CSC 584 HW3 Writeup

First Steps (4pts)

I used the state map of India as reference to create a “real world” graph. I used each state as a node, and whichever state shared a border, I assigned a weight. The weight was the distance between the centre of the state to the centre of another state with which it shares a border. On average, the graph had at least 3 outgoing vertices, fulfilling the requirement in the problem statement.

The second graph was generated using a third party python library called networkx. I used a random generator, wherein I specified the number of vertices and the probability that each node has an edge as an argument. Initially, I set the edge probability for a vertex to be 0.5, but I realised that setting it to a lower value allows me to exercise the limits of my algorithm since more computation would be necessary, so I set it to 0.1. It took me a while to dump the values onto a text file and parse them onto a CSV file. I used Python for parsing the text file with the adjacency matrix. Reading from the CSV file however, I reverted to C++. My final graph had 1000 nodes.

Both these graphs were parsed into a format that I used to implement Dijkstra and A* algorithm. I used the unordered_map data structure for the purpose of implementing the graph. The data structure used, is an unordered_map of unordered_maps. The reason for choosing this format was because I would save space in large maps. My choice of data structure allows me to add edges and vertices when needed, while preserving the O(1) lookup of an adjacency matrix.

Dijkstra’s Algorithm and A* (8pts)

As mentioned previously, I used a hashmap of hashmaps to represent my graph. The algorithm takes an input of starting node and ending node and returns the shortest path in reverse order. I store the list of nodes in an array and use the pop_heap operation on the node list to obtain the node closest to our node in consideration. If the popped node is equal to the finish node, we break from the loop and return the path. Dijkstra’s algorithm is a greedy algorithm, it chooses the best option at each iteration even if it is not the most optimal solution. We also break from the loop if the closest node that the algorithm has chosen is equal to infinity. If we find a better path, we update our weights and continue our algorithm.

A* algorithm is similar to Dijkstra, the only difference between A* and Dijkstra being the choice of our heuristic. f(v) = h(v) + g(v) - where h is the heuristic and g is the weight. For Dijkstra, the heuristic defaults to 0. The admissible heuristic I’ve chosen is the Manhattan distance for this assignment is the Manhattan distance.

For my small graph, I manually entered the coordinates of the centre of each state in India. I was only able to get the coordinates in terms of latitude and longitude, but I figured out that a simple addition of 54 to the latitude and 69 to the longitude does the trick to convert a latitude and longitude to a 2-dimensional cartesian coordinate system.

I used the same logic for the random generator graph, with my heuristic being expressed in terms of coordinates. I randomly generated coordinates in the form a 50 * 20 * 2 matrix, where the x-coordinate varied from 0 - 49 and y-coordinate from 0-19. I reshaped the matrix into a 1000 * 2 matrix. I did this to ensure that all my coordinates did not lie on the same plane, and were distributed across the 50 * 20 lattice.

```
sudz@sudz-pc:~/Desktop/Sem II/CSC 584/CSC-584/HW3/Part 2$ valgrind ./a.out 0 23 14
==150080== Memcheck, a memory error detector
==150080== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==150080== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==150080== Command: ./a.out 0 23 14
==150080==
14
5
11
12
Time taken: 0.0526411
Nodes Visited: 4
==150080==
==150080== HEAP SUMMARY:
==150080==     in use at exit: 0 bytes in 0 blocks
==150080==   total heap usage: 804 allocs, 804 frees, 116,957 bytes allocated
==150080==
==150080== All heap blocks were freed -- no leaks are possible
==150080==
==150080== For lists of detected and suppressed errors, rerun with: -s
==150080== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
sudz@sudz-pc:~/Desktop/Sem II/CSC 584/CSC-584/HW3/Part 2$ valgrind ./a.out 0 23 14
==150090== Memcheck, a memory error detector
==150090== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==150090== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==150090== Command: ./a.out 0 23 14
==150090==
14
5
11
Time taken: 0.0604404
Nodes Visited: 3
==150090==
==150090== HEAP SUMMARY:
==150090==     in use at exit: 0 bytes in 0 blocks
==150090==   total heap usage: 936 allocs, 936 frees, 128,725 bytes allocated
==150090==
==150090== All heap blocks were freed -- no leaks are possible
==150090==
==150090== For lists of detected and suppressed errors, rerun with: -s
==150090== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

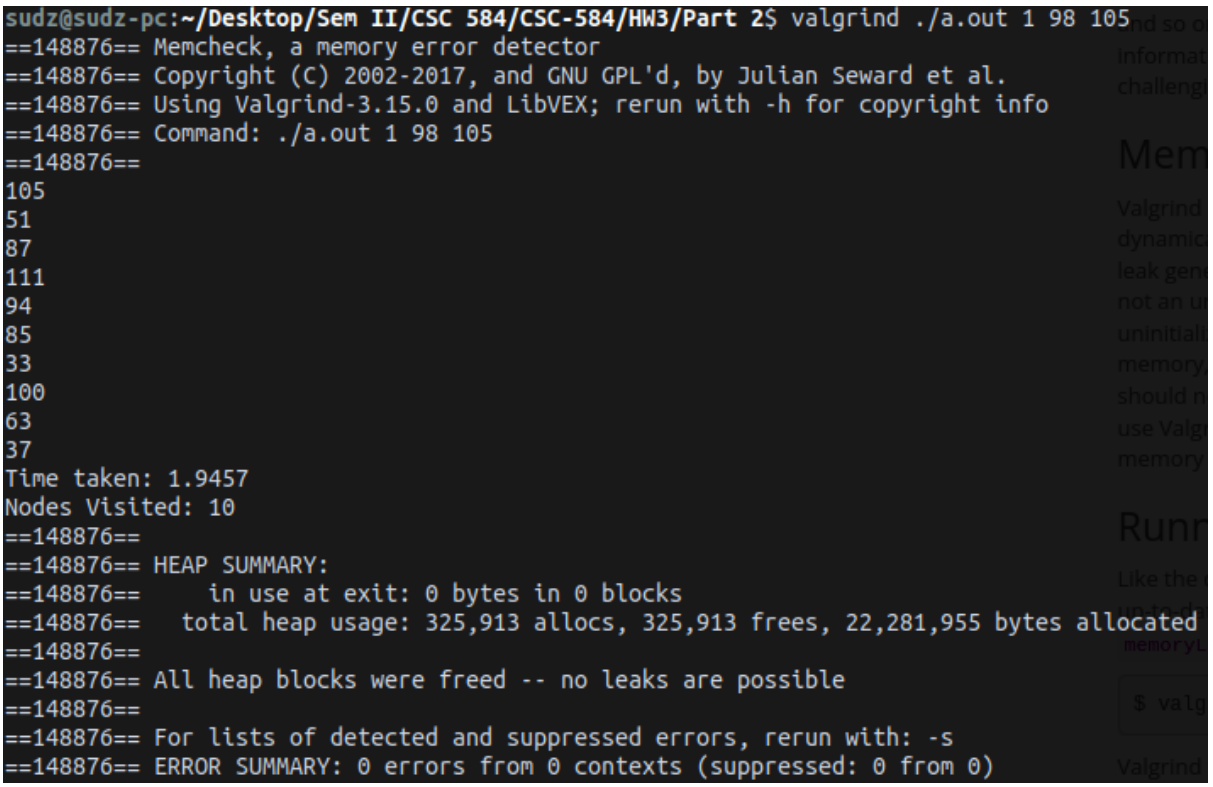

Fig - 1 Running valgrind on a) Small Graph (1) Dijkstra's algorithm and (2) A* algorithm with Manhattan heuristic b) Large Graph - (1) Dijkstra's algorithm and (2) A* algorithm with Manhattan heuristic

a) A* algorithm with Manhattan heuristic performs much better than the Greedy Best First Search Dijkstra's. We see that it takes 3 hops to travel from node 23 to 14 whereas in Dijkstra's, it takes 4 hops.

We observe that although A* algorithm takes more time to complete (0.060s) compared to Dijkstra's (0.052s), the amount of memory used is more or less the same with A* using slightly more. The tradeoff lies in the fact that Dijkstra mostly performs fairly well with small

graphs while taking lesser time, compared to A* which performs well but takes a little more time.

Note - Time taken for computation is while running valgrind, actual time taken for the large graph is in the region of 49 - 54 milliseconds.

b) A* algorithm with Manhattan heuristic performs much better than the Greedy Best First Search Dijkstra's. We see that it takes 3 hops to travel from node 98 to 105 whereas in Dijkstra's, it takes a whopping 10 hops.

We observe that although A* algorithm takes more time to complete (2.1546s) compared to Dijkstra's (1.9457s), the amount of memory used is more or less the same with A* using slightly more. But this tradeoff is acceptable since it takes only 0.2s more and uses a fraction more memory, but most importantly provides a superior solution.

Heuristics (4pts)

The admissible heuristic I used is the Manhattan distance. The Manhattan distance (or) cityblock distance allows for 4 different movement choices at a given point (left, right, top, bottom). It is computed by finding the absolute difference between the x and the y coordinates. In Manhattan distance, the scale between g(v) and h(v) matches, thus it is impossible to overestimate.

The inadmissible heuristic I considered however, is the Euclidean distance between two coordinates squared. Here, the problem lies in the fact that both g(v) and h(v) are in different scales. For long distances, this will approach the extreme of h(v) overly contributing to f(v), and A* will devolve into a Greedy Best-First-Search.

```
sudz@sudz-pc:~/Desktop/Sem II/CSC 584/CSC-584/HW3/Part 3$ c++ a_star.cpp
sudz@sudz-pc:~/Desktop/Sem II/CSC 584/CSC-584/HW3/Part 3$ valgrind ./a.out 1 98 105 0
==149602== Memcheck, a memory error detector
==149602== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==149602== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==149602== Command: ./a.out 1 98 105 0
==149602==
105
86
126
124
88
Time taken: 2.51768
Nodes Visited: 5
==149602==
==149602== HEAP SUMMARY:
==149602==     in use at exit: 0 bytes in 0 blocks
==149602==   total heap usage: 329,945 allocs, 329,945 frees, 22,391,959 bytes allocated
==149602==
==149602== All heap blocks were freed -- no leaks are possible
==149602==
==149602== For lists of detected and suppressed errors, rerun with: -s
==149602== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

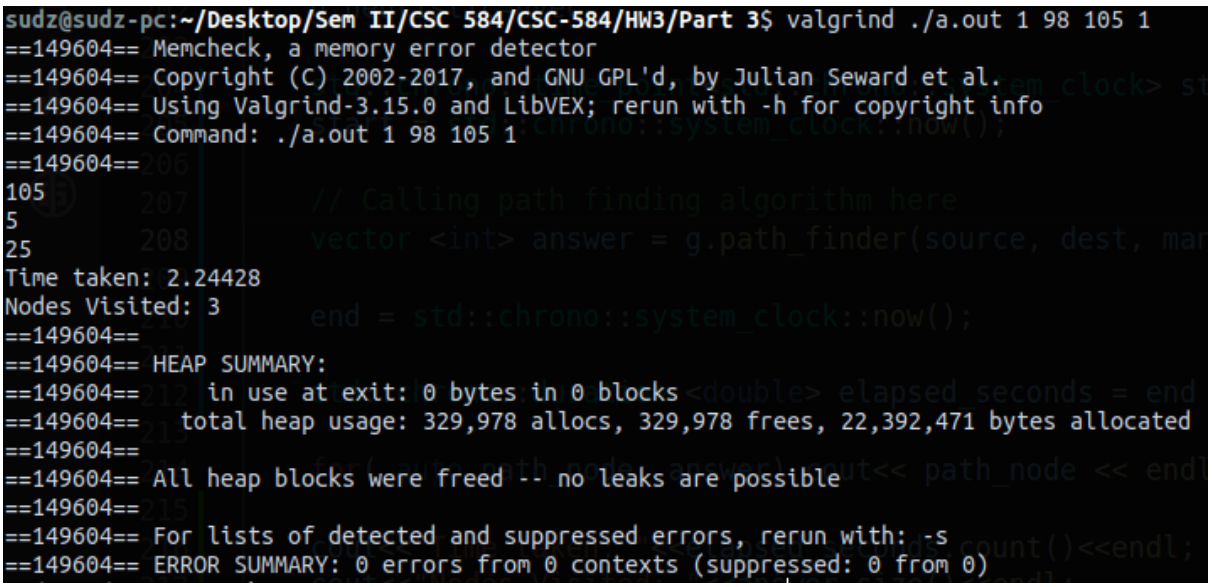

Fig 2 - Travelling from vertex 98 -> 105 using A* algorithm with 1) Euclidean squared and 2) Manhattan heuristic.

Note - Time taken for computation is while running valgrind, actual time taken for the large graph is in the region of 54 - 60 milliseconds.

From the above figure, we observe that using Euclidean distance squared as a heuristic makes us overestimate and take a longer path than necessary. Using Manhattan distance allows us to reach the destination in 3 hops whereas EDS takes 5 hops.

In terms of performance also, Manhattan distance outperforms EDS since it takes 0.27 seconds less to complete. This can be attributed to the calculation used while computing the Euclidean distance, since we perform 2 multiplications every iteration [(X2 - X1) and (Y2 - Y1)]. Memory usage however is nearly the same, which is expected.

Putting it All Together (7pts)

I designed my room environment purely using the SFML library and drew a total of 7 rooms with 11 walls. In order to achieve this design, I created a Room and wall class. Initially, I only had a wall class and I avoided creating a Room class since I thought everything else beside a wall would be a room, but soon I understood that pathfinding would be impossible without uniquely identifying every room. I assigned an unique integer to each room and mapped the coordinates of the room. Each Room has the following properties-

1. Room ID
2. Coordinates of the four corners of the room (TopLeft, TopRight, BottomRight, BottomLeft)
3. Room entrances associated with each room

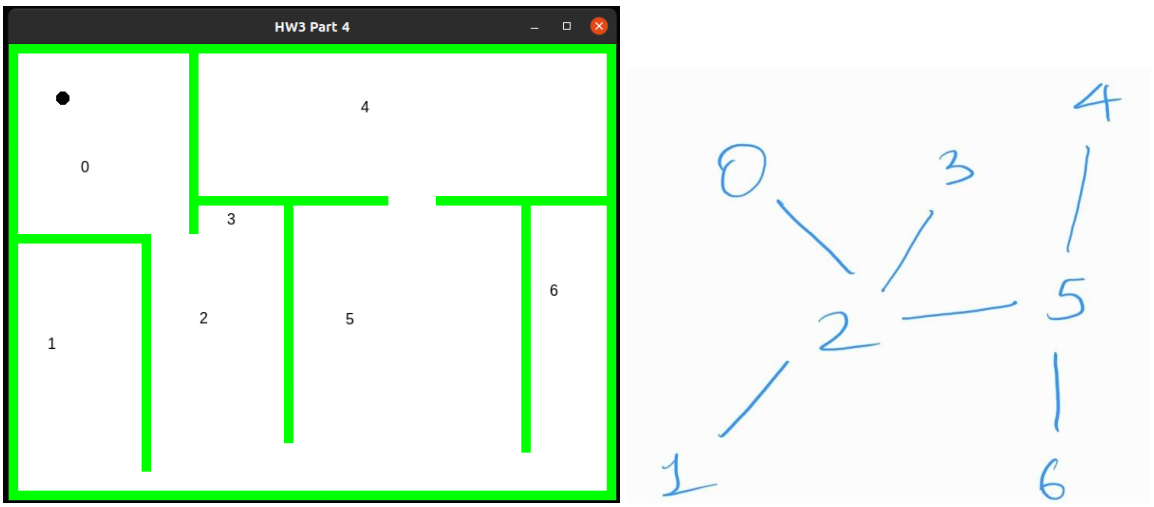

Fig 4 - Room environment (left) and its corresponding graphical sketch

The figure above shows the room environment and the sketch on the right is its graphical equivalent. I used Dijkstra's algorithm to find the path from source to destination. If I clicked on a position in the same room, I would directly move there. If my mouse click wasn't in the same room, in order to correctly identify the destination room, I iterated through each room and whichever room's coordinates was within the bounds of my mouse click was my destination.

Dijkstra's algorithm would provide an array of the shortest path. I would find the common entrance between the two rooms the sprite is supposed to travel and move there, popping off the previous room's room_id. Once the path array was empty, it means the sprite has reached the destination room. The sprite subsequently travels the offset between the room entrance and the destination mouse click.

For such an environment, assigning arbitrary weights will be sufficient. But in cases where there are 2 room entrance choices for the sprite to make, the sprite would use a distance metric(Euclidean/Manhattan) which would break the tie and provide the shortest path.

Appendix -

1	Andhra Pradesh	0	0	0	0	900	0	0	0	0	0	0	526.2	0	0	0	0	0	0	0	1030.2	0	0	0	677.6	338.7	0	0	0	0
2	Arunachal Pradesh	0	0	603.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	496	0	0	0	0	0	0	0	0	0	0
3	Assam	0	601.1	0	0	0	0	0	0	0	0	0	0	0	0	0	338.5	320.1	559	162	0	0	0	0	0	0	442.3	0	0	1104.9
4	Bihar	0	0	0	0	0	0	0	0	0	0	264.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	553	418.7
5	Chhattisgarh	1012.3	0	0	0	0	0	0	0	0	0	611.1	0	0	498	752.4	0	0	0	0	420.5	0	0	0	0	634.9	0	0	702.8	0
6	Goa	0	0	0	0	0	0	0	0	0	0	0	227.2	0	0	682	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	Gujarat	0	0	0	0	0	0	0	0	0	0	0	0	0	961.7	831.6	0	0	0	0	0	0	743.9	0	0	0	0	0	0	0
8	Haryana	0	0	0	0	0	0	0	0	353.1	0	0	0	0	0	0	0	0	0	0	0	320.6	360.7	0	0	0	0	0	676.2	0
9	Himachal Pradesh	0	0	0	0	0	0	0	353.1	0	624.1	0	0	0	0	0	0	0	0	0	0	253.1	0	0	0	0	0	486.9	0	0
10	Jammu and Kashmir	0	0	0	0	0	0	0	0	529.9	0	0	0	0	0	0	0	0	0	0	0	352.6	0	0	0	0	0	0	0	0
11	Jharkhand	0	0	0	238.8	613.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	466.1	0	0	0	0	0	0	0	733.8	333.9
12	Karnataka	585.2	0	0	0	0	229.2	0	0	0	0	0	0	673.7	0	0	0	0	0	0	0	0	0	0	693.2	594.8	0	0	0	0
13	Kerala	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	307.8	0	0	0	0	0
14	Madhya Pradesh	0	0	0	0	498	0	947.6	0	0	0	0	0	0	0	645.1	0	0	0	0	0	0	825.6	0	0	0	0	0	656.2	0
15	Maharashtra	0	0	0	0	752.4	682	831.6	0	0	0	0	0	0	645.1	0	0	0	0	0	0	0	0	0	0	618.9	0	0	0	0
16	Manipur	0	0	338.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	527.9	179	0	0	0	0	0	0	0	0	0	0
17	Meghalaya	0	0	320.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	Mizoram	0	0	559	0	0	0	0	0	0	0	0	0	0	0	0	527.9	0	0	0	0	0	0	0	0	0	360.4	0	0	0
19	Nagaland	0	496	162	0	0	0	0	0	0	0	0	0	0	0	0	179	0	0	0	0	0	0	0	0	0	0	0	0	0
20	Odisha	1030.7	0	0	0	418.8	0	0	0	0	0	454.6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	479.2
21	Punjab	0	0	0	0	0	0	0	322.5	253.1	352.6	0	0	0	0	0	0	0	0	0	0	0	587.6	0	0	0	0	0	0	0
22	Rajasthan	0	0	0	0	0	0	733.8	362.9	0	0	0	0	0	818.3	0	0	0	0	0	0	585.8	0	0	0	0	0	0	766.5	0
23	Sikkim	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	632.3
24	Tamil Nadu	677.6	0	0	0	0	0	0	0	0	0	0	693.2	307.8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	Telangana	335.1	0	0	0	638.2	0	0	0	0	0	0	585.5	0	0	618.9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	Tripura	0	0	442.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	360.4	0	0	0	0	0	0	0	0	0	0	0
27	Uttarakhand	0	0	0	0	0	0	0	0	486.9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	605.4
28	Uttar Pradesh	0	0	0	567.4	702.8	0	0	675	0	0	739.7	0	0	650.2	0	0	0	0	0	0	0	794.8	0	0	0	0	605.4	0	0
29	West Bengal	0	0	1104.9	418.7	0	0	0	0	0	0	333.9	0	0	0	0	0	0	0	0	479.2	0	0	632.3	0	0	0	0	0	0

Fig 5 - State map of India (above) and its corresponding adjacency matrix (below) in CSV format

```
sudz@sudz-pc:~/Desktop/Sem II/CSC 584/CSC-584/HW3/Part 2$ ./a.out 0 23 14
14
5
11
12
```

```
sudz@sudz-pc:~/Desktop/Sem II/CSC 584/CSC-584/HW3/Part 2$ ./a.out 0 23 14
14
5
11
```

Fig 6 - Output of Dijkstra's while travelling from TamilNadu (23) - Maharashtra (14) (23 TamilNadu -> 12 Kerala -> 11 Karnataka -> 5 Gujarat ->14 Maharashtra) and A* algorithm (23 TamilNadu -> 11 Karnataka -> 5 Gujarat ->14 Maharashtra). A* provides better performance when fed with the heuristic of the coordinates of each state's centre

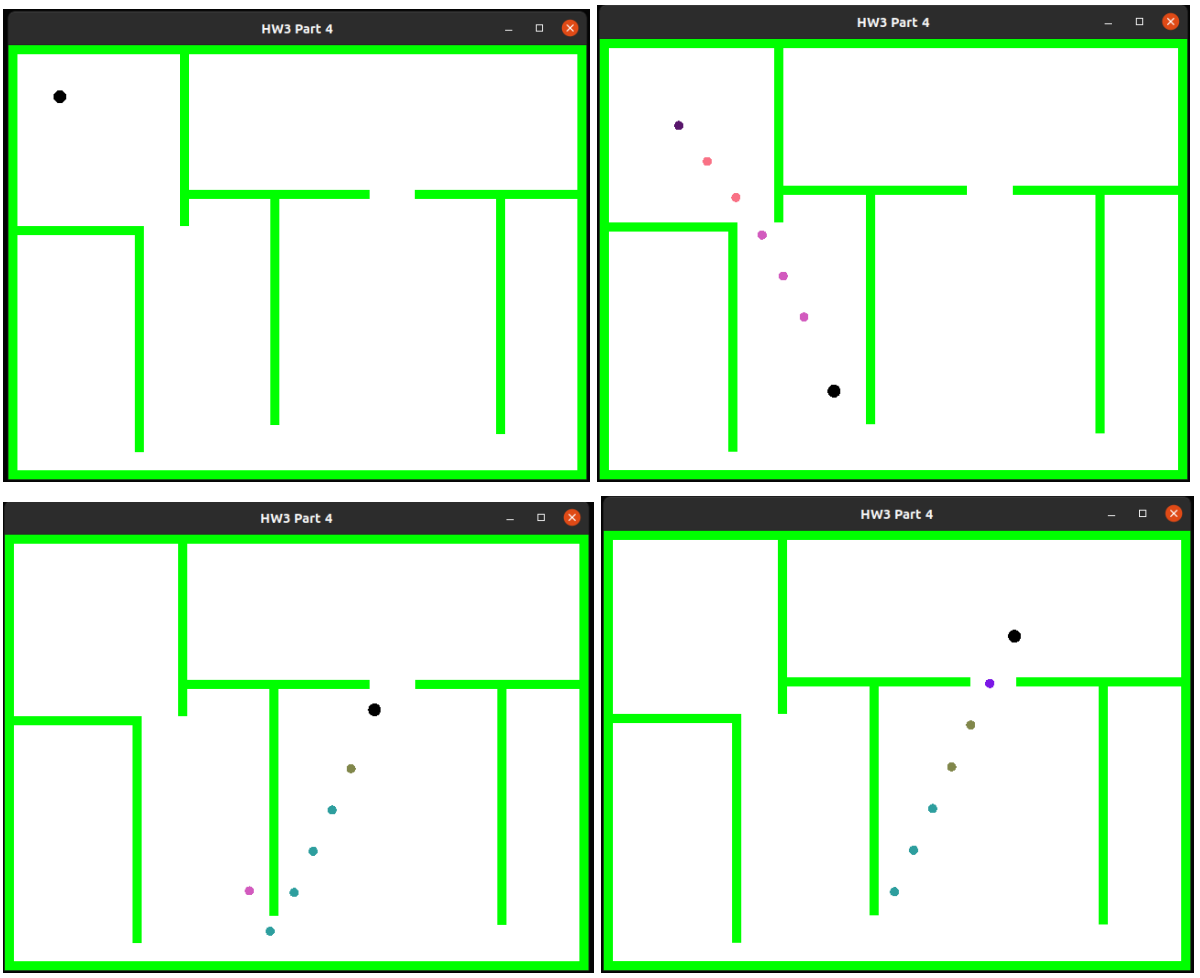

Fig 7 - Demonstration of sprite (black), moving from room 0 to room 4 (0 -> 2 -> 5 -> 4) evidenced by breadcrumbs