Lab2

Introduction to AI, M. MATEI
ISEP - December 2023
by Theophile Wemaere

Depth-first search and Breath-first search

Source code and markdown source file can be found here
or at github.com/Theophile-Wemaere/intro-to-ai

Part A

Question 1

The time complexity of DFS and BFS algorithms is **O(V + E)**, where V is the number of vertices (the nodes) and E (links between nodes) is the number of edges in the graph. This means that the time it takes for these algorithms to run is proportional to the size of the graph.

A *uses a heuristic function to guide its search. The heuristic function is an estimate of the distance between the current node and the goal node. This allows A* to focus on the most promising paths, which can make it more efficient than DFS and BFS.

The time complexity of A *is **O(V + E log V)** in the worst case, but it is often much better than this in practice. This is because A* is able to prune many of the nodes that would be explored by DFS and BFS.

Question 2

The memory requirements of DFS and BFS algorithms are **O(V)**, where V is the number of vertices in the graph. This is because these algorithms store the entire frontier of the search in memory.

The memory requirements of A *are also **O(V)**, but they can be higher in practice if the heuristic function is not very accurate. This is because A* may explore more nodes than DFS and BFS.

Question 3: Comparison and Applications

Here is a table summarizing the key differences between DFS, BFS, UCS, and A*:

Algorithm	Time Complexity	Memory Requirements	Description	Applications
DFS	O(V + E)	O(V)	Explores all paths in a depth-first manner	Finding connected components, detecting cycles
BFS	O(V + E)	O(V)	Explores all paths in a breadth-first manner	Finding shortest paths, checking if a graph is bipartite
UCS	O(V + E log V)	O(V)	Uninformed search algorithm that prioritizes nodes with the lowest cost	Finding shortest paths in graphs with non-negative edge costs
A*	O(V + E log V)	O(V)	Informed search algorithm that prioritizes nodes with the lowest estimated cost to the goal	Finding shortest paths in graphs with non-negative edge costs, planning navigation in robotics

In general, DFS is a good choice for finding connected components and detecting cycles. BFS is a good choice for finding shortest paths and checking if a graph is bipartite. UCS is a good choice for finding shortest paths in graphs with non-negative edge costs. A* is a good choice for finding shortest paths in graphs with non-negative edge costs, planning navigation in robotics, and other applications where a heuristic function is available.

To resume :

- DFS is more likely to get stuck in deep, narrow parts of the graph.
- BFS is more likely to explore a larger number of nodes.
- UCS is guaranteed to find the shortest path if one exists, but it may not be the most efficient algorithm.
- A* is not guaranteed to find the shortest path, but it is often much more efficient than UCS.

Part B

Question 1

The following python code is an example of the BFS algorithm implementation :

```
#!/bin/python

def bfs(graph, start,goal):
    visited = []
    queue = [[start]]
```

```
    while queue:
        path = queue.pop(0)
        current_node = path[-1]
        print(f"Visiting {current_node} from {' -> '.join(path)}")

        visited.append(current_node)

        if current_node == goal:
            print("\nOptimal path found : ",end ='')
            print(' -> '.join(path))
            return 0

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

graph = {
    'Start': ["A", "B","D"],
    'A': ["Start", "C"],
    'B': ["Start", "D"],
    'C': ["A", "D", "Goal"],
    'D': ["Start", "B", "C", "Goal"],
    'Goal': ["C", "D"]
}

bfs(graph, "Start", "Goal")
```

Question 2

The following python code is an example of the DFS algorithm implementation :

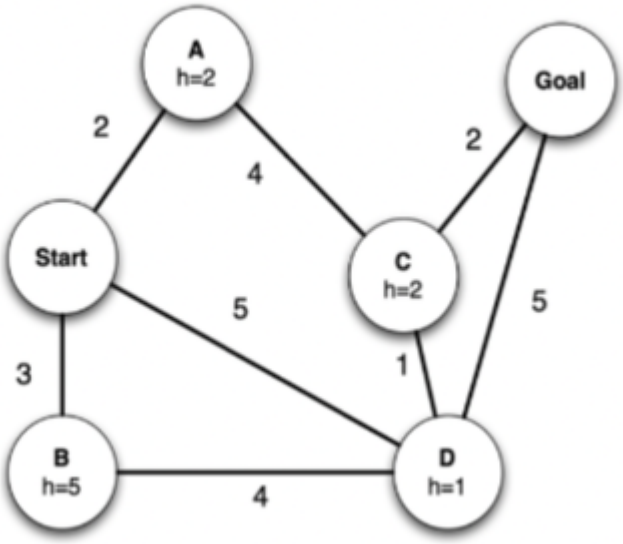
```
#!/bin/python

def dfs(graph,start,goal):
    visited = []
    queue = [[start]]

    while queue:
        path = queue.pop()
        current_node = path[-1]
        visited.append(current_node)
        print(f"Visiting {current_node} from {' -> '.join(path)}")

        if current_node == goal:
            print("\nOptimal path found : ",end ='')
            print(' -> '.join(path))
            return 0
```

```
        for neighbor in list(graph[current_node])[::-1]:
            if neighbor not in visited:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

graph = {
    'Start': ["A", "B", "D"],
    'A': ["Start", "C"],
    'B': ["Start", "D"],
    'C': ["A", "D", "Goal"],
    'D': ["Start", "B", "C", "Goal"],
    'Goal': ["C", "D"]
}

dfs(graph, "Start","Goal")
```

Part C

Question 1 & 2

For this exercise we will work on the following graph :



BFS :

Here is the result of the breath-first search algorithm on the graph :

```
 tihmz ~/Documents/Work/ISEP/intro-to-ai/Lab2  main !2 ?1
❯ ./Lab2.py --bfs

Running BFS algorithm :
Visiting Start from Start
Visiting A from Start -> A
Visiting B from Start -> B
Visiting D from Start -> D
Visiting C from Start -> A -> C
Visiting D from Start -> B -> D
Visiting C from Start -> D -> C
Visiting Goal from Start -> D -> Goal

Optimal path found : Start -> D -> Goal
```

DFS

Here is the result of the depth-first search algorithm on the graph :

```
 tihmz ~/Documents/Work/ISEP/intro-to-ai/Lab2  main !2 ?1
❯ ./Lab2.py --dfs

Running DFS algorithm :
Visiting Start from Start
Visiting A from Start -> A
Visiting C from Start -> A -> C
Visiting D from Start -> A -> C -> D
Visiting B from Start -> A -> C -> D -> B
Visiting Goal from Start -> A -> C -> D -> Goal

Optimal path found : Start -> A -> C -> D -> Goal
```

Question 3

The following output is the result of the A* algorithm on the same graph :

```
Examining : Start -> A
Examining : Start -> B
Examining : Start -> D
Examining : Start -> A -> C
Examining : Start -> D -> Goal

Optimal path found : Start -> A -> C -> Goal
```

Some for the graph we got the following results :

```
BFS : Start -> D -> Goal
DFS : Start -> A -> C -> D -> Goal
A*  : Start -> A -> C -> Goal
```

First we can see that both BFS and DFS use more iterations than A* to find a path :

- 8 iterations for BFS
- 6 iterations for DFS
- 5 iterations for A*
  (this also could be due to a bad algorithm implementation)

We can see that the BFS algorithm found the shortest path to the stop node (Goal) and that the DFS kind of got "stuck" in a branch and went deep to find the Goal node. The A *algorithm take account of heuristics so it found a path a bit longer than BFS but better that DFS. This complete what we presented at the beginning of this exercise that "A* is not guaranteed to find the shortest path" and it really depend of the graph structure.

Annexe : Complete python code

```
#!/bin/python
import argparse

graph = {
    'Start': ["A", "B", "D"],
    'A': ["Start", "C"],
    'B': ["Start", "D"],
    'C': ["A", "D", "Goal"],
    'D': ["Start", "B", "C", "Goal"],
    'Goal': ["C", "D"]
}

def dfs(graph,start,goal):
    visited = []
    queue = [[start]]

    while queue:
        path = queue.pop()
        current_node = path[-1]
        visited.append(current_node)
        print(f"Visiting {current_node} from {' -> '.join(path)}")

        if current_node == goal:
            print("\nOptimal path found : ",end ='')
            print(' -> '.join(path))
            return 0

        for neighbor in list(graph[current_node])[::-1]:
            if neighbor not in visited:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

def bfs(graph, start,goal):
    visited = []
```

```
    queue = [[start]]

    while queue:
        path = queue.pop(0)
        current_node = path[-1]
        print(f"Visiting {current_node} from {' -> '.join(path)}")

        visited.append(current_node)

        if current_node == goal:
            print("\nOptimal path found : ",end ='')
            print(' -> '.join(path))
            return 0

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

if __name__ == "__main__":
    try:
        parser = argparse.ArgumentParser(description='run BFS and DFS
algorithm')
        parser.add_argument('--dfs', action='store_true', help='Perform
depth-first search (DFS).')
        parser.add_argument('--bfs', action='store_true', help='Perform
breath-first search (BFS).')
        args = parser.parse_args()

        if args.bfs:
            print("\nRunning BFS algorithm : ")
            bfs(graph, "Start","Goal")
        if args.dfs:
            print("\nRunning DFS algorithm : ")
            dfs(graph, "Start","Goal")
        if not args.dfs and not args.bfs:
            parser.print_help()

    except KeyboardInterrupt:
        print("\nCtrl+C pressed, exiting...")
        exit(1)
```