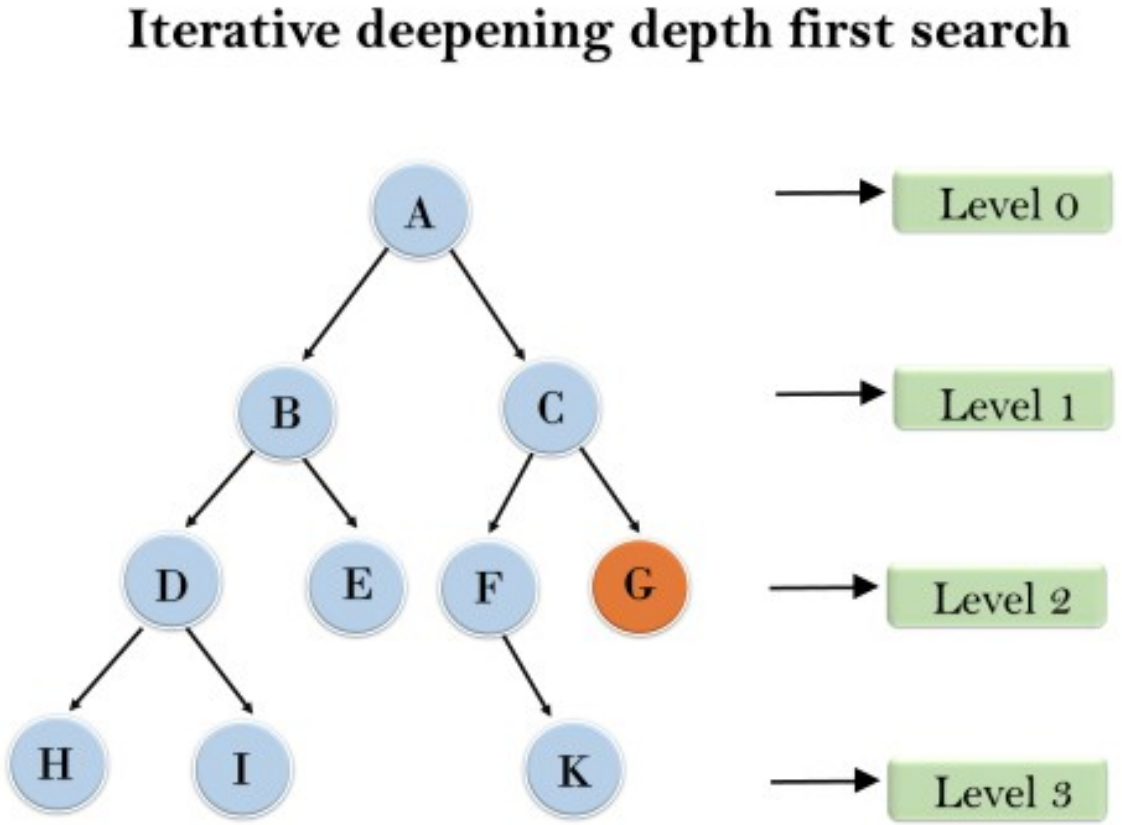
Cont..

- **Advantages:**
  - It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
- **Disadvantages:**
  - The main drawback of IDDFS is that it repeats all the work of the previous phase.
- **Example:**

- Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



Cont..


Iterative deepening depth first search
A
B
C
D
E
F
G
H
I
K
Level 0
Level 1
Level 2
Level 3

Cont..

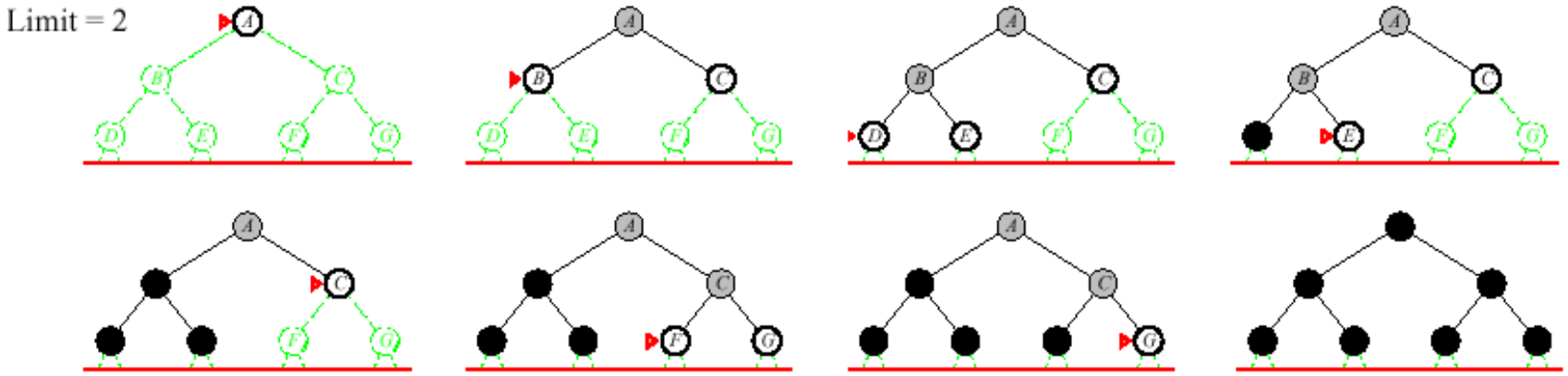
- 1'st Iteration-----> A
- 2'nd Iteration----> A, B, C
- 3'rd Iteration------>A, B, D, E, C, F, G
- 4'th Iteration------>A, B, D, H, I, E, C, F, K, G
  In the fourth iteration, the algorithm will find the goal node.

Iterative deepening search $l = 1$

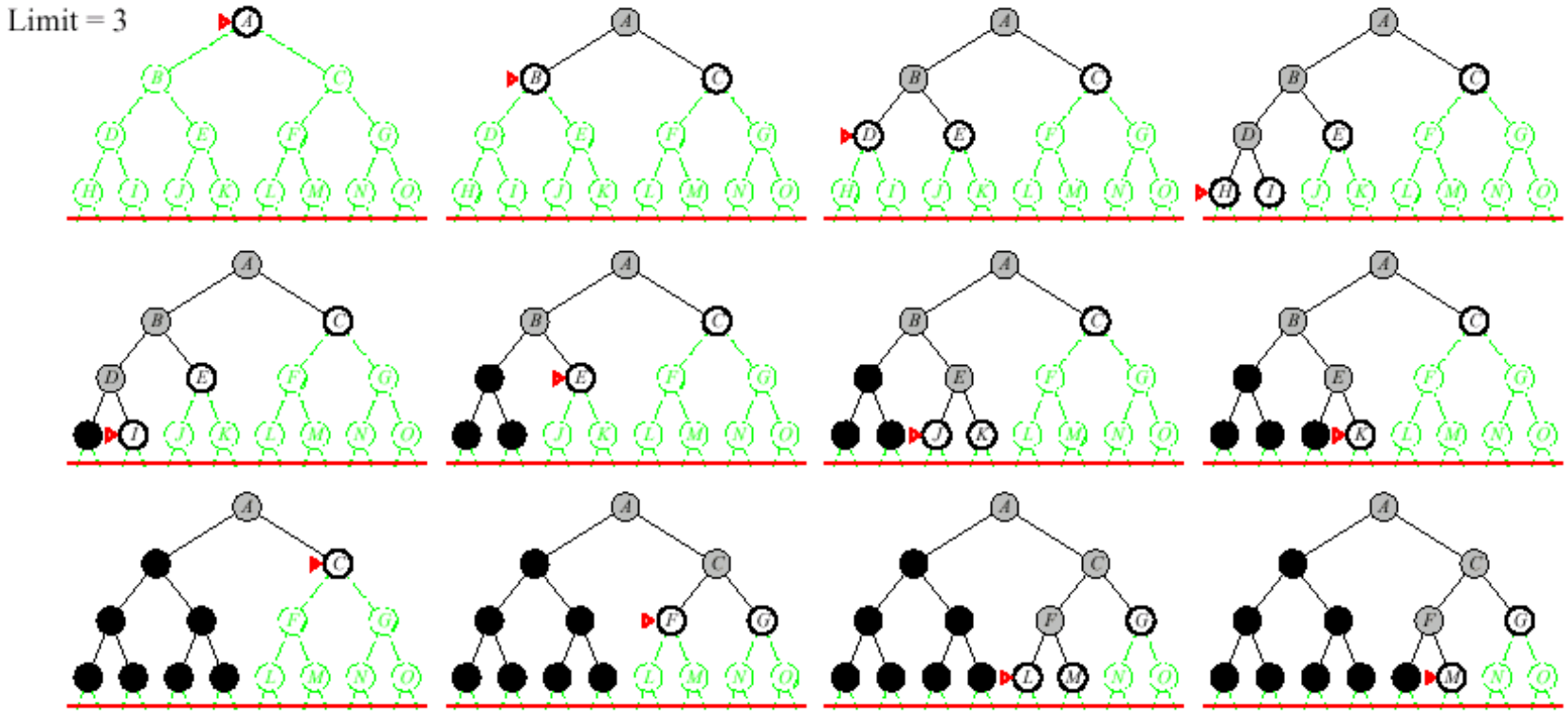
Limit = 1

Iterative deepening search $l = 2$

Limit = 2

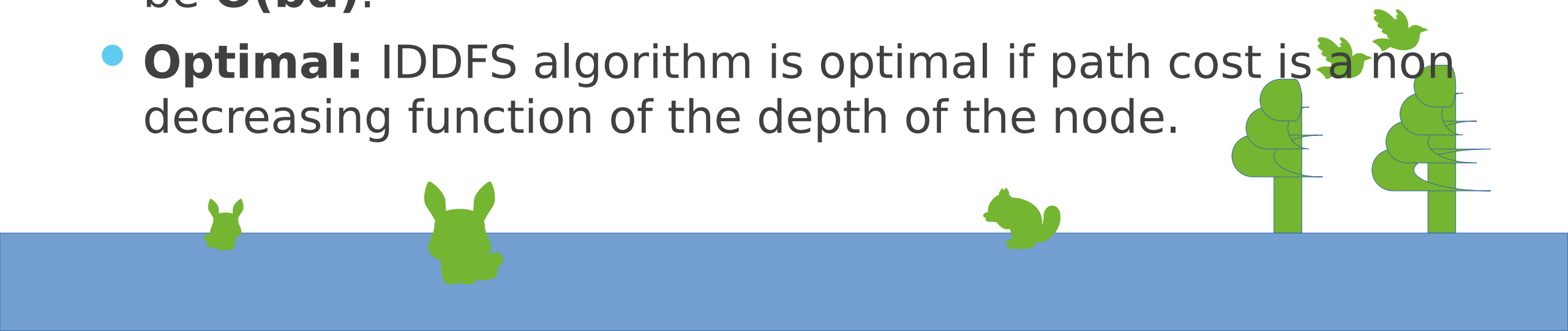


Iterative deepening search $l = 3$



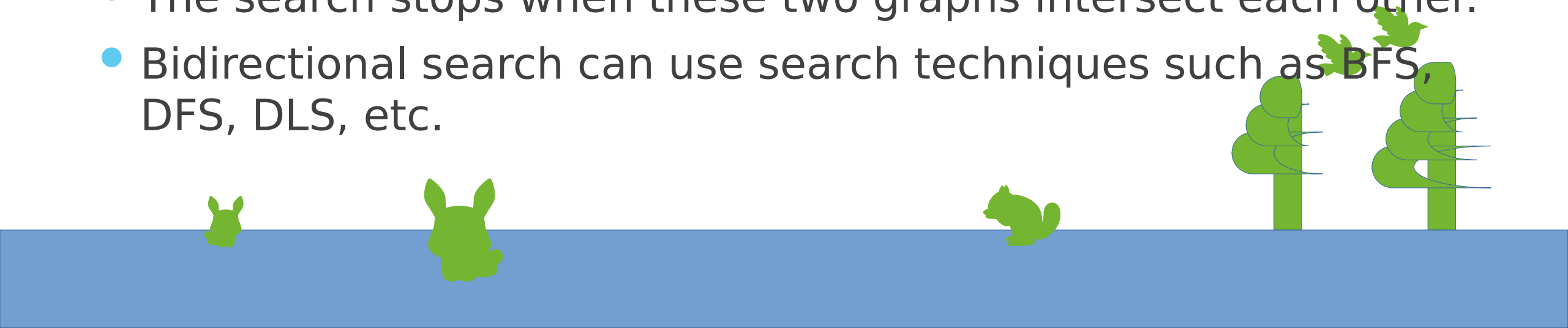
Cont..

- **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

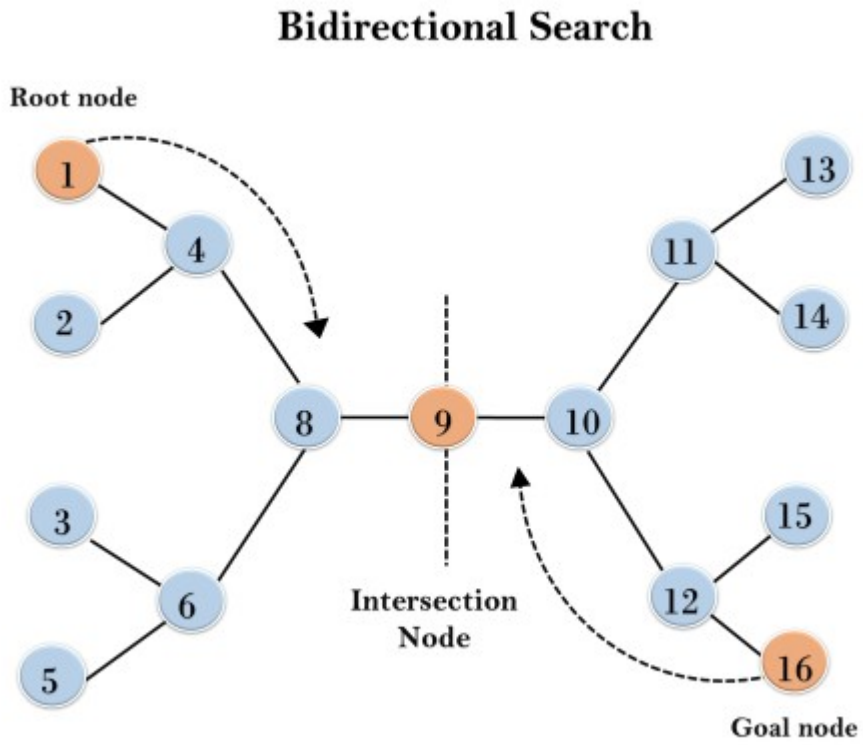
- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from 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.



Cont..

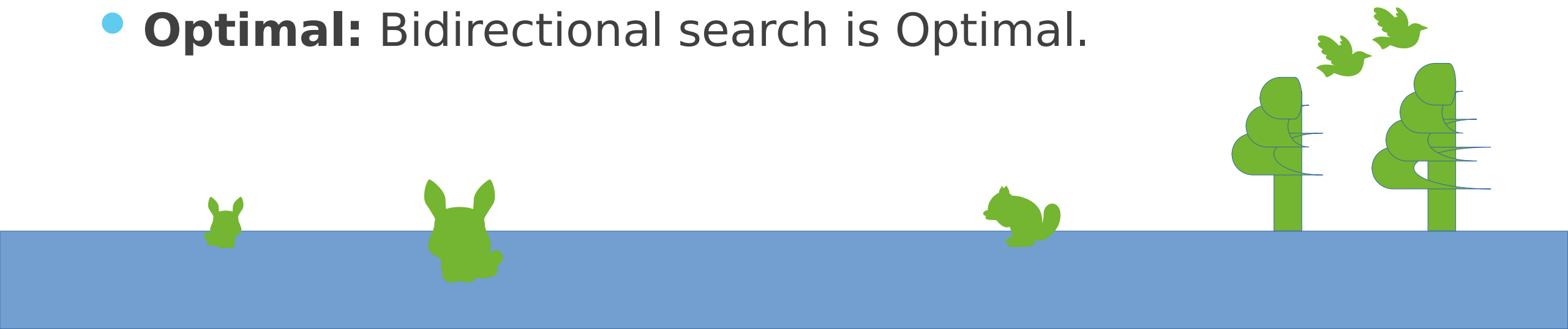
- **Advantages:**
  - Bidirectional search is fast.
  - Bidirectional search requires less memory
- **Disadvantages:**
  - Implementation of the bidirectional search tree is difficult.

Example:
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.



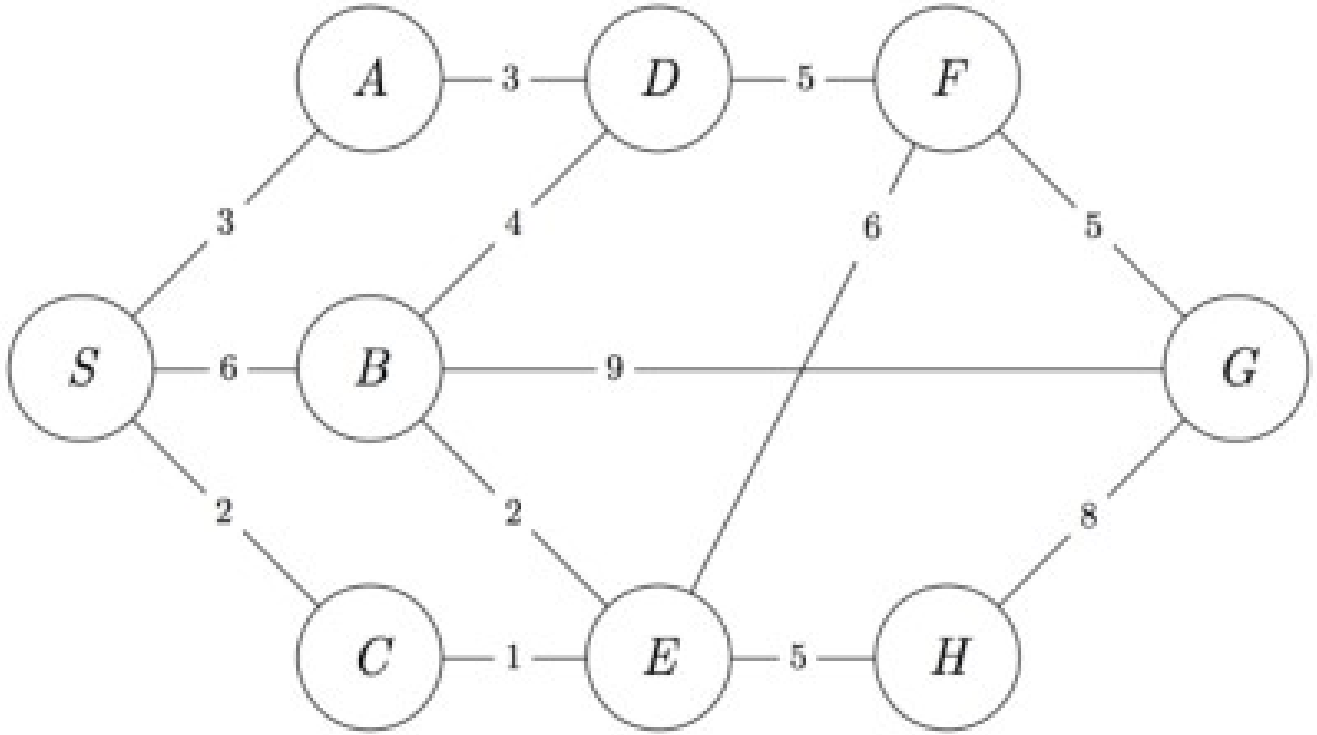
Cont..

- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is **$O(b^{d/2})$**.
- **Space Complexity:** Space complexity of bidirectional search is **$O(b^{d/2})$**.
- **Optimal:** Bidirectional search is Optimal.



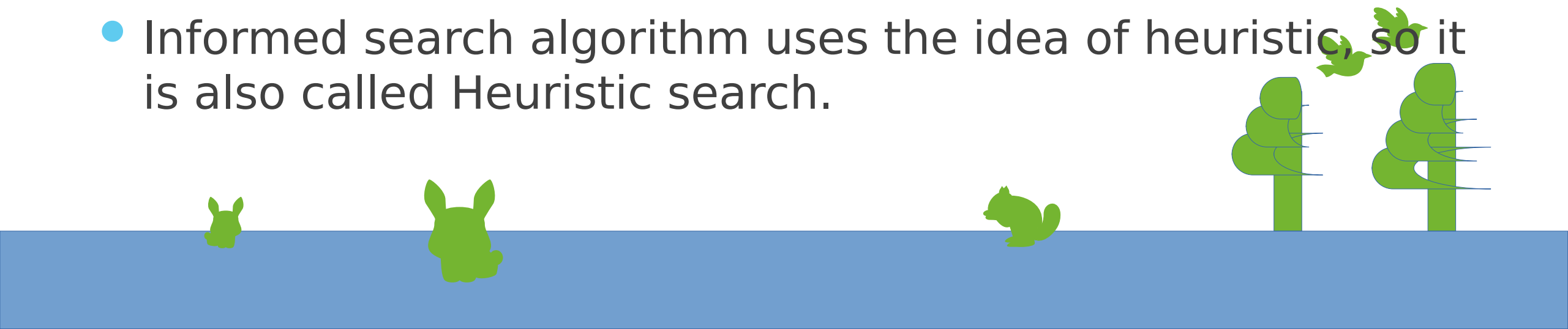
Exercise

Question: What is the order of visits of the nodes and the path returned by BFS, DFS and UCS?



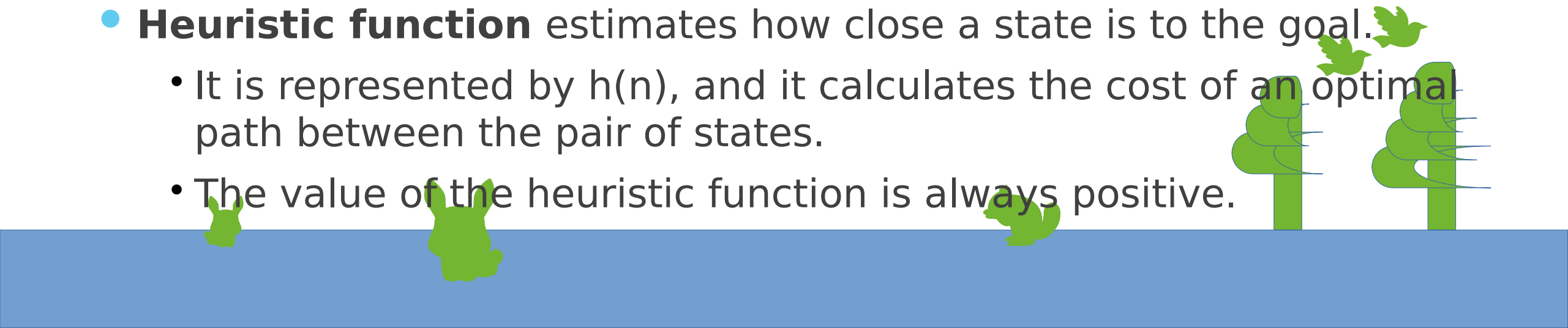
Informed Search Algorithms

- But **informed search** algorithm contains an array of **knowledge** such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This **knowledge** help agents to explore less to the search space and find more efficiently the goal node.
- The **informed search** algorithm is more useful for large search space.
- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.



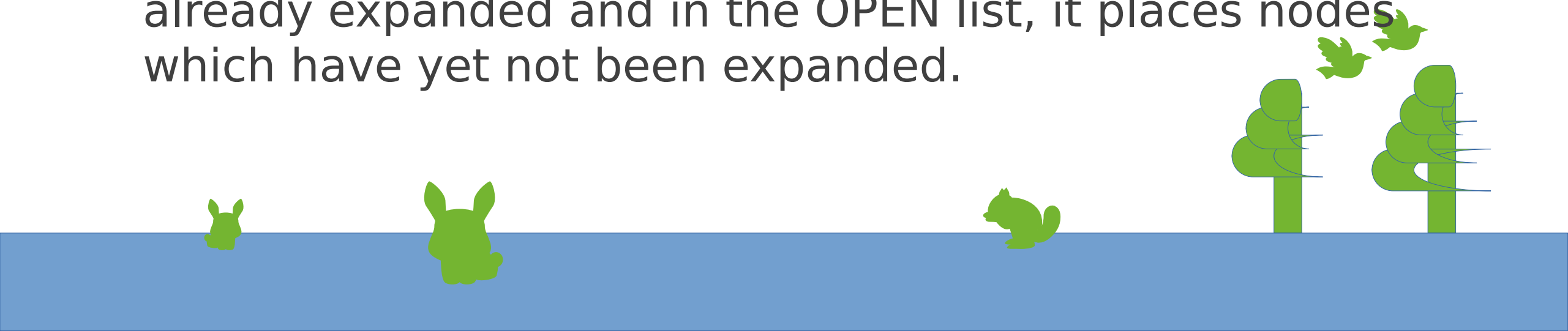
Cont..

- **Heuristics function:** Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the **current state** of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the **best solution**, but it guaranteed to find a good solution in reasonable time.
- **Heuristic function** estimates how close a state is to the goal.
  - It is represented by h(n), and it calculates the cost of an optimal path between the pair of states.
  - The value of the heuristic function is always positive.



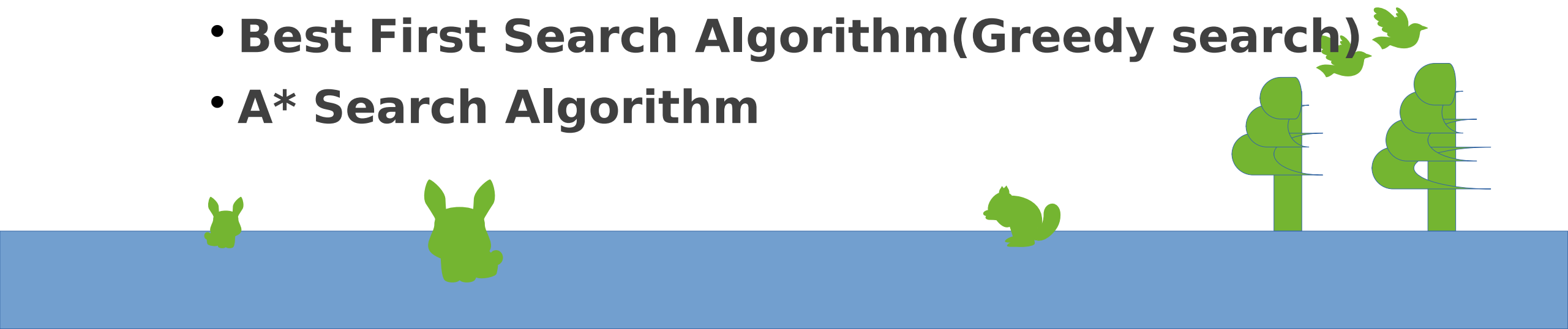
Pure Heuristic Search

- Pure heuristic search is the simplest form of heuristic search algorithms.
- It expands nodes based on their heuristic value h(n).
- It maintains two lists, **OPEN** and **CLOSED** list.
- In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.



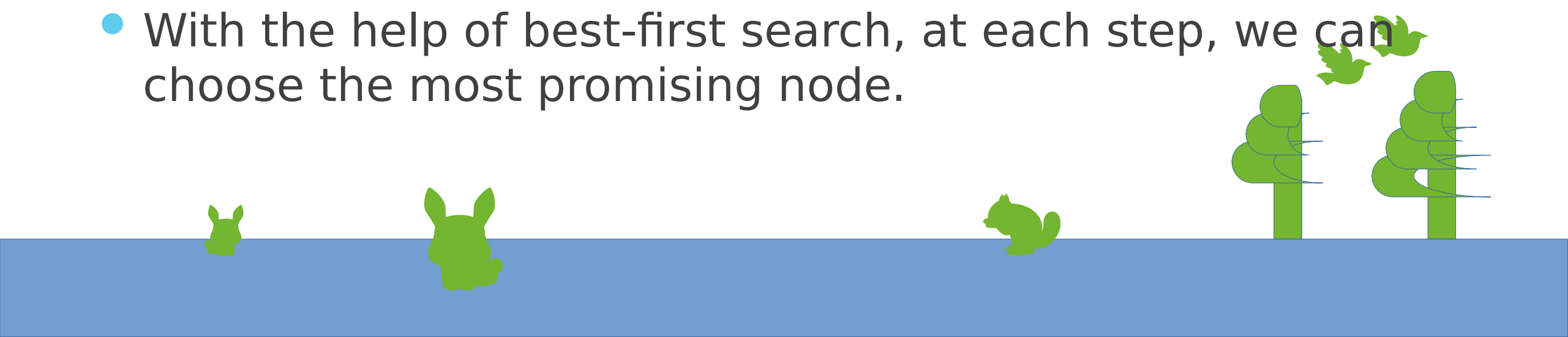
Cont..

- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list.
- The algorithm continues unit a goal state is found.
- In the informed search we will discuss two main algorithms which are given below:
  - **Best First Search Algorithm(Greedy search)**
  - **A* Search Algorithm**



Best-first Search Algorithm (Greedy Search)

- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms.
- It uses the heuristic function and search. Best-first search allows us to **take the advantages** of **both algorithms**.
- With the help of best-first search, at each step, we can choose the most promising node.



Cont..

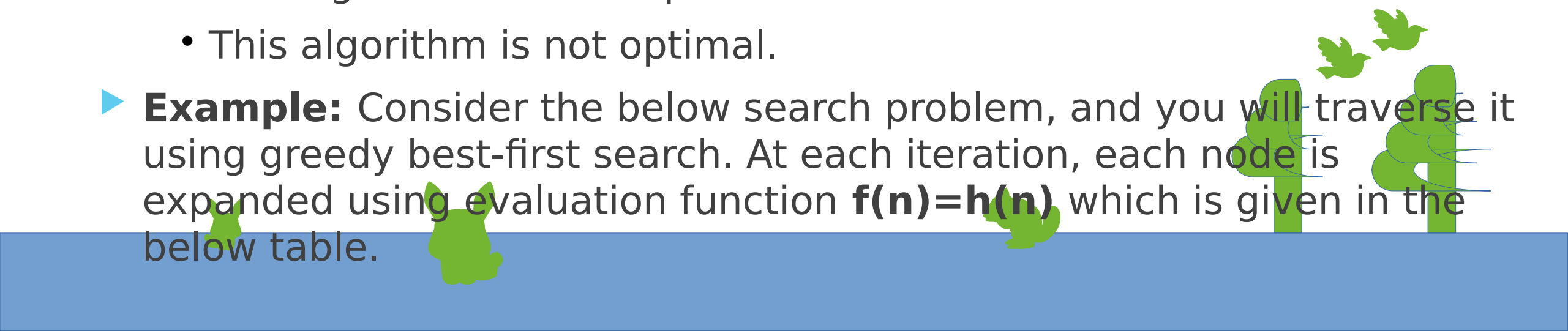
- **In** the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function.
- The greedy best first algorithm is implemented by the priority queue.

Cont..

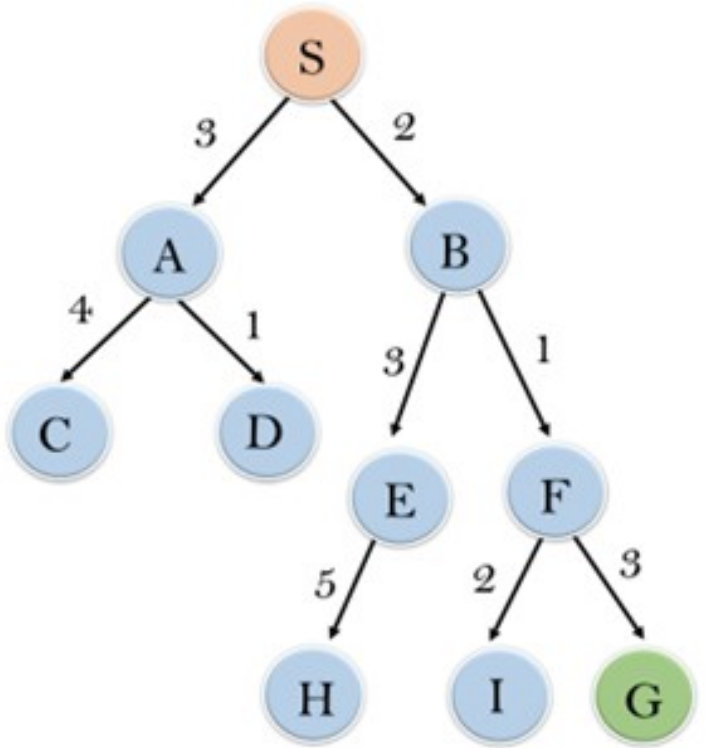
- **Best first search algorithm:**
  - **Step 1:** Place the starting node into the OPEN list.
  - **Step 2:** If the OPEN list is empty, Stop and return failure.
  - **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
  - **Step 4:** Expand the node n, and generate the successors of node n.
  - **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
  - **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
  - **Step 7:** Return to Step 2.

Cont..

- **Advantages:**
  - Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
  - This algorithm is more efficient than BFS and DFS algorithms.
- **Disadvantages:**
  - It can behave as an unguided depth-first search in the worst case scenario.
  - It can get stuck in a loop as DFS.
  - This algorithm is not optimal.
- **Example:** Consider the below search problem, and you will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function **f(n)=h(n)** which is given in the below table.

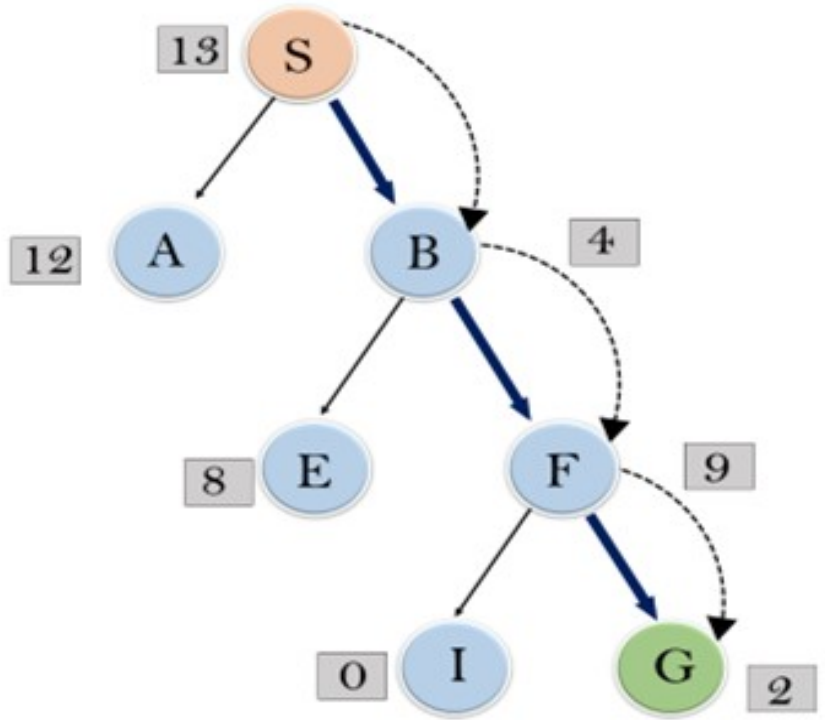


Cont..



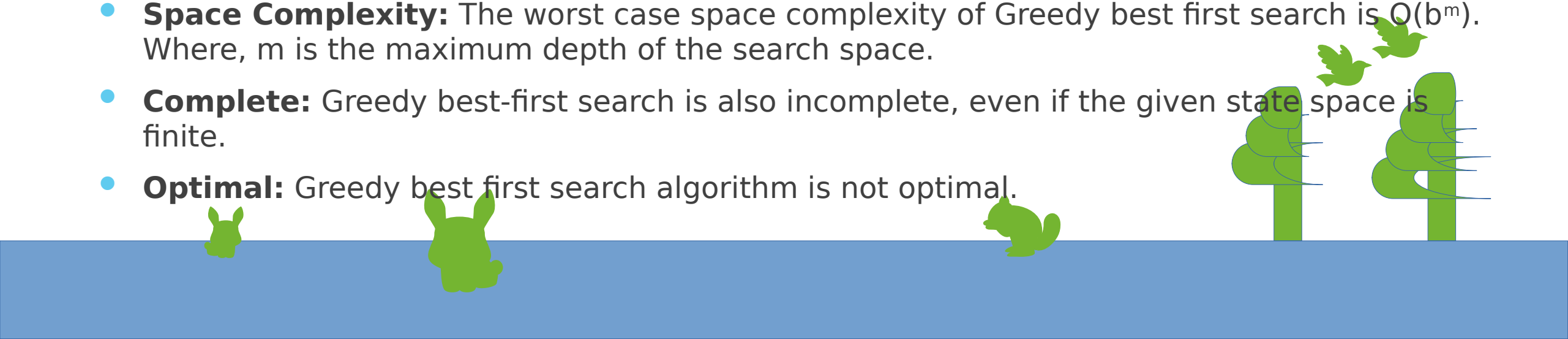
node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



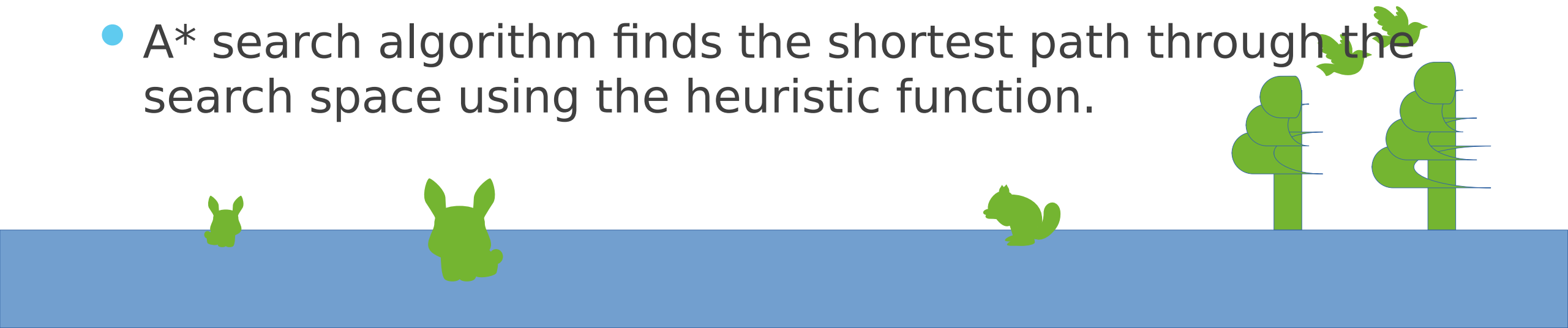
Cont..

- **Expand the nodes of S and put in the CLOSED list**
  - **Initialization:** Open [A, B], Closed [S]
  - **Iteration 1:** Open [A], Closed [S, B]
  - **Iteration 2:** Open [E, F, A], Closed [S, B]
    : Open [E, A], Closed [S, B, F]
  - **Iteration 3:** Open [I, G, E, A], Closed [S, B, F]
    : Open [I, E, A], Closed [S, B, F, G]
  - Hence the final solution path will be: **S----> B----->F----> G**
- **Time Complexity:** The worst case time complexity of Greedy best first search is $O(b^m)$.
- **Space Complexity:** The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.
- **Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.
- **Optimal:** Greedy best first search algorithm is not optimal.



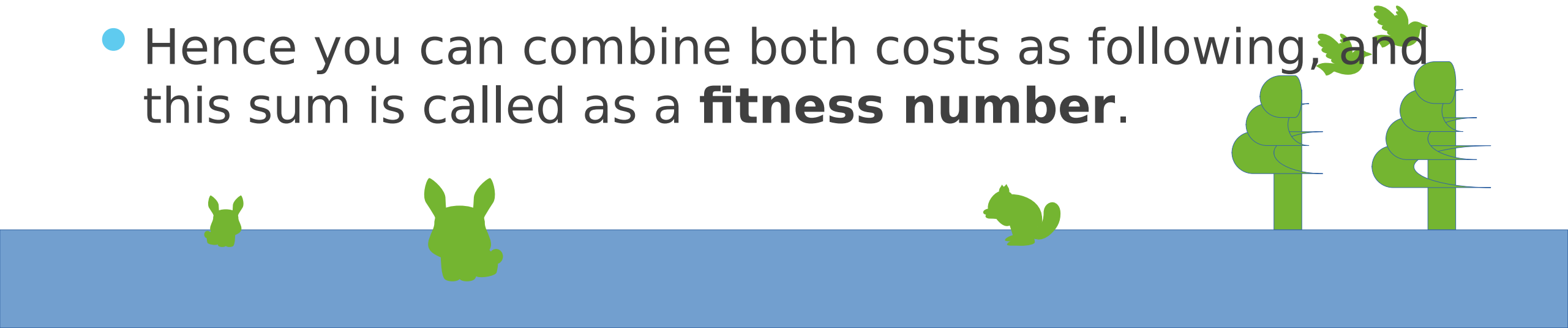
A* Search Algorithm

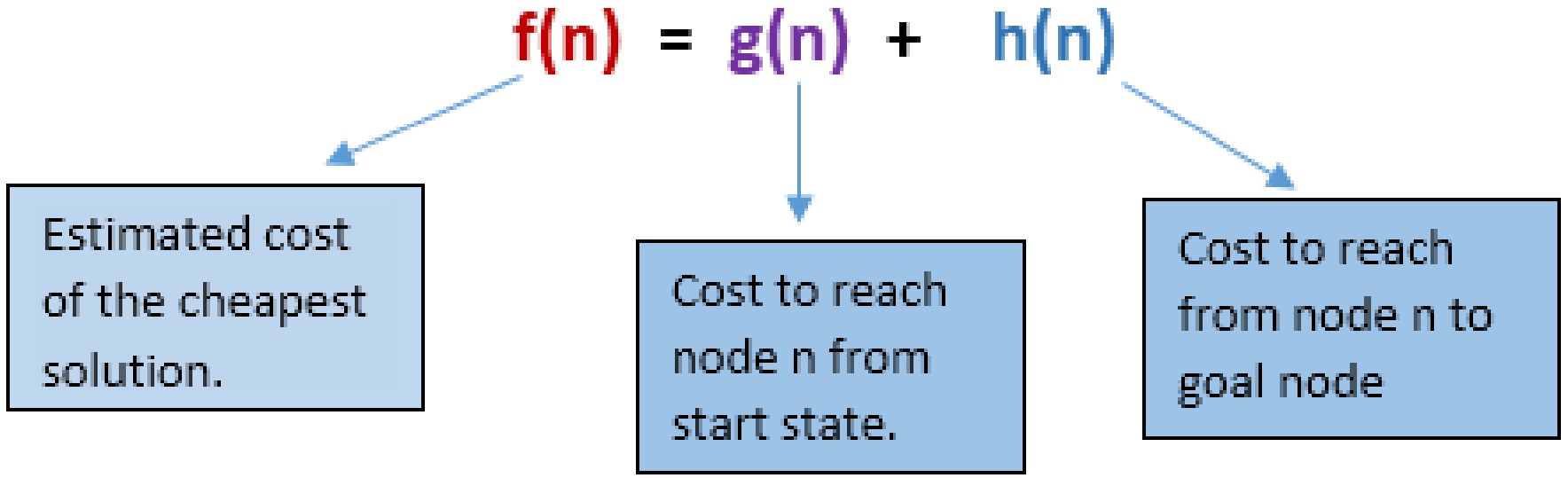
- A* search is the most commonly known form of best-first search.
- It uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- It has combined features of **UCS** and greedy **best-first search**, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function.



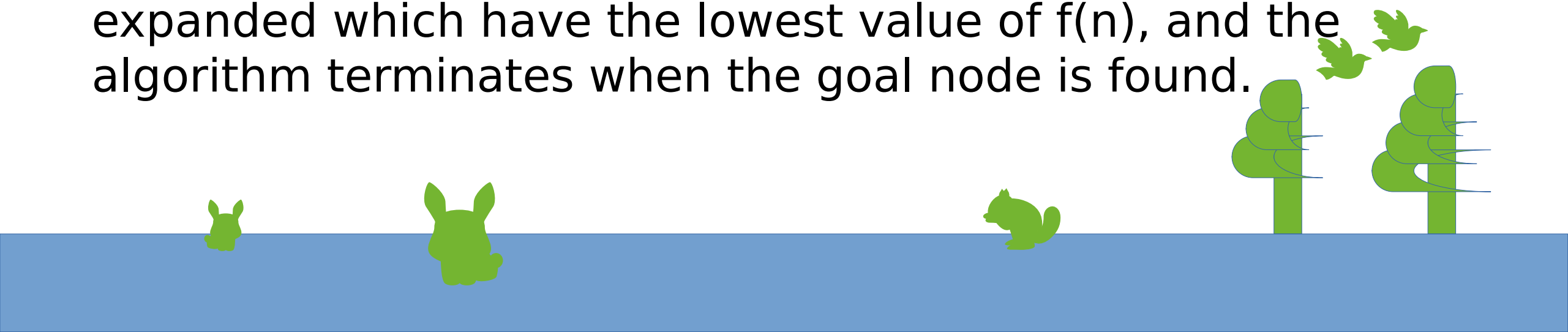
Cont..

- This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses g(n)+ h(n) instead of g(n).
- In A* search algorithm, you use search heuristic as well as the cost to reach the node.
- Hence you can combine both costs as following, and this sum is called as a **fitness number**.



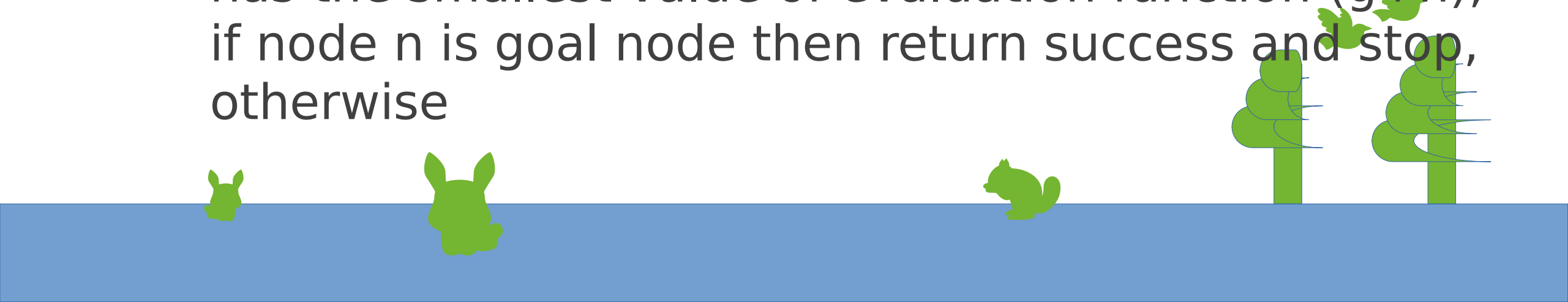


At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.



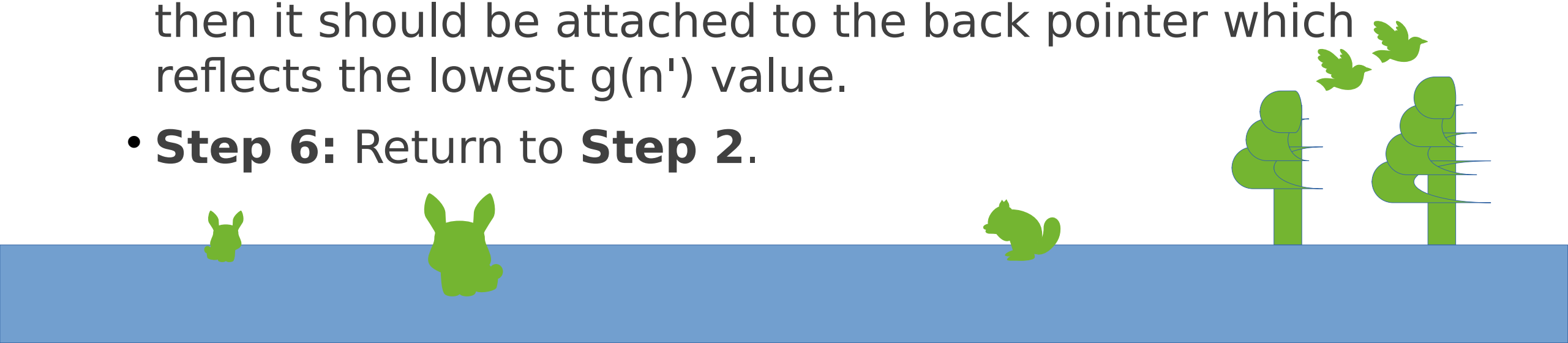
Cont..

- Algorithm of A* search:
  - **Step1:** Place the starting node in the OPEN list.
  - **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
  - **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise



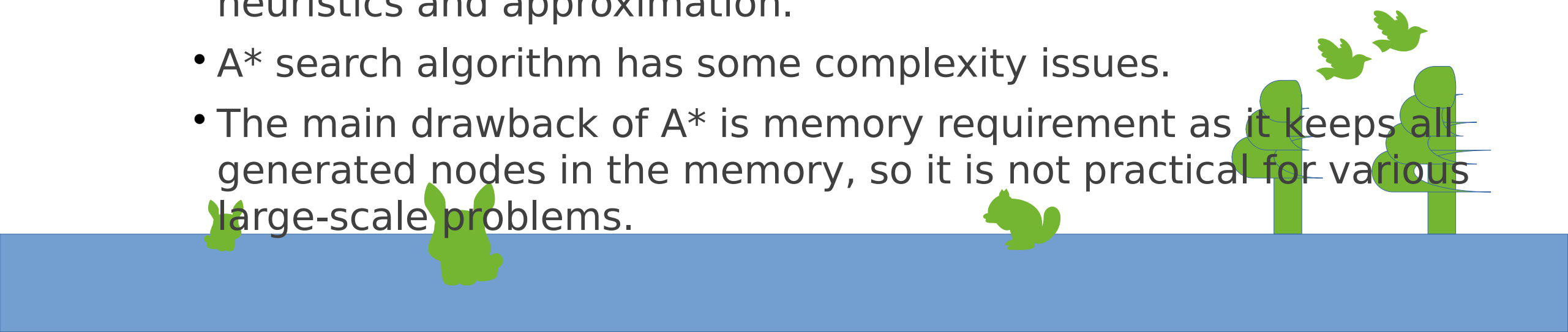
Cont..

- **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- **Step 6:** Return to **Step 2**.

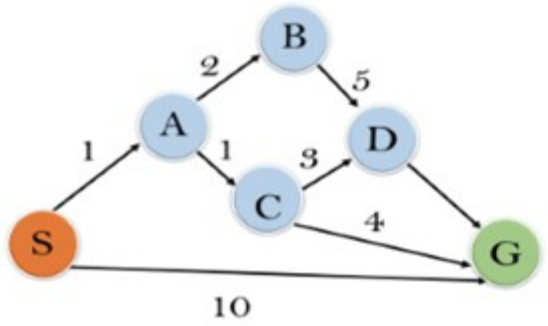


Cont..

- **Advantages:**
  - A* search algorithm is the best algorithm than other search algorithms.
  - A* search algorithm is optimal and complete.
  - This algorithm can solve very complex problems.
- **Disadvantages:**
  - It does not always produce the shortest path as it mostly based on heuristics and approximation.
  - A* search algorithm has some complexity issues.
  - The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

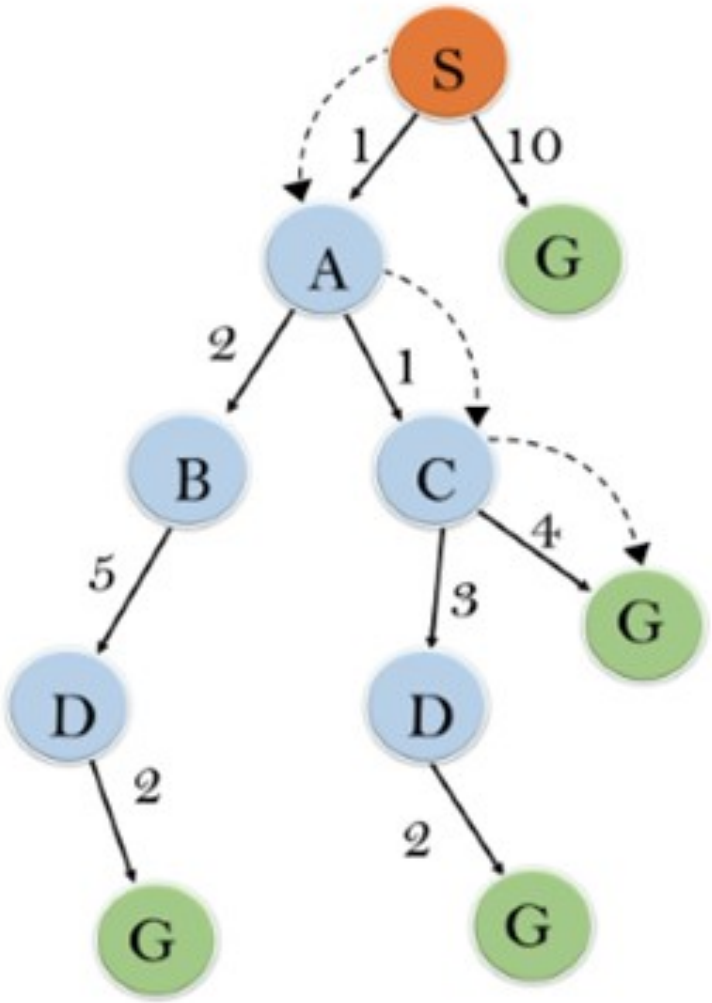


Example: In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state. Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

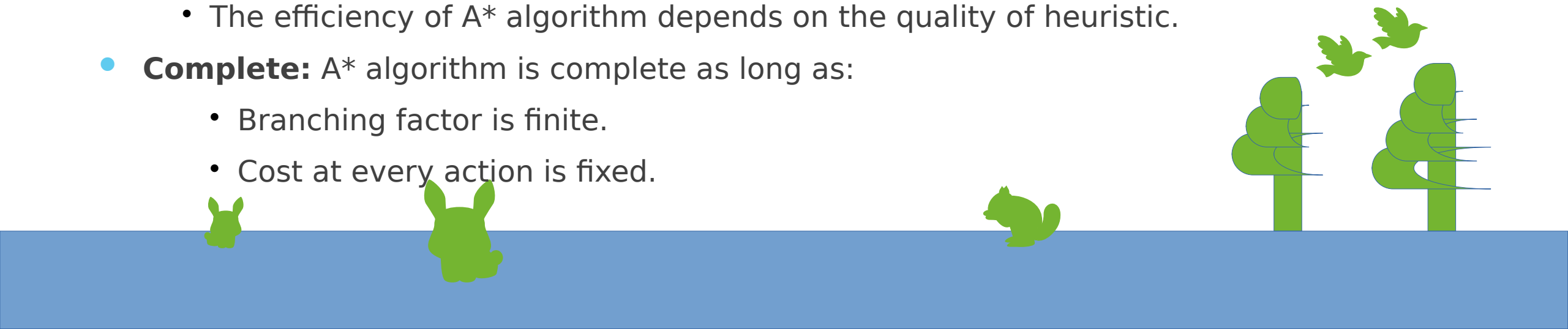
solution



Cont..

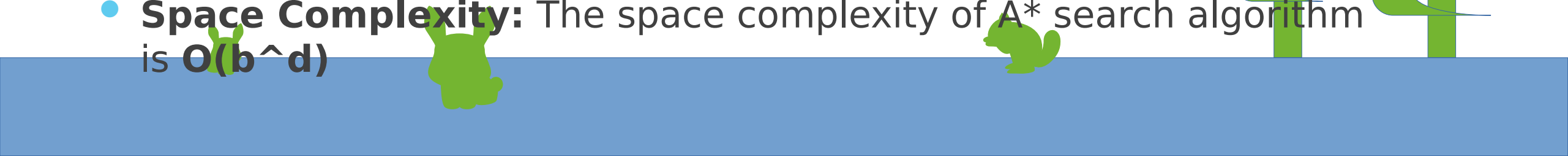
- **Initialization:** {(S, 5)}
- **Iteration1:** {(S--> A, 4), (S-->G, 10)}
- **Iteration2:** {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration3:** {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration 4** will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

- **Points to remember:**
  - A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
  - The efficiency of A* algorithm depends on the quality of heuristic.
- **Complete:** A* algorithm is complete as long as:
  - Branching factor is finite.
  - Cost at every action is fixed.

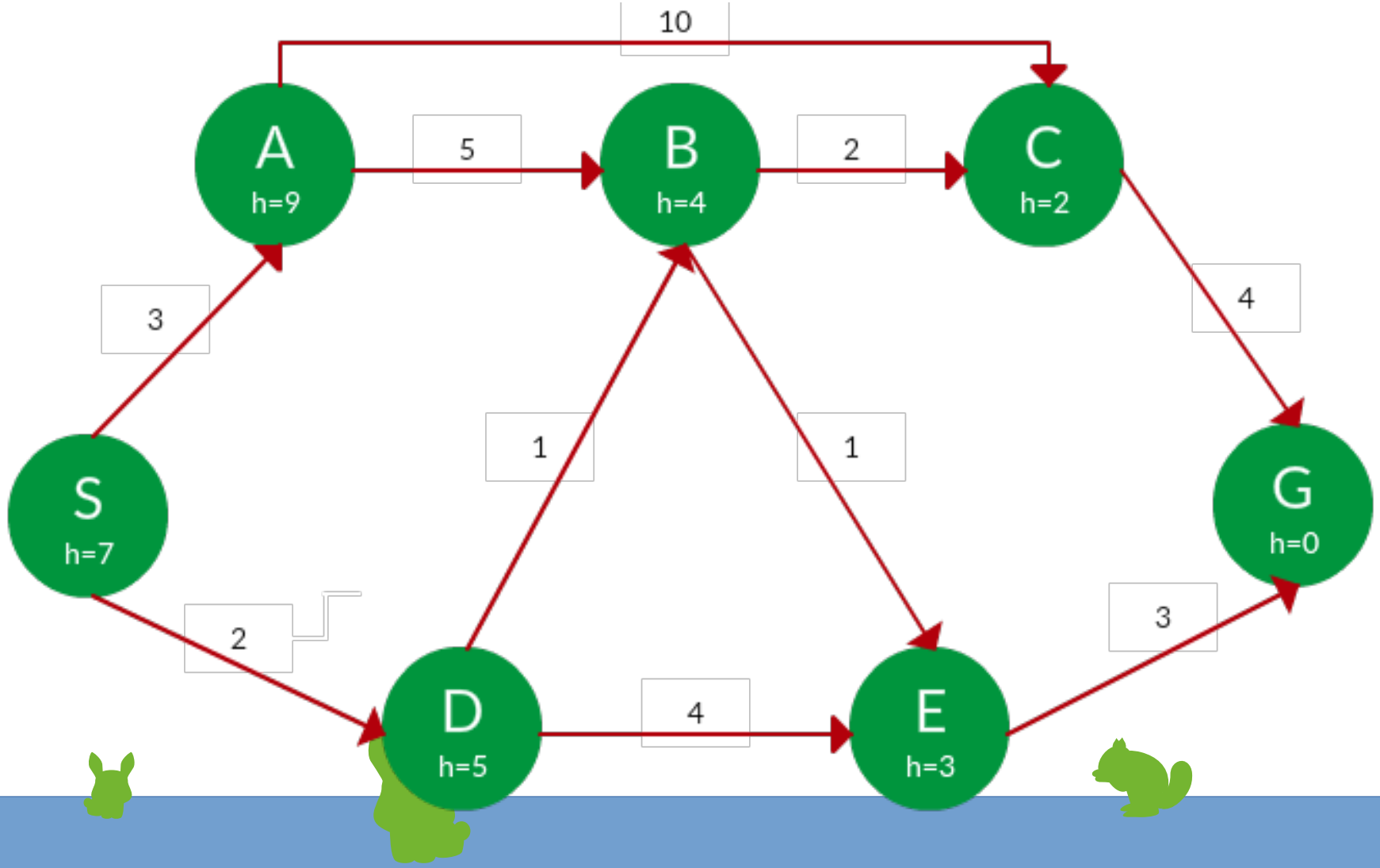


Cont..

- **Optimal:** A* search algorithm is optimal if it follows below two conditions:
  - **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
  - **Consistency:** Second required condition is consistency for only A* graph-search.
  - If the heuristic function is admissible, then A* tree search will always find the least cost path.
- **Time Complexity:** The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.
- **Space Complexity:** The space complexity of A* search algorithm is **O(b^d)**



Find the path from S to G using greedy search and A* search.

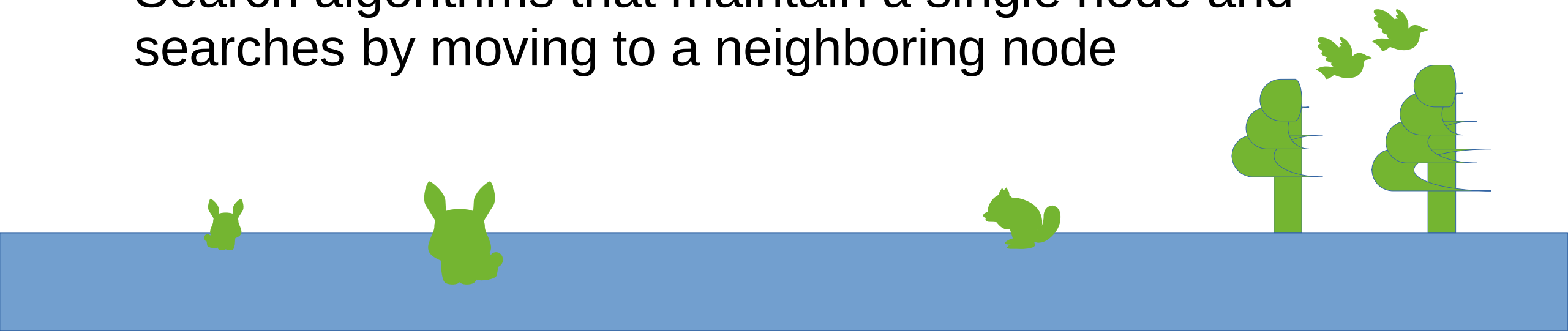


Search Algorithms So Far

- Designed to explore search space systematically:
  - keep one or more paths in memory
  - record which have been explored and which have not
  - a path to goal represents the solution

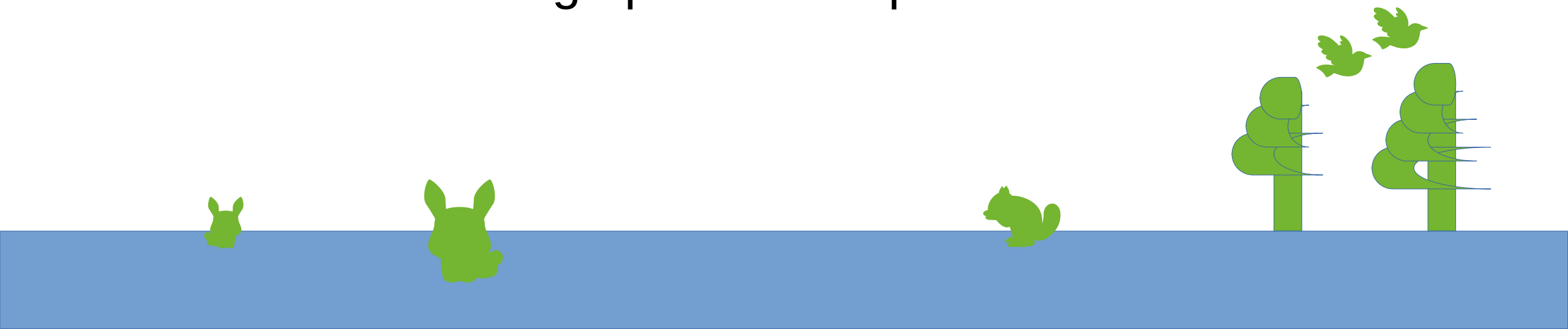
Local Search Algorithms

- In many **optimization problems**, the **path** to the **goal** is **irrelevant,** the goal state itself is the solution
- State space = set of "complete" configurations
- In such cases, we can use **local search algorithms**
- Search algorithms that maintain a single node and searches by moving to a neighboring node



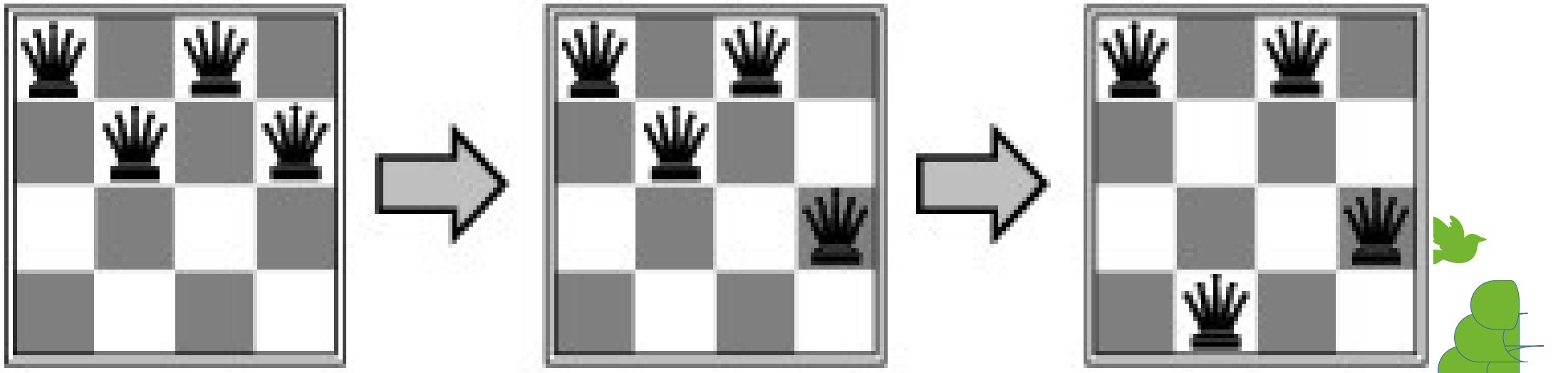
Local Search Algorithms

- Keep a single **"current"** state, try to improve it
  - use very little memory, usually a constant amount
  - find reasonable solutions in large or infinite state spaces for which systematic solutions are unsuitable
  - useful for solving optimization problems



Example: n-Queen Problem

- Put **n queens** on an **n × n** board with no two queens on the same row, column, or diagonal.

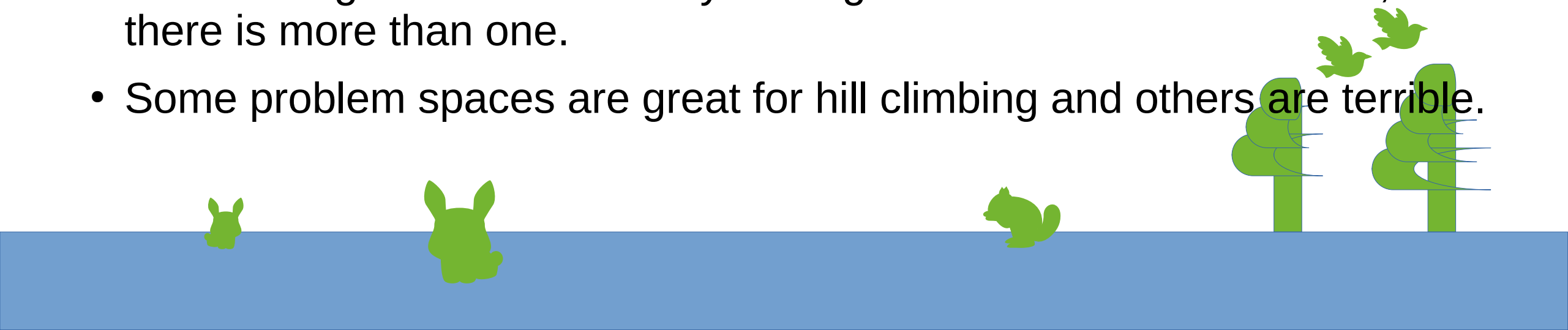


Types of local searches Algorithm

- Hill-climbing Search
- Simulated Annealing
- Local Beam Search
- Genetic Algorithm

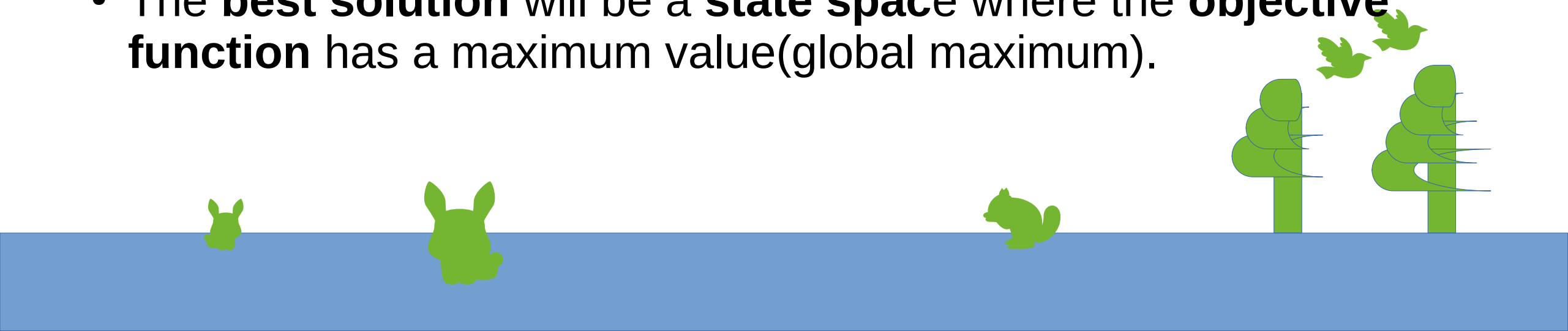
Hill-climbing Search

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Hill-climbing chooses randomly among the set of best successors, if there is more than one.
- Some problem spaces are great for hill climbing and others are terrible.

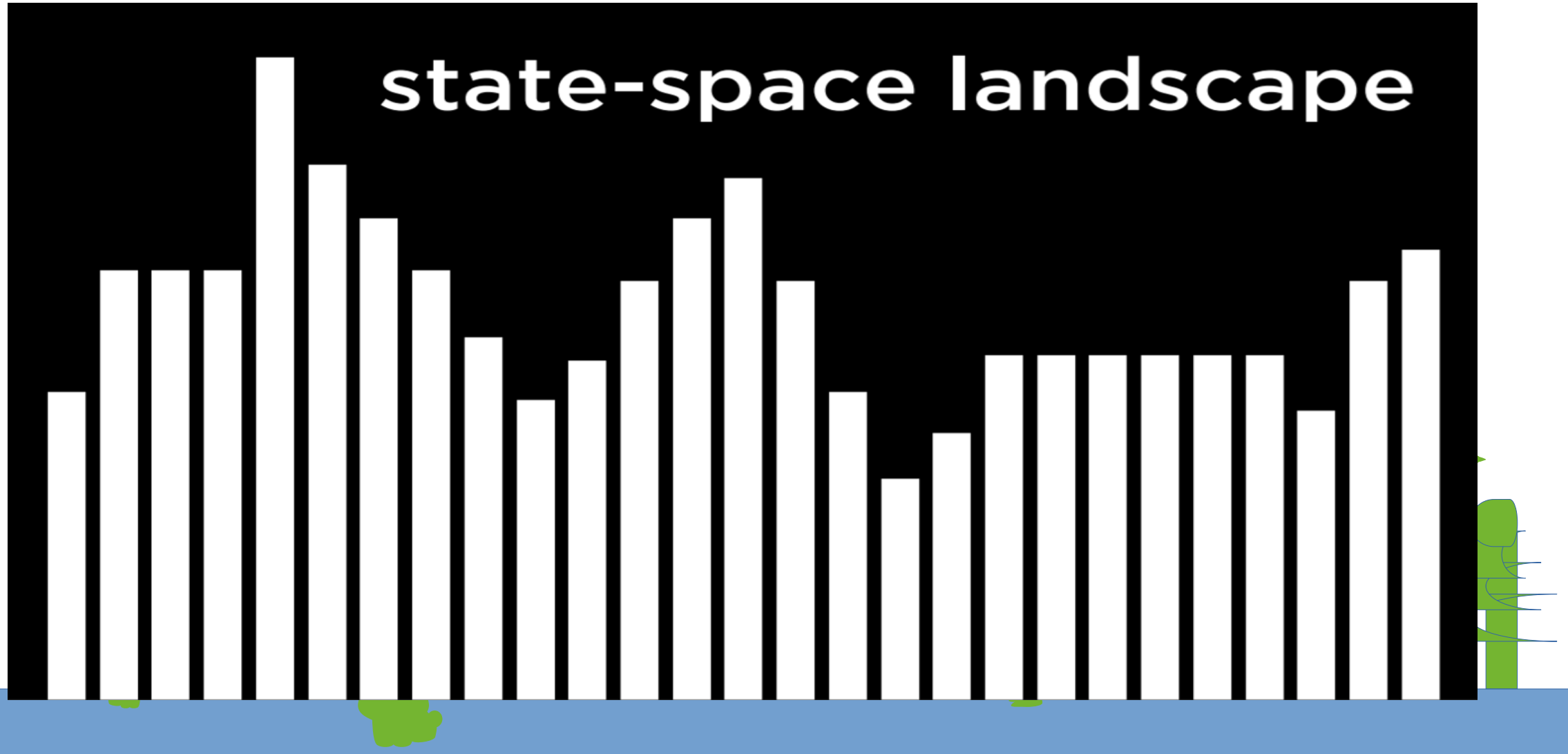


State-space Landscape of Hill climbing

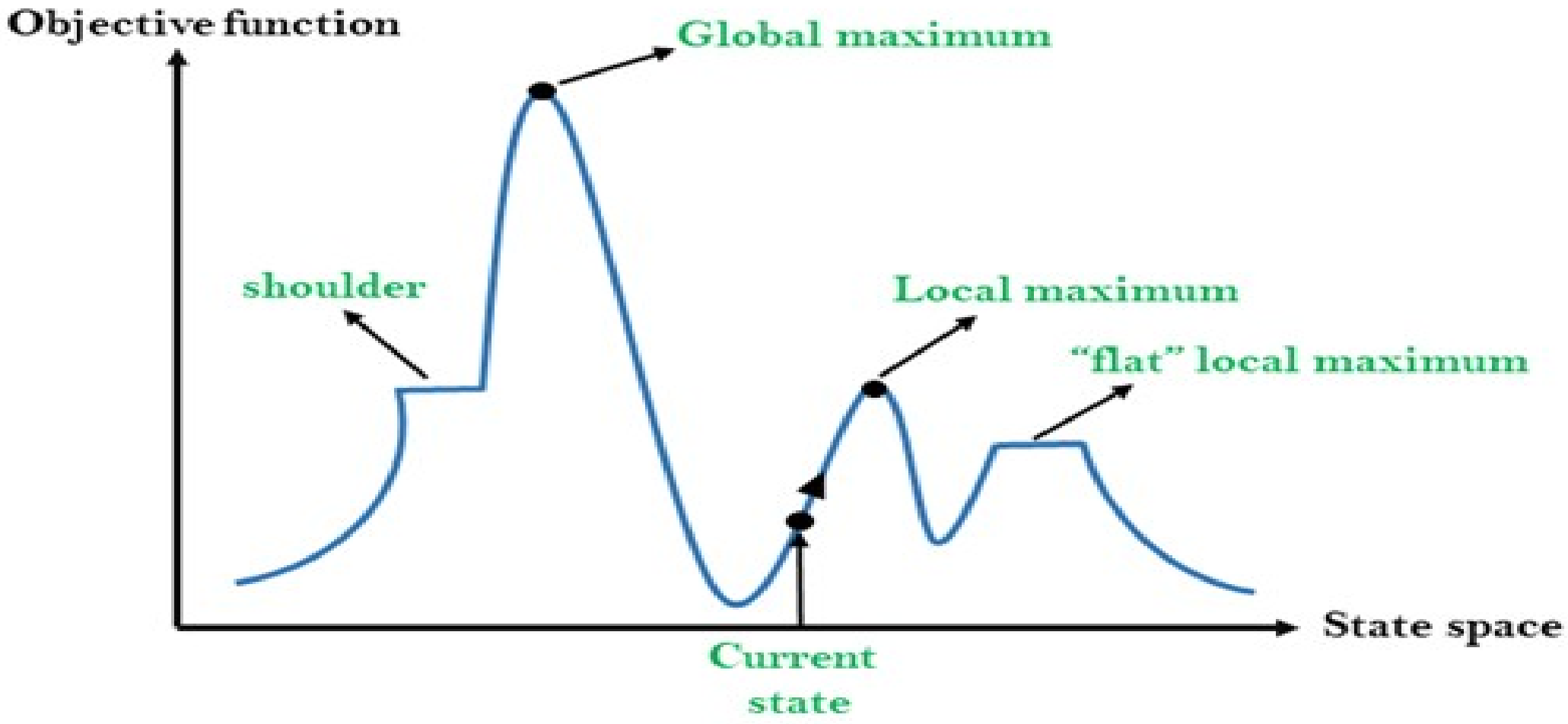
- The state-space diagram is a **graphical representation** of the **set of states** our search algorithm can reach vs the value of our **objective function**(the function which we wish to maximize).
  - **X-axis:** denotes the state space states or configuration our algorithm may reach.
  - **Y-axis**: denotes the values of objective function corresponding to a particular state.
- The **best solution** will be a **state spac**e where the **objective function** has a maximum value(global maximum).



State-space Landscape of Hill climbing

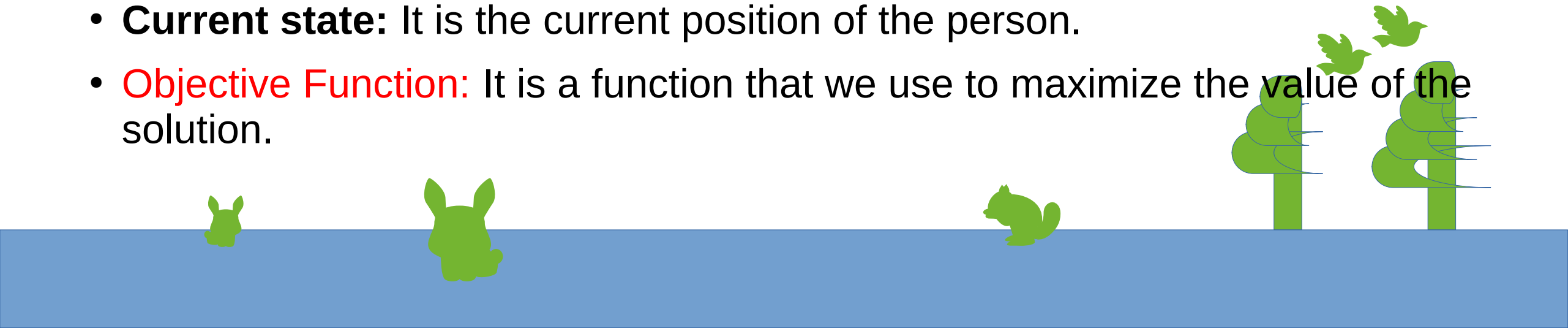


State-space Landscape of Hill climbing



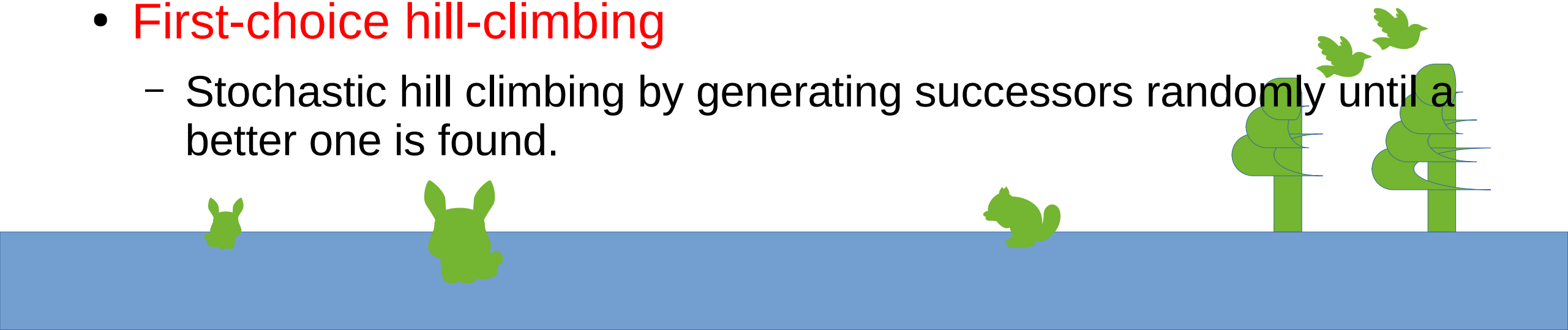
Different regions in the State Space

- **Global Maximum:** It is the highest point on the hill, which is the goal state.
- **Local Maximum:** It is the peak higher than all other peaks but lower than the global maximum.
- **Flat local maximum:** It is the flat area over the hill where it has no uphill or downhill. It is a saturated point of the hill.
- **Shoulder:** It is also a flat area where the summit is possible.
- **Current state:** It is the current position of the person.
- Objective Function: It is a function that we use to maximize the value of the solution.



Types of Hill climbing search algorithm

- Steepest-ascent: choose the highest-valued neighbor.
- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found.



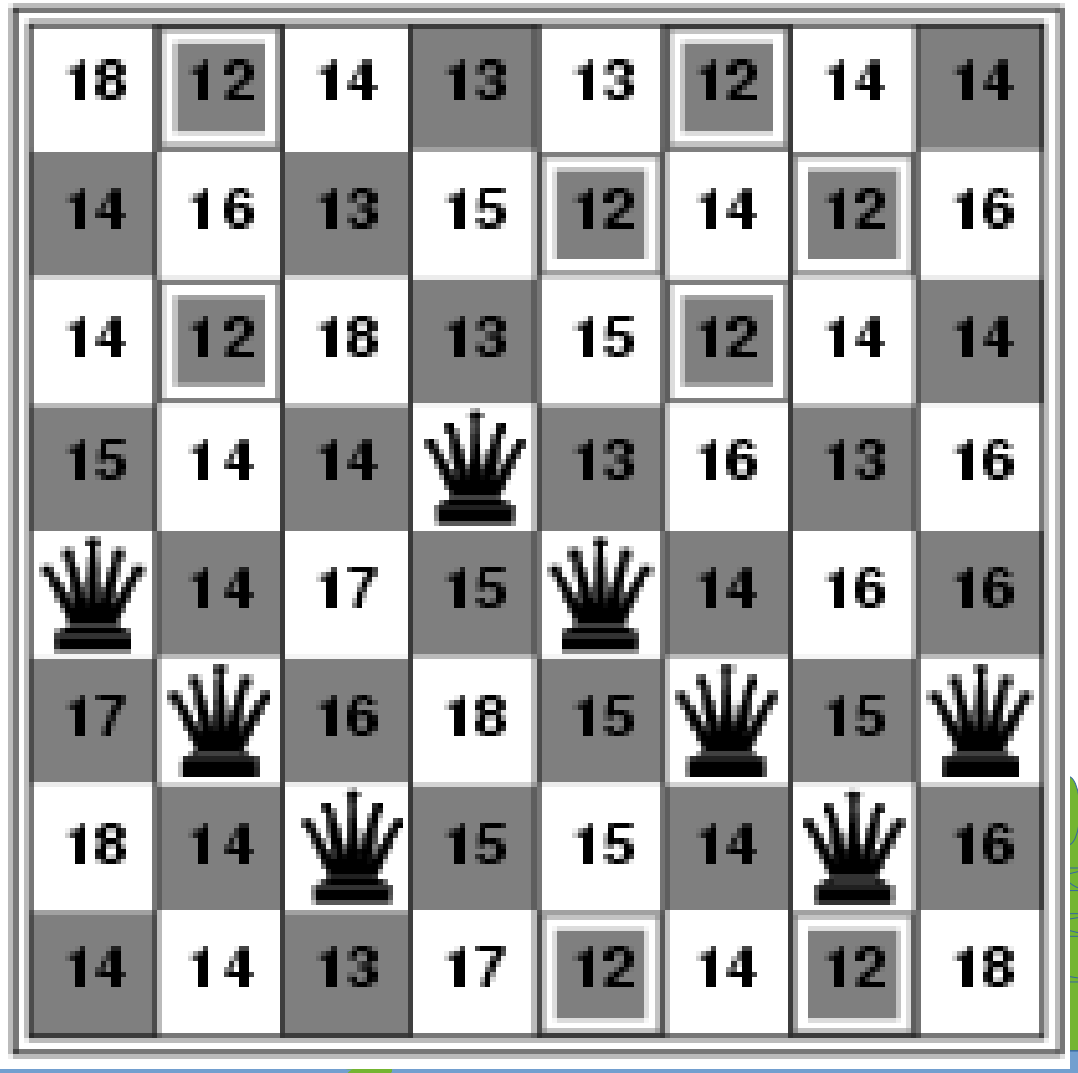
Cont..

- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.
  - If at first you don’t succeed, try, try again…

Example: 8-queens

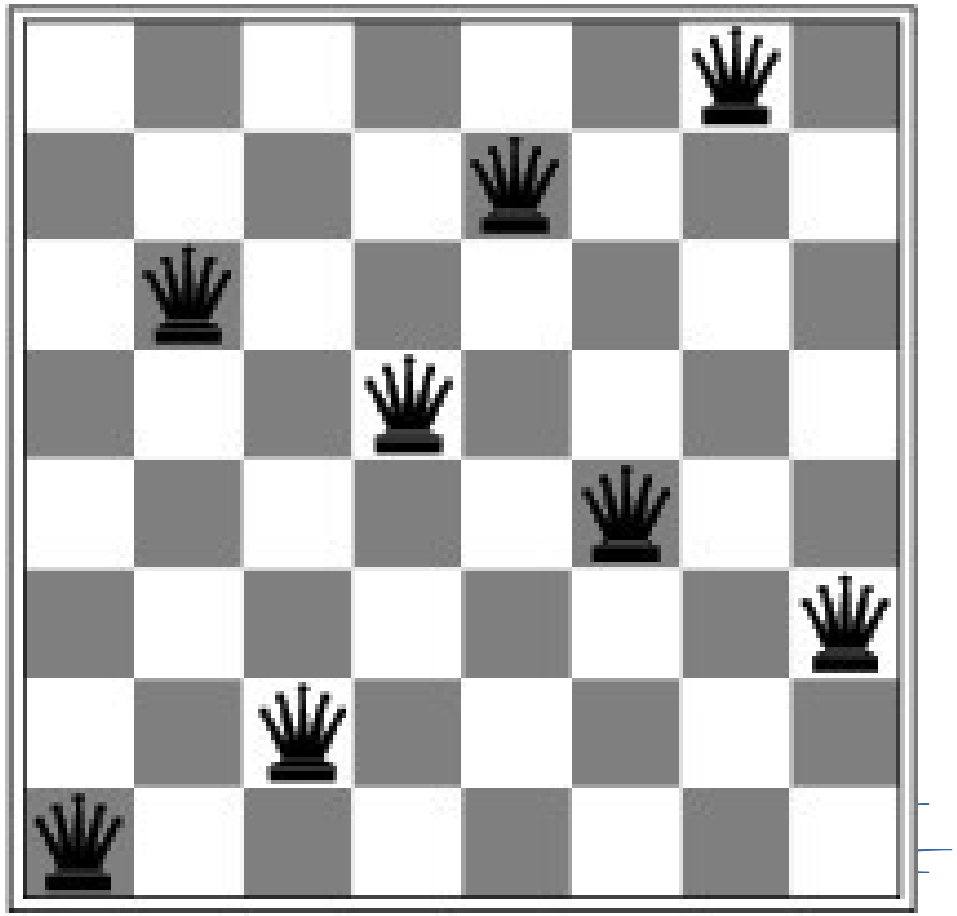
- h = number of pairs of queens that are attacking each other, either directly or indirectly
- h = 17 for the above state

- Each number indicates h if we move a queen in its column to that square
- 12 (boxed) = best h among all neighbors, select one randomly



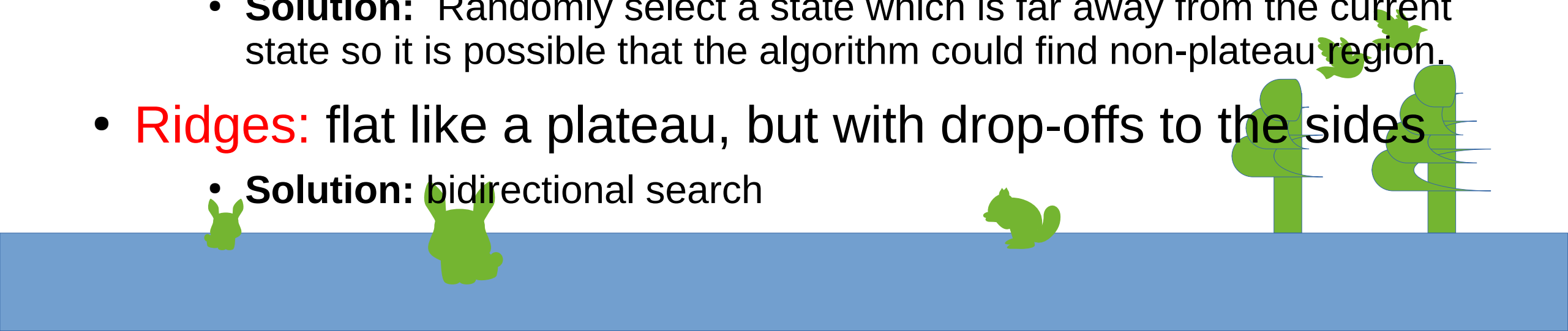
Cont..

- A local minimum with h = 1



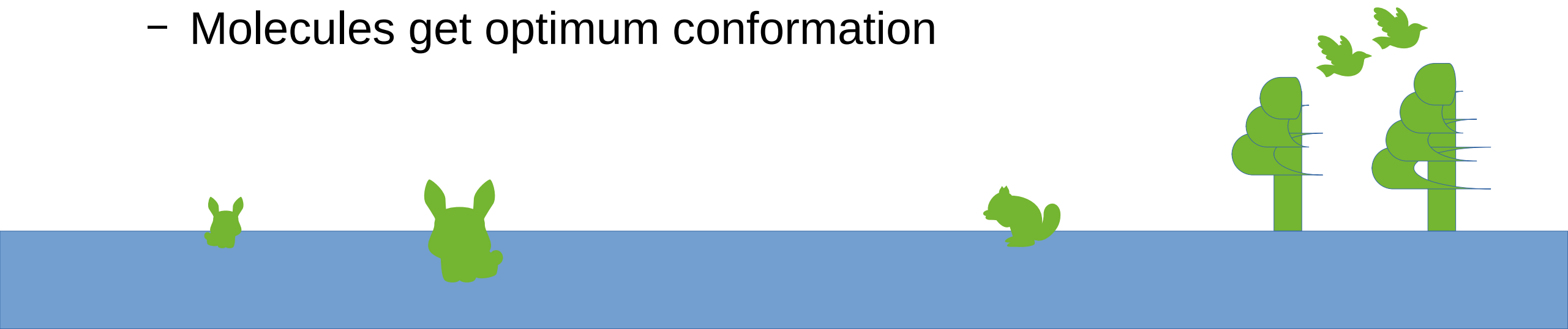
Drawbacks of hill climbing

- Local Maxima: depending on initial state, can get stuck in local maxima.
  - **Solution:** Backtracking technique
- Plateaus: the space has a broad flat region that gives the search algorithm no direction (random walk).
  - **Solution:** Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.
- Ridges: flat like a plateau, but with drop-offs to the sides
  - **Solution:** bidirectional search

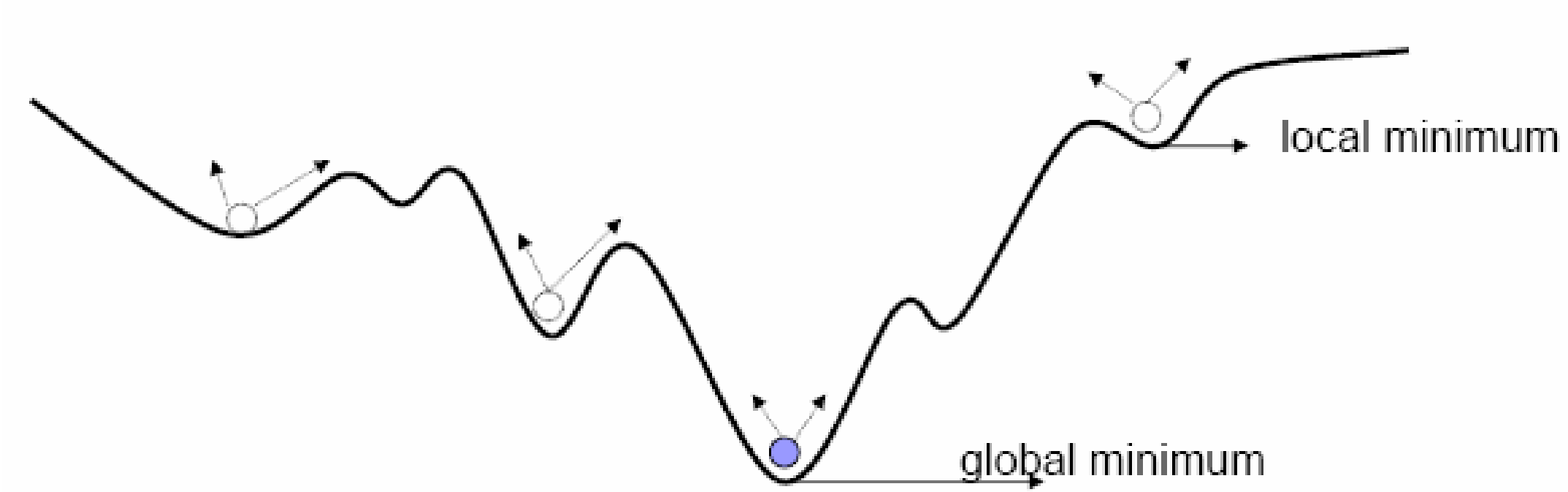


Simulated Annealing

- Simulates slow cooling of annealing process
- **What is annealing?**
  - Process of slowly cooling down a compound or a substance
  - Slow cooling let the substance flow around thermodynamic equilibrium
  - Molecules get optimum conformation



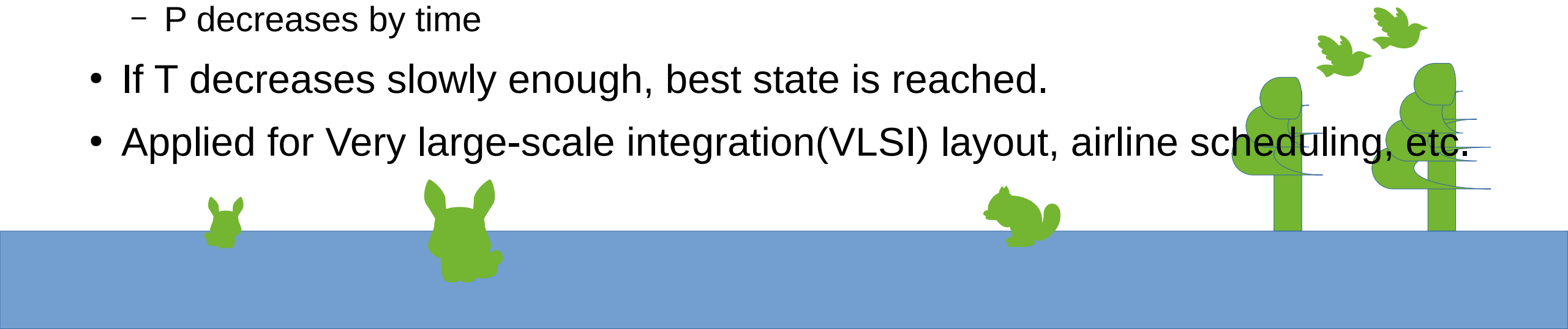
Simulated Annealing



- Gradually decrease shaking to make sure the ball escape from local minima and fall into the global minimum.

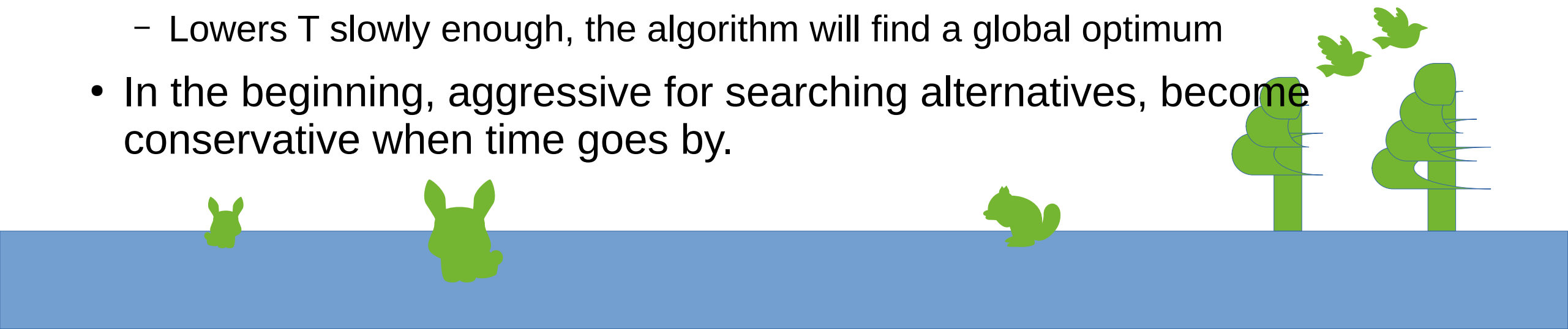
Simulated Annealing

- Escape local maxima by allowing “bad” moves.
  - **Idea:** but gradually decrease their **size** and **frequency.**
- **Origin: metallurgical annealing**
- Implement:
  - Randomly select a move instead of selecting best move
  - Accept a bad move with probability less than 1 (p<1)
  - P decreases by time
- If T decreases slowly enough, best state is reached.
- Applied for Very large-scale integration(VLSI) layout, airline scheduling, etc.



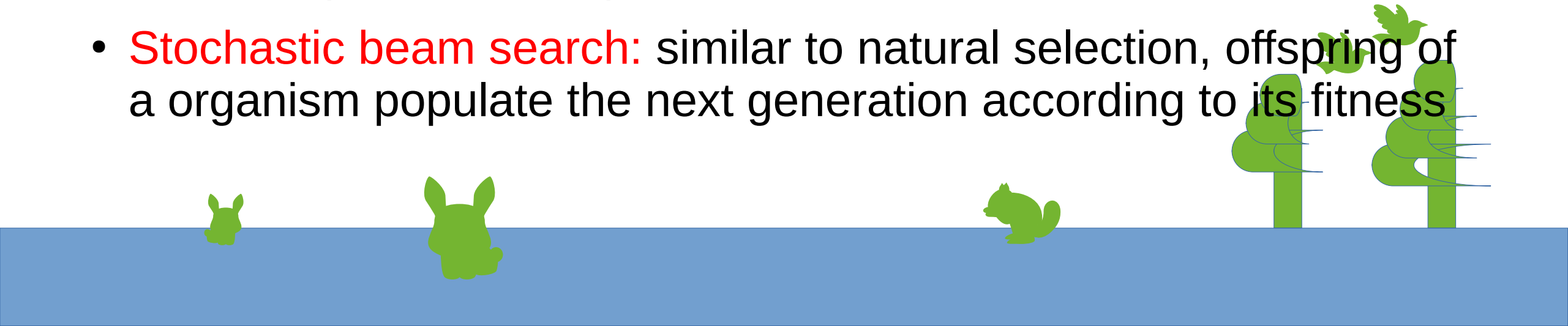
Simulated Annealing parameters

- **Temperature T**
  - Used to determine the probability
  - **High T** : large changes
  - **Low T** : small changes
- **Cooling Schedule**
  - Determines rate at which the temperature T is lowered
  - Lowers T slowly enough, the algorithm will find a global optimum
- In the beginning, aggressive for searching alternatives, become conservative when time goes by.



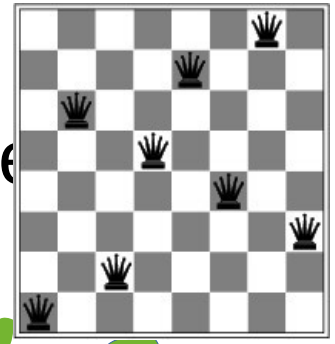
Local beam search

- **Idea:**
  - Keep track of k states rather than just one
  - Start with k randomly generated states
  - At each iteration, all the successors of all k states are generated
  - If any one is a goal state, stop; else select the k best successors from the complete list and repeat
- Stochastic beam search: similar to natural selection, offspring of a organism populate the next generation according to its fitness



Genetic Algorithms

- A successor state is generated by combining two parent states
- Start with k randomly generated states (population)
- A state is represented as a string over a finite alphabet 16257483 (often a string of 0s and 1s or digits)
- Evaluation function (fitness function). Higher values for bette states
- Produce the next generation of states by selection, crossover, and mutation

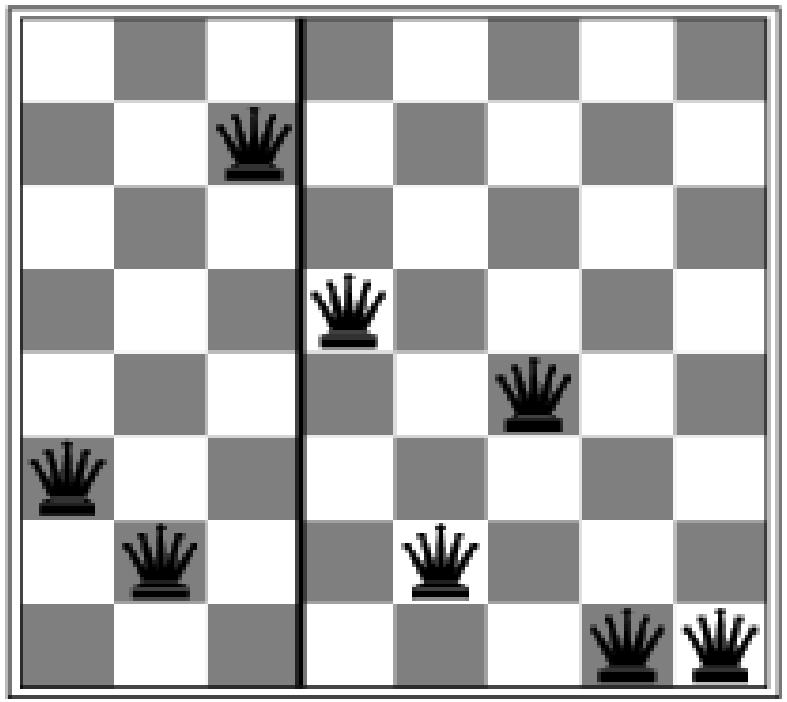


Genetic Algorithms

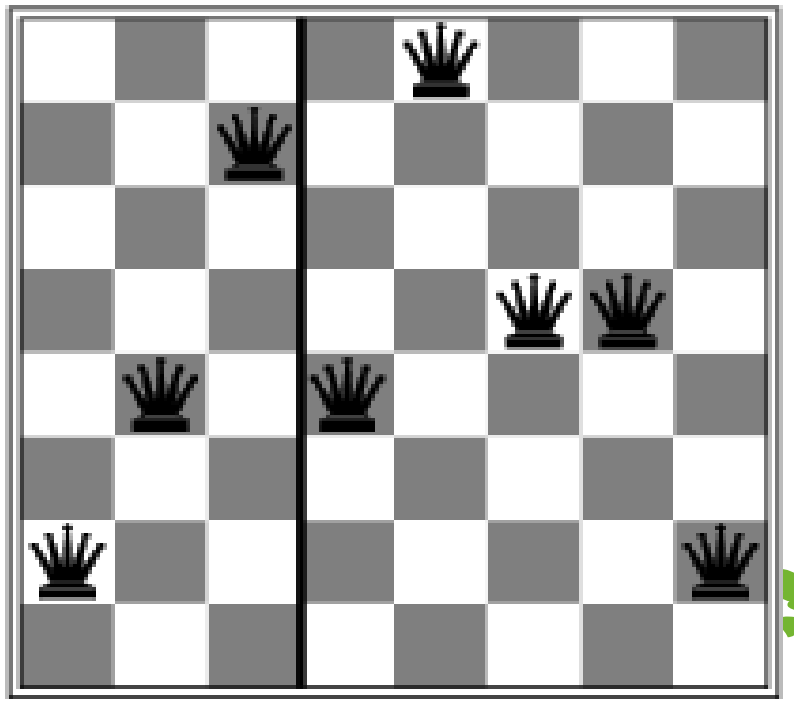
- Select individuals for next generation based on fitness
  - P(individual in next gen.) = individual fitness/total population fitness
- Crossover fit parents to yield next generation (offspring)
- Mutate the offspring randomly with some low probability

Example: N-Queens

- Each “individual” is a vector of each queen’s row



+

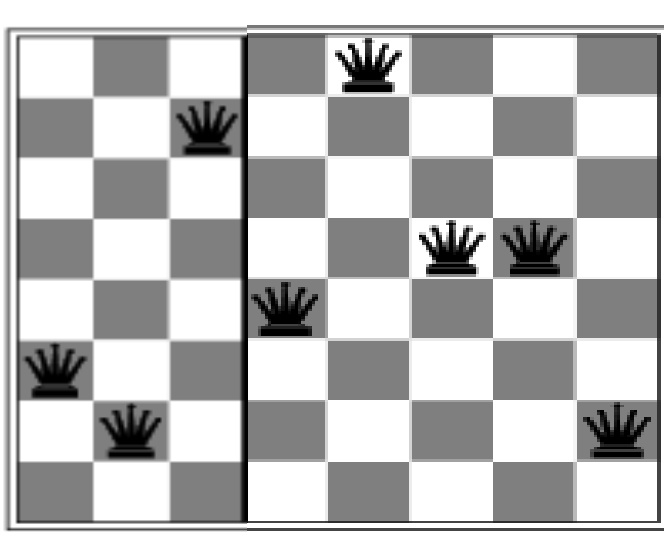


3	2	7	5	2	4	1	1

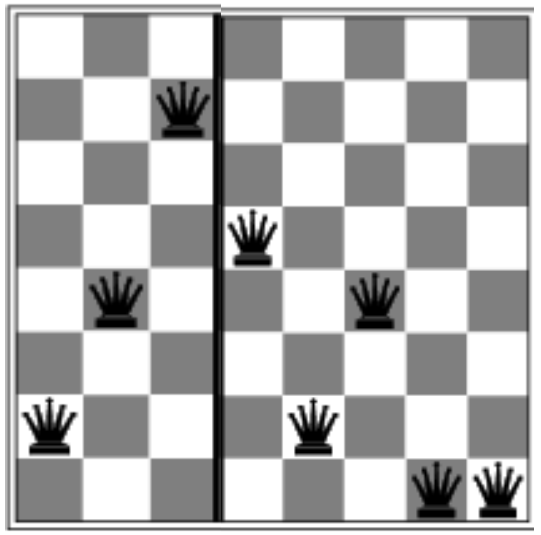
2	4	7	4	8	5	5	2

Example: N-Queens

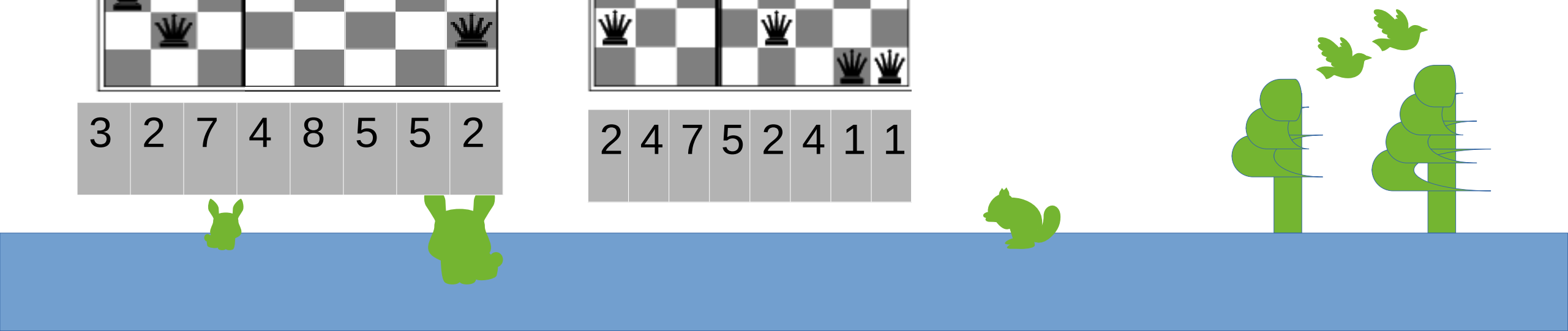
- Resulting individuals after cross-over



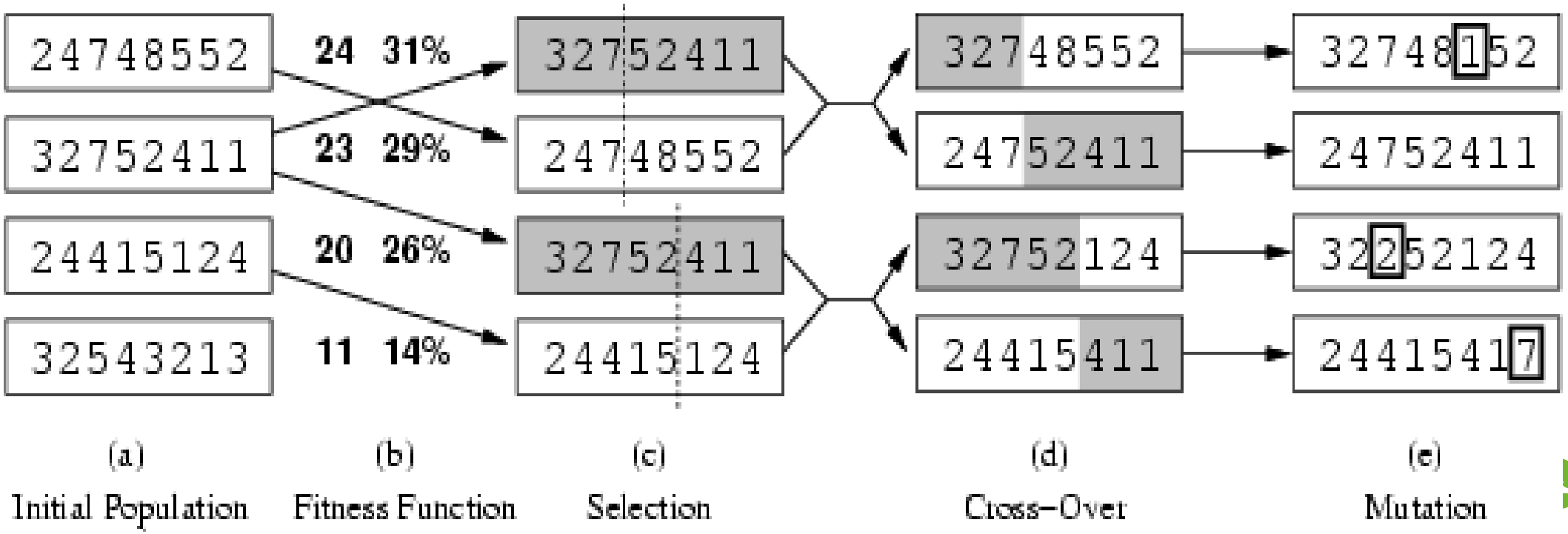
3	2	7	4	8	5	5	2



2	4	7	5	2	4	1	1

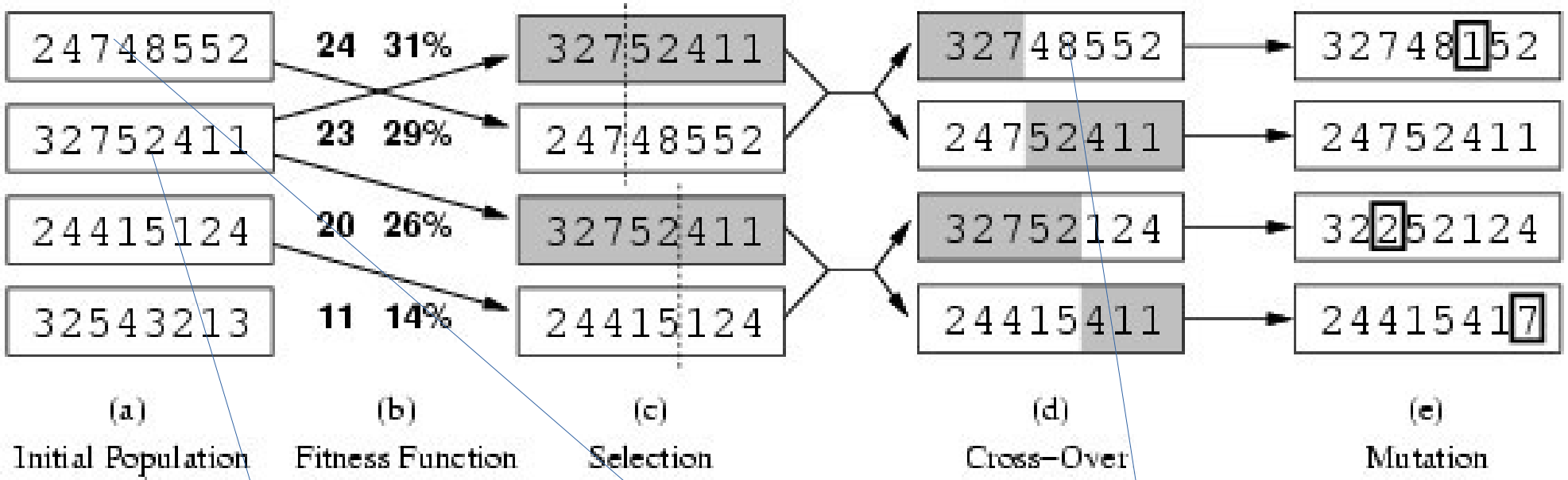


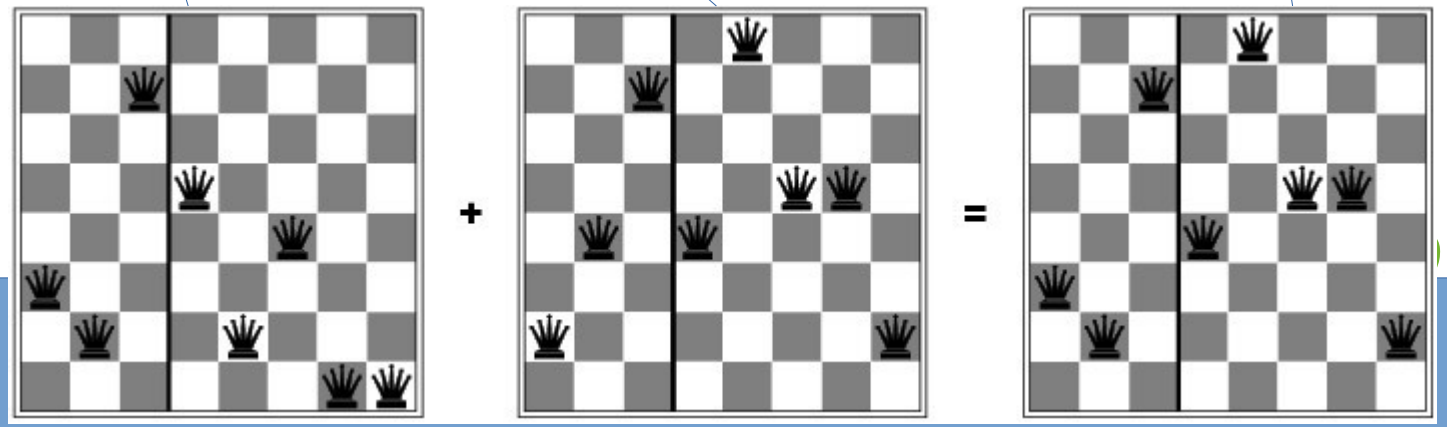
Genetic algorithms

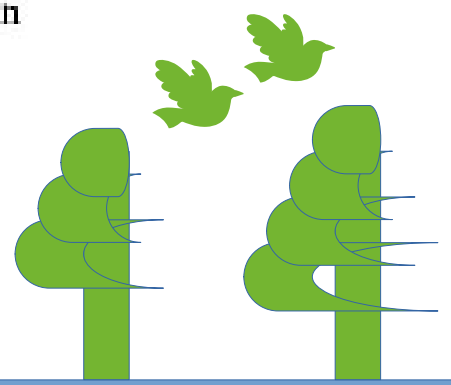


- Fitness function (value): number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)

Example: 8-Queen

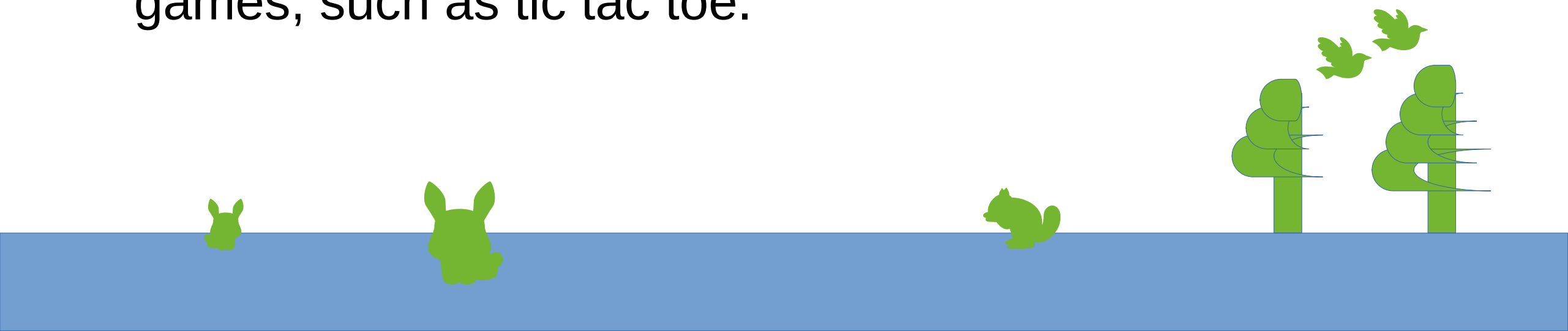






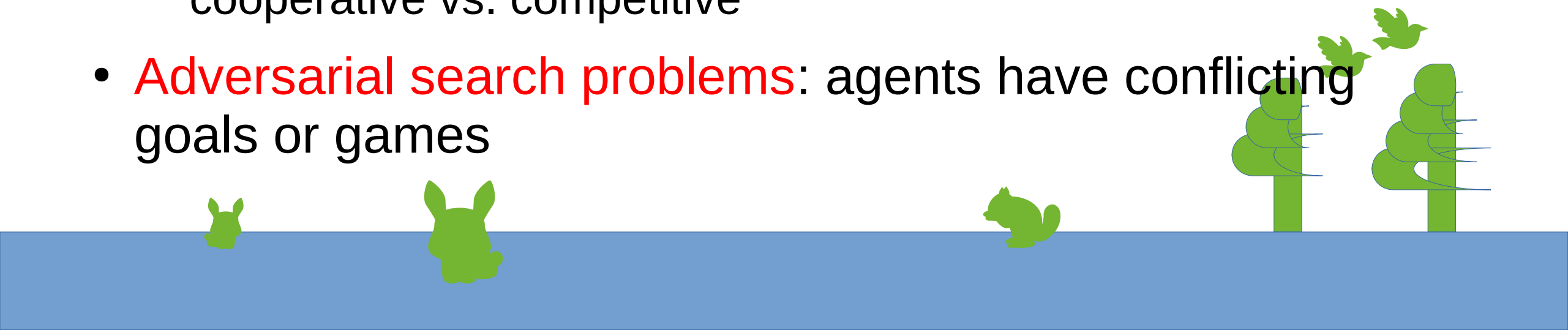
Adversarial Search Strategies

- Previously, we have discussed algorithms that need to find an answer to a question, in adversarial search the algorithm faces an opponent that tries to achieve the opposite goal.
- Often, AI that uses adversarial search is encountered in games, such as tic tac toe.



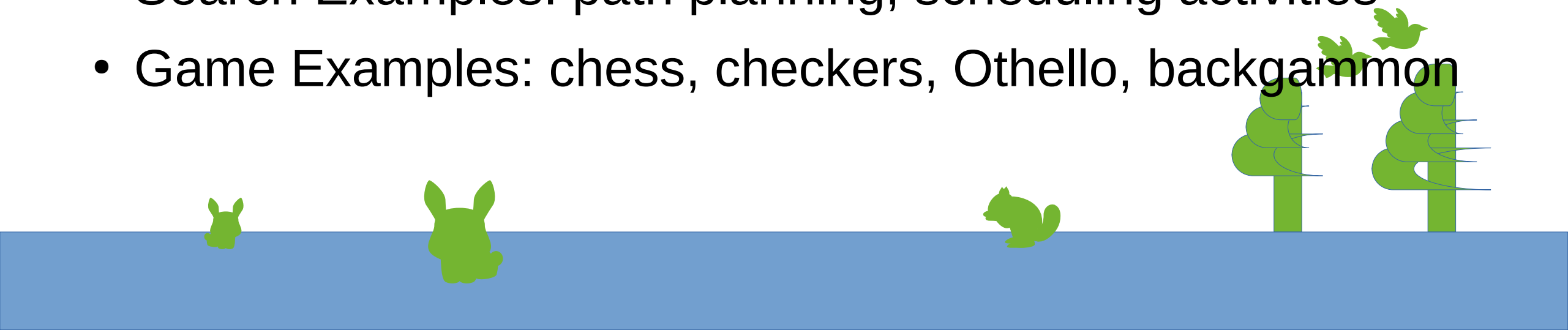
Adversarial Search

- Multi-agent environment:
  - any given agent needs to consider the actions of other agents and how they affect its own welfare
  - introduce possible contingencies into the agent's problem-solving process
  - cooperative vs. competitive
- Adversarial search problems: agents have conflicting goals or games



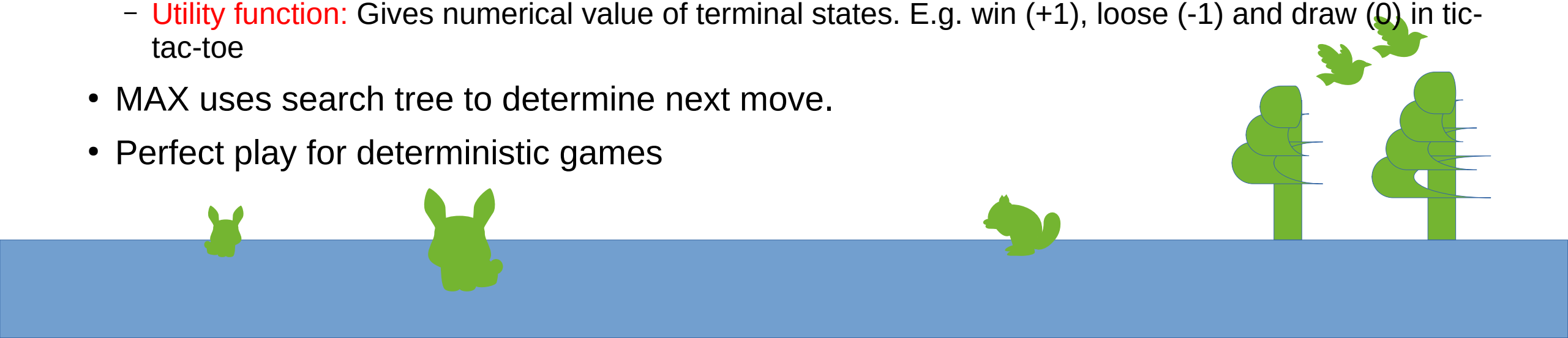
Games vs. Search Problems

- "Unpredictable"
  - Opponent specifying a move for every possible opponent reply
- Time limits
  - unlikely to find goal, must approximate
- Search Examples: path planning, scheduling activities
- Game Examples: chess, checkers, Othello, backgammon

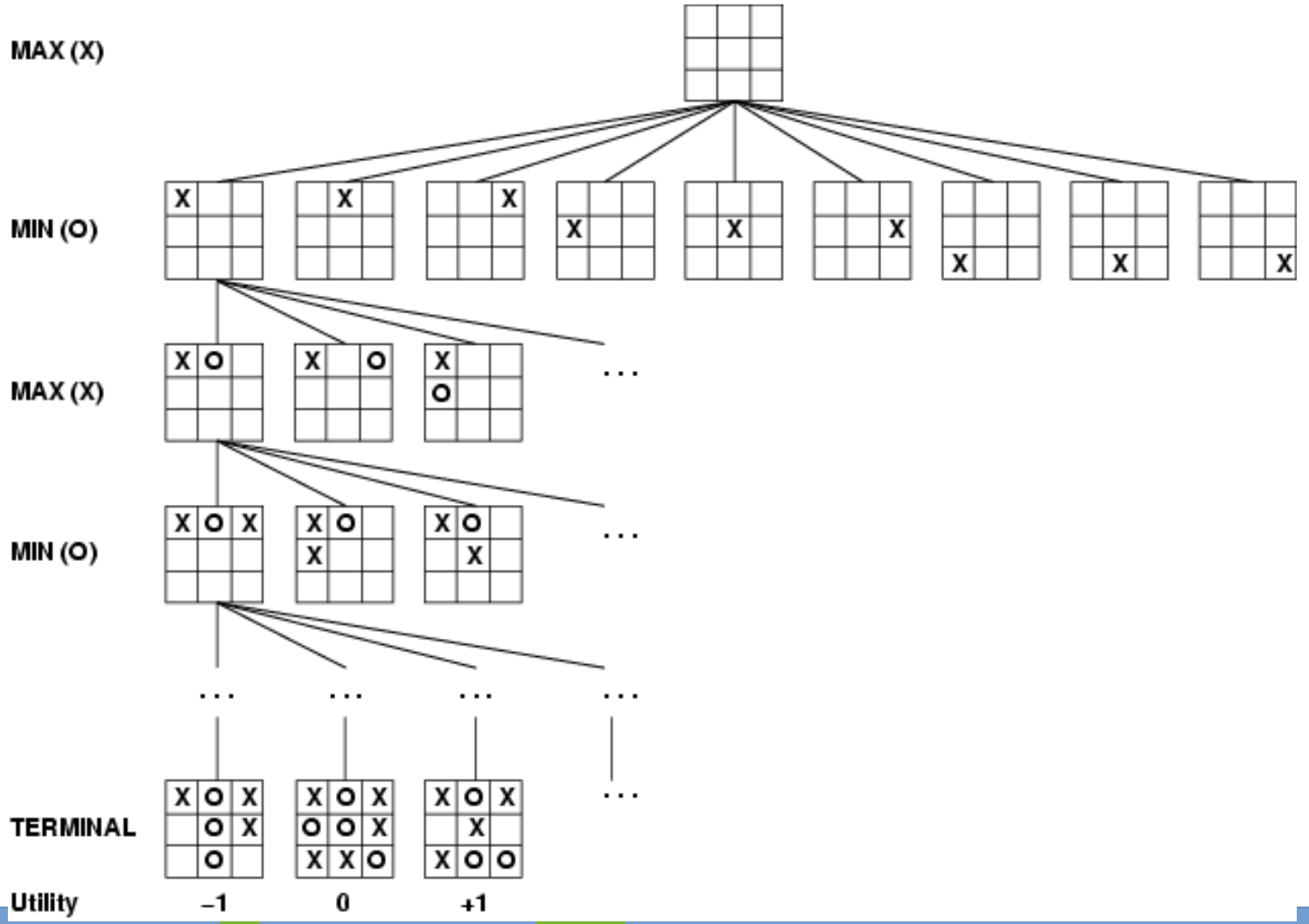


Minimax

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over. Winner gets award, looser gets penalty.
- Games as search:
  - Initial state: e.g., board configuration of chess
  - Successor function: list of (move, state) pairs specifying legal moves.
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states. E.g. win (+1), loose (-1) and draw (0) in tic-tac-toe
- MAX uses search tree to determine next move.
- Perfect play for deterministic games



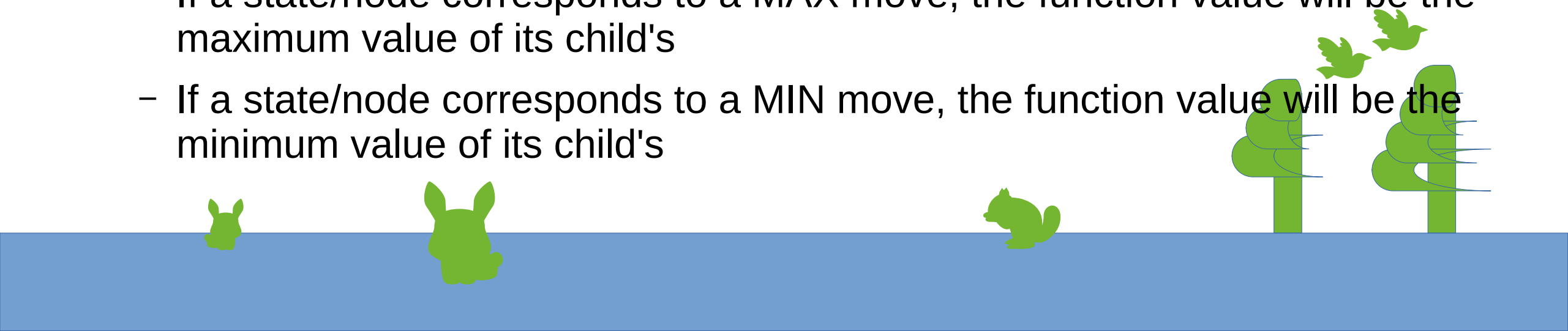
Minimax



- From among the moves available to you, take the best one
- The best one is determined by a search using the MiniMax strategy

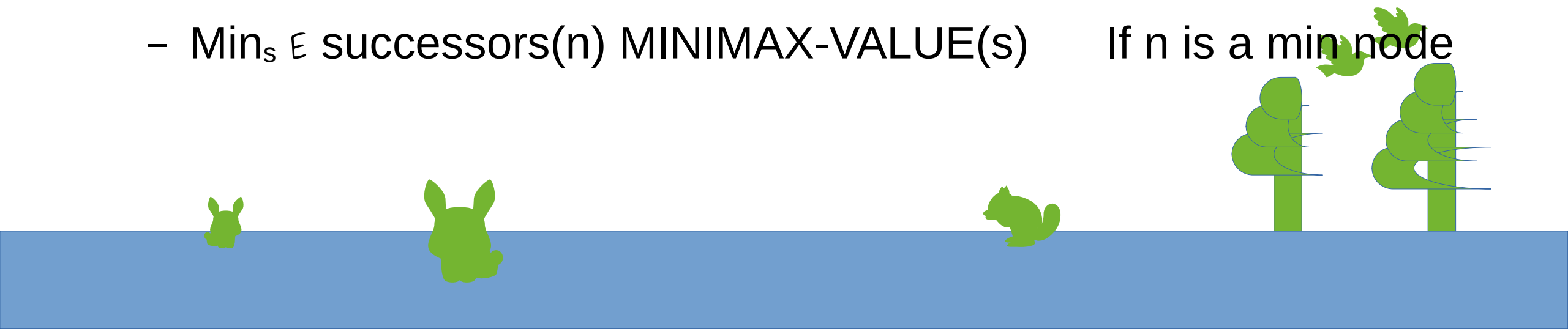
Optimal strategies

- MAX maximizes a function: find a move corresponding to max value
- MIN minimizes the same function: find a move corresponding to min value
- **At each step**:
  - If a state/node corresponds to a MAX move, the function value will be the maximum value of its child's
  - If a state/node corresponds to a MIN move, the function value will be the minimum value of its child's

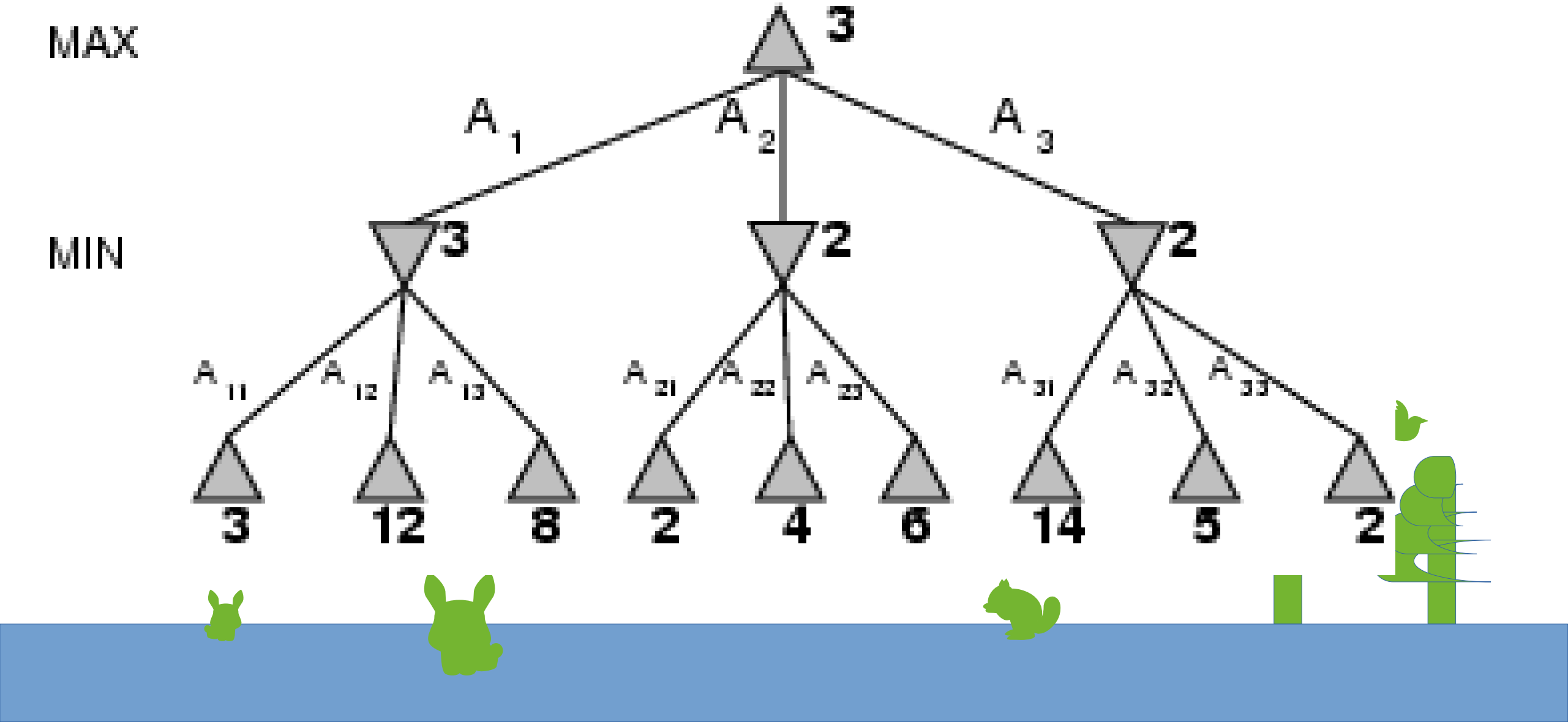


Optimal strategies

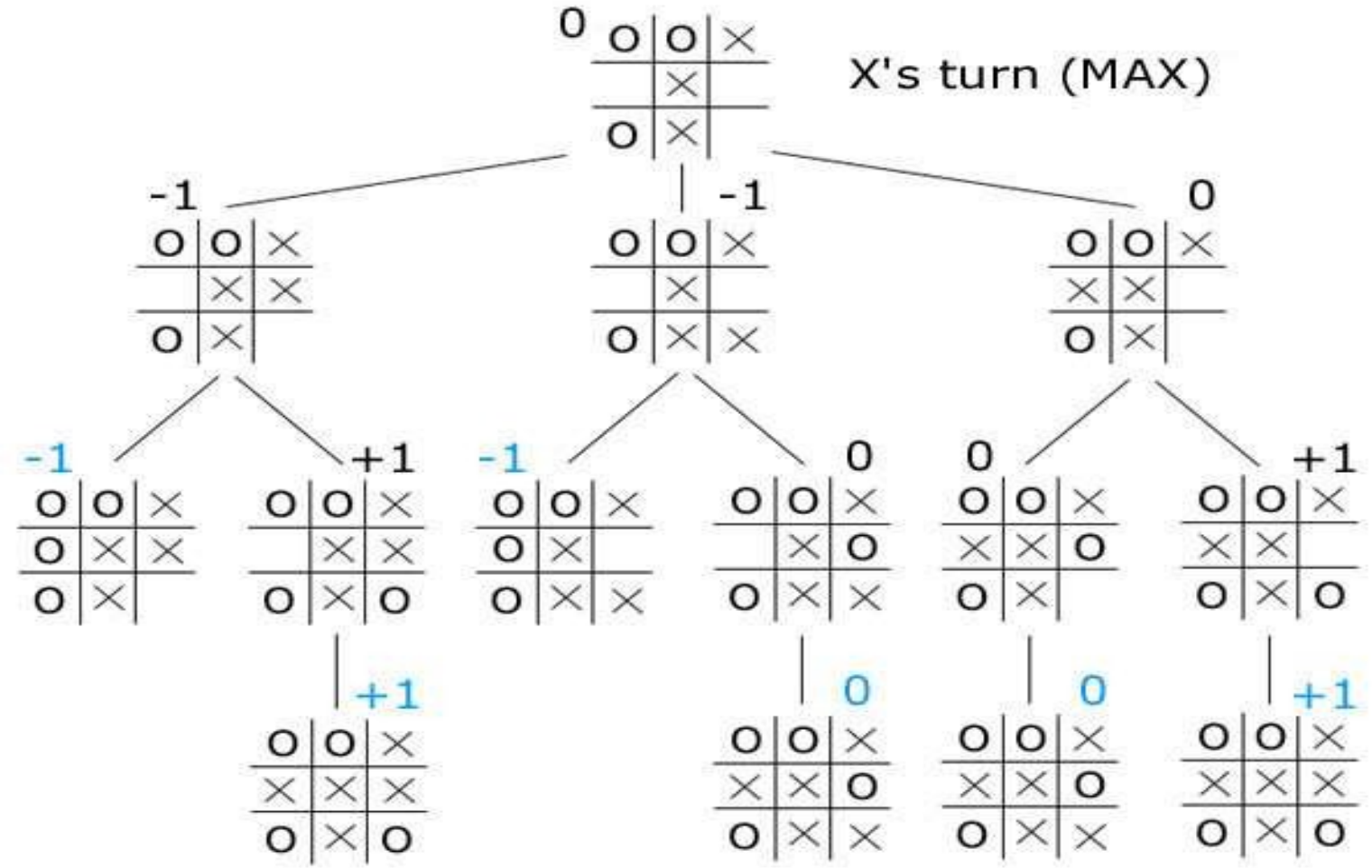
- Given a game tree, the optimal strategy can be determined by using the minimax value of each node:
- MINIMAX-VALUE(n) =
  - UTILITY(n) If n is a terminal
  - $\max_{s \in \text{successors}(n)}$ MINIMAX-VALUE(s) If n is a max node
  - $\text{Min}_{s \in \text{successors}(n)}$ MINIMAX-VALUE(s) If n is a min node



Minimax


MAX
MIN
3
A 1
A 2
A 3
3
2
2
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2

A Partial Game Tree for Tic-Tac-Toe



Properties of minimax

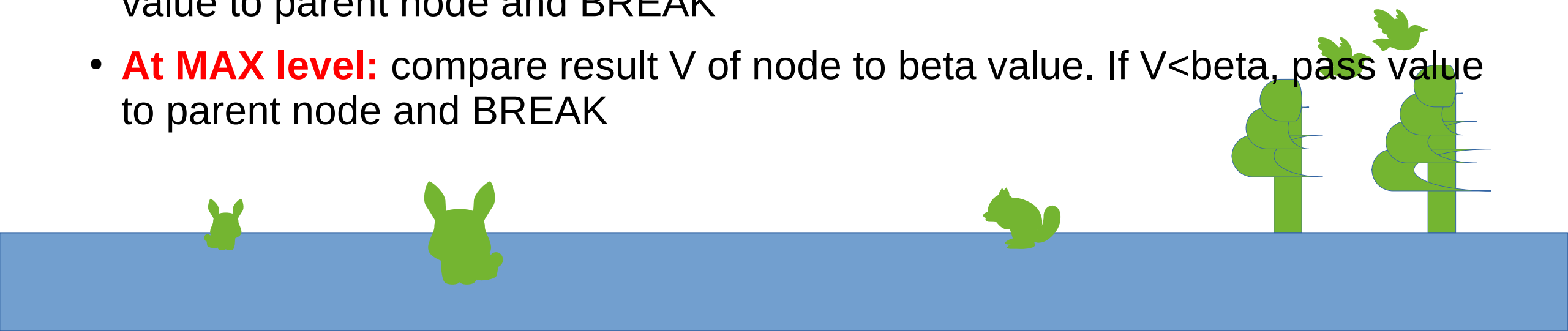
- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? O(bm)
- Space complexity? O(bm) (depth-first exploration)

Problem of minimax search

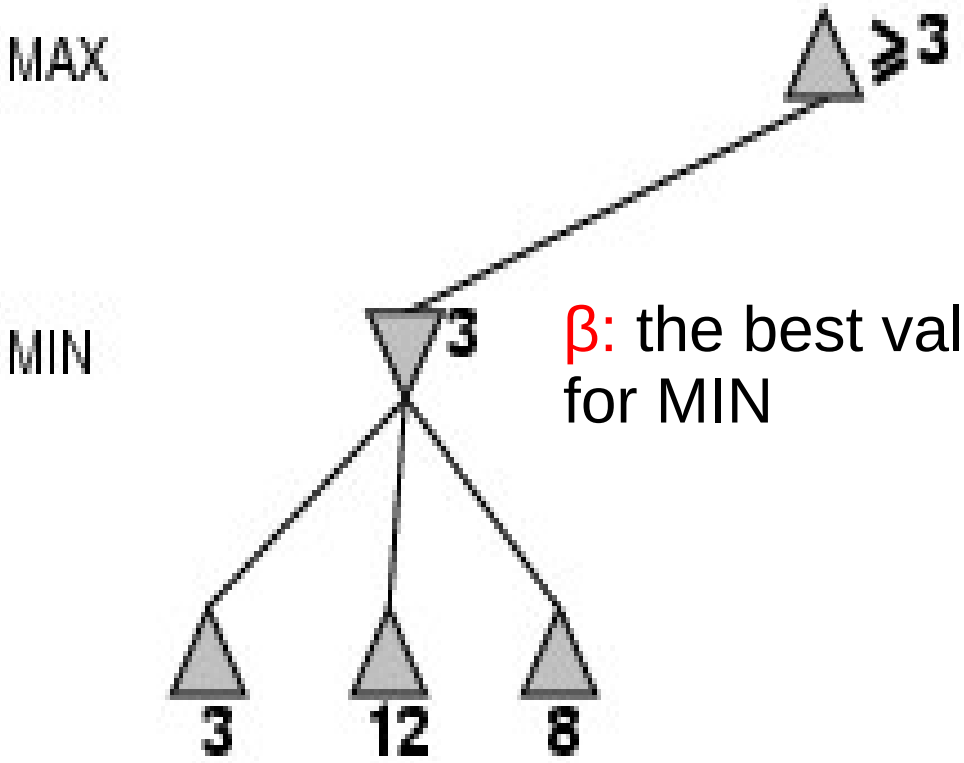
- Number of games states is exponential to the number of moves.
  - Solution: Do not examine every node
- Alpha-beta pruning:
  - Remove branches that do not influence final decision
  - Revisit example

ALPHA-BETA PRUNING

- **Pruning:** A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
- **Alpha values:** the best values achievable for MAX, hence the max value so far
- **Beta values:** the best values achievable for MIN, hence the min value so far
- **At MIN level:** compare result V of node to alpha value. If V>alpha, pass value to parent node and BREAK
- **At MAX level:** compare result V of node to beta value. If V<beta, pass value to parent node and BREAK



α-β pruning

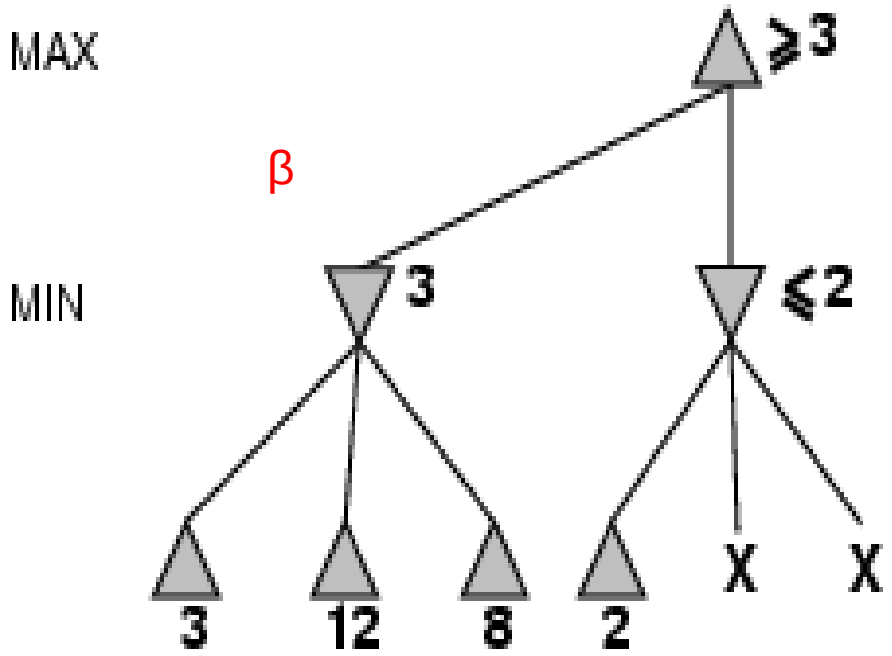


α: the best values achievable for MAX

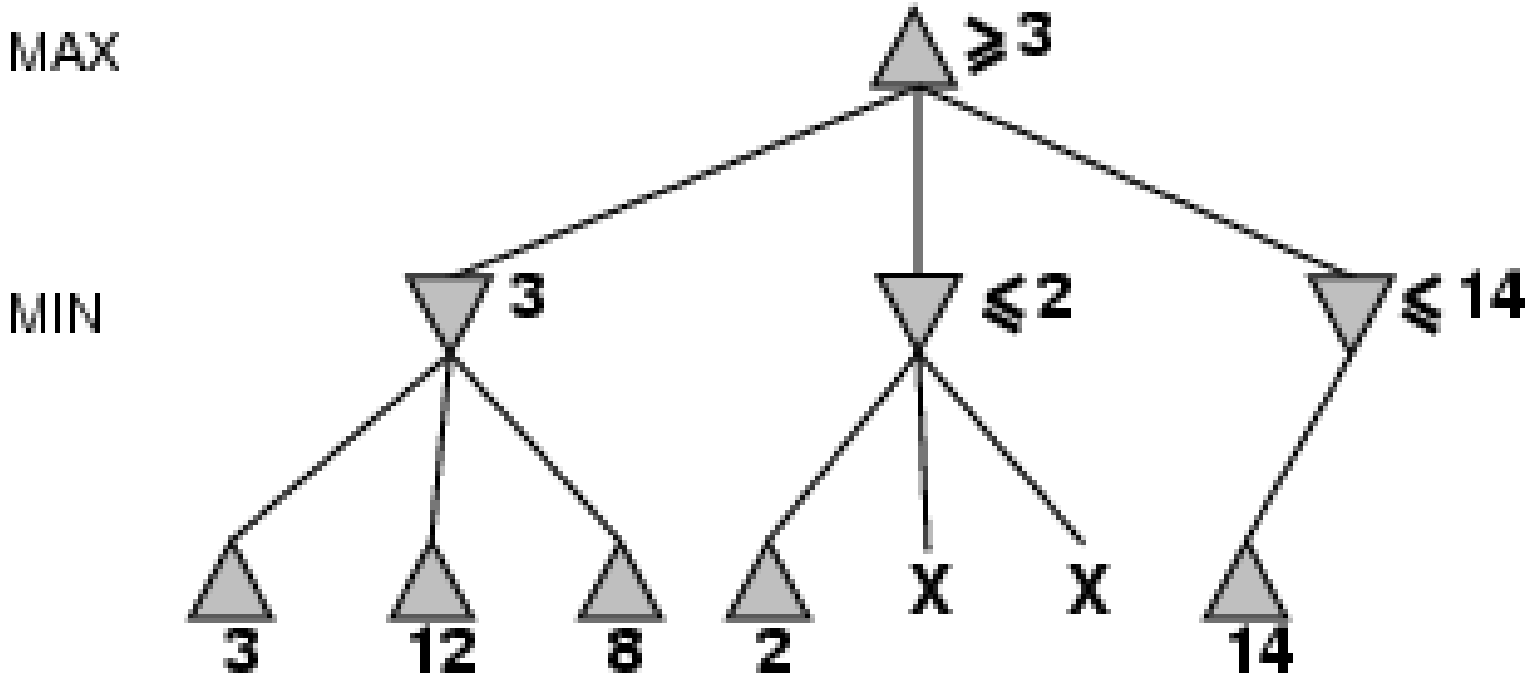
β: the best values achievable for MIN

α-β pruning example

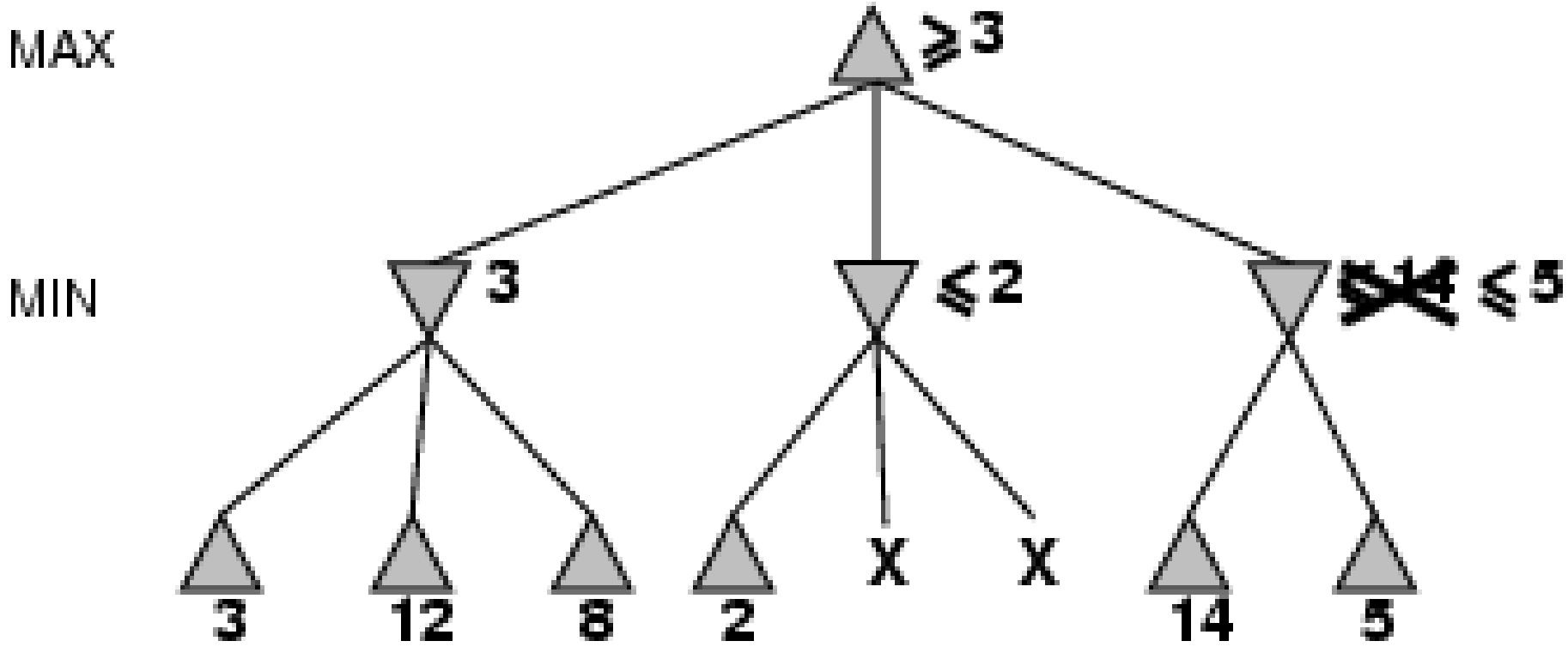
- Compare result V of node to β. If V< β, pass value to parent node and BREAK.



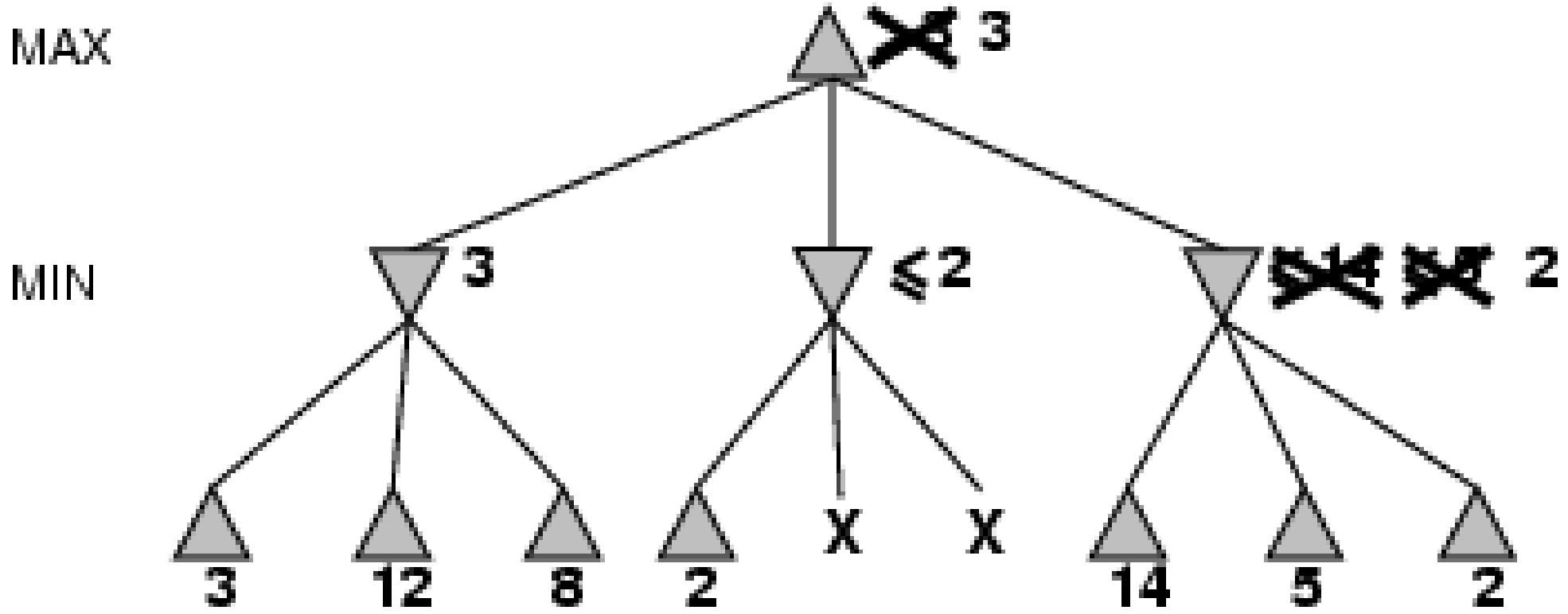
α-β pruning example



α-β pruning example

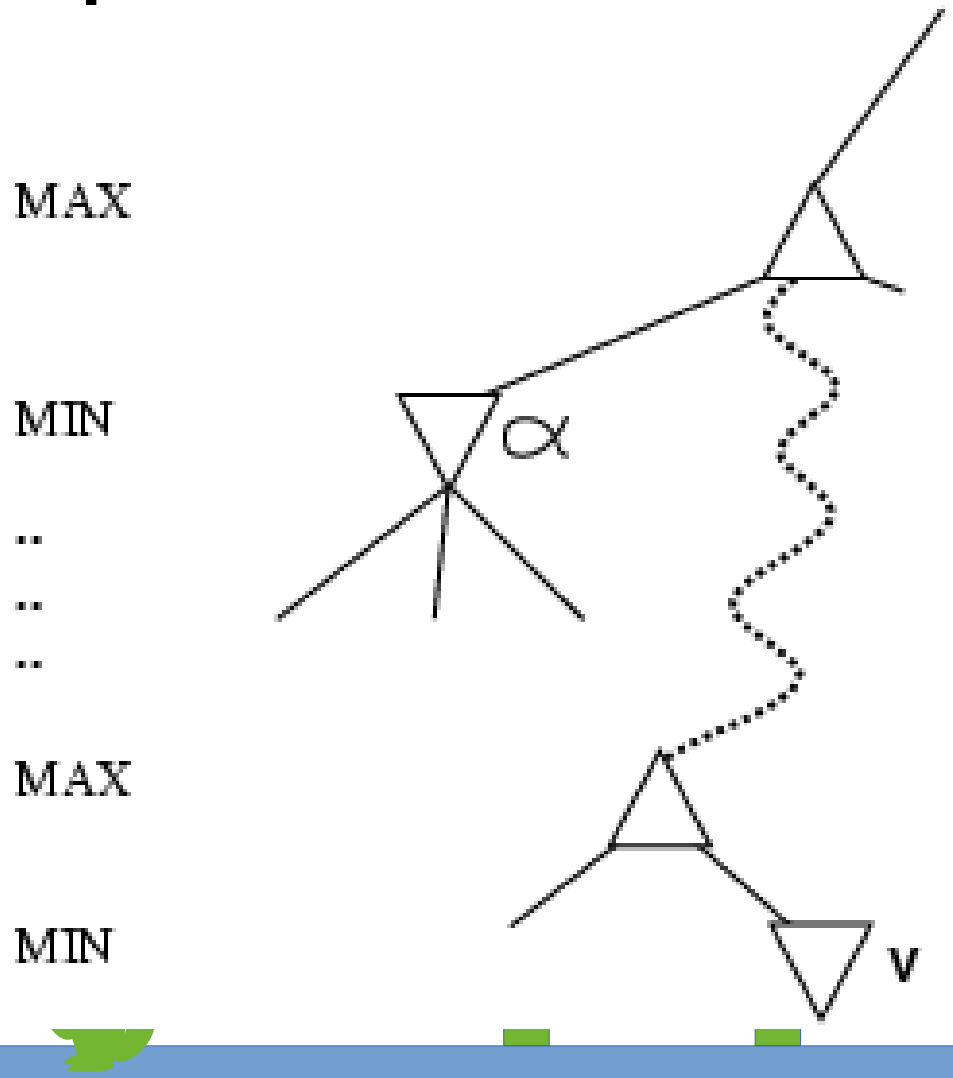


α-β pruning example



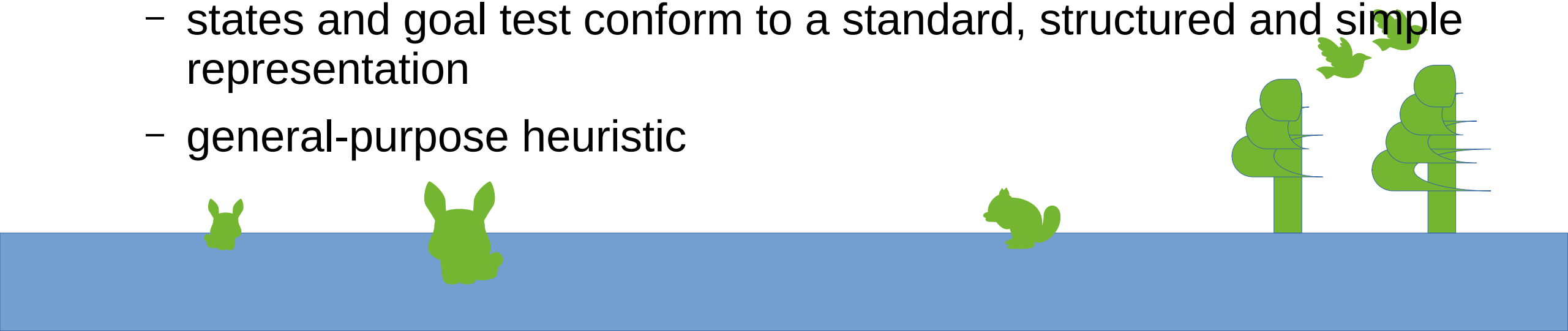
Basic Idea of α-β

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for max
- If v is worse than α, max will avoid it
  - prune that branch
- Define β similarly for min



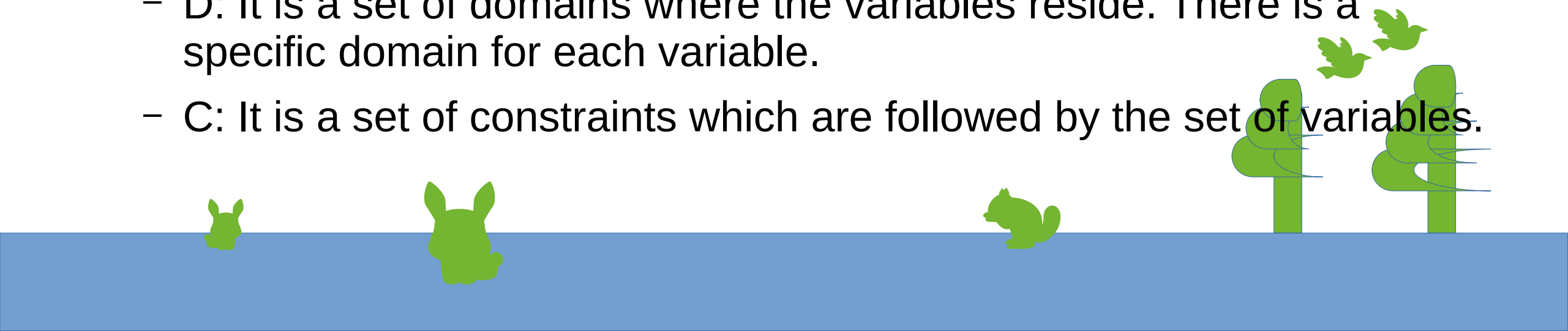
Constraint Satisfaction Problems

- Previous search problems:
  - Problems can be solved by searching in a space of states
  - state is a "black box" – any data structure that supports successor function, heuristic function, and goal test – problem-specific
- Constraint satisfaction problem:
  - states and goal test conform to a standard, structured and simple representation
  - general-purpose heuristic



Constraint Satisfaction Problems

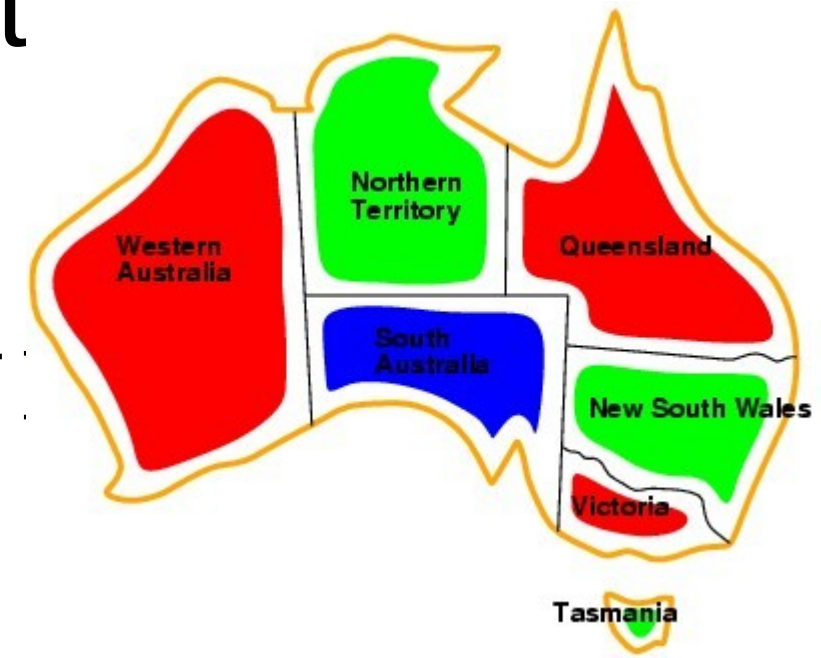
- It is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem.
- Constraint satisfaction depends on three components, namely:
  - X: It is a set of variables.
  - D: It is a set of domains where the variables reside. There is a specific domain for each variable.
  - C: It is a set of constraints which are followed by the set of variables.



Example: Map Coloring

- Variables: X = {WA, NT, Q, NSW, V, SA, T }
- Domains: Di = {red, green, blue}
- Constraints: adjacent regions must have different colors

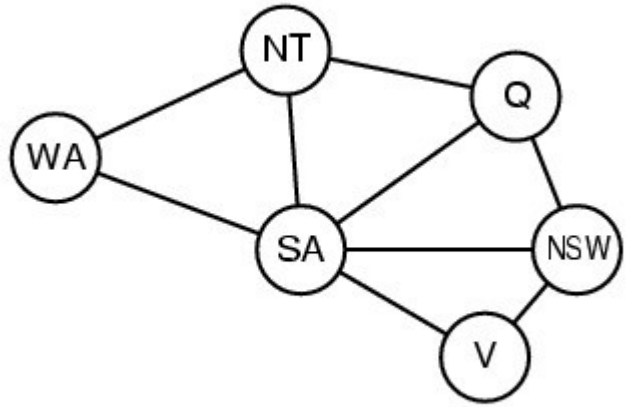
Solution: Complete and Consistent Assignment



- Variables: X = {WA, NT, Q, NSW, V, SA, T }
- Domains: Di = {red, green, blue}
- Constraints: adjacent regions must have different colors
- Solution? {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}.

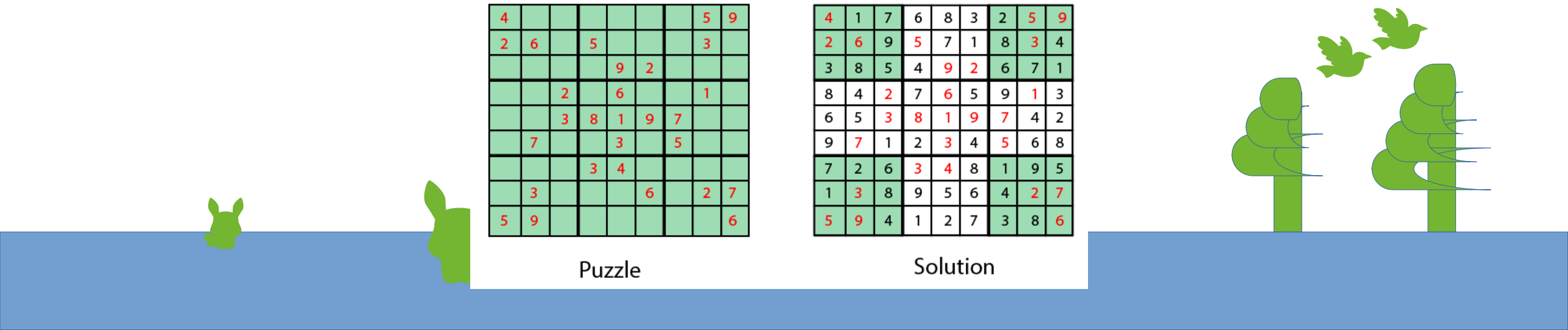
Constraint Graph

- Constraint graph: nodes are variables, arcs are constraints
- Binary CSP: each constraint relates two variables
- CSP conforms to a standard pattern
  - a set of variables with assigned values
  - generic successor function and goal test
  - generic heuristics
  - reduce complexity



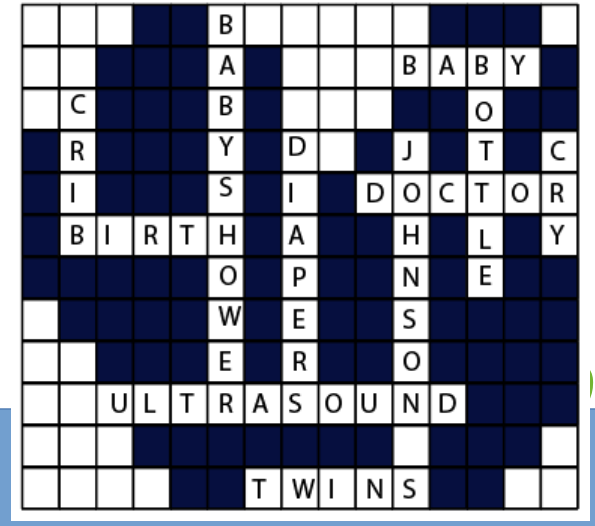
CSP Problems

- Graph Coloring: The problem where the constraint is that no adjacent sides can have the same color.
- Sudoku Playing: The gameplay where the constraint is that no number from 0-9 can be repeated in the same row or column.



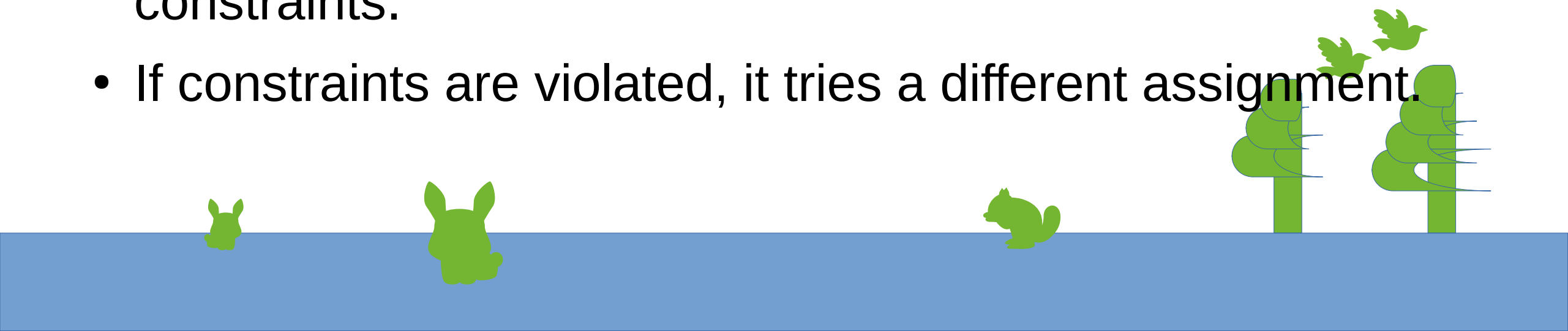
CSP Problems

- n-queen problem: In n-queen problem, the constraint is that no queen should be placed either diagonally, in the same row or column.
- Crossword: In crossword problem, the constraint is that there should be the correct formation of the words, and it should be meaningful.



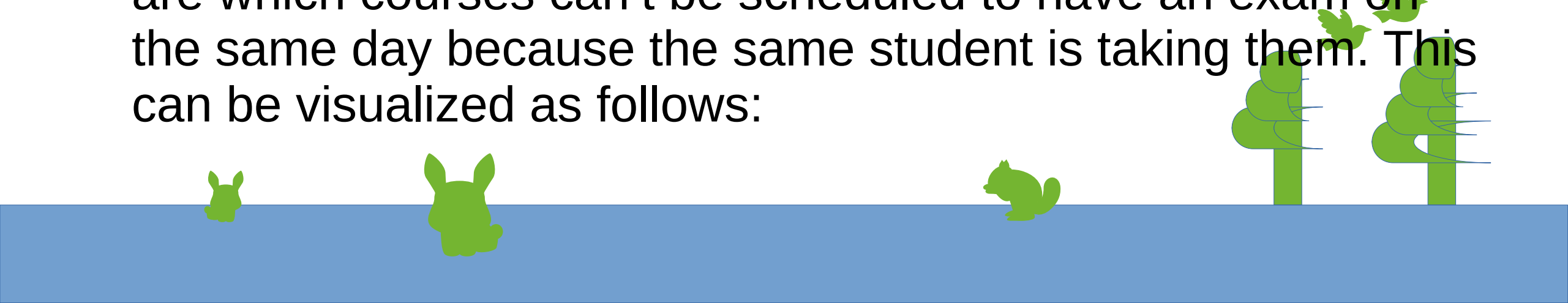
Backtracking Search

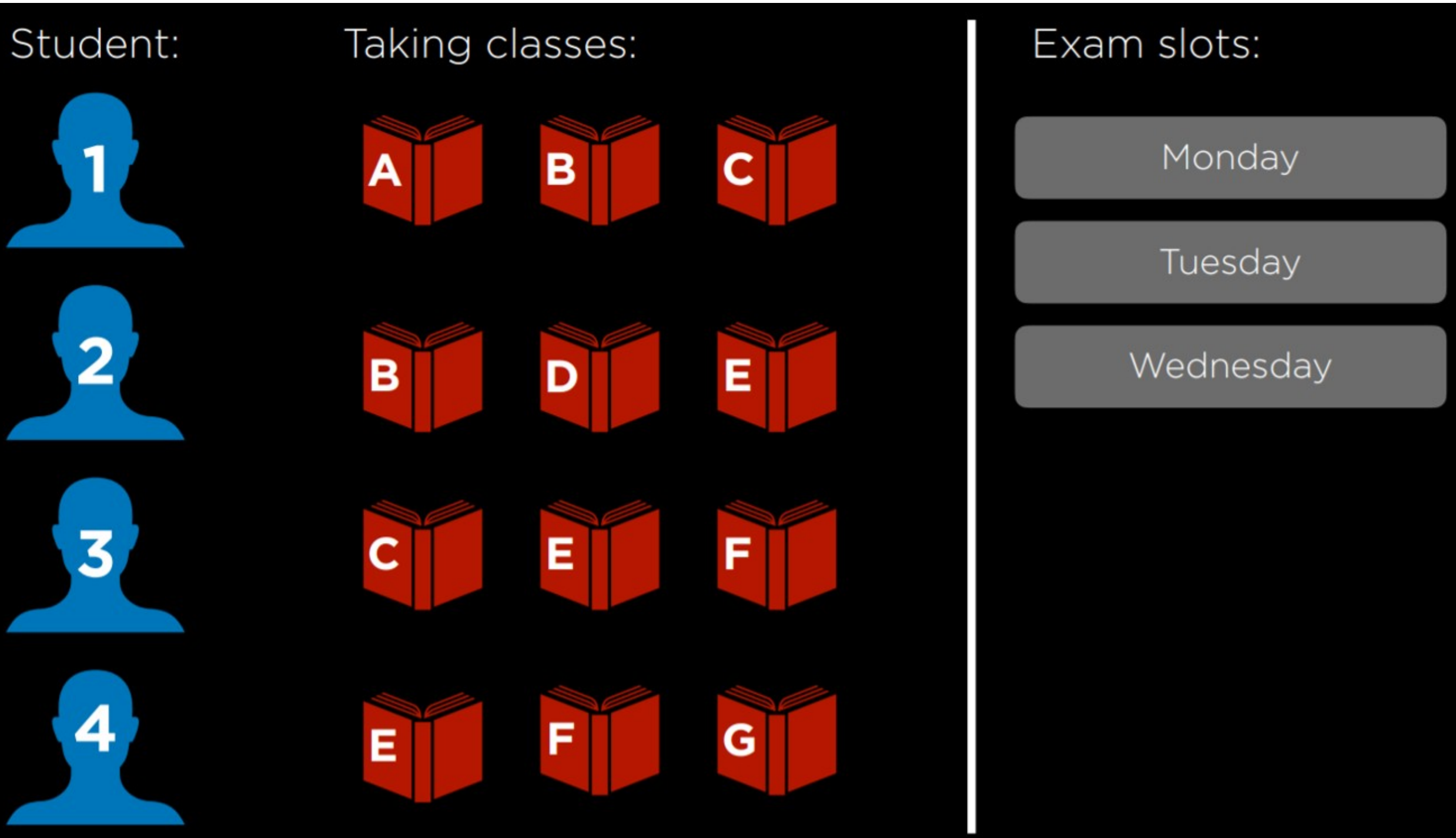
- Backtracking search is a type of a search algorithm that takes into account the structure of a constraint satisfaction search problem.
- In general, it is a recursive function that attempts to continue assigning values as long as they satisfy the constraints.
- If constraints are violated, it tries a different assignment.



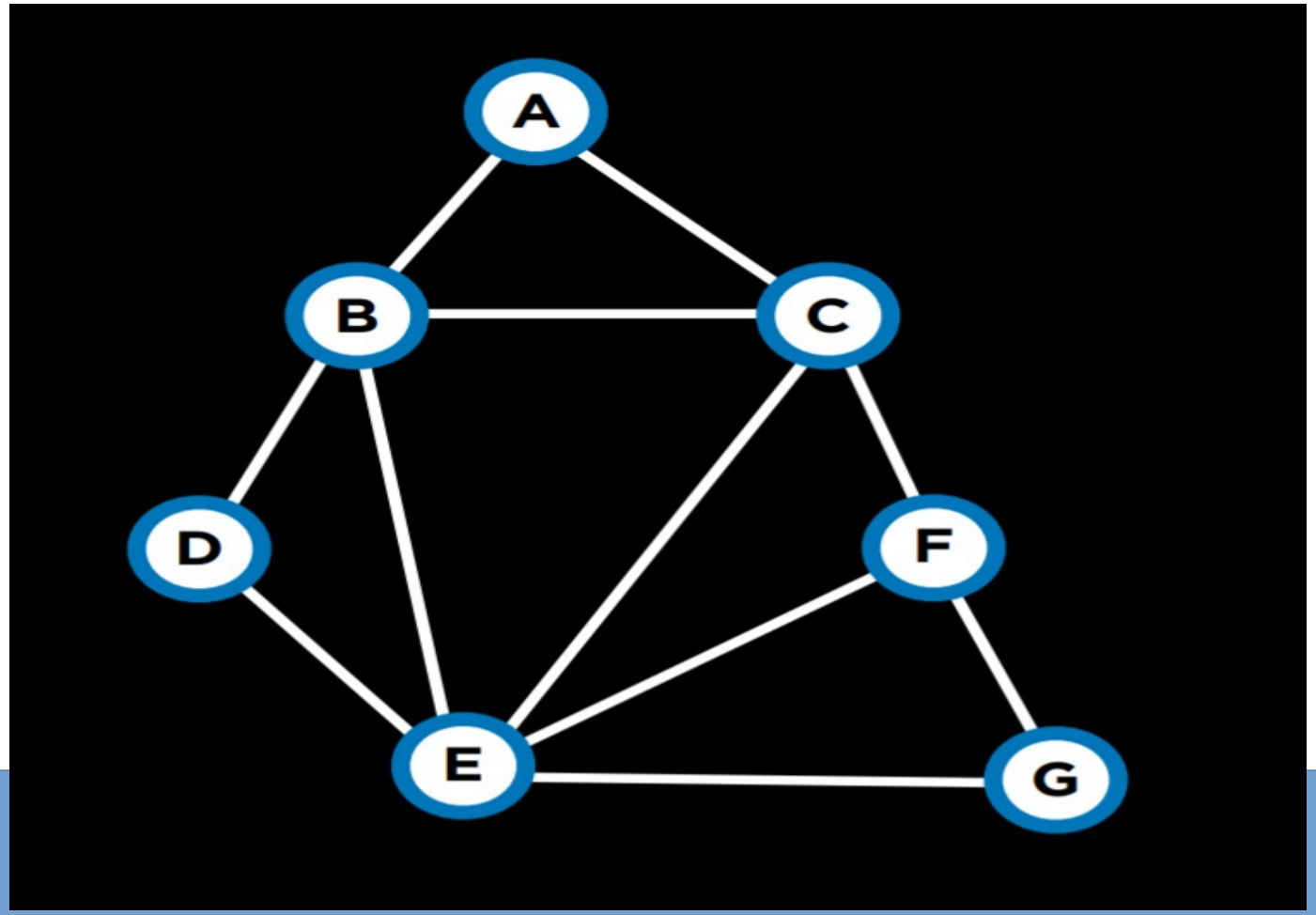
Example

- Consider another example. Each of students 1-4 is taking three courses from A, B, …, G. Each course needs to have an exam, and the possible days for exams are Monday, Tuesday, and Wednesday. However, the same student can't have two exams on the same day. In this case, the variables are the courses, the domain is the days, and the constraints are which courses can't be scheduled to have an exam on the same day because the same student is taking them. This can be visualized as follows:




Student:
Taking classes:
Exam slots:
1
A
B
C
2
B
D
E
3
C
E
F
4
E
F
G
Monday
Tuesday
Wednesday

This problem can be solved using constraints that are represented as a graph. Each node on the graph is a course, and an edge is drawn between two courses if they can't be scheduled on the same day. In this case, the graph will look this:



Variables: {A, B, C, D, E, F, G}
Domains: {Monday, Tuesday, Wednesday}
for each variable
Constraints: {A≠B, A≠C, B≠C, B≠D, B≠E, C≠E, C≠F, D≠E, E≠F, E≠G, F≠G}

END