AI & Robotics

Blind Search

AI & ROBOTICS LAB

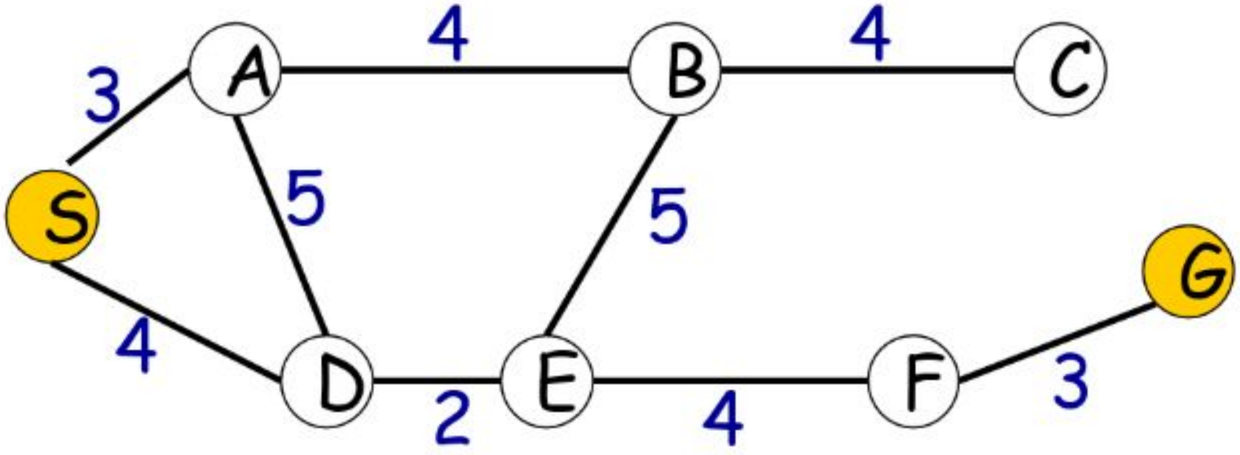
Goals

The **junior-colleague**

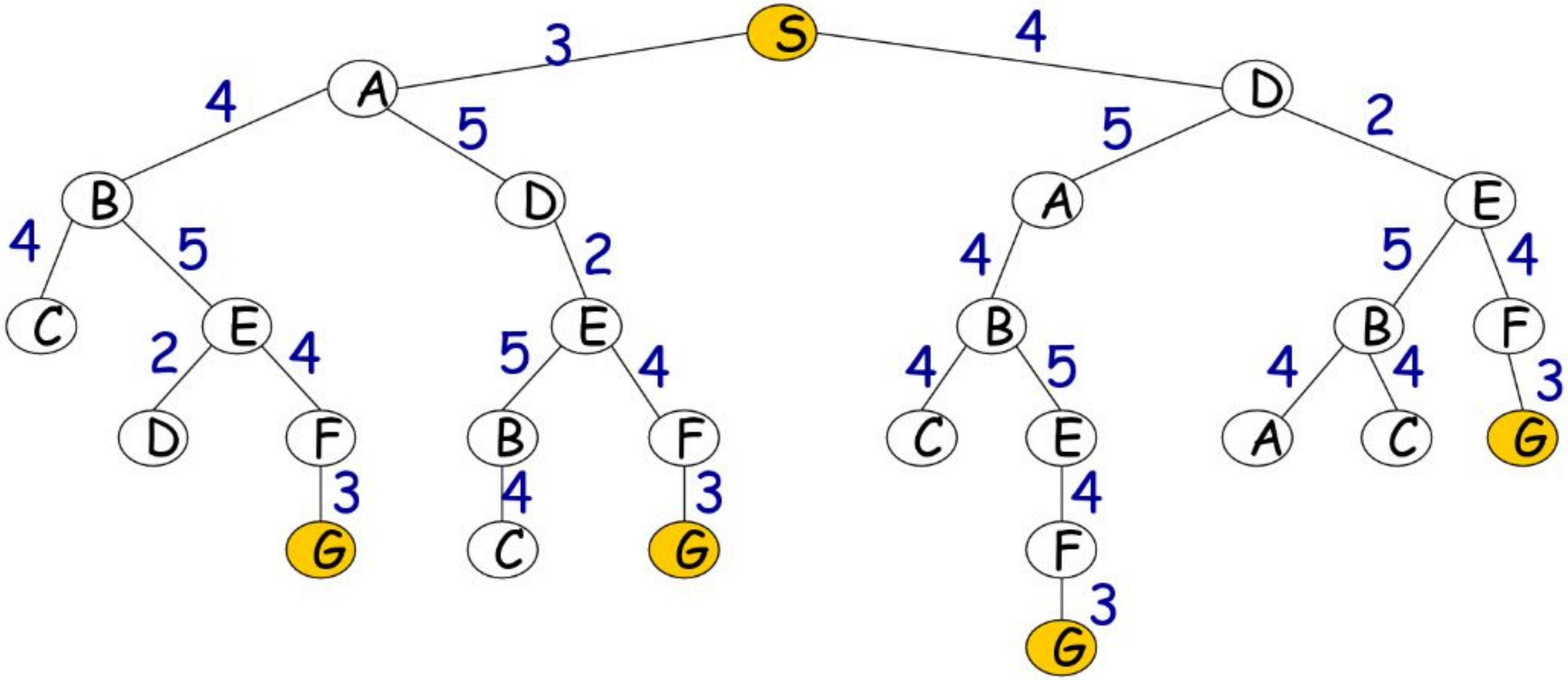
- can describe and transform a graph with a start and end node into a partial paths tree representation
- can explain and implement depth-first search
- can analyze the completeness of depth-first search
- can evaluate the time complexity of depth-first search
- can evaluate the space complexity of depth-first search
- can explain and implement breadth-first search
- can analyze the completeness of breadth-first search
- can evaluate the time complexity of breadth-first search
- can evaluate the space complexity of breadth-first search
- can explain and implement iterative deepening search
- can analyze the completeness of iterative deepening search
- can evaluate the time complexity of iterative deepening search
- can evaluate the space complexity of iterative deepening search
- can compare the different blind search techniques

Representation

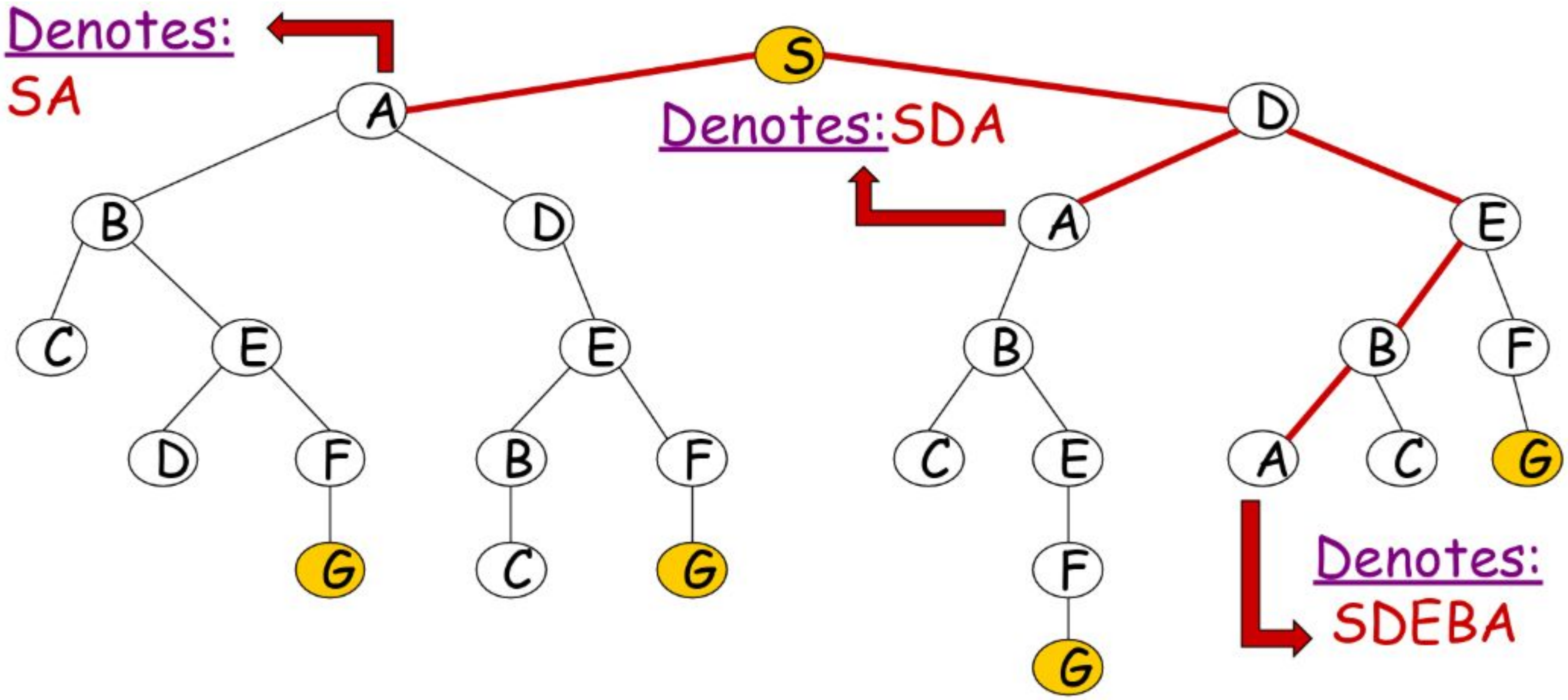
- Running example



Representation: loop-free tree of partial paths



Representation: loop-free tree of partial paths

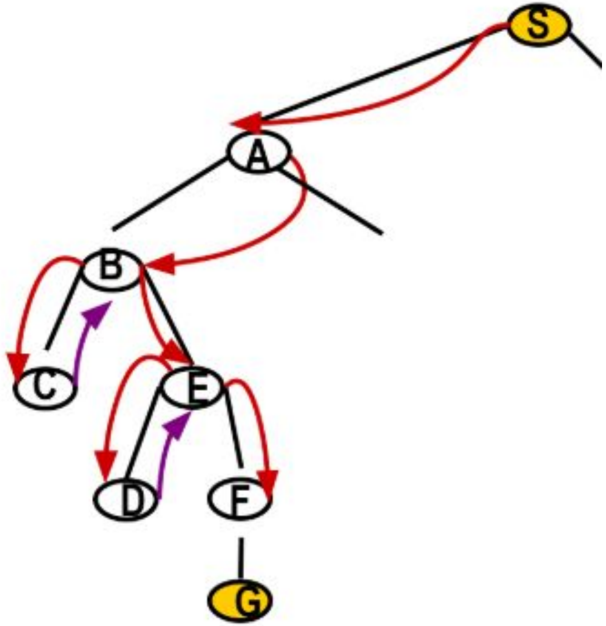


We are not interested in optimal paths *for now*, so we drop the costs

Depth-first Search

Depth-first search

1. Start at the root node
2. Select a child (convention: left-to-right)
3. Explore as far as possible along child branch
4. Backtrack: return to upper levels
5. Go back to step 2



Depth-first search

```
function depthFirst(G,v):
   stack.push(v)
   while (stack is not empty
            and goal is not reached)
      v = stack.pop()
      if v not discovered:
         label v as discovered
      for all edges from v to w in
            G.adjacentEdges(v) do
             stack.push(w)
```

```
function depthFirst(G,v):
   label v as discovered
   for all edges from v to w in
           G.adjacentEdges(v) do
       if vertex w not discovered
 then
     depthFirst(G,w)
```

Depth-first search

Analysis

- Completeness
- Worst case complexity (time and space)
  - We do not take the built-in loop-detection into account.
  - We do not take the length (space) of representing paths into account
  - Multiple ways to look at this, we evaluate based on these terms:

    d = depth of the tree

    b = (average) branching factor of the tree

    m = depth of the shallowest solution

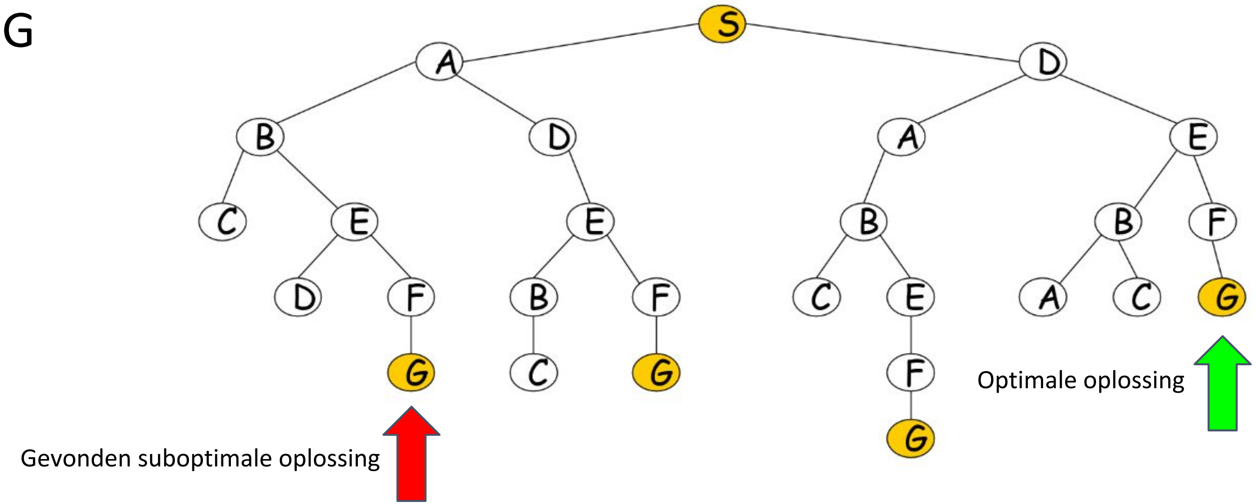
Depth-first search

Completeness

- Complete for finite graphs (finite amount of nodes):

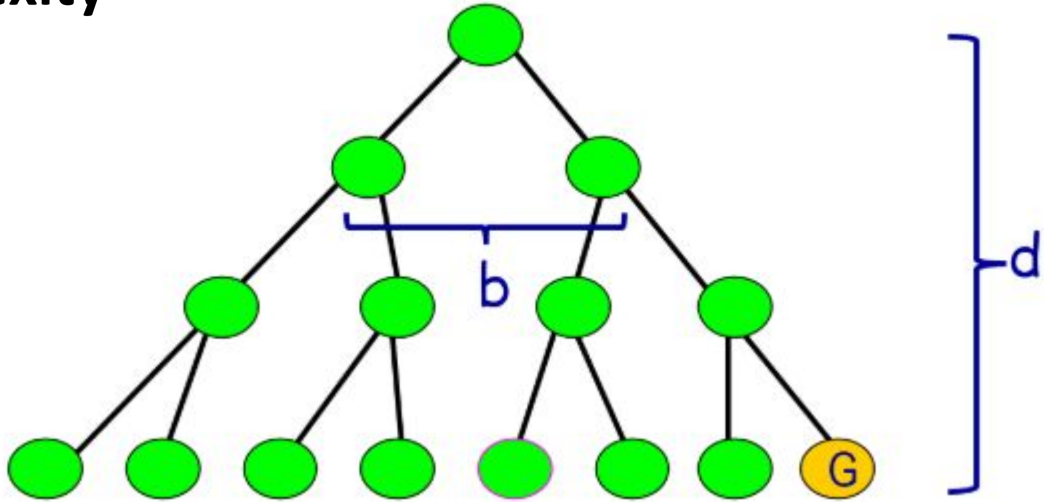
  => Always finds a path
- Does not necessarily find the shortest path

  => SABEFG vs. SDEFG



Depth-first search

Time complexity

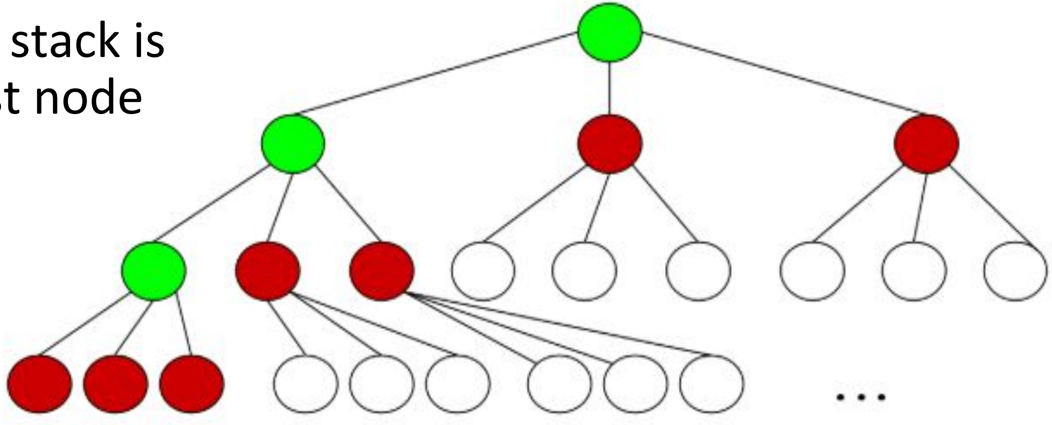


$O(b^d + b^{d-1} + b^{d-2} + ... + 1) = \mathbf{O(b^d)}$

Depth-first search

Space complexity

- Largest number of nodes in stack is reached at bottom left-most node
- Example: (b = 3, d = 3)



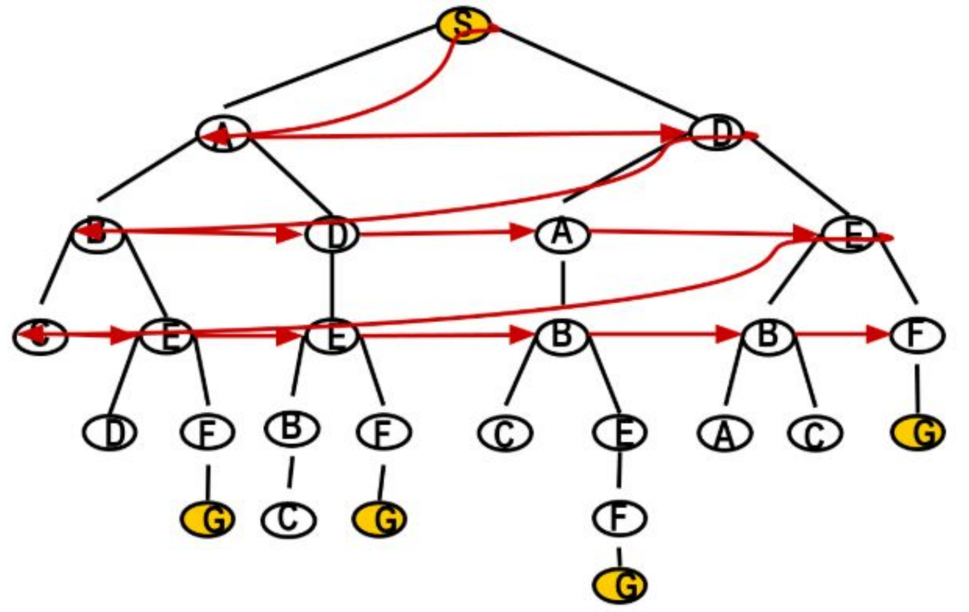
Stack contains all nodes => 7

O((b-1)*d + 1) = **O(b*d)**

Breadth-first Search

Breadth-first search

1. Start at the root node
2. Go to the next level
3. Iterate over the nodes at this depth level
4. Go to step 2



Breadth-first search

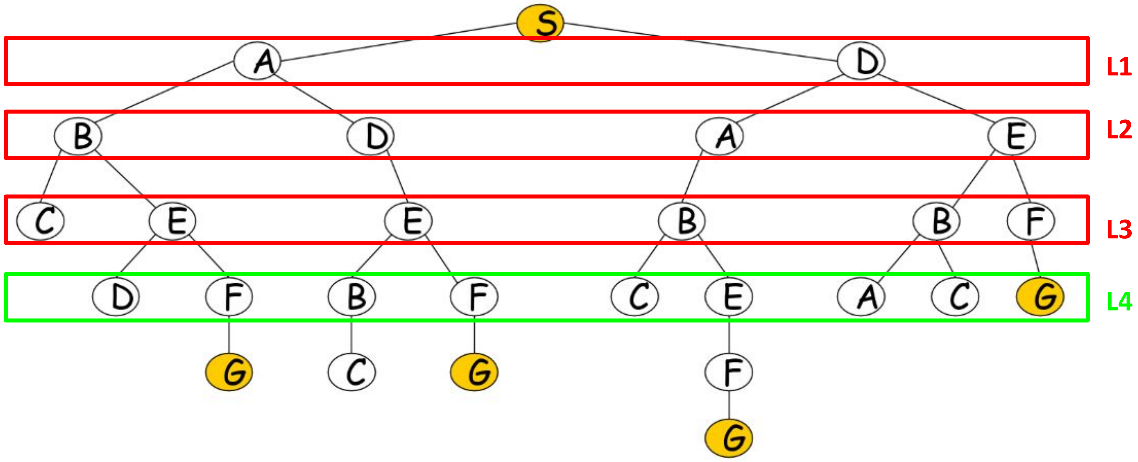
```
function breadthFirst(G,v):
    queue.push(v)
    while (queue is not empty and goal is not reached)
        v = queue.pop()
        if v not discovered:
            label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
            queue.push(w)
```

Only difference!

Breadth-first search

Completeness

- Complete, even for infinite graphs
  => Always finds a path
- Complete, even without loop-detection
- **<u>Always</u>** finds the shortest path



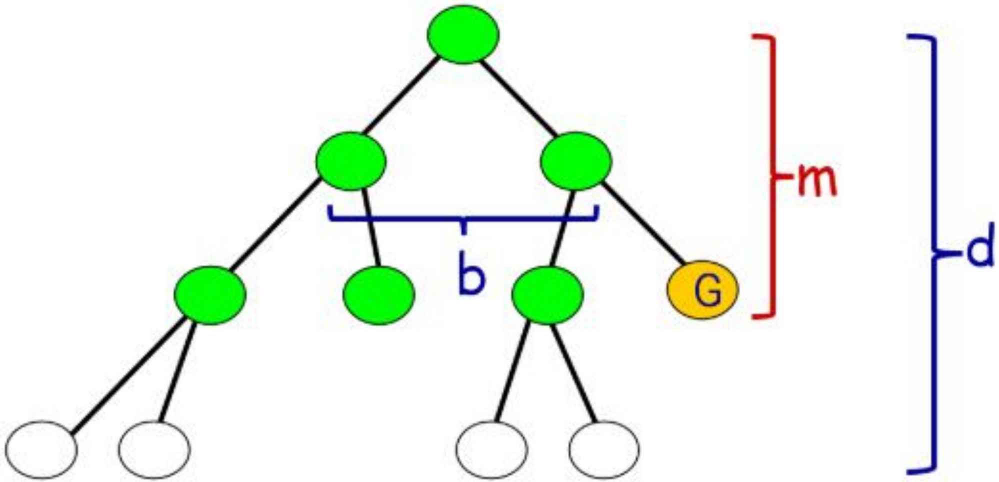
Breadth-first search

Time complexity

Goal at depth **m**

=> all nodes until that point have been checked

O(b^m)

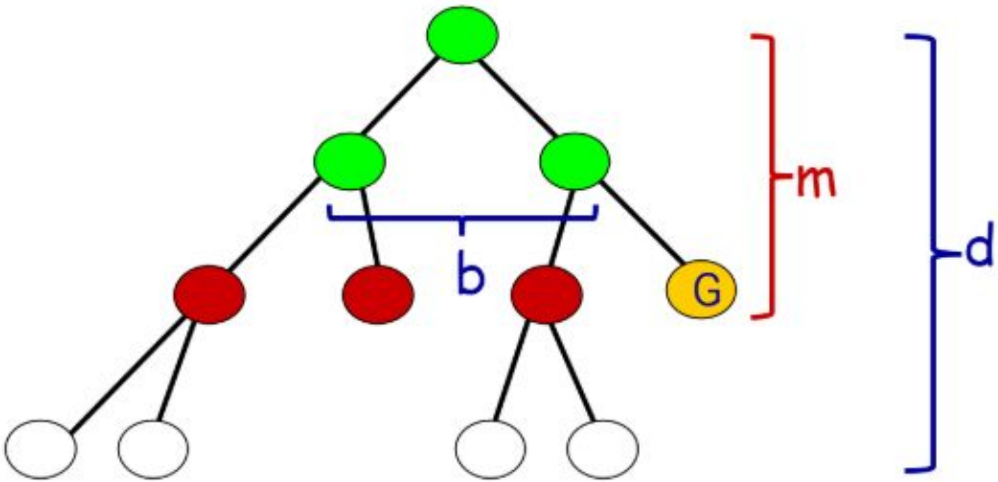


Breadth-first search

Space complexity

Queue contains all and G nodes => 4

O(b^m)



Comparison

	Depth-first	**Breadth-first**
Time complexity	$O(b^d)$	$O(b^m)$
Space complexity	$O(b*d)$	$O(b^m)$

b: branching factor
d: depth of the search tree
m: depth of the shallowest solution

- Depth-first
  - search space contains very deep branches without solution

    => bad time complexity
- Breadth-first
  - High memory demands

[*https://seanperfecto.github.io/BFS-DFS-Pathfinder/*](https://seanperfecto.github.io/BFS-DFS-Pathfinder/)

Iterative deepening Search

Iterative deepening search

- Restrict a depth-first search to a fixed depth.
- If no path was found, increase the depth and restart the search.

=> best of both depth-first and breadth-first

Iterative deepening search

```
function deep(G,v,depth):
   stack.push(v)
   while (stack is not empty
           and goal is not reached):
       v = stack.pop()
       if v not discovered:
           label v as discovered
           if path_length < depth
           for all edges from v to w in
                  G.adjacentEdges(v) do
                   stack.push(w)
```

```
function iterativeDeep(G,v):
   depth: 1

   while goal is not reached do
        perform deep(G,v,depth)
        increase depth by 1
```

Iterative deepening search

Completeness

- Complete

  => **<u>Always</u>** finds the shortest path (like breadth-first)

Iterative deepening search

Time complexity

If the path is found for Depth = **m**, how much time spent on all **< m** trees?

$O(b^{m-1} + b^{m-2} + b^{m-3} + \ldots + 1) = \mathbf{O(b^{m-1})}$

Time spent on Depth = **m**:

$\mathbf{O(b^{m})}$

Space complexity

$\mathbf{O(b*m)}$

Comparison

	Depth-first	**Breadth-first**	**Iterative deepening**
Time complexity	$O(b^d)$	$O(b^m)$	$O(b^m)$
Space complexity	$O(b*d)$	$O(b^m)$	$O(b*m)$

b: branching factor
d: depth of the search tree
m: depth of the shallowest solution