COM3103

Artificial Intelligence

4. Uninformed Search:

*Depth First Search* and *Breadth First Search*

Searching

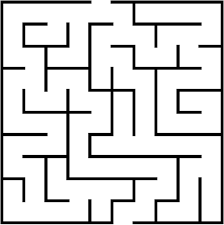
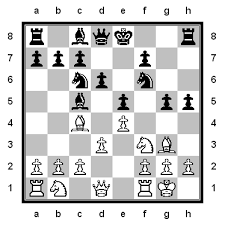
* Searching are the techniques for systematically finding or constructing solutions to problems.
* Basic technique: *generate-and-test*.

1. Generate (or find) a possible candidate solution.
2. Test the candidate solution.
   1. IF solution is found, announce the result
   2. ELSE go to step 1.

Searching in AI

* Searching techniques are used in many AI problems
* A picture containing text, table, desk

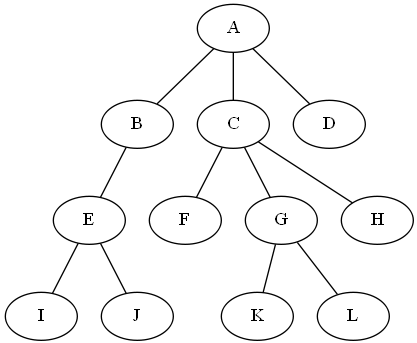
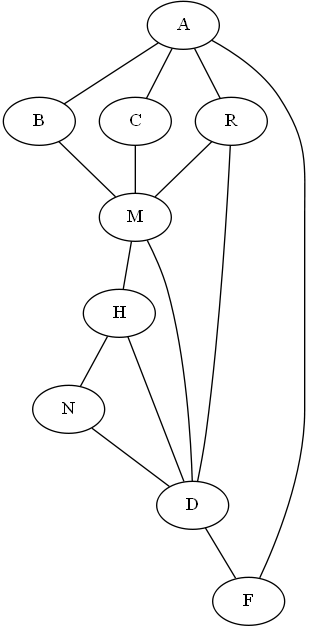
  Description automatically generatedExamples
  + Path planning
  + Two players Games
  + Natural Language Processing

Basic of Searching in AI: Trees and Graphs

* **A close-up of some leaves

  Description automatically generated with low confidence**Many searching algorithms in AI are based on trees and graphs

**Tree Graph**



Tree

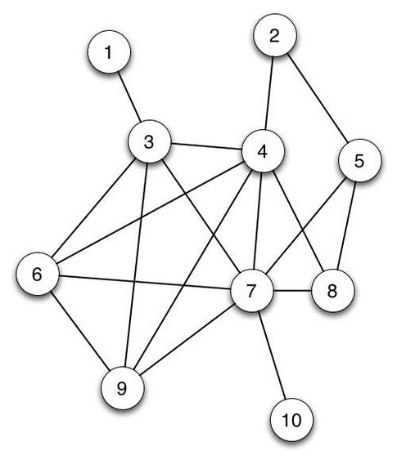
* A tree has a number of nodes (A *root node*, leaf nodes, and other nodes)
* Each node may have links leading to other nodes.
* Each link connects a *parent* node with a *child* node
* The route from the *root node* to a certain node is called its path
* A tree cannot contain cycles (only one path form root to any node)

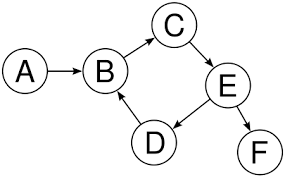


Graph

* A graph also consists of nodes and links
* However, cycles (loops) are allowed in graphs (which can be problematic for tree-based searching algorithms)
* Examples
  + Road networks in a country

**Undirected graph**

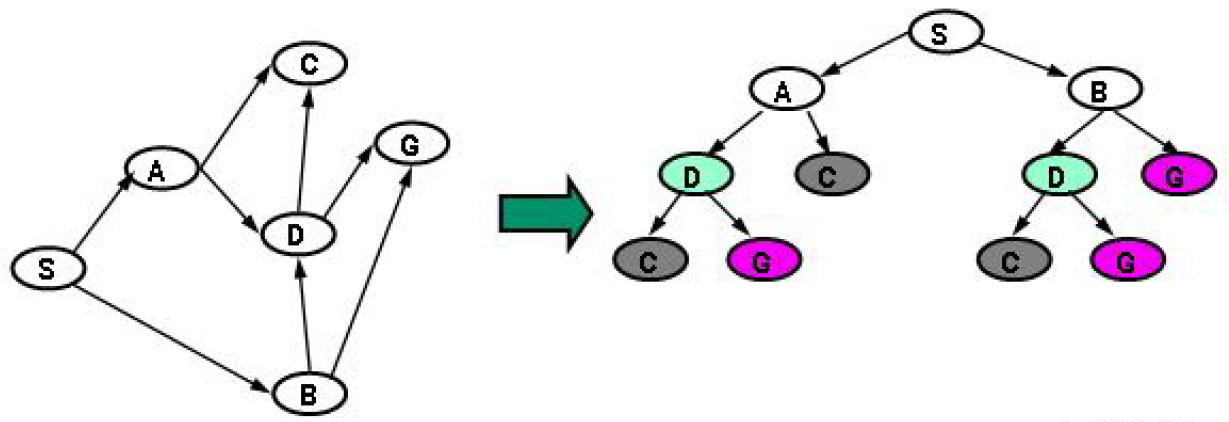
* + Computer networks
  + Social media friend connections

****

**Directed graph**

Graphs and Trees

* Many searching algorithms are tree-based (no loops allowed).
* However, many problems are actually graphs.
* Solution: represent a graph using a tree, repeating some nodes if necessary

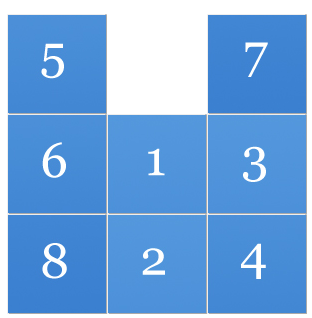


* + The tree on the right is a tree-representation of the graph on the left (note that some nodes are repeated).

Representing Problems using Trees (or Graphs)

Example: 8-puzzle problem

We want to find a way to transform an initial state (e.g., the example on the left) to a goal state (the example on the right) by applying a series of operators (e.g., moving an adjacent number into the blank square)



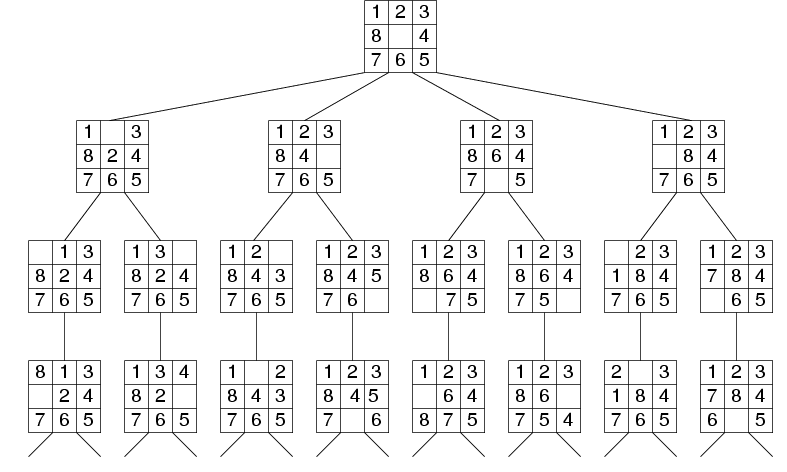
**Goal State**

**Initial State**

Try it out:

<https://murhafsousli.github.io/8puzzle/#/>

* In this case, we are interested in the path that leads from the root to a goal state (hidden somewhere down the tree below)
* Node: a possible state. Link: operation (i.e., moving a number).

 Tree:

* + Note: We can only examine one node (state) at a time!

8-puzzle problem

Input:

* What does each node represent?
  + A state of the puzzle (the position of each piece (0 – 8, blank))
* Start state (The root node)
  + A given starting position (varies case by case)
* Operators
  + Swapping the blank square with a neighbour square (UP/DOWN/LEFT/RIGHT)
  + Cost per move: uniform cost (1 per move)
* Test for goal state
  + Are the pieces in pre-defined goal positions?

Output:

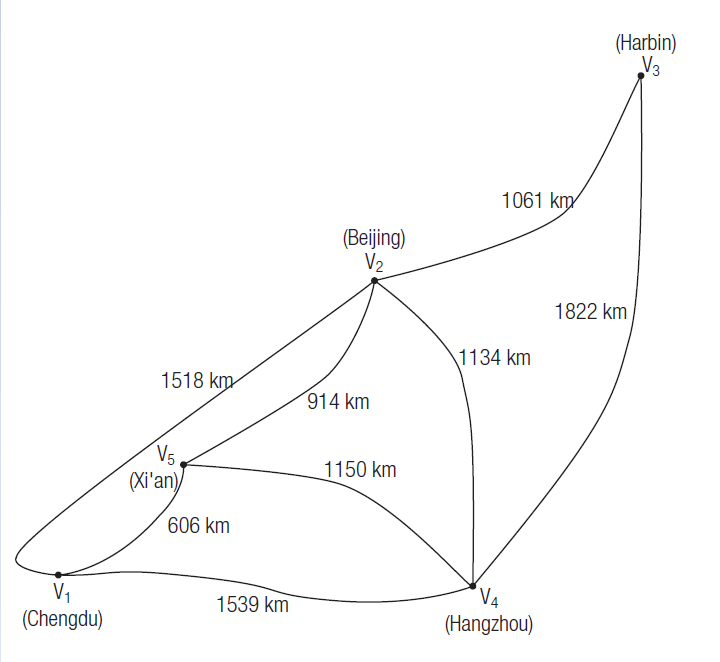
* A path from start state to goal state

[Some problem may require the shortest path]

Example: Route planning problem

E.g., find the shortest path from Chengdu to Harbin

Graph:



Route planning problem

Input:

* What does each node represent?
  + A state of the puzzle (the location of the traveller)
* Start state (The root node)
  + A given city (e.g., Chengdu)
* Operators
  + Travel to a connected city
  + Cost per move: the distance travelled
* Test for goal state
  + The current state equals some given goal state (e.g., Harbin)

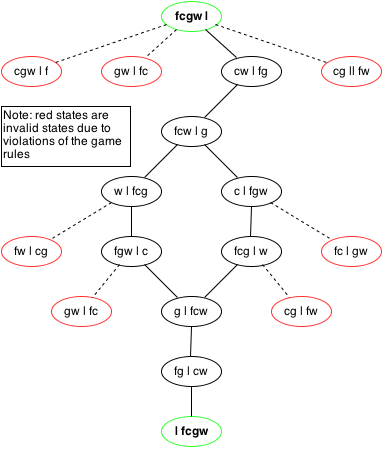
Output:

* + Path: a path from start-state (e.g., Chengdu) to goal-state (e.g., Harbin)
  + Shorter paths are more preferable

Example: *Farmer, Goat, Wolf and Cabbage problem* revisited

A farmer who is carrying a wolf, a goat and some cabbage needs to cross a river by boat. However, the boat is so small that he can carry only one item on the boat at a time. If left unattended, the wolf would eat the goose, and the goose would eat the cabbage. How can the farmer carry all items across the river?

Farmer, Goat, Wolf and Cabbage problem

Graph:

Input:

* What does each node represent?
* A state of the puzzle (Position of the farmer, goat, wolf and cabbage)
* Start state (The root node)
  + Farmer and all items on the left
* Operators
  + Famer travel to the other side, either alone or carrying one item
  + Cost per move: uniform (1 per move)
* Test for goal state
  + Farmer and all items on the Right

Output:

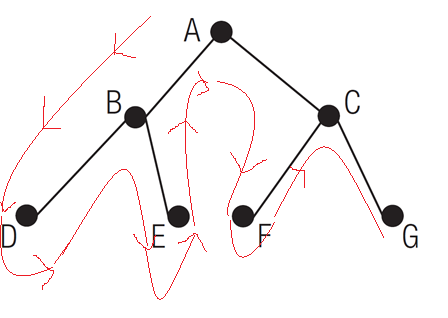
* A path from start state to goal state (with shorter paths more preferable).

Common Searching Algorithms

* Uninformed search (blind search)
  + - Given no extra knowledge on how to find the solution.
    - Examples
      * *Depth first search*
      * *Breadth first search*
* Informed search (heuristic)
  + - Utilize problem-specific knowledge to speed up the search
    - Example: Hill climbing

Depth First Search (DFS)

Try to follow a certain branch of the tree as deep as possible, before trying the next branch

For example: DFS in this example would inspect the nodes in the order

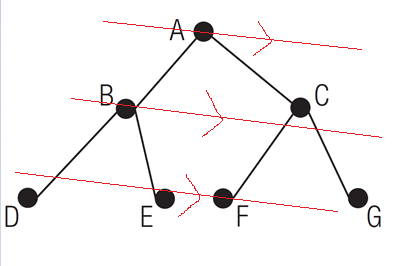
[A, B, D, E, C, F, G]

Or, alternatively, from right to left as

[A, C, G, F, B, E, D]

Breadth First Search (BFS)

Nodes are visited level by level (from top to bottom, from left to right OR from left to right)

BFS would inspect the nodes in the order

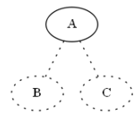
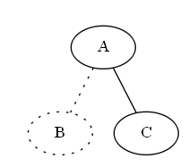
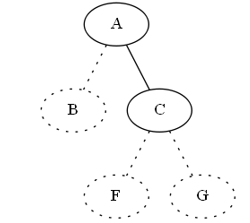
A, B, C, D, E, F, G

or, Alternatively, from right to left as

A, C, B, G, F, E, D

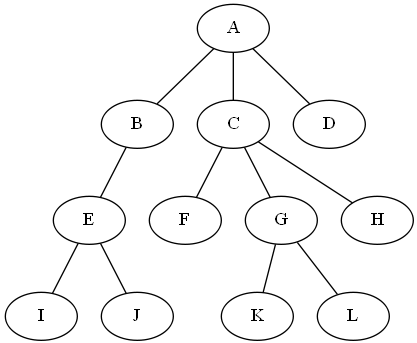
Note:

* We can only inspect one node at a time.
* We can only check whether a node is the goal state when we get there.
* We often do not know what further options are available (i.e., whether a node has children) until we arrive at the node.



Exercise 1

Show the order of visiting each node if the tree is searched using i) Depth First Search, ii) Breadth First Search



DFS:

[A, B, E, I, J, C, F, G, K, L, H, D]

or, from right to left:

[A, D, C, H, G, L, K, F, B, E, J, I]

BFS

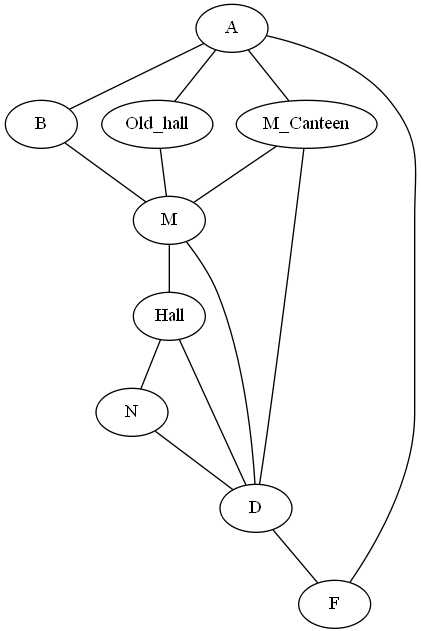
[A, B, C, D, E, F, G, H, I, J, K, L]

or from right to left:

[A, D, C, B, H, G, F, E, L, K, J, I]

Exercise 2:

Consider the following graph representation of the HSUHK campus.

1. Convert the graph into tree format
2. Show the search order using DFS and BFS

Tree Searching Algorithm

(Basic version)

1. Start at the root node (called the *next node*).
2. Setup a list of *visited* nodes (initially empty)
3. Setup a list of *to\_visit* nodes (initially empty)
4. Repeat until SUCCESS or FAILURE:
   1. If the *next node* has NOT been visited before:
      1. Add the *next node* to *visited* nodes list
      2. If it is the goal, report SUCCESS, and stop.
      3. If it is not the goal, add its children to the end of the *to\_visit* list
5. Look at the *to\_visit* list
6. If there are more nodes in the list

Remove one of the nodes from the list and select it as the *next node*

1. Else if no more nodes in the list

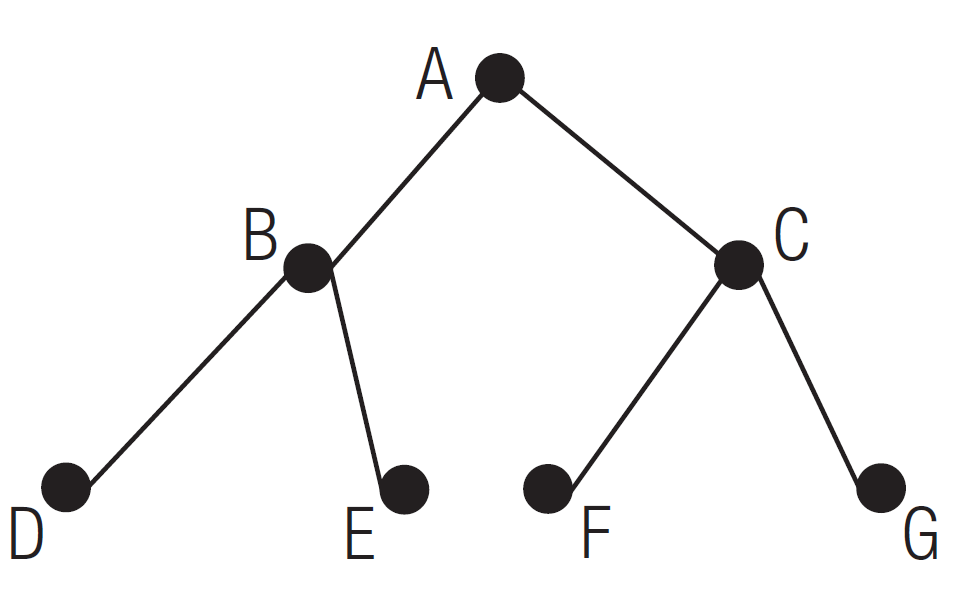
Report FAILURE

Note: more formally:

* + - *to\_visit* is also known as the ***frontier set***
    - *visited* is also known as the ***explored set***
    - *finding a node’s children* is called ***expanding*** the node

Tree Searching Algorithm (Basic version)

We will now illustrate the idea using this tree



Start

Goal

Tree Searching algorithm: Basic version

1. Start at the root node (called the *next node*).
2. Setup a list of *visited* nodes (initially empty)
3. Setup a list of *to\_visit* nodes (initially empty)

**Tree:**

* **next\_node = A**
* **visited = [ ]**
* **to\_visit = [ ]**

Tree Searching algorithm: Basic version

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**[Not shown here]**

**Tree:**

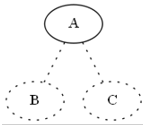
* **next\_node = A**
* **visited = [A]**
* **to\_visit = [ ]**

Tree Searching Algorithm: Basic Idea

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**[Not shown here]**

**Tree:**

* **next\_node = A**
* **visited = [A]**
* **to\_visit = [B,C]**

Tree Searching Algorithm: Basic Idea

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

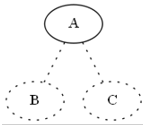
**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove one of the nodes from the list and select it as the *next node*

1. Else If no more nodes in the list

**Tree:**

Report FAILURE

* **next\_node = ?**
* **visited = [A]**
* **to\_visit = [ ? ] #(was: [B, C])**

1. If there are more nodes in the list

Remove one of the nodes from the list and select it as the *next node*

**Question:** which node to remove from the **to\_visit** List?

The first one? The last one? Random?

**Answer:** It depends:

DFS: remove the last one (i.e., from the END of the LIST) (a stack!)

*i.e. nodes from the next level will be explored first (if any)*

BFS: remove the first one (i.e., from the FRONT of the LIST)(a queue!)  
 *i.e. nodes from the same level will be explored first (if any)*

Depth First Search Algorithm

1. Start at the root node (called the *next node*).
2. Setup a list of *visited* nodes (initially empty)
3. Setup a list of *to\_visit* nodes (initially empty)
4. Repeat until SUCCESS or FAILURE:
   1. If the *next node* has NOT been visited before:
5. Add the *next node* to *visited* nodes list
6. If it is the goal, report SUCCESS, and stop.
7. If it is not the goal, add its children to the end of the *to\_visit* list
8. Look at the *to\_visit* list
9. If there are more nodes in the list

Remove one node from the END of the list and select it as the *next node*

1. Else If no more nodes in the list

Report FAILURE

Breadth First Search Algorithm

1. Start at the root node (called the *next node*).
2. Setup a list of *visited* nodes (initially empty)
3. Setup a list of *to\_visit* nodes (initially empty)
4. Repeat until SUCCESS or FAILURE:
   1. If the *next node* has NOT been visited before:
5. Add the *next node* to *visited* nodes list
6. If it is the goal, report SUCCESS, and stop.
7. If it is not the goal, add its children to the end of the *to\_visit* list
8. Look at the *to\_visit* list
9. If there are more nodes in the list

Remove one node from the FRONT of the list and select it as the *next node*

1. Else If no more nodes in the list

Report FAILURE

We will now continue the previous example using Depth First Search

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

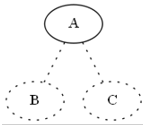
**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the LAST nodes from the list and select it as the *next node*

1. If no more nodes in the list

**Tree:**

Report FAILURE

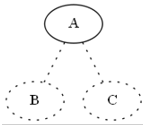
* **next\_node = ?**
* **visited = [A]**
* **to\_visit = [ ? ] #(was: [B, C])**

Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**[Not shown here]**

**Tree:**

* **next\_node = C**
* **visited = [A, C]**
* **to\_visit = [B C]**

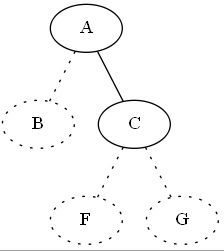
Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**

* **next\_node = C**
* **visited = [A, C]**
* **to\_visit = [B, F, G]**



Tree Searching Algorithm: Depth First Search

4. Repeat until SUCCESS or FAILURE:

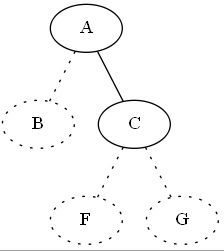
1. If the *next node* has NOT been visited before:

**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the LAST nodes from the list and select it as the *next node*

1. Else If no more nodes in the list

Report FAILURE

**Tree:**

* **next\_node = G**
* **visited = [A, C]**
* **to\_visit = [B,F ] #(was: [B, F, G])**

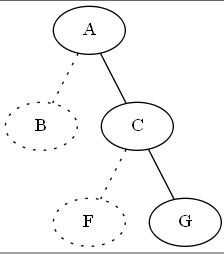
Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**

* **next\_node = G**
* **visited = [A, C, G]**
* **to\_visit = [B, F]**

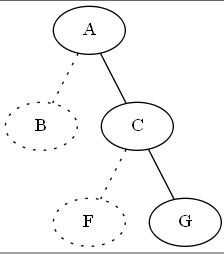


Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**

* **next\_node = G**
* **visited = [A, C, G]**
* **to\_visit = [B, F] (No child to add)**

Tree Searching Algorithm: Depth First Search

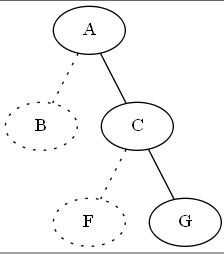
4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the LAST nodes from the list and select it as the *next node*

1. Else if no more nodes in the list

**Tree:**

Report FAILURE

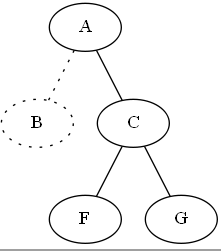
* **next\_node = F**
* **visited = [A, C, G]**
* **to\_visit = [B] (was: B, F)**

Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**

* **next\_node = F**
* **visited = [A, C, G, F]**
* **to\_visit = [B] (F has no child to add)**

Tree Searching Algorithm: Depth First Search

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

**[Not shown here]**

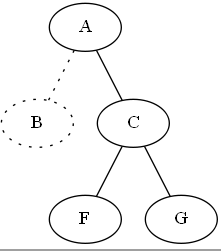
1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the LAST nodes from the list and select it as the *next node*

1. Else if no more nodes in the list

**Tree:**

Report FAILURE

* **next\_node = B**
* **visited = [A, C, G, F]**
* **to\_visit = [ ] (was: B)**

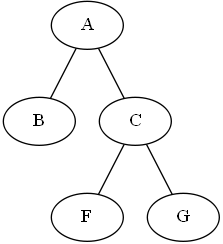
Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**

* **next\_node = B**
* **visited = [A, C, G, F, B]**
* **to\_visit = [ ]**

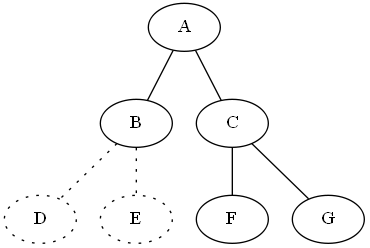


Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**

* **next\_node = B**
* **visited = [A, C, G, F, B]**
* **to\_visit = [D, E]**

Tree Searching Algorithm: Depth First Search

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

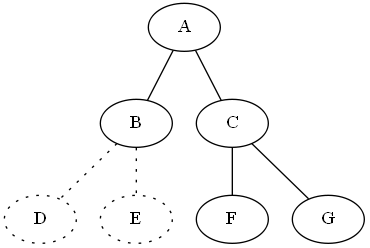
**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the LAST nodes from the list and select it as the *next node*

1. Else if no more nodes in the list

**Tree:**

Report FAILURE  
**next\_node = E**

**visited = [A, C, G, F, B]**

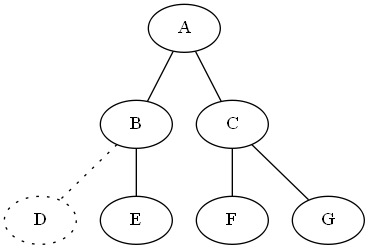
**to\_visit = [D ]**

Tree Searching Algorithm: Depth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**



**next\_node = E (Goal!)**

**visited = [A, C, G, F, B, E]**

**to\_visit = [D ]**

Note: The path from root to goal is [A, B, E]

Goal

Next, we repeat the search using Breadth First Search

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

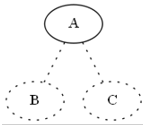
**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the FIRST nodes from the list and select it as the *next node*

1. If no more nodes in the list

**Tree:**

Report FAILURE

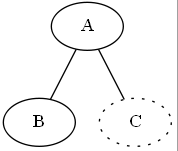
* **next\_node = ?**
* **visited = [A]**
* **to\_visit = [ ? ] #(was: [B, C])**

Tree Searching Algorithm: Breadth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the to\_visit list

**Tree:**

**[Not shown here]**

**next\_node = B**

**visited = [A, B]**

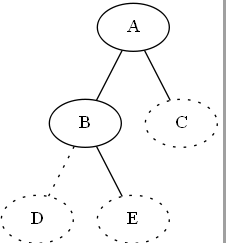
**to\_visit = [ C ]**

Tree Searching Algorithm: Breadth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**



**next\_node = B**

**visited = [A, B]**

**to\_visit = [ C, D, E ]**

Tree Searching Algorithm: Breadth First Search

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

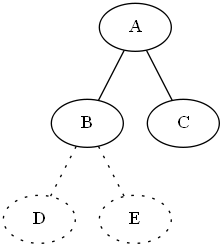
**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the FIRST nodes from the list and select it as the *next node*

1. If no more nodes in the list

**Tree:**

Report FAILURE

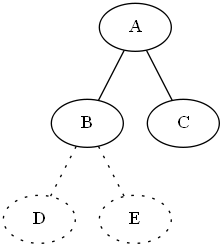
* **next\_node = C**
* **visited = [A, B ]**
* **to\_visit = [ D,E] (was: C, D, E)**

Tree Searching Algorithm: Breadth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the to\_visit list

**Tree:**

**[Not shown here]**



**next\_node = C**

**visited = [A, B, C]**

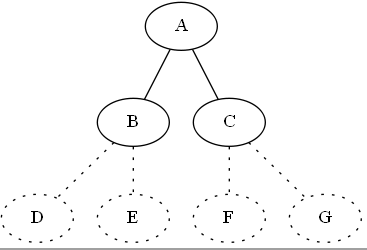
**to\_visit = [ D,E] (was: C, D, E)**

Tree Searching Algorithm: Breadth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the *to\_visit* list

**Tree:**

**[Not shown here]**



**next\_node = C**

**visited = [A,B,C ]**

**to\_visit = [D E,F,G]**

Tree Searching Algorithm: Breadth First Search

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

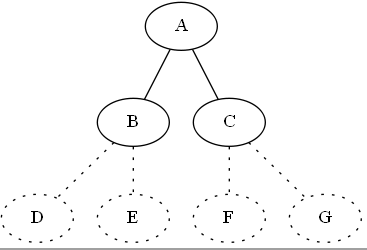
**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the FIRST nodes from the list and select it as the *next node*

1. If no more nodes in the list

**Tree:**

Report FAILURE

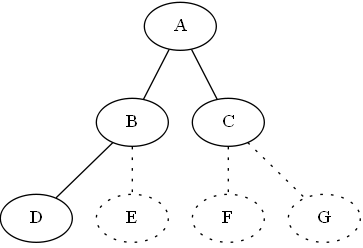
* **next\_node = D**
* **visited = [A, B, C]**
* **to\_visit = [E,F,G] (was:[D,E,F,G])**

Tree Searching Algorithm: Breadth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the to\_visit list

**Tree:**

**[Not shown here]**



**next\_node = D**

**visited = [A, B, C, D]**

**to\_visit = [E, F, G ]**

Tree Searching Algorithm: Breadth First Search

4. Repeat until SUCCESS or FAILURE:

1. If the *next node* has NOT been visited before:

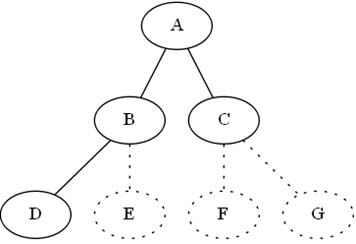
**[Not shown here]**

1. Look at the *to\_visit* list
2. If there are more nodes in the list

Remove the FIRST nodes from the list and select it as the *next node*

1. If no more nodes in the list

**Tree:**

****Report FAILURE

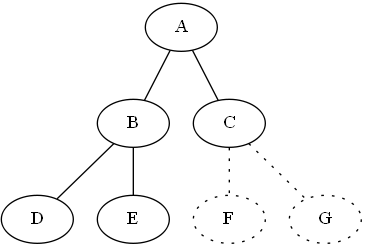
* **next\_node = E**
* **visited = [A, B, C, D ]**
* **to\_visit = [F,G] (was: E, F, G)**

Tree Searching Algorithm: Breadth First Search

1. Repeat until SUCCESS or FAILURE:
2. If the *next node* has NOT been visited before:
3. Add the *next node* to *visited* nodes list
4. If it is the goal, report SUCCESS, and stop.
5. If it is not the goal, add its children to the end of the *to\_visit* list
6. Look at the to\_visit list

**Tree:**

**[Not shown here]**



**next\_node = E GOAL REACHED**

**visited = [A, B, C, D, E]**

**to\_visit = [F, G]**

Goal

Depth First Search Inplementation (Version 1)

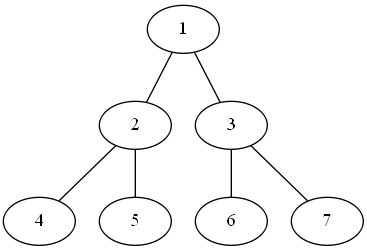
In this example, we representing a tree in a Python program using a dictionary as follow:

graph = {1: [2 , 3],

2: [4, 5],

3: [6, 7]}

goal = 5



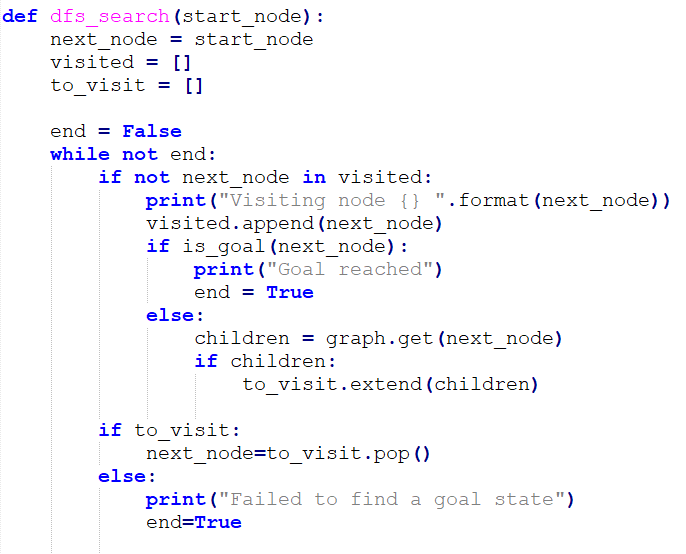
Goal

And a simple fucntion for checking whether a given node (or a state) is the goal:

def is\_goal(x):

return x==goal

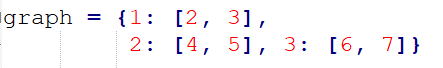
Depth First Search Inplementation (Version 1)

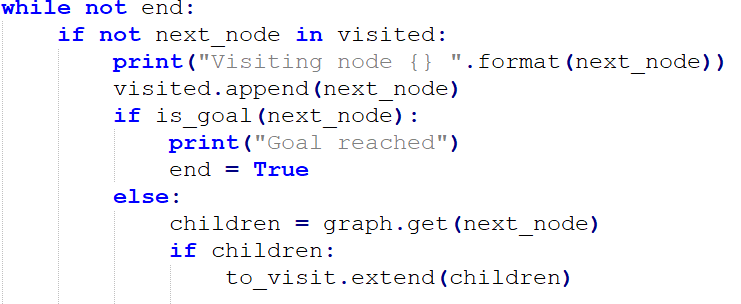
And the search function closely follows the algorithm described above.

* The complete program is listed in Moodle

# 

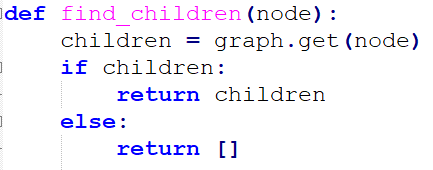
* Note that, in DFS version 1, we are directly refering to the pre-defined dictionary for the graph data:

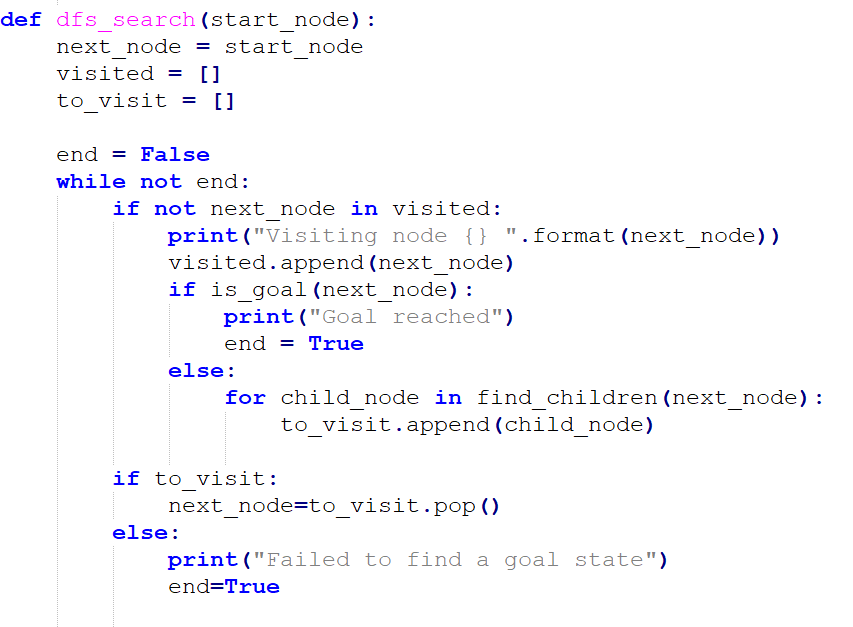




* However, in some problems, it is not practical to pre-define the whole graph in advance.
* To take care of this possibility, we can replace the line marked in red by a more generic find\_children( )function for generating the children, which can be changed for different problems, as follows:

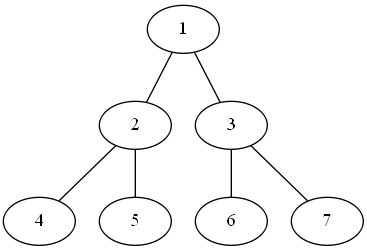
Depth First Search Inplementation (Version 2)



The complete program is listed in Moodle

Depth First Search Inplementation (Version 3): Remember the path

* The above versions work fine in finding a goal state. However, it does not remember the path from the root to the goal.



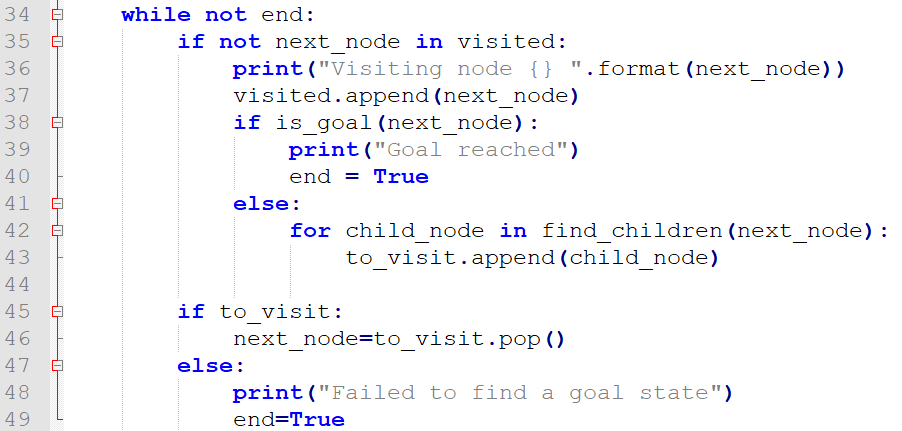
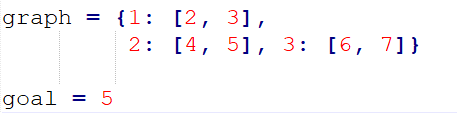
The path is

[1, 2, 5]

Goal

* To do so, we can record the path from the root to each node, by storing it next to each state in the to\_visit List.
* Thus, each element of the to\_visit list is now a 2-part tuple (state, path), where
  + state is a state in the problem (same as before)
  + path **i**s a list of the states from the root to the current node
* The complete program is listed in Moodle

Exercise 3: Breadth First Search Inplementation

* Implementation BFS by modifying the program of DFS version-2.
* Name your program BFS.py
* Hint: Which line(s) (see below) should we change?
* Test your program using the same graph as before. Show the order in which the nodes are visited.
* 

Exercise 4: Maze

Write a program (Maze.py) in Pyhon to find a path through the following maze using DFS!

A screenshot of a game

Description automatically generated with low confidence

Hints:

* DFS version 3 could be useful.
* What are the states?
* What does the tree looks like?
* How can we represent the tree represented in Python?

Oh, in case anyone cannot figure it out, the answer is [A, B, D, I, K, M, N] 😁

Challenge Exercise: Three Man and Three Barbarians. (Ten bonus coupons)

Solve the following puzzle by adapting and modifying the program codes we shown in class.

Graphical user interface

Description automatically generated with medium confidence"Three people and three barbarians need to cross a river by boat. The boat has only two seats. At any time, if the barbarians outnumber the people in either side of the river, the barbarians would harm the tourists . Also, The boat cannot cross the river by itself with no people on board. How do they cross the river?"

<https://www.novelgames.com/en/missionaries/>

Challenge Exercise: Three Man and Three Barbarians. (Hints)

* How do we represent a state of the problem in the program?
* What is the start state?
* What is the goal state?
* What operators are available?
  + Moving 1 man to the opposite side
  + Moving 1 barbarian to the opposite side
  + Moving 1 man and 1 barbarian to the opposite side
  + Moving 2 men to the opposite side
  + Moving 2 barbarians to the opposite side

Note: you may reference the Wolf Goat Cabbage program, but do not blindly follow all details as the problems are not the same one!

Any Questions?