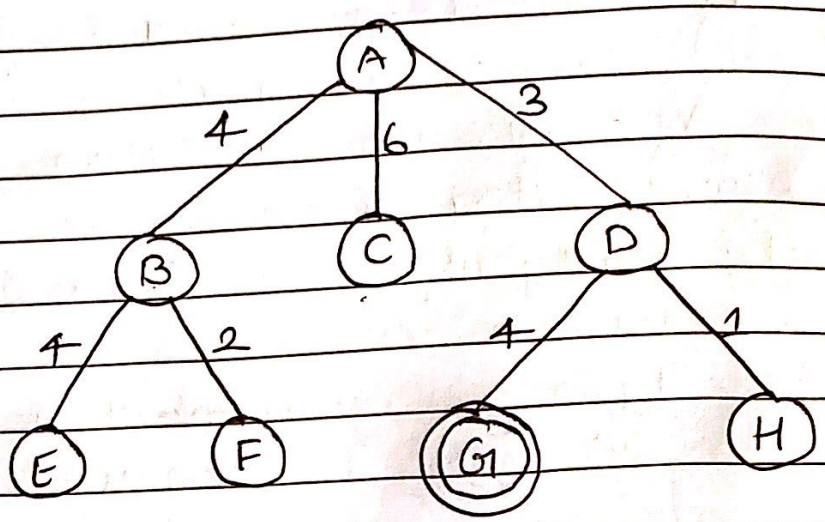
CSE 5360-001.

PRATIK PRAMOD SINGH
UTA ID :- 1001670417.

Page No. 1
Date

ASSIGNMENT - 2.

Q.1.)



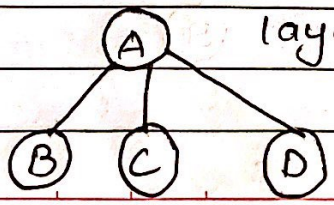
• Breadth - First Search (BFS)

→ The BFS uses a First in First out (FIFO) queue. As we know, A is the starting node & G1 is the goal node.

→ BFS will work in the following order. It traverse the graph horizontally & visit all nodes of that layer & proceed to next layer.

• Iteration for above Eg:-

1.) A is popped from the queue & checked for goal node. As it is not the goal node, it's mark visited & next layer nodes are traversed (B, C & D).



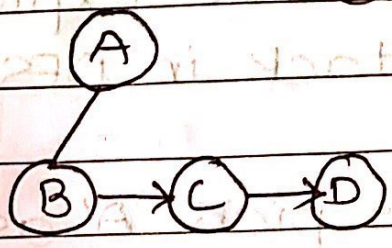
layer 0 visited : [A].

layer 1.

2.) As in iteration 1, A is not the goal node, BFS moves to the neighbours in layer 1 & traverse each node But none of the nodes in layer 1 are the goal node, so they are marked as visited & BFS moves to the next layer.

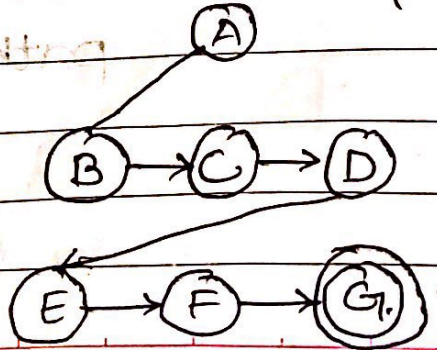
Current Path (C.P.) = [A, B, C, D]

visited = [A, B, C, D].



3.) Now we traverse layer 3. The nodes are E, F, G1 & H. Every node is checked if it is the goal node or not. If the goal is not same as the traversed node, then it's marked visited & we go on to the next node. If the node traversed is the goal node, then we mark it as visited & return the path. In this layer we first traverse E & F & then go to the node G1 & it's the goal node. So we mark it visited & return the path.

path = [A, B, C, D, E, F, G1].



Date

• Depth-First Search [DFS]

→ DFS uses backtracking. The algorithm moves on the current path & then backtracks when there are no paths available. Before selecting the next path all nodes on current, which are not visited are explored. We use stack in DFS implementation

→ Here, we pick 'A' as the start node & the adjacent nodes are pushed in the stack. Every node is popped & check if it's the goal node.

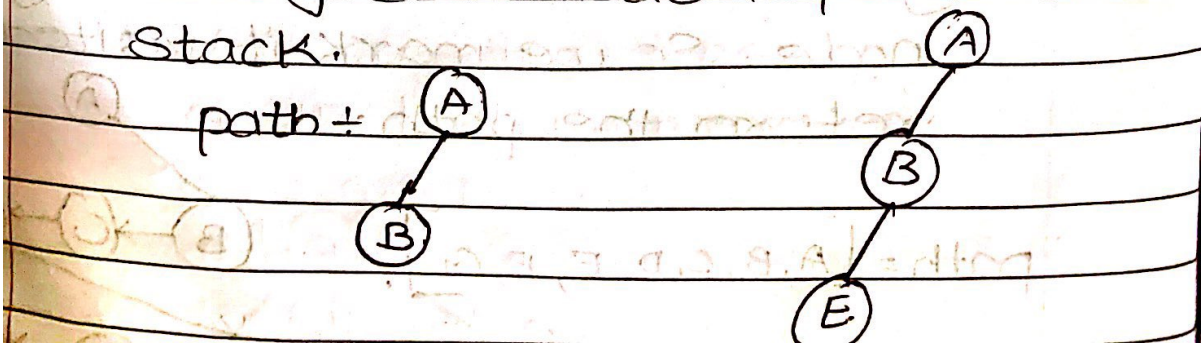
• Iteration for above Eg:-

1.) A is selected & adjacent nodes are pushed to stack.

path :- (A).

2.) DFS explores the tree from left to right. Hence 'B' is popped & it's adjacent node is pushed to stack.

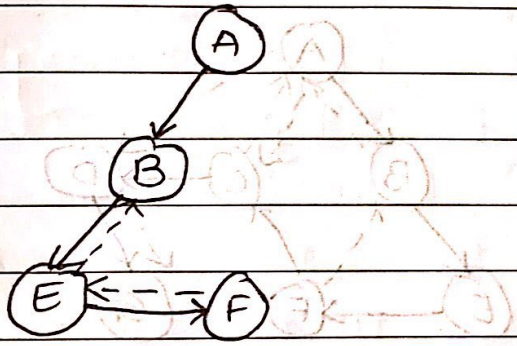
path :- (A) (B)



3.) The pop process is done again for 'E' & since it's not the goal node, the algorithm backtracks & checks for adjacent node of 'B'. Since, 'B' is traversed & marked visited; It is not added to path again.

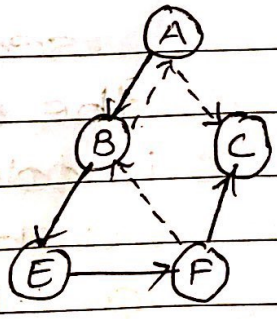
path :- [A, B, E, F].

Note :- ⟶ : Explore path
⇢ : traverse path(backtrack).



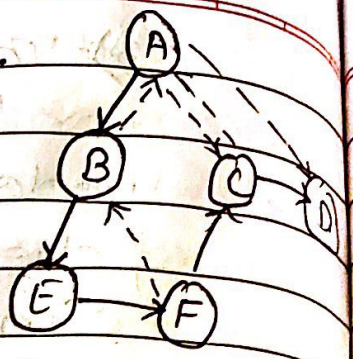
4.) As F is not the goal node, BFS traverse back to 'A' to check for other adjacent nodes.

path : [A, B, E, F, C].



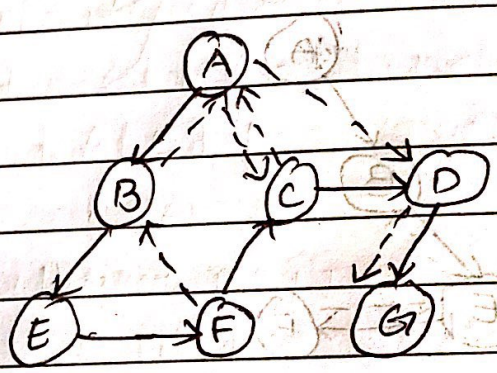
5.) DFS backtracks to 'A' as 'C' doesn't have any adjacent node. It then pop 'D' & push adjacent node to stack

path :- [A, B, E, F, C, D].



6.) As, D is explored & it is not goal node. Hence 'G' is popped. Now, the algorithm will terminate as the goal node is found & the path is returned.

path :- [A, B, E, F, C, D, G].



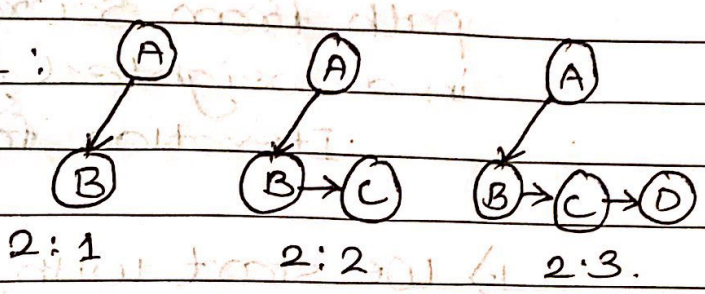
- Iterative Deepening Search (IDS).

→ IDS is a combination of depth first search & breadth first search. In IDS, in every iteration the depth is increased & then the nodes are traversed in a way such that it performs DFS in a BFS manner. In IDS, the top level node is visited multiple times & the last level node is visited once.

Iteration for the above Eg:-

1.) level 0 : (A) path : [A].

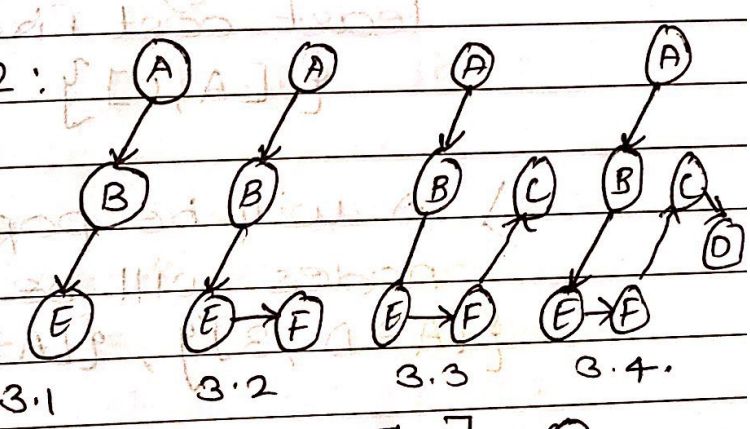
2.) level 1 :



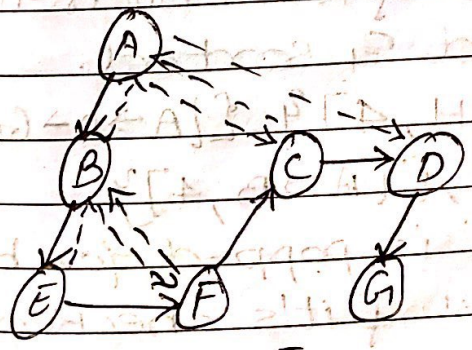
In the above iteration 2:1, 2:2, 2:3 IDS is limited to level 1. Hence it uses BFS on level 1.

path : [A, B, C, D]

3.) level 2 :



path : [A, B, E, F, C, D, G].



As we see from 3.1 to 3.5. IDS has it's max depth at 2 & explores from start node to max depth node using DFS & then uses BFS to explore neighbouring nodes

- Uniform Cost Search (UCS)
  → UCS is best for search problem
  UCS always allow to find optimal path from source to goal node in a weighted graph.
  - Iteration for above Eg:-

1.) We start with node A & will check for goal node in every iteration. We store visited nodes & cost ~~with~~ to visit each node. A priority queue is made in a least cost first way & arranged.
{[A, 0]}

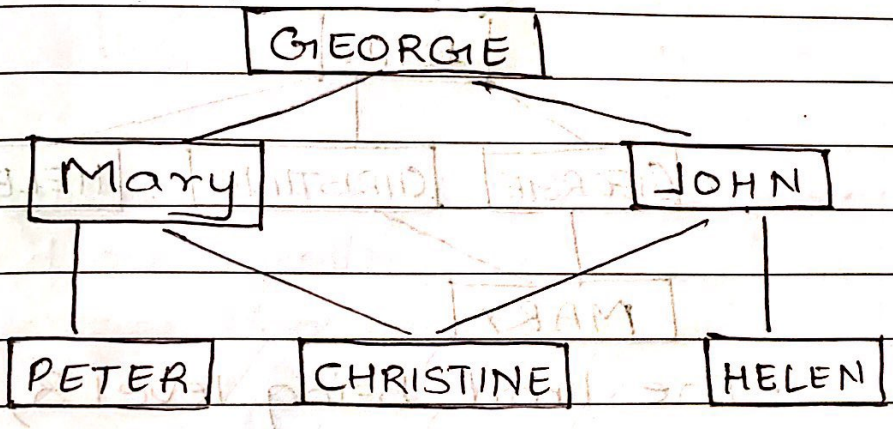
2.) A will be popped & children nodes will be pushed & stored
{[A→D, 3]}, {[A→C, 6]}, {[A→B, 4]}

3.) Node 'D' will be popped being the cheapest & it's children will be pushed & sorted.
{[A→D→H, 4]}, {[A→D→G, 7]}, {[A→C, 6]}, {[A→B, 4]}

4.) Node 'H' is popped for being the cheapest & it's checked for goal node. As it is not the goal node, the next cheapest node is selected & iterated

for being the goal node. And in the final iteration it will reach the goal node 'G' with the total path cost of '7'. Hence, the optimal path is [A → D → G, 7].

Q.2)

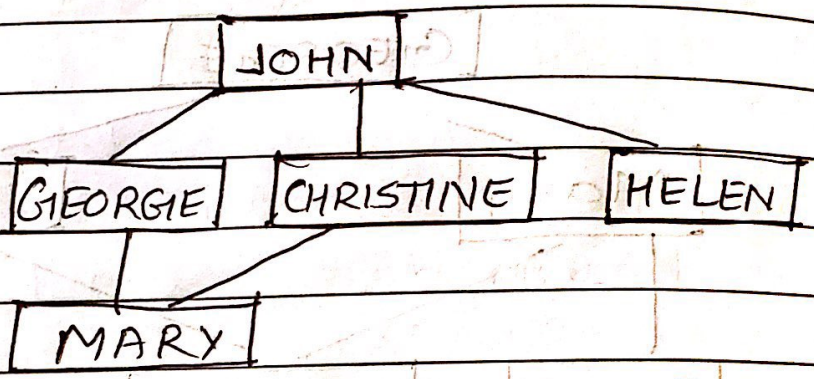


i) In the above given example, if we use the search strategies such as BFS, DFS, Iterative Deepening & Uniform Cost Search without any changes. Then the most optimal algorithm will be the uniform cost search, which will guarantee find correct number of degrees of seperation between any two people/node in graph.

The reason is UCS only gives the optimal path of nodes but will explore every node

BFS, DFS, IDS find the path, explore all nodes but doesn't gives the optimal path.

ii) Considering John as the starting point, the first three levels are



. The limit being level 3, so Peter is not considered as it is at level 4.

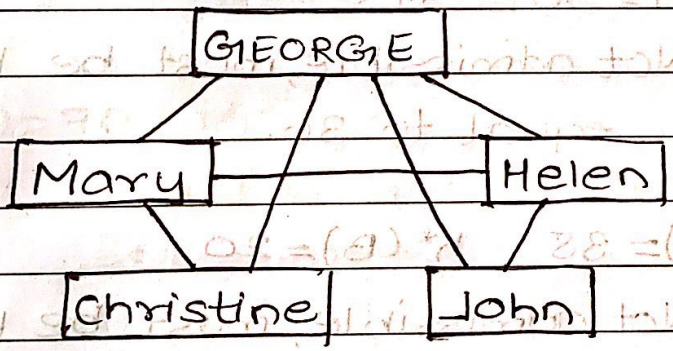
. SNG & Search tree both does not have one to one correspondence

Nodes with 1 connection: SNG:-2 ST:-1
Nodes with 2 connection: SNG:-2 ST:-3
Nodes with 3 connection: SNG:-2 ST:-1

iii) SNG containing 5 people with 4 degree of seperation between them (Atleast two people)

Here, George is connected to Helen is four degree of seperation. Also Helen is connected to george with 4 degree of seperation.

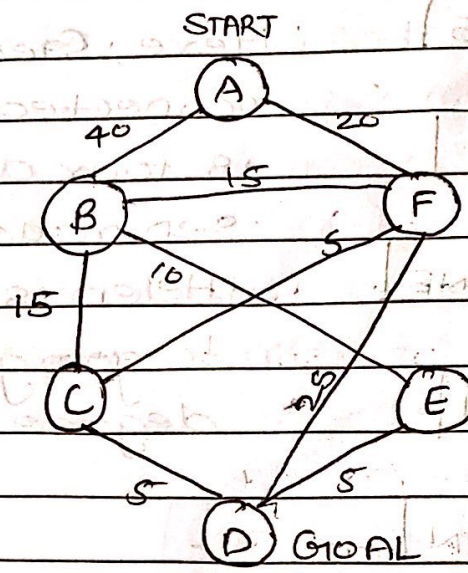
iv) SNG containing exactly 5 people & all people have 1 degree of seperation between them.



v) Since, every node in the search tree takes 1KB memory, one million people will use 10^6 KB that is approx 1GB.

The Space Complexity of BFS is $O(b^{d+1})$. If we enhance the complexity such that (10^{5+1}). Such that 10 is branching factor & 5 is the max depth, the memory req. to store node, will not exceed 1 GB.

Q.3)



• Heuristic 1 : For admissible
$h(n) <= h^*(n)$.

→ $h(A) = 50$ $h^*(A) = 30$
Not admissible, must be below or equal to 30.

→ $h(B) = 35$ $h^*(B) = 20$
Not admissible, must be less than or equal to 20.

→ $h(C) = 5$ $h^*(C) = 5$, Admissible

→ $h(D) = 0$ $h^*(D) = 0$ Admissible.

→ $h(E) = 45$ $h^*(E) = 5$
Not Admissible, must be less than or equal to 5

→ $h(F) = 10$ $h^*(F) = 10$ Admissible

• Heuristic 2 : for admissible
$h(n) <= h^*(n)$.

→ $h(A) = 70$ $h^*(A) = 30$.
Not Admissible, must be less then or equal to 30.

→ $h(B) = 70$, $h^*(B) = 20$
Not Admissible, must be less then or equal to 20.

→ $h(C) = 70$, $h^*(C) = 5$
Not admissible, must be less than or equal to 5.

→ $h(D) = 70$, $h^*(D) = 0$.
Not admissible, D is the goal state. Heuristic should be 0.

→ $h(E) = 70$, $h^*(E) = 5$
Not admissible, must be less than or equal to 5

→ $h(F) = 70$, $h^*(F) = 10$
Not admissible, must be less than or equal to 10.

Page No. 13
Date

• Heuristic C : For admissible
$h(n) <= h^*(n)$,

→ $h(A) = 40$, $h^*(A) = 30$ Admissible
Not Admissible, must be less than or equal to 30.

→ $h(B) = 20$, $h^*(B) = 20$
Admissible.

→ $h(C) = 5$, $h^*(C) = 5$
Admissible.

→ $h(D) = 0$, $h^*(D) = 0$
Admissible

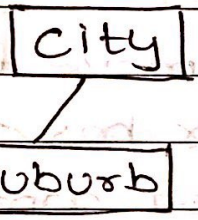
→ $h(E) = 5$, $h^*(E) = 5$
Admissible.

→ $h(F) = 20$, $h^*(F) = 10$
Not admissible, must be less than or equal to 10.

Q.4] Design Search Space from following information.

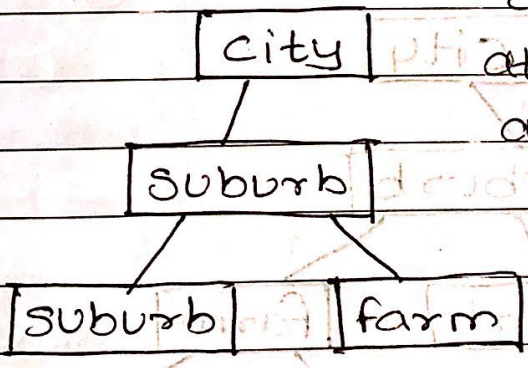
i) Successors of a city are always suburbs.

· Each city has atleast one suburb as a successor.



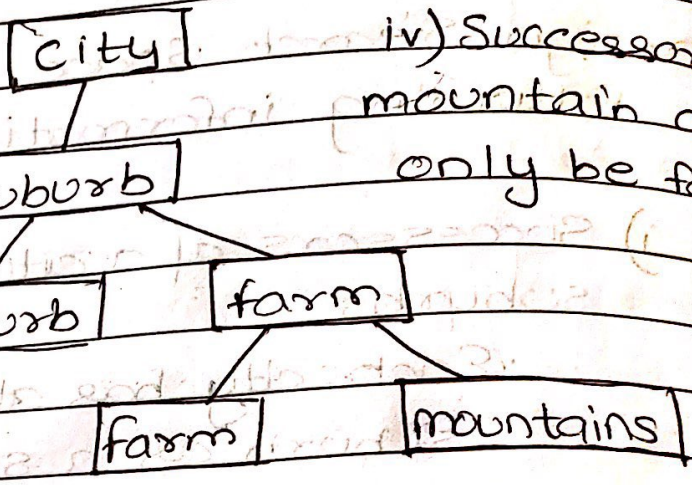
ii) Successors of a subub can only be cities, suburb or farms

· Each suburb has atleast one city as a successor.



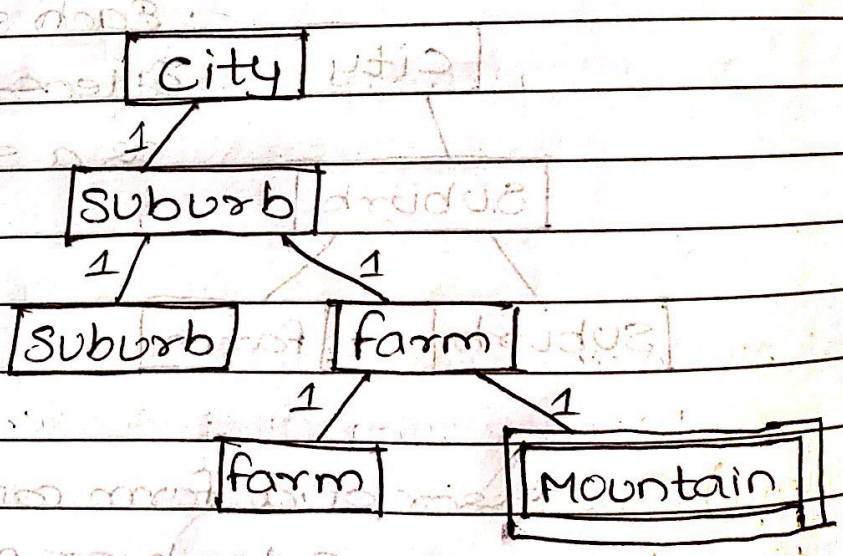
iii) Successor of a farm can be only farms, or suburb or mountain.

· Each farm has atleast one other farm as successor.



iv) Successor of mountain can only be farms

- Every move from one node to another has a cost of 1.
- The best admissible heuristic is nothing but the most optimal path from any node to a goal node. (i.e. mountain).



h (city) = 3
h (farm) = 1
h (Suburb) = 2
h (mountain) = 0.

Q.5) Shortest Solution is no longer than 100 moves (Some initial states)
Shortest solution is no longer than 208 moves (all initial states).

Memory required = 1KB for 1 node.

a] Using BFS
→ Space Complexity = $O(b^{d+1})$.

best case = 3.2^{101+1} [3.2 is the branching factor & 101 is the max depth.]

Total memory req. will exceed 50KB. And the average case = 3.2^{208+1} will be far greater than the best case needing more memory.
Also the best case = 3.2^{101+1} will also need more than 1200KB memory

b) Using DFS
→ Space Complexity = $O(b.d)$.
best case = $O(3.2 \times 201)$
Average case = $O(3.2 \times 208)$.

Here, the memory requirement will exceed 50KB & 1200KB as the max depth is '∞'.

The value is ∞ because, the goal state cannot be presumed to be at any level

c) Using IDS
→ Space Complexity : O(b.d).
best case = O(3.2 x 101)
Average case = O(3.2 x 208).

IDS will guarantee to find a solution & store nodes under the memory requirement of 1200kb.
The result is guaranted for both best & average case of the problem.

d) Using UCS.
→ Space Complexity : $O(b^{C^*/\epsilon})$.

best case : $O(3.2^{101})$
Average case. : $O(3.2^{208})$

UCS will be comparitively better than BFS in memory storage.
However it will not guarantee to store nodes under 1200 or 50kb of memory as it will need more than 1200 Kb memory in any case.

(9.6) · A* :- The value of $f(n) = g(n) + h(n)$. where $h(n)$ is the Euclidean distance between two nodes & $g(n)$ is the actual cost of travelling to each node, as $g(n)$ is same for every node in the given figure. Greedy Search will only calculate the euclidean distance $h(n)$ between two nodes & will choose the shortest value among the options.

Soln for Fig. 5

→ consider start node (0, 6)
End node (3, 7).

$0,6 - 0,7$, $h(0,6) = 3.16$, $h(0,7) = 3$
$1,6 - 1,7$, $h(1,6) = 2.23$, $h(1,7) = 2$
$2,6 - 2,7$, $h(2,6) = 1.41$, $h(2,7) = 1$.
$3,6 - 3,7$, $h(3,6) = 1$ $h(3,7) = 0$

From above values, considering source node as (0,6) greedy search will choose two options (1,6) or (0,7) & will have value 2.23 & 3. So it will choose 2.23.

Now in the second iteration it'll choose from $h(1,7)$ & $h(2,6)$ with values 2 & 1.41. It'll choose 1.41 i.e. $h(2,6)$.

Now from h(2,6) it again has two options, i.e h(2,7) & h(3,6) both with value of 1. Here, the values will be choosen at random. Then it'll move ahead to the goal state.

- In A* algorithm it'll start from h(0,0) & goal node h(3,7). It will have same method & same path as greedy search

Therefore, in fig.5 Greedy search will alway peforme same as A*, irrespective of it's start & end nodes.

* For Fig.6 :- Consider, start node : (3,1) end node : (3,2).

h(3,1) = 1 h(2,0) = 2.23

h(3,0) = 2 h(4,0) = 2.23.

Considering h(3,1) as start node, Greedy search will only have one option i.e h(3,0) with value 2.

In next step the node h(3,0) will have three options h(2,0), h(3,0) & h(4,0) with values 2.23, 2, 2.23.

The algorithm will select h(3,1) & go back to the start state.

This makes it fall into infinite loop.

Now, In A* algorithm considering h(3,1) as start node. The algorithm will find f(n) [g(n)+h(n)]

∴ f(3,1) = 1 , f(4,0) = 4.23
f(3,0) = 3 , f(4,1) = 4.44
f(4,2) = 5 , f(3,2) = 5.

The A* algorithm will follow the following path (3,1)-(3,0)-(4,0)-(4-1)-(4,2)-(3,2) as it has the lowest f(n) values.

At f(4,1), it has three option i.e f(4,0), f(5,1) & f(4,2) with values 4.23, 6.23 & 5. Here it'll not go back to f(4,0) as it is already visited. So it selects f(4,2) with value of 5 being lowest.

∴ Greedy Search performs worse or same as A* algorithm, depending upon the start & end nodes.