Madan Bhandari Memorial College

Lab No. 10

Title: Write a program to implement Greedy Best First Search.

Greedy Best-First Search is an AI search algorithm that attempts to find the most promising path from a given starting point to a goal. It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.

Key Features:

- **Heuristic-based**: Uses a heuristic function to estimate the cost from current node to goal
- **Greedy approach**: Always selects the node that appears best according to the heuristic
- **Informed search**: Unlike uninformed searches, it has knowledge about the goal location
- **Time complexity**: O(b^m) where b is branching factor and m is maximum depth
- **Space complexity**: O(b^m) in worst case

Greedy Best-First Search Working:

- Greedy Best-First Search works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.
- The algorithm uses a heuristic function to determine which path is the most promising.
- The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths.
- If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

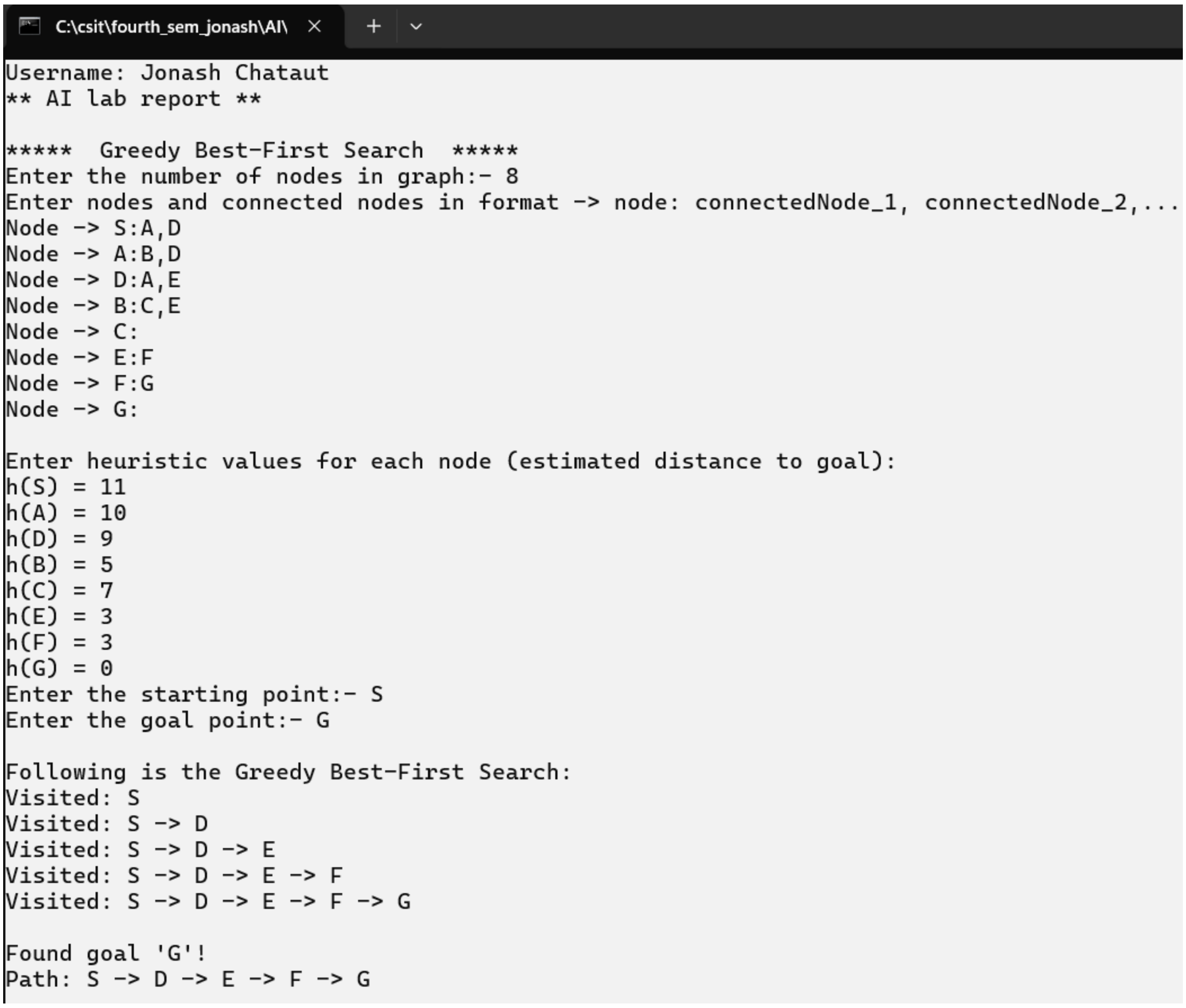
Source Code:

```
import heapq
def createGraphGBFS(graph):
    print("*****  Greedy Best-First Search  *****")
    n = int(input("Enter the number of nodes in graph:- "))
    print("Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...")
    for _ in range(n):
        node = input("Node -> ").split(":")
        graph[node[0].strip()] = [n.strip() for n in node[1].split(",") if n.strip()]
    heuristics = dict()
    print("\nEnter heuristic values for each node (estimated distance to goal):")
    for node in graph.keys():
        heuristics[node] = int(input(f"h({node}) = "))
    return graph, heuristics

def GBFS(graph, heuristics, start, goal):
    visited = set()
    path = []
    queue = []
    heapq.heappush(queue, (heuristics[start], start, [start]))  # (heuristic, node, path)
    while queue:
        h_val, node, path = heapq.heappop(queue)
        print("Visited:", " -> ".join(path))
        if node == goal:
            print(f"\nFound goal '{goal}'!")
            print("Path:", " -> ".join(path))
            return True
        visited.add(node)
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(queue, (heuristics[neighbor], neighbor, path + [neighbor]))
    print(f"\nGoal '{goal}' not found.")
    return False

# Main driver
graph = dict()
graph, heuristics = createGraphGBFS(graph)
start = input("Enter the starting point:- ").strip()
goal = input("Enter the goal point:- ").strip()
print("\nFollowing is the Greedy Best-First Search:")
GBFS(graph, heuristics, start, goal)
```

Output:



C:\csit\fourth_sem_jonash\AI\

```
Username: Jonash Chataut
** AI lab report **

*****  Greedy Best-First Search  *****
Enter the number of nodes in graph:- 8
Enter nodes and connected nodes in format -> node: connectedNode_1, connectedNode_2,...
Node -> S:A,D
Node -> A:B,D
Node -> D:A,E
Node -> B:C,E
Node -> C:
Node -> E:F
Node -> F:G
Node -> G:

Enter heuristic values for each node (estimated distance to goal):
h(S) = 11
h(A) = 10
h(D) = 9
h(B) = 5
h(C) = 7
h(E) = 3
h(F) = 3
h(G) = 0
Enter the starting point:- S
Enter the goal point:- G

Following is the Greedy Best-First Search:
Visited: S
Visited: S -> D
Visited: S -> D -> E
Visited: S -> D -> E -> F
Visited: S -> D -> E -> F -> G

Found goal 'G'!
Path: S -> D -> E -> F -> G
```

Lab No. 11

Title: Write a program to implement A* Search Algorithm.

The A* algorithm is highly effective and well-known search technique utilized for finding the most efficient path between two points in a graph. It is applied in scenarios such as pathfinding in video games, network routing and various artificial intelligence (AI) applications. It was developed in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael as an improvement on Dijkstra's algorithm.

- While Dijkstra's algorithm explores all possible directions around the starting node uniformly
- A* combines actual travel cost with an estimated future cost(heuristics)

This optimizes the search process, reduces computational load and ensures the most efficient path is found with minimal unnecessary exploration.

Key Components of A* Algorithm

A* uses two important parameters to find the cost of a path:

1. $g(n)$**:** Actual cost of reaching node n from the start node. This is the accumulated cost of the path from the start node to node n.
2. $h(n)$**:** The heuristic finds of the cost to reach the goal from node n. This is a weighted guess about how much further it will take to reach the goal.

The function, $f(n)=g(n)+h(n)$ is the total estimated cost of the cheapest solution through node n. This function combines the path cost so far and the heuristic cost to estimate the total cost guiding the search more efficiently.

A* Algorithm:

A* is an informed search algorithm, means it uses the $f(n)$ function to prioritize which nodes to explore next. The process can be broken down into the following steps:

1. **Initialization:** The initial node is added to the open set, a collection of nodes that are yet to be explored. The $f(n)$value for the start node is calculated using the heuristic.
2. **Loop**: A* selects the node with the lowest $f(n)$ value from the open set. This node is expanded and its neighbors are examined.
3. **Goal Check**: If the node being processed is the goal node, the search terminates and the algorithm returns the path to the goal.
4. **Node Expansion**: Each neighbor of the current node is evaluated based on the $g(n)$, $h(n)$ and $f(n)$ values. If a better path to a neighbor is found i.e a lower $f(n)$ then the neighbor is added to the open set or its values are updated.
5. **Repeat**: This process continues until the goal is found or the open set is empty which means there is no solution.

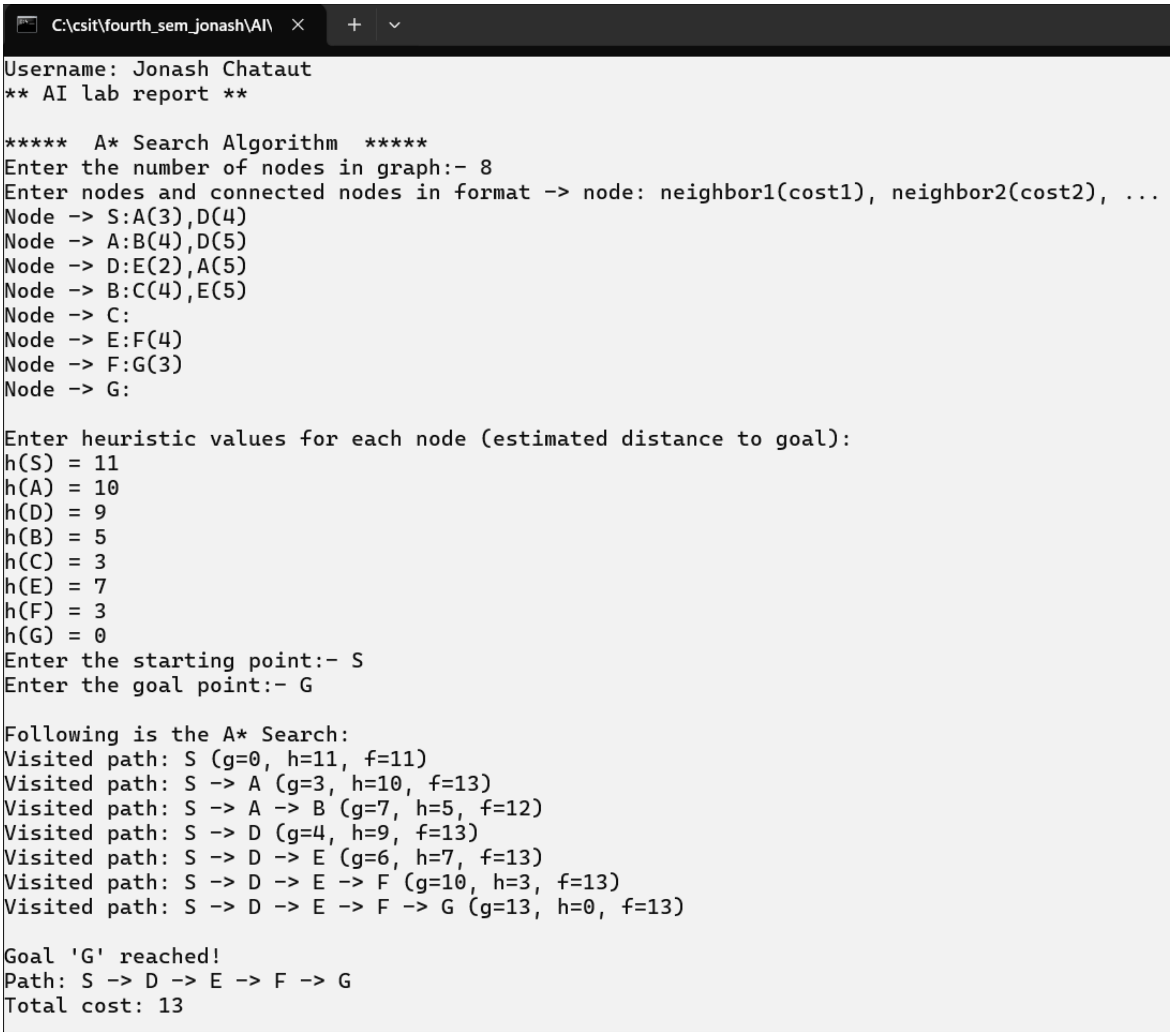
Source Code:

```
import heapq
def createGraphAStar(graph):
  print("*****  A* Search Algorithm  *****")
  n = int(input("Enter the number of nodes in graph:- "))
  print("Enter nodes and connected nodes in format -> node: neighbor1(cost1), neighbor2(cost2), ...")
  for _ in range(n):
    node_input = input("Node -> ").split(":")
    node = node_input[0].strip()
    neighbors = {}
    for neighbor in node_input[1].split(","):
      if neighbor.strip():
        n_name, n_cost = neighbor.strip().split("(")
        neighbors[n_name.strip()] = int(n_cost[:-1])
    graph[node] = neighbors
  heuristics = {}
  print("\nEnter heuristic values for each node (estimated distance to goal):")
  for node in graph.keys():
    heuristics[node] = int(input(f"h({node}) = "))
  return graph, heuristics

def AStar(graph, heuristics, start, goal):
  visited = set()
  queue = []
  heapq.heappush(queue, (heuristics[start], 0, start, [start]))  # (f_score, g_score, node, path)
  while queue:
    f_val, g_val, node, path = heapq.heappop(queue)
    print("Visited path:", " -> ".join(path), f"(g={g_val}, h={heuristics[node]}, f={f_val})")
    if node == goal:
      print(f"\nGoal '{goal}' reached!")
      print("Path:", " -> ".join(path))
      print(f"Total cost: {g_val}")
      return True
    visited.add(node)
    for neighbor, cost in graph.get(node, {}).items():
      if neighbor not in visited:
        g_neighbor = g_val + cost
        f_neighbor = g_neighbor + heuristics.get(neighbor, 0)
        heapq.heappush(queue, (f_neighbor, g_neighbor, neighbor, path + [neighbor]))
  print(f"\nGoal '{goal}' not found.")
  return False

# Main driver
graph = dict()
graph, heuristics = createGraphAStar(graph)
start = input("Enter the starting point:- ").strip()
goal = input("Enter the goal point:- ").strip()
print("\nFollowing is the A* Search:")
AStar(graph, heuristics, start, goal)
```

Output:



```
Username: Jonash Chataut
** AI lab report **

*****  A* Search Algorithm  *****
Enter the number of nodes in graph:- 8
Enter nodes and connected nodes in format -> node: neighbor1(cost1), neighbor2(cost2), ...
Node -> S:A(3),D(4)
Node -> A:B(4),D(5)
Node -> D:E(2),A(5)
Node -> B:C(4),E(5)
Node -> C:
Node -> E:F(4)
Node -> F:G(3)
Node -> G:

Enter heuristic values for each node (estimated distance to goal):
h(S) = 11
h(A) = 10
h(D) = 9
h(B) = 5
h(C) = 3
h(E) = 7
h(F) = 3
h(G) = 0
Enter the starting point:- S
Enter the goal point:- G

Following is the A* Search:
Visited path: S (g=0, h=11, f=11)
Visited path: S -> A (g=3, h=10, f=13)
Visited path: S -> A -> B (g=7, h=5, f=12)
Visited path: S -> D (g=4, h=9, f=13)
Visited path: S -> D -> E (g=6, h=7, f=13)
Visited path: S -> D -> E -> F (g=10, h=3, f=13)
Visited path: S -> D -> E -> F -> G (g=13, h=0, f=13)

Goal 'G' reached!
Path: S -> D -> E -> F -> G
Total cost: 13
```