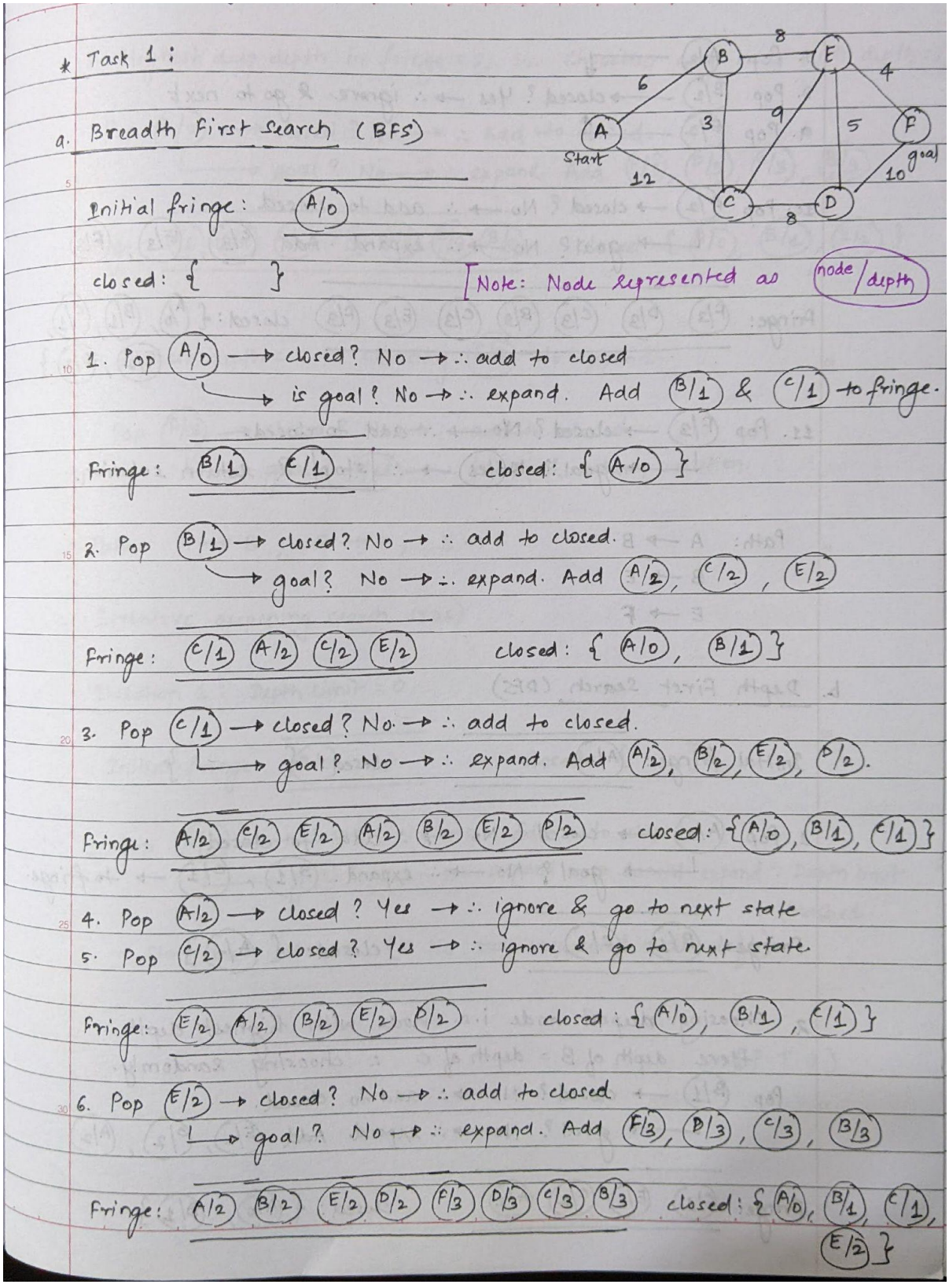
**Task 1: Solution**



7. Pop (A/2) ⟶
8. Pop (B/2) ⟶ closed? Yes ⟶ ∴ ignore & go to next
9. Pop (E/2) ⟶

10. Pop (D/2) ⟶ closed? No ⟶ ∴ add to closed
└⟶ goal? No ⟶ ∴ expand. Add (F/3), (E/3), (F/3)

Fringe: (F/3) (D/3) (C/3) (B/3) (C/3) (E/3) (F/3) closed: { (A/0), (B/1), (C/1), (E/2), (D/2) }

11. Pop (F/3) ⟶ closed? No ⟶ ∴ add to closed
└⟶ goal? Yes ⟶ ∴ stop & return solution.

Path: A ⟶ B
B ⟶ E
E ⟶ F

b. Depth First Search (DFS)

Initial fringe: (A/0) closed: { }

1. Pop (A/0) ⟶ closed? No ⟶ ∴ add to closed.
└⟶ goal? No ⟶ ∴ expand. (B/1), (C/1) ⟶ to fringe.

Fringe: (B/1) (C/1) closed: { (A/0) }

2. Choosing deepest node i.e. node with highest depth
Here depth of B = depth of C ∴ choosing randomly.
Pop (B/1) ⟶ closed? No ⟶ ∴ add to closed.
└⟶ goal? No ⟶ ∴ expand. Add (E/2), (C/2), (A/2)

Fringe: (C/1) (E/2) (C/2) (A/2) closed: { (A/0), (B/1) }

3. Highest ~~deep~~ depth in fringe = 2. ∴ Choosing any node with depth = 2

Pop (E/2) → closed ? No → ∴ Add to closed.
↳ goal ? No → ∴ expand. Add (F/3), (D/3), (C/3), (B/3)

Fringe: (F/1) (C/2) (A/2) (F/3) (D/3) (C/3) (B/3)  closed: { (A/0), (B/1), (E/2) }

4. Highest depth = 3 ∴ choosing (F/3) to expand.

Pop (F/3) → closed ? No → ∴ add to closed.
↳ goal ? (Yes) → ∴ [stop] & return solution.

Path: A → B , B → E , E → F

a. Iterative deepening search (IDS)

Iteration 1 : Depth Limit = 0

Initial fringe: (A/0)  closed: { }

1. Pop (A/0) → closed ? No → ∴ add to closed.
↳ goal ? No → ∴ expand. → Cannot expand ∵ Depth Limit Reached.

∴ Start next iteration (∵ No ~~one~~ unexplored nodes left)

(P.T.O.)

Iteration 2 : Depth Limit (DL) = 1.

Initial fringe: (A/0) closed: { }

1. Pop (A/0) → closed? No → ∴ add to closed.
   └→ goal? No → ∴ expand. Add (B/1), (C/1) to fringe

Fringe: (B/1) (C/1) closed: { (A/0) }

2. Choose deepest node.
   Pop (B/1) → closed? No → ∴ add to closed
   └→ goal? No → ∴ expand → Not possible ∵ DL reached.

Fringe: (C/1) closed: { (A/0), (B/1) }

3. Choose next deepest node.
   Pop (C/1) → closed? No → ∴ add to closed.
   └→ goal? No → ∴ expand → Not possible ∵ DL reached

Fringe : closed : { (A/0), (B/1), (C/1) }

No new nodes to explore ∵ fringe is empty
∴ increased DL to 2 and start new iteration.

Iteration 3 : Depth Limit (DL) = 2.

Initial fringe: (A/0) closed : { }

1. Pop (A/0) → closed? No → ∴ add to closed
   └→ goal? No → ∴ expand. Add (B/1), (C/1) to fringe.

Fringe: (B/1) (C/1) closed: { (A/0) }

2. Pop (B/1) → closed? No → ∴ add to closed.
↳ goal? No → ∴ expand. Add (E/2), (C/2), (A/2).

Fringe: (E/1) (E/2) (C/2) (A/2) closed: { (A/0), (B/1) }

3. Pop (E/2) → closed? No → ∴ add to closed.
↳ goal? No → ∴ expand → Not possible ∵ DL reached.

Fringe: (E/1) (E/2) (A/2) closed: { (A/0), (B/1), (E/2) }

4. Pop (E/2) → closed? No → ∴ add to closed
↳ goal? No → ∴ expand → Not possible ∵ DL reached.

Fringe: (E/1) (A/2) closed: { (A/0), (B/1), (E/2), (C/2) }

5. Pop (A/2) } closed? Yes → ∴ ignore & go to next
6. Pop (E/1) }

Fringe: closed: { (A/0), (B/1), (E/2), (C/2) }

Iteration 4: Depth limit (DL) = 3.

Initial fringe: (A/0) closed: { }

1. Pop (A/0) → closed? No → ∴ add to closed.
↳ goal? No → ∴ expand. Add (B/1), (C/1) to fringe.

Fringe: (B/1) (C/1) closed: { (A/0) }

2. Pop (B/1) → closed ? No → ∴ add to closed
↳ goal ? No → ∴ expand · Add (E/2), (C/2), (A/2)

Fringe: (C/1) (E/2) (C/2) (A/2) closed : { (A/0), (B/1) }

3. Pop (E/2) → closed ? No → ∴ add to closed
↳ goal ? No → ∴ expand · Add (F/3), (D/3), (C/3), (B/3)

Fringe: (C/1) (C/2) (A/2) (F/3) (D/3) (C/3) (B/3) closed : { (A/0), (B/1), (E/2) }

4. Pop (F/3) → closed ? No → ∴ add to closed
↳ goal ? (Yes) → ∴ [stop] & return solution.

Path : A → B, B → E, E → F.

d. Uniform cost search (UCS):

Initial fringe: (A/0) closed : { }

[Note : Node represented as (node/cost)]

1. Pop (A/0) → closed ? No → ∴ add to closed
↳ goal ? No → ∴ expand · Add (B/6), (C/12)

Fringe: (B/6) (C/12) closed : { (A/0) }

2. Pop (B/6) → closed ? No → ∴ add to closed
↳ goal ? No → ∴ expand . Add (E/14), (C/9), (A/12)

Fringe: (C/12) (E/14) (C/9) (A/12) closed : { (A/0), (B/6) }

3. Pop (C/9) → closed? No → add to closed
↳ goal? No → ∴ expand. Add (A/21), (B/12), (E/18), (D/17)

Fringe: (E/12) (E/14) (A/12) (A/21) (B/12) (E/18) (D/17) closed: {(A/0), (B/16), (C/9)}

4. Pop (E/12)
5. Pop (A/12)
6. Pop (B/12)
} closed? Yes → ∴ ignore & go to next

7. Pop (E/14) → closed? No → ∴ add to closed
↳ goal? No → ∴ expand. Add (F/18), (D/19), (C/23), (B/22)

Fringe: (A/21) (E/18) (D/17) (F/18) (D/19) (C/23) (B/22)

closed: {(A/0), (B/16), (C/9), (E/14)}

8. Pop (D/17) → closed? ~~Yes~~ No → ∴ add to closed.
↳ goal? No → ∴ expand. Add (F/27), (E/22), (C/25)

Fringe: (A/21) (E/18) (F/18) (D/19) (C/23) (B/22) (F/27) (E/22) (C/25)

closed: {(A/0), (B/16), (C/9), (E/14), (D/17)}

9. Pop (E/18) → closed? Yes → ∴ ignore & go to next.

Fringe: (A/21) (F/18) (D/19) (C/23) (B/22) (F/27) (E/22) (C/25)

closed: {(A/0), (B/16), (C/9), (E/14), (D/17)}

10. Pop (F/18) → closed? No → ∴ add to closed.
↳ goal? (Yes) → ∴ stop & return solution.

Path: A → B, B → E, E → F.

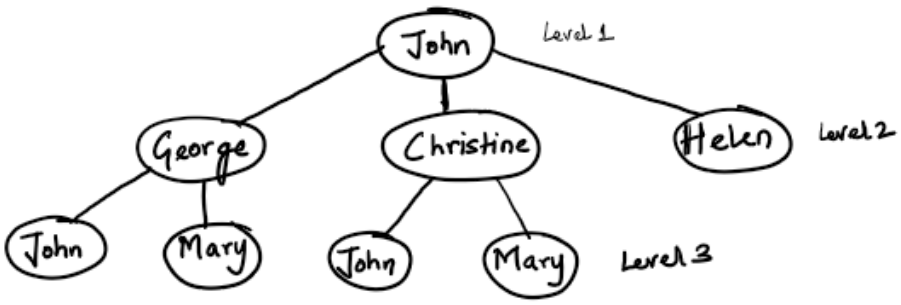
**Task 2: A social network graph (SNG) is a graph where each vertex is a person and each edge represents an acquaintance. In other words, an SNG is a graph showing who knows who. For example, in the graph shown on Figure 2, George knows Mary and John, Mary knows Christine, Peter and George, John knows Christine, Helen and George, Christine knows Mary and John, Helen knows John, Peter knows Mary.**
**The degrees of separation measure how closely connected two people are in the graph. For example, John has 0 degrees of separation from himself, 1 degree of separation from Christine, 2 degrees of separation from Mary, and 3 degrees of separation from Peter.**

**1. From among general tree search using breadth-first search, depth-first search, iterative deepening search, and uniform cost search, which one(s) guarantee finding the correct number of degrees of separation between any two people in the graph?**
**Ans.**

a. Breadth First Search
b. Iterative Deepening Search
c. Uniform Cost Search

Since the above three search strategies give optimal solutions (BFS and IDS: path cost should be 1), they can be used to find the correct number of degrees of separation between any two people in the graph.

**2. For the SNG shown in Figure 2, draw the first three levels of the search tree, with John as the starting point (the first level of the tree is the root).**
**Ans.**



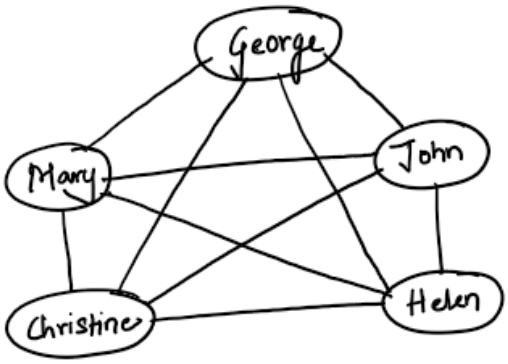

**3. Is there a one-to-one correspondence between nodes in the search tree and vertices in the SNG (i.e. does every node in the search tree correspond to a vertex in the SNG)? Why, or why not? In your answer here, you should assume that the search algorithm does not try to avoid revisiting the same state.**
**Ans.**
There is no one-to-one correspondence between the given graph (SNG) and the search tree because in a search tree the same person can be repeated multiple times. The reason for this is that, in SNG the relation between 2 people is bidirectional. For example: If John knows George then George knows John.

In question 2 answer diagram, we can see the same. George is the child node of John in Level 2 and John is the child node of George again in Level 3.

**4. Draw an SNG containing exactly 5 people, where at least two people have 4 degrees of separation between them.**
**Ans.**
Assuming that this question is not in reference to the given diagram (Figure 2 as per website).


Here, Peter and Christine are having degree of separation = 4.

**5. Draw an SNG containing exactly 5 people, where all people have 1 degree of separation between them.**
**Ans.**




Here the degree of separation between each person is 1.

**6. In an implementation of breadth-first tree search for finding degrees of separation, suppose that every node in the search tree takes 1KB of memory. Suppose that the SNG contains one million people. Outline (briefly but precisely) how to make sure that the memory required to store search tree nodes will not exceed 1GB (the correct answer can be described in one-two lines of text). In your answer here you are free to enhance/modify the breadth-first search implementation as you wish, as long as it remains breadth-first (a modification that, for example, converts breadth-first search into depth-first search or iterative deepening search is not allowed).**
**Ans.**
In such a case, we can implement the Breadth First using Graph search.

Basically,

a. We need to keep track of the nodes that are already visited by listing them as closed (since further expansion of these nodes will only cause generation of multiple nodes for the same person).
b. And before expansion of any node, we should check if that node belongs to a closed list or not. If it belongs then ignore it and go to the next node or else expand and add the new nodes to the fringe.

**Task 3: Consider the search space shown in Figure 3. D is the only goal state. Costs are undirected. For each of the following heuristics, determine if it is admissible or not. For non-admissible heuristics, modify their values as needed to make them admissible.**
**Ans:**

For a heuristic function $h(n)$ to be admissible, it should not overestimate i.e. if $h^*(n)$ is the actual/ true cost of going from node $n$ to goal then value of $h(n)$ should be in the following range.

$$0 \leq h(n) \leq h^*(n)$$

**Heuristic 1:**

| Node n | h(n) | h*(n) | Admissible or not? If not, why? | Corrected h(n) |
|---|---|---|---|---|
| A | 5 | 27 | Admissible | 5 |
| B | 20 | 17 | Not admissible, since h(n) > h*(n) | ≤17 |
| C | 15 | 5 | Not admissible, since h(n) > h*(n) | ≤5 |
| D | 0 | 0 | Admissible | 0 |
| E | 10 | 25 | Admissible | 10 |
| F | 0 | 9 | Admissible | 0 |

**Heuristic 2:**

| Node n | h(n) | h*(n) | Admissible or not? If not, why? | Corrected h(n) |
|---|---|---|---|---|
| A | 40 | 27 | Not admissible, since h(n) > h*(n) | ≤27 |
| B | 40 | 17 | Not admissible, since h(n) > h*(n) | ≤17 |
| C | 40 | 5 | Not admissible, since h(n) > h*(n) | ≤5 |
| D | 40 | 0 | Not admissible, since h(n) > h*(n) | 0 |
| E | 40 | 25 | Not admissible, since h(n) > h*(n) | ≤25 |
| F | 40 | 9 | Not admissible, since h(n) > h*(n) | ≤9 |

**Heuristic 3:**

| Node n | h(n) | h*(n) | Admissible or not? If not, why? | Corrected h(n) |
|---|---|---|---|---|
| A | 10 | 27 | Admissible | 10 |
| B | 15 | 17 | Admissible | 15 |
| C | 0 | 5 | Admissible | 0 |
| D | 0 | 0 | Admissible | 0 |
| E | 25 | 25 | Admissible | 25 |
| F | 5 | 9 | Admissible | 5 |

**Heuristic 4:**

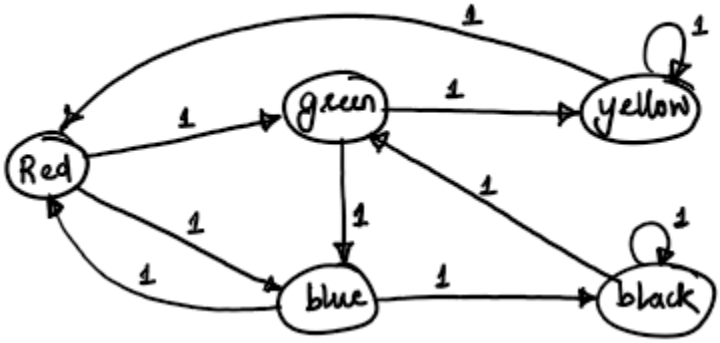
| Node n | h(n) | h*(n) | Admissible or not? If not, why? | Corrected h(n) |
|---|---|---|---|---|
| A | 0 | 27 | Admissible | 0 |
| B | 0 | 17 | Admissible | 0 |
| C | 0 | 5 | Admissible | 0 |
| D | 0 | 0 | Admissible | 0 |
| E | 0 | 25 | Admissible | 0 |
| F | 0 | 9 | Admissible | 0 |

**Task 4: Consider a search space, where each state can be red, green, blue, yellow, or black. Multiple states may have the same color. The goal is to reach any black state. Here are some rules on the successors of different states, based on their color (these successor functions are unidirectional):**

- **Red states can only have green or blue children.**
- **Blue states can only have red or black children.**
- **Green states can only have blue or yellow children.**
- **Yellow states can only have yellow or red children.**
- **Black states can only have green or black children.**

**Define a maximally admissible heuristic that assigns a value to each state based only on the color of that state.**
**Assume that any move from one state to another has cost 1.**
**Ans:**

An admissible heuristic $h(n)$ is less than or equal to $h^*(n)$. Therefore a maximally admissible heuristics is nothing but $h^*(n)$ of a relaxed version of the original problem.
In the original problem there can be multiple states of the same color. A maximally admissible heuristics can be derived from a simpler relaxed version of the same problem. Below is a relaxed version of the original problem where there is only one state of each color.




**Heuristics function**

| Node $n$ | $h(n)$ |
|---|---|
| black | 0 |
| blue | 1 |
| red | 2 |
| green | 2 |
| yellow | 3 |

**Task 5: Figures 4 and 5 show maps where all the towns are on a grid. Each town T has coordinates (Ti, Tj), where Ti Tj are non-negative integers. We use the term Euclidean distance for the straight-line distance between two towns, and the term driving distance for the length of the shortest driving route connecting two towns. The only roads that exist connect towns that have Euclidean (straight-line) distance 1 from each other (however, there may be towns with Euclidean distance 1 from each other that are NOT directly connected by a road, for example in Figure 6).**
**Consider greedy search, where the node to be expanded is always the one with the shortest Euclidean distance to the destination. Also consider A* search, where h(n) is the Euclidean distance from n to the destination (remember that the next node is picked not based on h(n) but based on f(n) = g(n) + h(n)). For each of the maps showing on Figures 4 and 5, which of the following statements is true?**

- **Greedy search always performs better than or the same as A*.**
- **Greedy search always performs worse than or the same as A*.**
- **Greedy search performs sometimes better, sometimes worse, and sometimes the same as A*, depending on the start and end states.**

**Justify your answer. For the purposes of this question, the performance of a search algorithm is simply measured by the number of nodes visited by that algorithm. Note that you have to provide separate answers for Figure 4 and for Figure 5.**

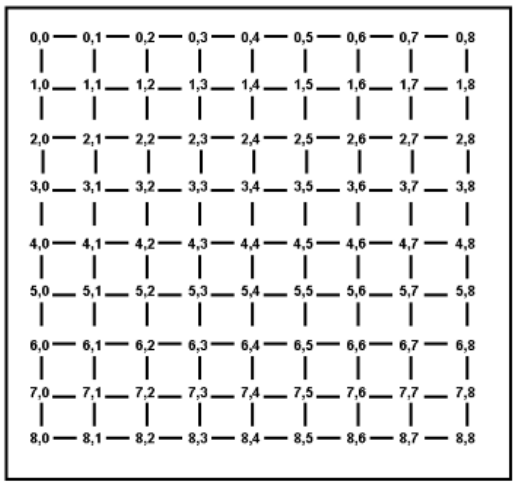


Figure 4: A map of cities on a fully connected grid. Every city is simply named by its coordinates.

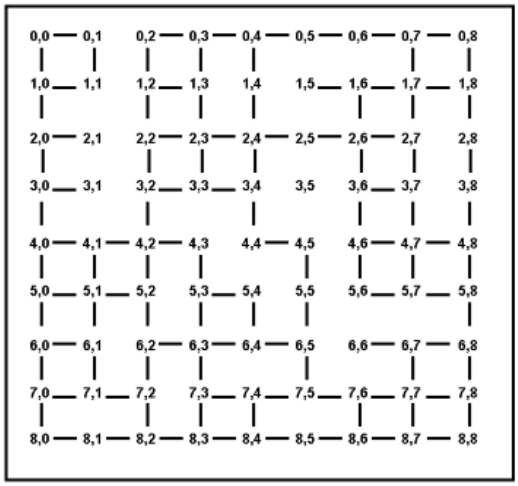


Figure 5: A map of cities on a partially connected grid. Every city is simply named by its coordinates.

**Ans:**
**Figure 4:**
For map in figure 4, "Greedy search always performs better than or the same as A*" is true.

In greedy search, the agent will always try to find the path through the node that is closest to the goal node using the heuristics h(n) which here is Euclidean distance. As in figure 4 all the nodes are connected, greedy search will always find an optimal path. But in A* search, the agent will find the path using heuristics h(n) and cost function g(n) i.e. f(n) = g(n) + h(n). While doing so A* search can lead to expansion of more nodes as compared to greedy search.

For example, for finding a path from node (0,0) to (5,5). Here the greedy search will expand only one of the nodes (1,0) and (0,1) whereas in A* search, it will expand both the nodes.

**Figure 5:**
For map in figure 5, "Greedy search performs sometimes better, sometimes worse, and sometimes the same as A*, depending on the start and end states" is true.

This map is not completely connected and thus greedy search will not be able to find the optimal solution always. Sometimes it can cause the greedy search to get stuck in a loop. For example, for finding the path from (3,0) to (3,2). Greedy search will expand (3,0) to (3,1) as it is closer to (3,2) but there is no path between (3,1) and (3,2) causing (3,1) to expand to (3,0) again. Sometimes greedy will perform better than A* for the areas where the nodes are completely connected. For example, for finding paths between the nodes (0,2) and (3,3).