Artificial Intelligence

Tutorial week 2 – Search

COMP9414

1 Introduction

In this tutorial, we will explore five different search algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS), Best-First Search, and A*. We will use NetworkX, a powerful Python library for creating, manipulating, and analysing complex graphs [1], to construct the graph and evaluate the performance of each algorithm. Additionally, we will visualise the step-by-step iterations of these algorithms to provide a deeper understanding of their inner workings.

NetworkX is a versatile tool that allows us to easily create various types of graphs, perform complex operations on them, and visualize the results. It supports many graph theory-related algorithms, making it an ideal choice for this tutorial. Throughout this guide, NetworkX will help us to define graphs, implement search algorithms, and collect performance metrics efficiently.

To get started with NetworkX, ensure it is installed in your Python environment. You can install it via pip if you haven't already:

```
pip install networkx
```

2 Search Algorithms

2.1 Depth-First Search (DFS)

Depth-First Search (DFS) is an algorithm used for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. DFS uses a stack to keep track of the nodes to be visited next [2].

Algorithm:

- Initialize a stack with the starting node and an empty path.
- While the stack is not empty:
  - Pop a node and its path from the stack.

  - If the node has been visited, skip it.
  - Mark the node as visited.
  - Append the current node to the path.
  - If the goal is reached, return the path.
  - Add all unvisited neighbours to the stack.

2.2 Breadth-First Search (BFS)

Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and explores the neighbour nodes at the present depth prior to moving on to nodes at the next depth level. BFS uses a queue to keep track of the nodes to be visited next [2].

Algorithm:

- Initialize a queue with the starting node and an empty path.
- While the queue is not empty:
  - Dequeue a node and its path.
  - If the node has been visited, skip it.
  - Mark the node as visited.
  - Append the current node to the path.
  - If the goal is reached, return the path.
  - Add all unvisited neighbours to the queue.

2.3 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) is an algorithm used for finding the shortest path in a weighted graph where the cost of edges can vary. It expands the least cost node first, ensuring that the shortest path is found. UCS uses a priority queue to keep track of the nodes to be visited next, with priority given to nodes with the lowest cumulative cost [2].

Algorithm:

- Initialize a priority queue with the starting node, cost 0, and an empty path.
- While the queue is not empty:
  - Dequeue the node with the lowest cost.
  - If the node has been visited, skip it.
  - Mark the node as visited.

  - Append the current node to the path.
  - If the goal is reached, return the path and cost.
  - Add all unvisited neighbours to the priority queue with their cumulative cost.

2.4 Best-First Search

Best-First Search is an algorithm that uses heuristics to guide the search process. It selects the node that appears to be closest to the goal according to the heuristic function. Best-First Search uses a priority queue to keep track of the nodes to be visited next, with priority given to nodes with the lowest heuristic value [2].

Algorithm:

- Initialize a priority queue with the starting node, its heuristic value, and an empty path.
- While the queue is not empty:
  - Dequeue the node with the lowest heuristic value.
  - If the node has been visited, skip it.
  - Mark the node as visited.
  - Append the current node to the path.
  - If the goal is reached, return the path.
  - Add all unvisited neighbours to the priority queue with their heuristic value.

2.5 A* Search

A* Search is a widely used algorithm that combines the strengths of Uniform Cost Search and Best-First Search. It uses both the actual cost from the start and the heuristic cost to the goal to guide the search. A* uses a priority queue to keep track of the nodes to be visited next, with priority given to nodes with the lowest combined cost [2].

Algorithm:

- Initialize a priority queue with the starting node, cost 0, its heuristic value, and an empty path.
- While the queue is not empty:
  - Dequeue the node with the lowest cost + heuristic value.
  - If the node has been visited, skip it.

- Mark the node as visited.
- Append the current node to the path.
- If the goal is reached, return the path and cost.
- Add all unvisited neighbours to the priority queue with their cumulative cost + heuristic value.

3 Experiments

3.1 Step 1: Graph Definition

First, we will define an undirected graph with weighted edges using NetworkX (See Figure 1). This graph will be used for all the algorithms to ensure a fair comparison.

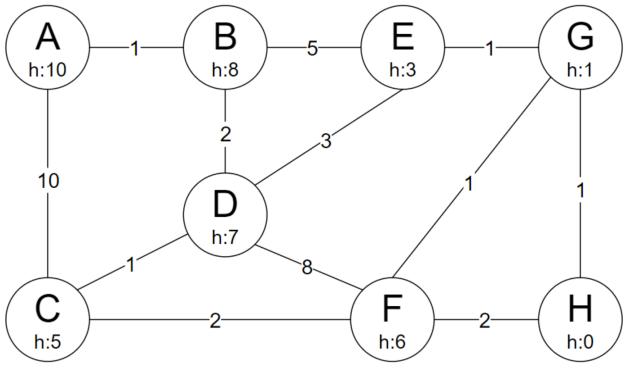

Figure 1: Membership function for output variable injection.

3.2 Step 2: Algorithm Implementations

In this tutorial, each algorithm is provided with a skeleton code that you must complete by filling in the TODO parts. The algorithms are implemented to work with NetworkX graphs. Some of these algorithms rely on priority queues to efficiently manage the order in which nodes are explored. To implement priority queues in Python, we will use the `heapq` module, which provides an easy-to-use and efficient way to maintain a priority queue using a binary heap [3].

The `heapq` module is ideal for implementing priority queues because it maintains the heap invariant efficiently, allowing us to push and pop the smallest item in $O(\log n)$ time. This makes it perfect for search algorithms that need to expand the least-cost node first, such as UCS, Best-First Search, and A* [4].

3.3 Step 3: Visualisation

We will visualise each algorithm's step-by-step iterations to enhance understanding using a graphical user interface. For this purpose, we will utilise PyQt5, a library that allows us to create cross-platform GUI applications in Python. The visualisation will help illustrate how the algorithms explore the graph, make decisions at each step, and ultimately find the path to the goal.

We will use the `QApplication` class from PyQt5 to manage our GUI application's control flow and handle the main settings. Ensure PyQt5 is installed in your Python environment:

```
pip install PyQt5
```

PyQt5 provides a flexible and powerful framework for creating custom visualisations of the graph algorithms. By using `QApplication`, we can manage the event loop and display GUI windows that show the step-by-step progress of each algorithm.

The visualisations will illustrate the following aspects:

- **Current node:** Highlight the current node being explored.
- **Visited nodes:** Show the nodes that have already been visited.
- **Frontier nodes:** Indicate the nodes that are in the stack/queue and are yet to be visited.
- **Path construction:** Demonstrate how the path is being constructed as the algorithm progresses.
- **Node expansion:** Visualise the process of expanding nodes and exploring their neighbours.

3.4 Step 4: Running the Algorithms and Collecting Metrics

To evaluate the performance of each search algorithm, we will use a helper function, `run_algorithm`, which automates the process of running the algorithms on the graph and collecting key metrics. This function takes care of executing the search algorithms, timing their execution, and calculating various performance metrics, including the path found, cost, node expansions, time complexity, and space complexity.

The helper function (`run_algorithm`) is designed to handle both heuristic-based algorithms (such as A* and Best-First Search) and non-heuristic algorithms (like DFS, BFS, and UCS). It dynamically adjusts its execution based on whether the chosen algorithm requires a heuristic function.

```
import time

def run_algorithm(algorithm, graph, start, goal, heuristic=None):
```

```
start_time = time.time()  # Record the start time
# Run the algorithm, with or without heuristic as needed
if algorithm in [DFS, BFS, UCS]:
    path, cost, node_expansions = algorithm(graph, start, goal)
else:
    path, cost, node_expansions = algorithm(graph, start, goal, heuristic)
end_time = time.time()  # Record the end time

# Calculate performance metrics
time_complexity = end_time - start_time
space_complexity = len(path)

# Return a dictionary with the collected metrics
return {
    "path": path,
    "cost": cost,
    "node_expansions": node_expansions,
    "time_complexity": time_complexity,
    "space_complexity": space_complexity,
}
```

3.4.1 How it Works:

- **Timing execution:** The function records the start and end times of the algorithm execution to compute the time complexity.
- **Dynamic algorithm execution:** Depending on whether the algorithm is heuristic-based, it either calls the algorithm with or without the heuristic parameter.
- **Metric collection:** After running the algorithm, the function collects the following metrics:
  - **Path:** The sequence of nodes from the start to the goal.
  - **Cost:** The total cost associated with the path found.
  - **Node Expansions:** The number of nodes expanded during the search.
  - **Time Complexity:** The total execution time of the algorithm.
  - **Space Complexity:** The length of the path, which serves as a simple measure of space complexity in this context.

3.4.2 Using the Helper Function:

We will use the `run_algorithm` function to run each of the algorithms on our graph, collect the metrics, and print the results for comparison. This approach

allows us to standardise the evaluation process across all algorithms, making it easier to analyse their performance under the same conditions.

This standardised approach provides a consistent and efficient way to compare the performance of different search algorithms using the same graph structure defined earlier in the tutorial.

4 Tasks

(a) Fill in the TODO part in the DFS skeleton code to add all unvisited neighbours to the stack.

(b) Fill in the TODO part in the BFS skeleton code to add all unvisited neighbours to the queue.

(c) Fill in the TODO part in the UCS skeleton code to add all unvisited neighbours to the priority queue with their cumulative cost.

(d) Fill in the TODO part in the Best-First Search skeleton code to add all unvisited neighbours to the priority queue with their heuristic value.

(e) Fill in the TODO part in the A* skeleton code to add all unvisited neighbours to the priority queue with their cumulative cost + heuristic value.

References

[1] Aric Hagberg, Pieter Swart, and Daniel Chult. Exploring network structure, dynamics, and function using networkx. 06 2008.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 3rd edition, 2010.

[3] Python Software Foundation. heapq — heap queue algorithm. `https://docs.python.org/3/library/heapq.html`, 2020.

[4] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.