Kyle Tranfaglia

COSC 411

Project 01

11 Oct. 2024

Graph Search Report

Program Description:

1. BFS (Breadth First Search):

The algorithm is designed to find the shortest path (the least amount of edges traveled | the least number of cities traveled) between a given start and destination city in a graph using a queue-based (FIFO) approach. The function begins by recording the start time to measure the time taken for the search. It initializes a queue containing the start city, an empty list to track the path, and a total distance of 0 as a tuple. A visited list is maintained to track explored cities, preventing revisits and infinite loops. For each iteration, the algorithm dequeues the first city from the queue, first exploring the earliest added city. If the destination city is reached, the algorithm calculates and prints the path taken, the total distance traveled, and the time consumed by the search. Otherwise, it iterates over all neighboring cities of the current city. If a neighbor has not been visited, it is enqueued along with the updated path and total distance. Once all neighbors of a city are explored, the city is marked as visited. If no path is found, the function prints an error message, though this is unlikely in a connected graph, especially with input validation. The algorithm explores cities in a breadth-first manner, meaning it checks all neighbors of a city before moving to the next level. As long as there is a valid path between the start and destination cities, the algorithm will find the shortest path (the least cities travelled).

2. DFS (Depth First Search):

The algorithm is designed to find a path between a given start and destination city in a graph using a stack-based (LIFO) approach. The function begins by recording the start time to measure the time taken for the search. It initializes a stack containing the start city, an empty list to track the path, and a total distance of 0 as a tuple. A visited list is maintained to track the explored cities, preventing cycles and revisits. For each iteration, the algorithm pops the city from the top of the stack, first exploring the most recently added city. If the destination city is reached, it calculates and prints the path taken, the total distance traveled, and the time consumed by the search. Otherwise, the algorithm iterates over all neighbors of the current city. If a neighbor has not been visited, it is appended to the stack along with the updated path and total distance. Once all neighbors of a city are explored, the city is marked as visited. If no path is found, the function prints an error message, though this is unlikely in a connected graph, considering there is input validation. The algorithm explores the leftmost path as it processes neighbors in the order they are stored in the graph dictionary. Note that the implemented DFS will find the first valid path from the start city to the destination city without guaranteeing the shortest or optimal path. Thus, given a valid start and destination city, the algorithm will find a path (first, the leftmost path found in the graph) as long as a path between the cities exists.

3. UCS (Uniform Cost Search):

The algorithm is designed to find the optimal (least cost/distance) path between a given start and destination city in the weighted graph. It uses a priority queue to explore cities based on the cumulative distance traveled (cost). The function begins by recording the start time to measure the search time. It initializes a priority queue containing the start city, the path so far, and a total distance of 0 as a tuple. A visited dictionary is used to store the minimum cost required to reach each city, ensuring that cities are only revisited if a lower cost path is found. At each iteration, the algorithm dequeues the city with the smallest total distance (lowest cost) from the priority queue. If the destination city is reached, the algorithm calculates and prints the path taken, the total distance traveled, and the time consumed by the search. Otherwise, it iterates over all neighboring cities of the current city. For each neighbor, it calculates the new total distance to reach that neighbor. If the neighbor has not been visited or if the new distance is smaller than the previously recorded distance, the neighbor is added to the priority queue along with the updated path and distance. If no path is found, the function prints an error message, though this is unlikely in a connected graph with input validation. The priority queue ensures that cities are explored in increasing order of cost, guaranteeing that the first time the destination is reached, the path discovered is optimal in terms of total distance.

4. Input Validation

The input validation function ensures that the cities entered by the user are valid by checking whether both the start and destination cities exist in the graph. It continuously prompts the user for input until valid cities are provided, helping to prevent errors caused by invalid or misspelled city names.

5. Driver Program

The driver porgram (main) tests each of the search algorithms by storing the user input once validated and then usies the start and destination cities to call each of the functions. All three search functions are called by the driver program with the same start and destination city from the user, along with the city graph. The program servers to test and compare the functionality and performance of the search algorithms above.

Source Code:

```
# Kyle Tranfaglia

# COSC411 - Project01

# Last updated 10/09/24

# This program uses a BFS, DFS, and UCS algorithm to find a path given a start

# city and destination city in the provided undirected & weighted graph


import time

from collections import deque

from queue import PriorityQueue


# Undirected city graph with distances (miles) in alphabetical order with
shortest to the longest paths (edges)

city_graph = {

   'Baltimore': {'Washington DC': 45, 'Salisbury': 117, 'Philadelphia': 101,
'Pittsburgh': 248},

   'Boston': {'New York': 216, 'Buffalo': 450},

   'Buffalo': {'Pittsburgh': 219, 'Boston': 450},

   'New York': {'Philadelphia': 94, 'Boston': 216, 'Pittsburgh': 370},

   'Norfolk': {'Richmond': 93, 'Salisbury': 132},

   'Philadelphia': {'New York': 94, 'Baltimore': 101, 'Salisbury': 138,
'Pittsburgh': 304},

   'Pittsburgh': {'Buffalo': 219, 'Baltimore': 248, 'Philadelphia': 304, 'New
York': 370},

   'Richmond': {'Washington DC': 110, 'Norfolk': 93},

   'Salisbury': {'Baltimore': 117, 'Philadelphia': 138, 'Washington DC': 116,
'Norfolk': 132},

   'Washington DC': {'Baltimore': 45, 'Salisbury': 116, 'Richmond': 110}
```

```
}



# Breadth First Search algorithm to find the shortest path (the least cities
traveled) between a start and end city
def bfs(graph, start, destination):
   start_time = time.perf_counter()  # Start timer

   # Queue (doubled-ended | FIFO) for BFS that stores a tuple: (current city,
path so far, total distance traveled)
   queue = deque([(start, [start], 0)])
   # List to track visited cities and prevent revisits & infinite loops
   visited = []

   # Continue search as long as there are cities in the queue left to explore
   while (queue):
       # Remove first city from queue (FIFO) and store the tuple individually |
O(1) compared to O(n) for .pop(0)
       city, path, distance = queue.popleft()

       # Check if destination has been reached
       if (city == destination):
           # End timer and calculate time taken to complete search
           end_time = time.perf_counter()
           time_taken = end_time - start_time

           # Print results from BFS
           print("\nBFS Path:")
```

```
            print(f"Path: {' --> '.join(path)}")  # join list elements of path 
into a single string separated by ->
            print(f"Total distance: {distance} miles")
            print(f"Time taken: {time_taken:.7f} seconds")

            return
        else:
            # Explore all neighboring cities if not already visited
            for neighbor, miles in graph[city].items():
                # Check if a neighbor has been visited, then add it to the queue 
with an updated path and total distance
                if (neighbor not in visited):
                    queue.append((neighbor, path + [neighbor], distance + 
miles))

            visited.append(city)  # Add current city to set to avoid revisit

    # Since the graph is connected, this should never happen with valid input: 
displays message on a failed search
    print(f"No path found from {start} to {destination}")



# Depth First Search algorithm to finds a path (leftmost | first found) between 
a start and end city
def dfs(graph, start, destination):
    start_time = time.perf_counter()  # Start timer

    # Stack (LIFO) for DFS that stores a tuple: (current city, path so far, 
total distance traveled)
```

```
    stack = [(start, [start], 0)]

    # List to track visited cities and prevent revisits & infinite loops

    visited = []


    # Continue search as long as there are cities in the stack left to explore

    while (stack):

        # Remove first city from queue (FIFO) and store the tuple individually

        city, path, distance = stack.pop()


        # Check if destination has been reached

        if (city == destination):

            # End timer and calculate time taken to complete search

            end_time = time.perf_counter()

            time_taken = end_time - start_time


            # Print results from DFS

            print("\nDFS Path:")

            print(f"Path: {' --> '.join(path)}")  # join list elements of path
into a single string separated by ->

            print(f"Total distance: {distance} miles")

            print(f"Time taken: {time_taken:.7f} seconds")


            return

        else:

            # Explore all neighboring cities if not already visited

            for neighbor, miles in graph[city].items():

                # Check if a neighbor has been visited, then add it to the stack
with an updated path and total distance

                if (neighbor not in visited):
```

```
                    stack.append((neighbor, path + [neighbor], distance +
miles))

            visited.append(city)  # Add current city to set to avoid revisit

    # Since the graph is connected, this should never happen with valid input:
displays message on a failed search
    print(f"No path found from {start} to {destination}")


# Uniform Cost Search algorithm to finds the optimal (the least cost) path
between a start and end city
def ucs(graph, start, destination):
    start_time = time.perf_counter()  # Start timer

    # Priority queue for UCS that stores a tuple: (total distance traveled
*priority item*, current city, path so far)
    priority_queue = PriorityQueue()
    priority_queue.put((0, start, [start]))  # Initialize with the start city
and distance as 0 (distance is priority)

    # Dictionary to track the least cost to reach each city
    visited = {}

    # Continue search as long as there are cities in the priority queue left to
explore
    while (not priority_queue.empty()):
        # Get the city with the lowest cost from the priority queue and store
the tuple individually
```

```
        distance, city, path = priority_queue.get()

        # Check if destination has been reached
        if (city == destination):
            # End timer and calculate time taken to complete search
            end_time = time.perf_counter()
            time_taken = end_time - start_time

            # Print results from DFS
            print("\nUCS Path:")
            print(f"Path: {' --> '.join(path)}")  # join list elements of path
into a single string separated by ->
            print(f"Total distance: {distance} miles")
            print(f"Time taken: {time_taken:.7f} seconds")

            return
        else:
            # Visit the city if not yet visited or revisit if a smaller distance
(to ensure optimality) is found
            if (city not in visited or distance < visited[city]):
                visited[city] = distance  # Set the minimum cost to reach this
city

                # Explore all neighboring cities
                for neighbor, miles in graph[city].items():
                    new_distance = distance + miles  # Calculate the new total
distance (cost) to reach the neighbor
```

```
                   # Check if it's a new city or has a lower cost, then add the
neighbor to the priority queue
                   if (neighbor not in visited or new_distance <
visited[neighbor]):
                       priority_queue.put((new_distance, neighbor, path +
[neighbor]))

   # Since the graph is connected, this should never happen with valid input:
displays message on a failed search
   print(f"No path found from {start} to {destination}")



# Get and validate user input until a city in the graph is entered
def get_city_input(prompt, graph):
   city = input(prompt).strip()  # Get for input and strip any surrounding
whitespace

   # Prompt for a city until a valid city (in the graph) is entered
   while (city not in graph):
       print(f"'{city}' is not in the graph. Please enter a valid city.")
       city = input(prompt).strip()  # Get for input and strip any surrounding
whitespace

   return city



# Driver program (main)
if (__name__ == "__main__"):
   # Program welcome message
```

```
    print("Find a path from city x to city y using the following: Baltimore, "
          "Boston, Buffalo, New York, Norfolk, Philadelphia, Pittsburgh, 
Richmond, Salisbury, Washington DC")

    # Get and validate user input for the start and destination city
    start_city = get_city_input("Enter the start city: ", city_graph)
    destination_city = get_city_input("Enter the destination city: ", 
city_graph)

    # Call BFS function to find & display the shortest path (the least cities 
traveled)
    bfs(city_graph, start_city, destination_city)

    # Call DFS function to find & display a path (leftmost | first found)
    dfs(city_graph, start_city, destination_city)

    # Call UCS function to find & display the optimal path (the lowest 
cost/distance)
    ucs(city_graph, start_city, destination_city)
```

Test Results:

```
Find a path from city x to city y using the following: Baltimore, Boston, Buffalo, New York, Norfolk, Philadelphia, Pittsburgh, Richmond, Salisbury, Washington DC
Enter the start city: Washington DC
Enter the destination city: Philadelphia

BFS Path:
Path: Washington DC --> Baltimore --> Philadelphia
Total distance: 146 miles
Time taken: 0.0000348 seconds

DFS Path:
Path: Washington DC --> Richmond --> Norfolk --> Salisbury --> Philadelphia
Total distance: 473 miles
Time taken: 0.0000089 seconds

UCS Path:
Path: Washington DC --> Baltimore --> Philadelphia
Total distance: 146 miles
Time taken: 0.0001055 seconds

Process finished with exit code 0
```

```
Find a path from city x to city y using the following: Baltimore, Boston, Buffalo, New York, Norfolk, Philadelphia, Pittsburgh, Richmond, Salisbury, Washington DC
Enter the start city: Pittsburgh
Enter the destination city: Boston

BFS Path:
Path: Pittsburgh --> Buffalo --> Boston
Total distance: 669 miles
Time taken: 0.0000471 seconds

DFS Path:
Path: Pittsburgh --> New York --> Boston
Total distance: 586 miles
Time taken: 0.0000078 seconds

UCS Path:
Path: Pittsburgh --> New York --> Boston
Total distance: 586 miles
Time taken: 0.0001055 seconds

Process finished with exit code 0
```

```
Find a path from city x to city y using the following: Baltimore, Boston, Buffalo, New York, Norfolk, Philadelphia, Pittsburgh, Richmond, Salisbury, Washington DC
Enter the start city: Richmond
Enter the destination city: Baltimore

BFS Path:
Path: Richmond --> Washington DC --> Baltimore
Total distance: 155 miles
Time taken: 0.0000225 seconds

DFS Path:
Path: Richmond --> Norfolk --> Salisbury --> Washington DC --> Baltimore
Total distance: 386 miles
Time taken: 0.0000056 seconds

UCS Path:
Path: Richmond --> Washington DC --> Baltimore
Total distance: 155 miles
Time taken: 0.0000537 seconds

Process finished with exit code 0
```

```
Find a path from city x to city y using the following: Baltimore, Boston, Buffalo, New York, Norfolk, Philadelphia, Pittsburgh, Richmond, Salisbury, Washington DC
Enter the start city: Bostin
'Bostin' is not in the graph. Please enter a valid city.
Enter the start city: Bostonn
'Bostonn' is not in the graph. Please enter a valid city.
Enter the start city: Boston
Enter the destination city: Salisbury

BFS Path:
Path: Boston --> New York --> Philadelphia --> Salisbury
Total distance: 448 miles
Time taken: 0.0004108 seconds

DFS Path:
Path: Boston --> Buffalo --> Pittsburgh --> New York --> Philadelphia --> Salisbury
Total distance: 1271 miles
Time taken: 0.0000200 seconds

UCS Path:
Path: Boston --> New York --> Philadelphia --> Salisbury
Total distance: 448 miles
Time taken: 0.0001407 seconds

Process finished with exit code 0
```

```
Find a path from city x to city y using the following: Baltimore, Boston, Buffalo, New York, Norfolk, Philadelphia, Pittsburgh, Richmond, Salisbury, Washington DC
Enter the start city: Buffalo
Enter the destination city: Norfolk

BFS Path:
Path: Buffalo -> Pittsburgh -> Baltimore -> Salisbury -> Norfolk
Total distance: 716 miles
Time taken: 0.0000795 seconds

DFS Path:
Path: Buffalo -> Boston -> New York -> Pittsburgh -> Philadelphia -> Salisbury -> Norfolk
Total distance: 1610 miles
Time taken: 0.0000196 seconds

UCS Path:
Path: Buffalo -> Pittsburgh -> Baltimore -> Washington DC -> Richmond -> Norfolk
Total distance: 715 miles
Time taken: 0.0002441 seconds

Process finished with exit code 0
```