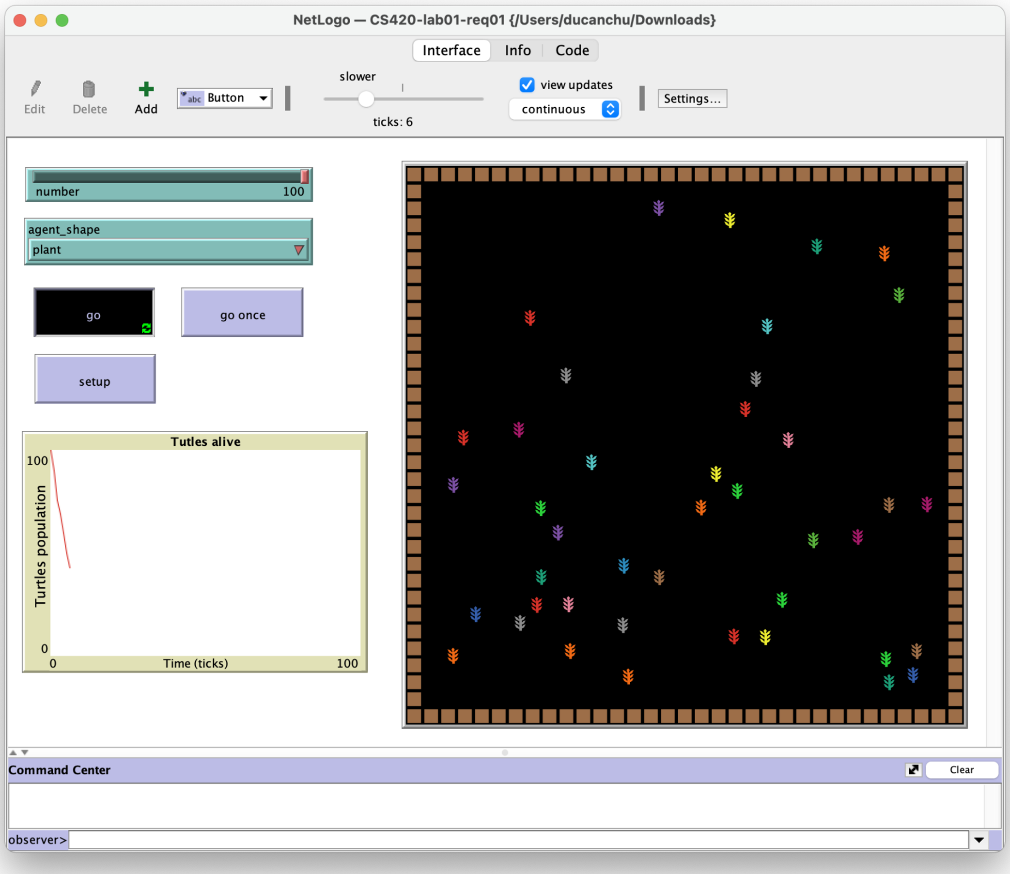
CS420 - Project 01
Searching Solver with NetLogo

1. Members and Assignment plan

- **Nguyễn Thiệu Khang. Student ID: 19125051**
  - Requirement 1 - Demo in NetLogo and Report
  - Requirement 2 - Algorithms for level 3 and commenting
- **Từ Tấn Phát. Student ID: 19125064**
  - Requirement 2 - Algorithms for levels 1 and 2, commenting and report
  - Report for Requirement 2
- **Chu Đức An. Student ID: 19125001**
  - Requirement 1 - Video clip and commenting
  - Requirement 2 - Generator algorithm and video clip for all 3 levels

Video clips drive folder:
drive.google.com/drive/folders/1Nu2UJQfzpS88m0FcyvQ13HesJbVyNWfO?usp=sharing

2. Requirement 01


NetLogo — CS420-lab01-req01 {/Users/ducanchu/Downloads}

ticks: 6

number 100

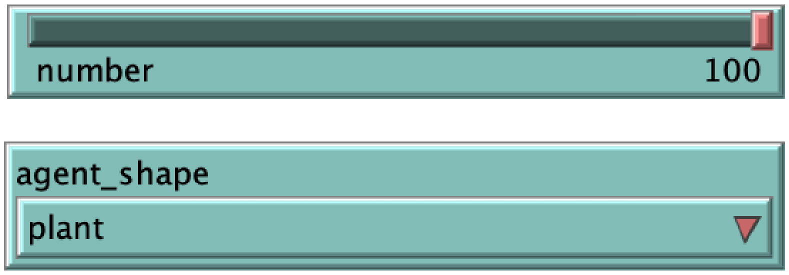
agent_shape: plant

Tutles alive

In this part, we have explored the features of NetLogo and created a sample application in which these features are applied to simulate the execution of a search algorithm. All of the tasks we have finished in this part consist of:

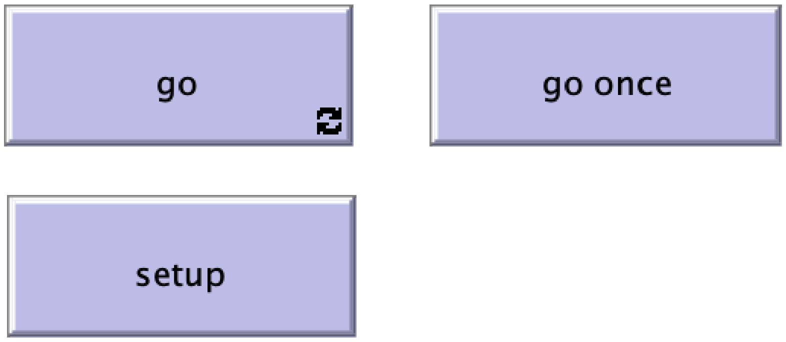
- Building a user interface including
  - A screen that shows the movement of agents in the simulator environment.
  - A chart that plots the number of agents alive in a timeframe.
  - A dropdown menu that allows users to select the agent shape (turtle, person, sheep, or plant).
  - A slider that allows users to select the initial number of agents.
  - Three buttons to set up the agents' initial position and to let the agents move one step or repeatedly.
- Creating a beautiful graphic design for the simulator environment:
  - The agents are initially generated with different colors
  - The fence and background are colored brown and black respectively so that the users can focus on the movement of the agents.
- Implementing the function of the three buttons:
  - setup: Set up the initial state of the simulator environment.
  - go: Allow all agents to move one step.
- Implementing other utility functions:
  - create-agents: Generate all agents with the selected shape.
  - create-walls: Create all of the wall blocks.

The user interface application is shown above.

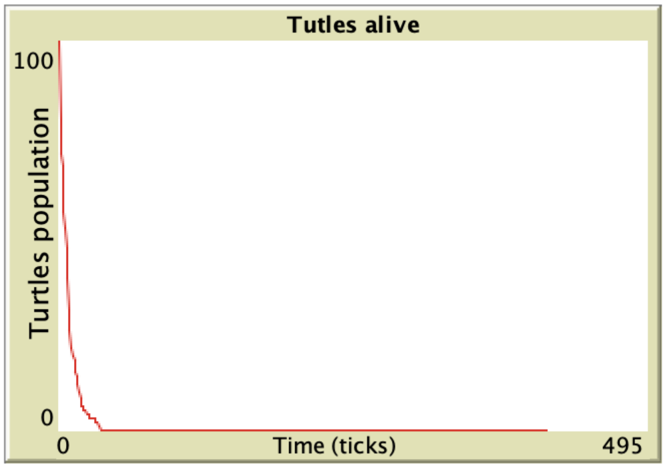

number 100

agent_shape plant

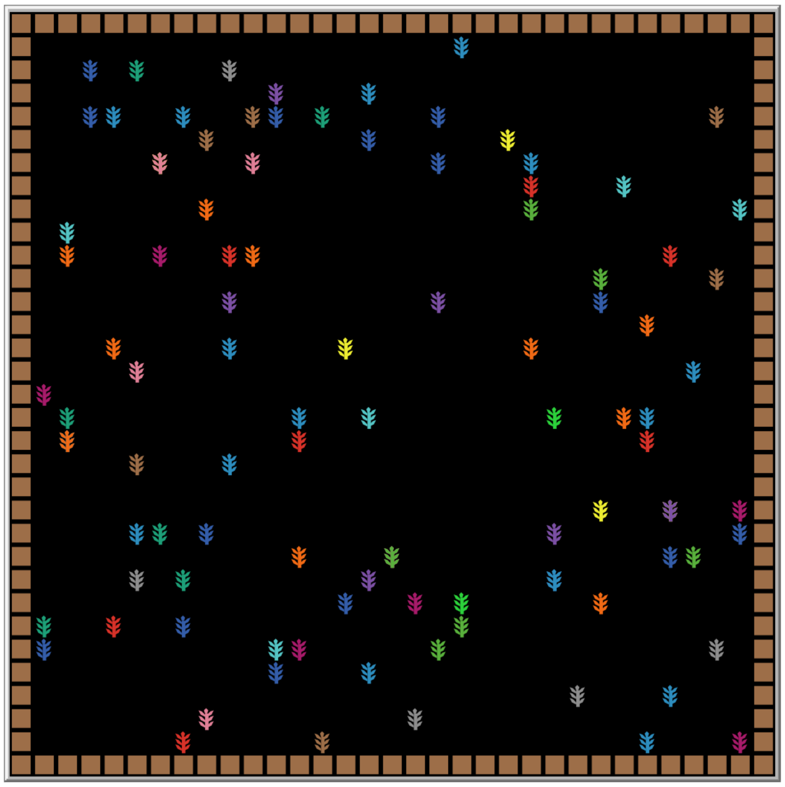
Users can easily customize the simulator environment using the dropdown menu and the slider to control the shape and number of agents.



All tasks can be easily executed with three buttons.



The agent population is intuitively plotted on a chart.



The movement of agents in the simulator environment is displayed on a big screen.

In the implementation of this part, we mainly focus on creating the functions which initiate all elements including agents and wall blocks in the environment and allow the agents to move in one step.

```
globals [
  agent_set ; The set of all the moving agents
  wall_set ; The set of all wall elements
  wall_shape
]
```

Initially, we declare three variables agent_set, wall_set, and wall_shape to store the set of all moving agents, the set of all wall blocks, and the shape of the wall respectively. Those variables are all declared globally so that all functions below can access and use their data.

```
to setup
  clear-all
  set wall_shape "square"
  create-walls
  create-agents
  set agent_set turtles with [shape = agent_shape]
  set wall_set turtles with [shape = wall_shape]
  reset-ticks
end
```

In the setup function, all of the tasks in the process of simulator environment initialization are performed. Firstly, we call the utility clear-all to swipe all kinds of stuff being shown out of the main screen. Thereafter, a new square wall is created with the following two functions. Then all agents with different colors are put on screen randomly one by one. Finally, the application timer is reset to prepare for the execution.

```
to go
  ask agent_set [
    ; Every agent moves randomly in one
    ; tick until they hit the wall and die
    rt (random 90) - (random 90)
    fd random 10
    if any? wall_set in-radius 1 [
      die
    ]
  ]
  tick
end
```

The go function performs the movement of the agent in one step and sticks with the go and go once buttons. The go once call the go function one time per click, white the go button does repeatedly. Inside the go function, we force every agent to move randomly in one direction until they hit something. If one agent hits the wall, it can be considered to be out of the map.

Besides those above main functions, there are two more utility functions that create the wall blocks and the agents at the beginning stage of each experiment. The main idea is to use the random function to generate the position of each element.

```
to create-agents
  create-turtles number [
    setxy (random (max-pxcor - min-pxcor - 1) + min-pxcor + 1)
          (random (max-pycor - min-pycor - 1) + min-pycor + 1)
    set shape agent_shape
  ]
end
```

```
to create-walls
  let cnt max-pycor
  while [cnt >= min-pycor] [
    create-turtles 1 [
      setxy max-pxcor cnt
      set shape wall_shape
      set color brown
    ]
    create-turtles 1 [
      setxy min-pxcor cnt
      set shape wall_shape
      set color brown
    ]
    set cnt cnt - 1
  ]

  set cnt max-pxcor - 1
  while [cnt >= min-pxcor + 1] [
    create-turtles 1 [
      setxy cnt max-pycor
      set shape wall_shape
      set color brown
    ]
    create-turtles 1 [
      setxy cnt min-pycor
      set shape wall_shape
      set color brown
    ]
    set cnt cnt - 1
  ]

end
```

3. Requirement 02

Requirement: Implement the uninformed and informed search algorithms on the NetLogo and then give comments and comparisons.

Environment design: We implemented a simple grid world, in which different colors represent a different kind of entity. The color-to-role mapping is as follows:

- Yellow tile: Starting points of agents (i.e. where the agent starts finding their path, the path cost is by default 1).

- Green tile: Goal points (the tile that the agents should reach in order to be verified as solved).
- White tile: Wall (there is no going through the wall, they act as a complete hindrance to agents).
- Black tile: Space (agents go through them freely with a uniform path cost of 1)
- Blue tile: Water (agents can go on the water, but the path cost will be 5 for each water tile).
- Red tile: Lava (agents can go through lava with a path cost of 10, they function as a trap)

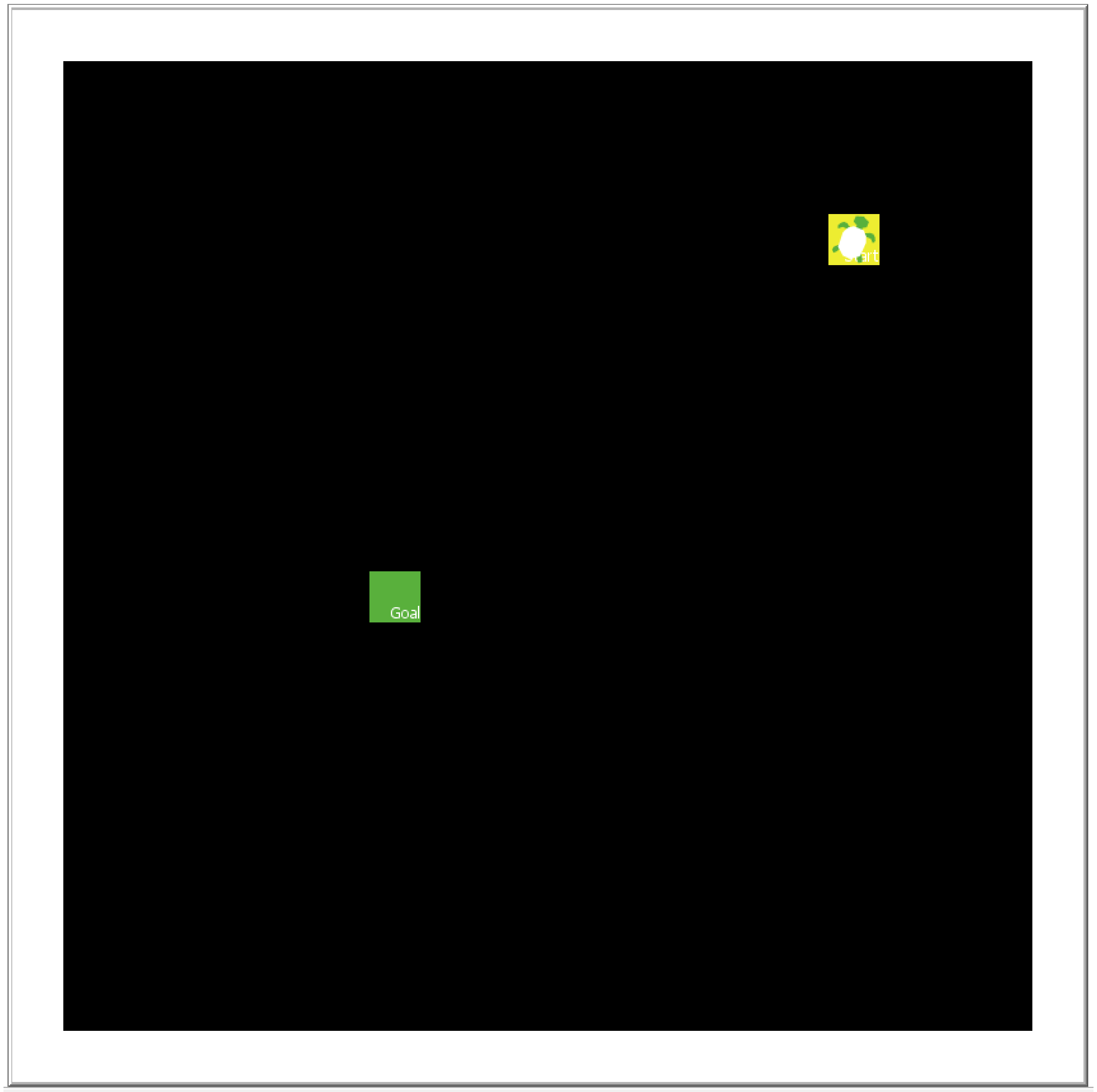
In addition, the agents are only allowed to move in the adjacent 4 neighbors (north, east, west, south). They are not allowed to move diagonally, but multiple agents are allowed to stand on the same tile. There will be 5 different maps with increasing difficulty. These maps are generated instead of hard-coded.

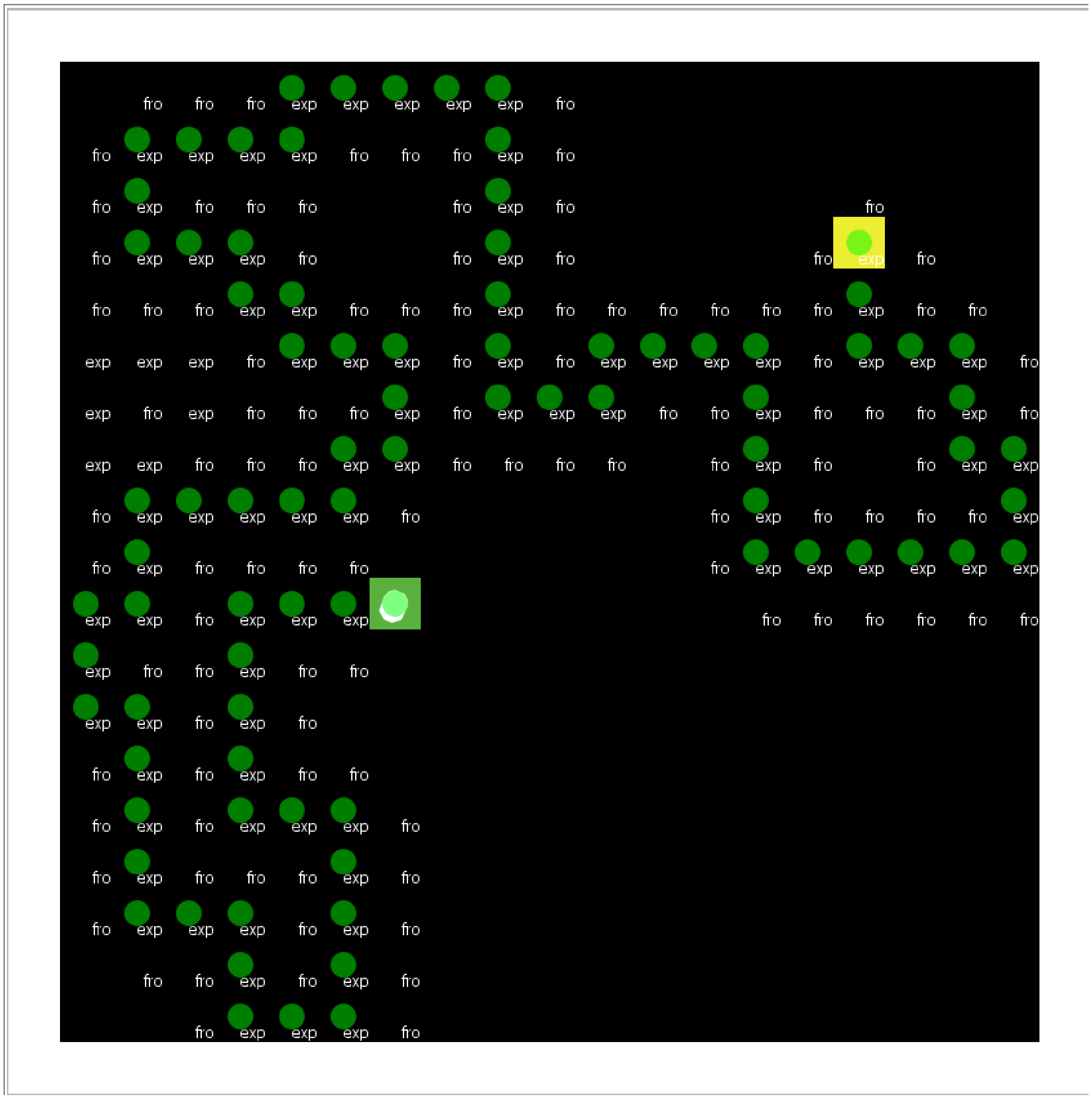
To visualize the algorithms, we also add some labels and symbols to each tile:
- “Fro”: Stands for frontier, meaning the search algorithm pushes that tile into its frontier but has not expanded it yet.
- “Exp”: Stands for expanded, meaning the algorithm already expands this tile.
- Different color circles: Denote the path of the agent, from the starting point to the goal.

The heuristic of each tile is calculated using the Pythagorean distance between the considered tile and the goal point. The heuristic, in this case, is admissible and consistent because it never overestimates the cost to the goal and it also does not overestimate the step cost (the minimum distance between two tiles in the grid world is the Pythagorean distance or the Manhattan distance).

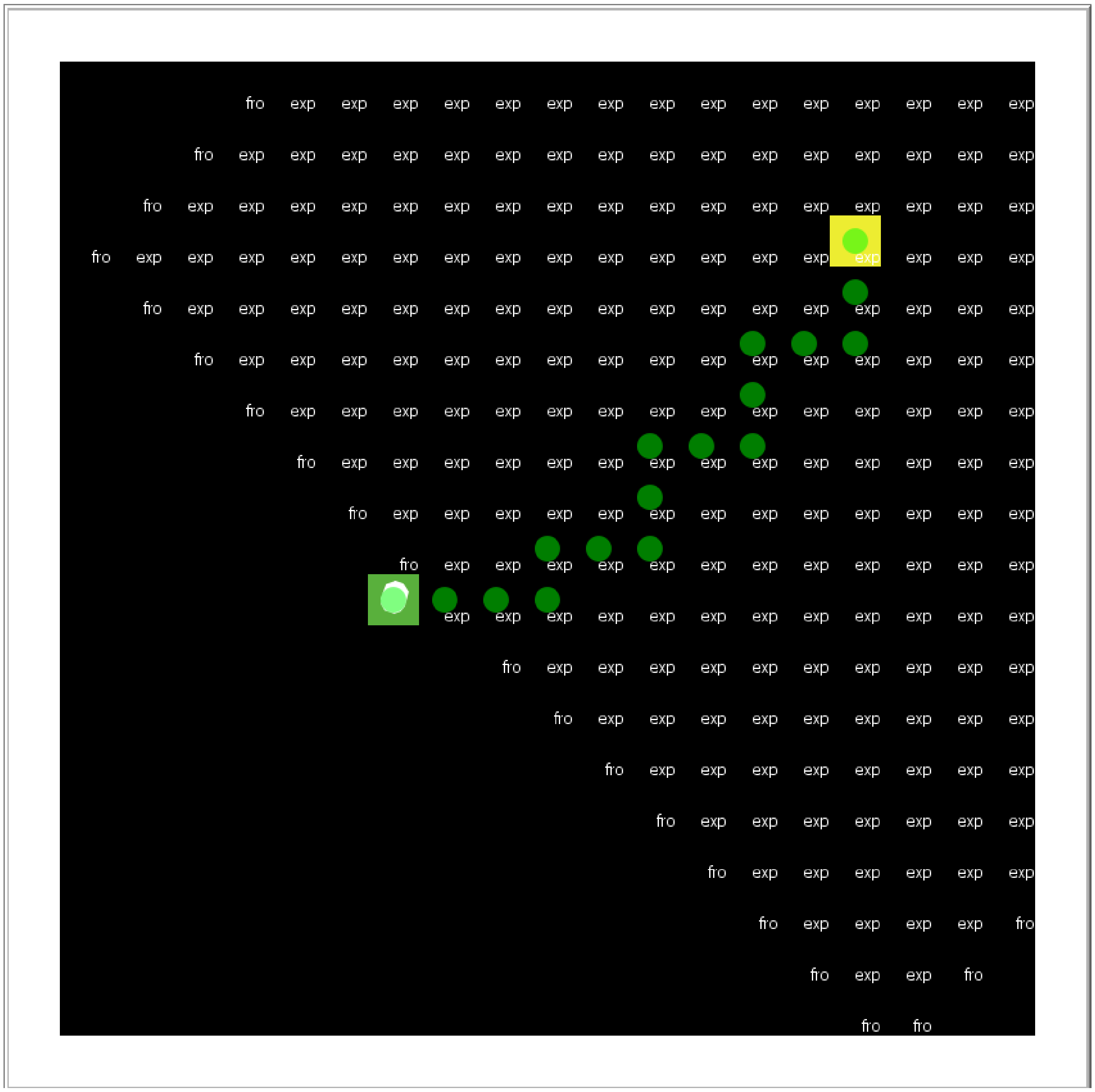
3.1. Level 1: One algorithm, one agent, one start, one goal

3.1.1. Map 1: No walls, no obstacles (except boundaries)

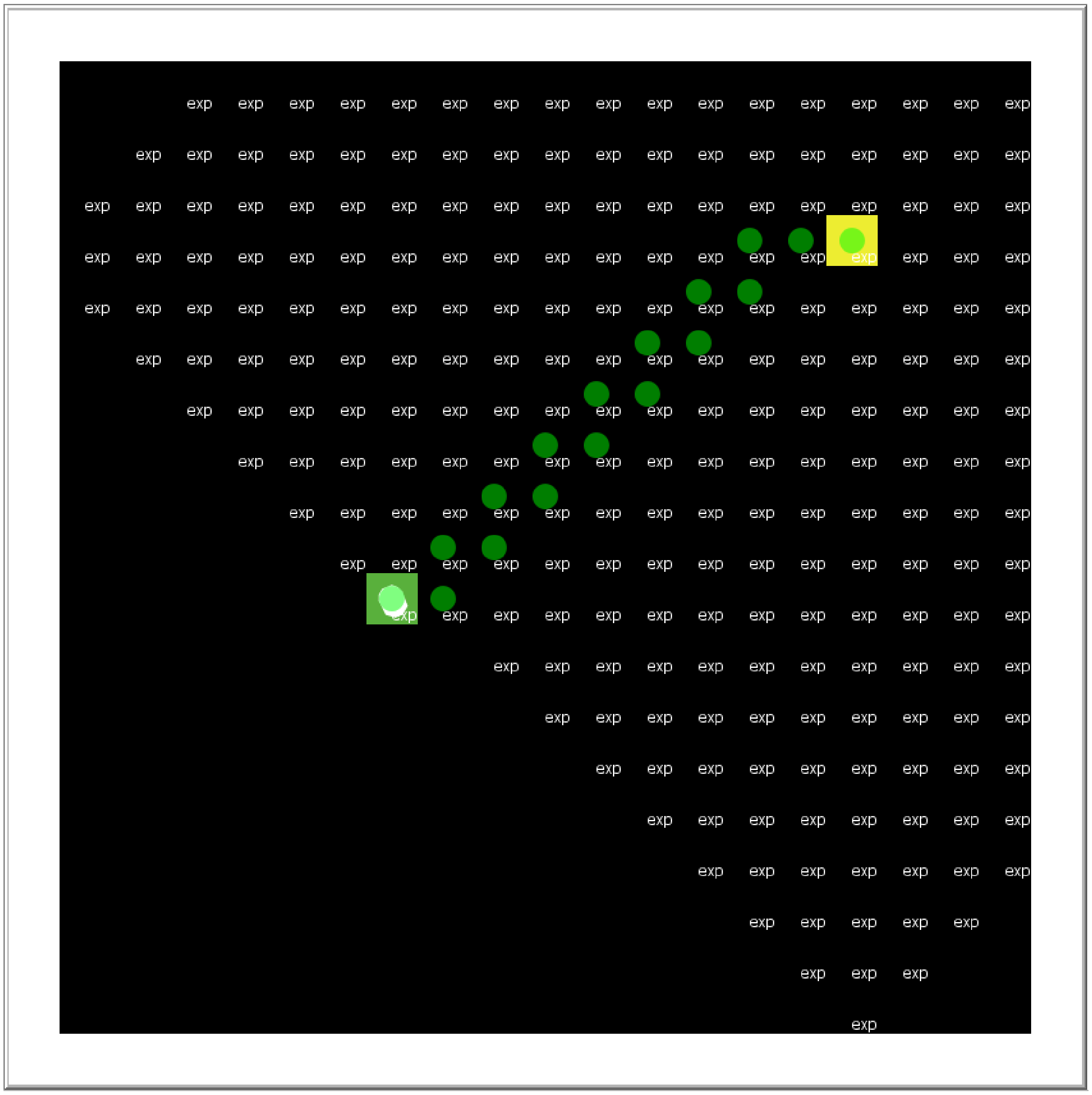




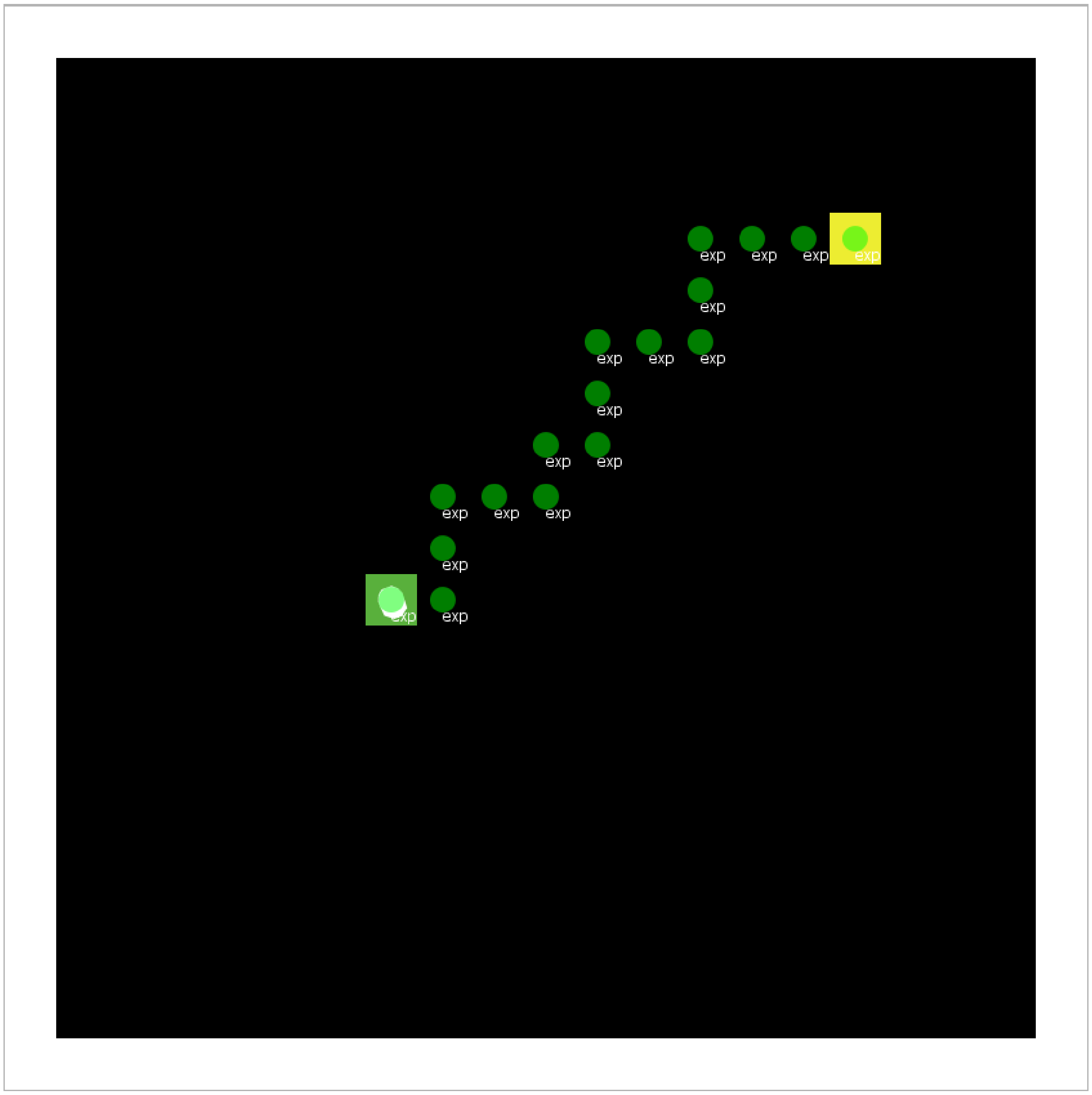
DFS: Path cost = 85, Time cost = 13.62 ms



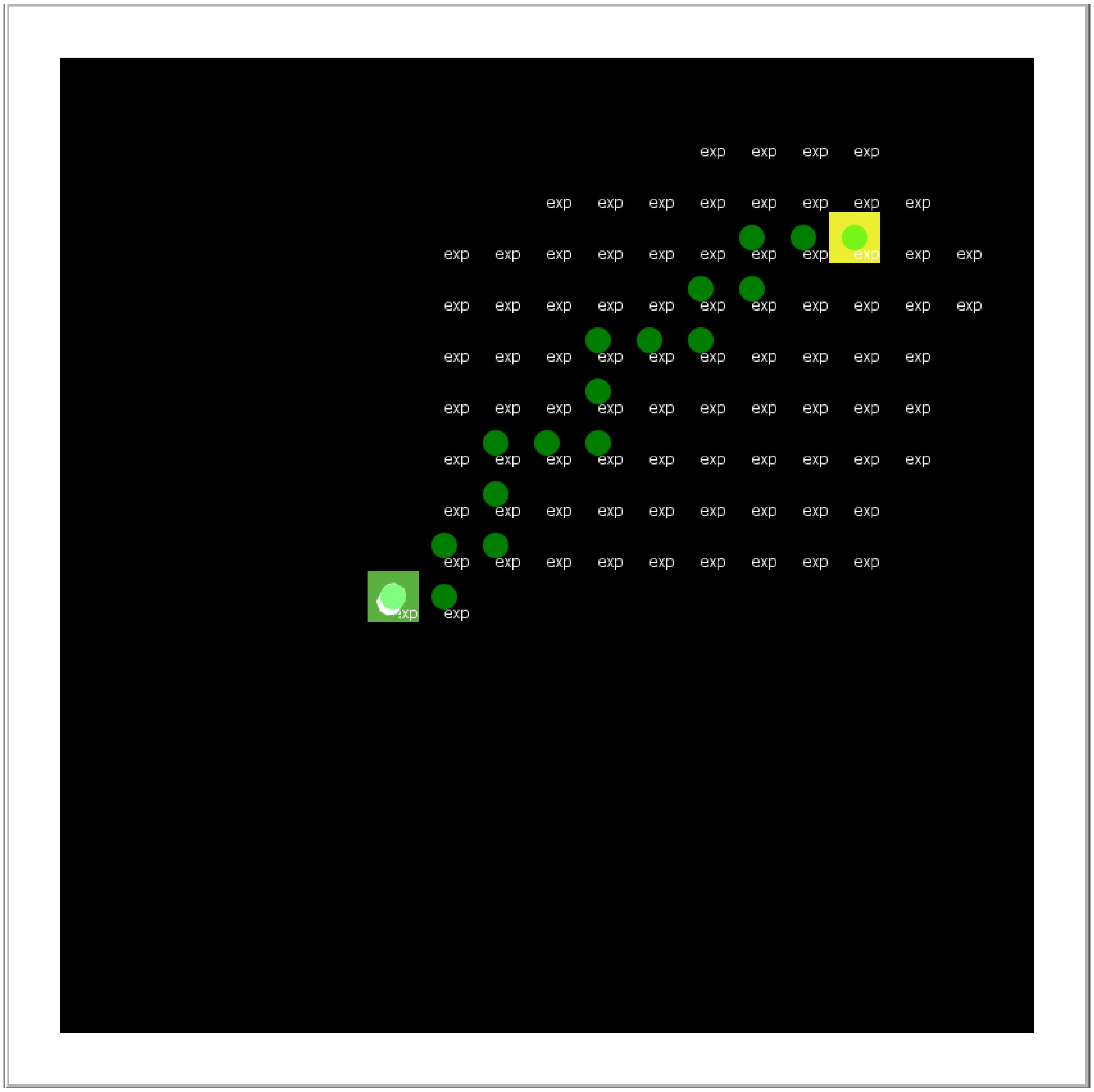
BFS: Path cost = 17, Time cost = 9.8 ms



UCS: Path cost = 17, Time cost = 11.28 ms



GBFS: Path cost = 17, Time cost = 0.75 ms



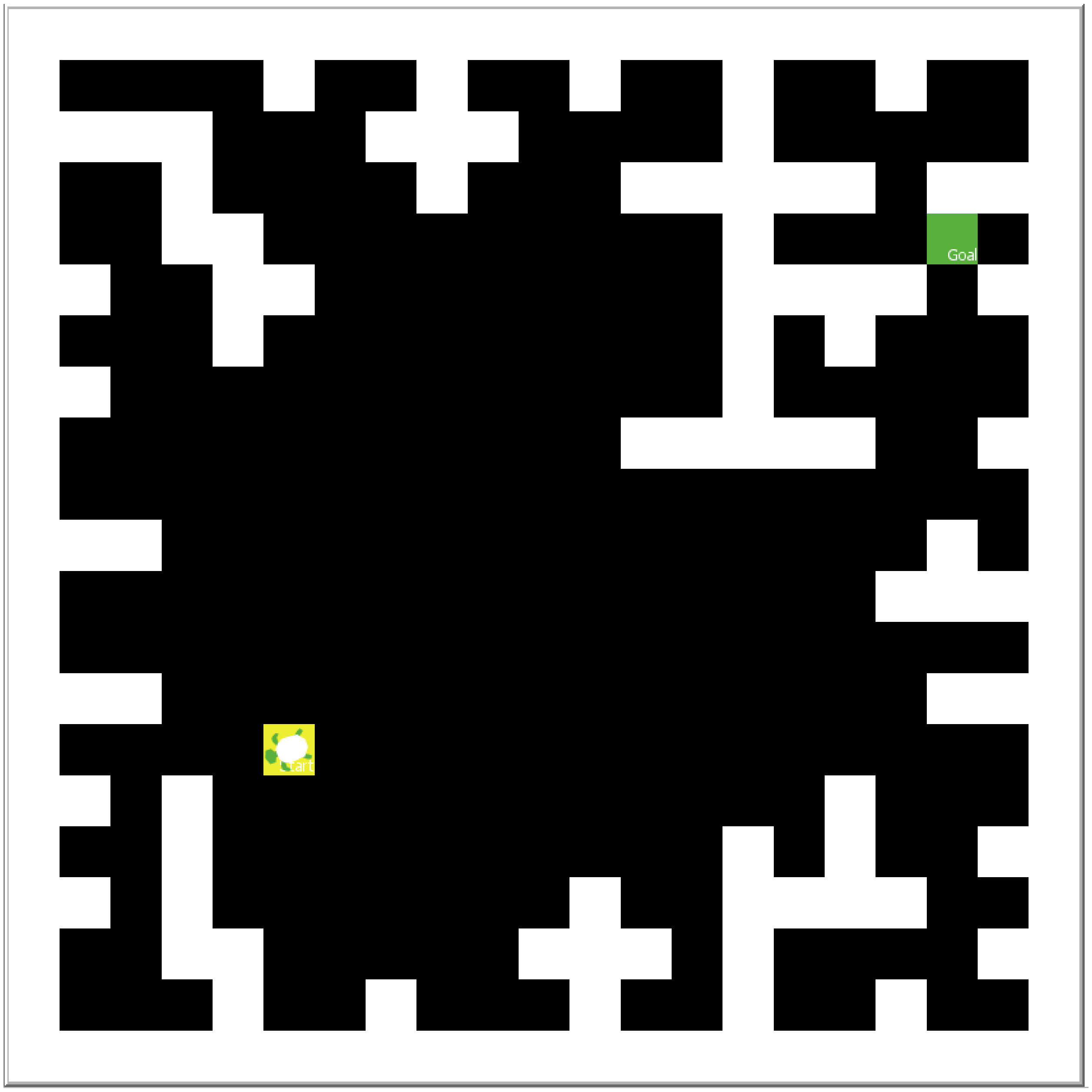
A: Path cost: 17, Time cost: 3.9ms*

This is the easiest map, with no blockades between the path from start to goal. From the above statistics, we can see that Greedy best-first search is easily the best choice out of all of them since the heuristic always decreases as the agent approaches the goal, the agent does not have to consider all adjacent nodes. There is no trap yet, so the path cost will be 1 no matter what, and GBFS achieves the best solution in the shortest time.

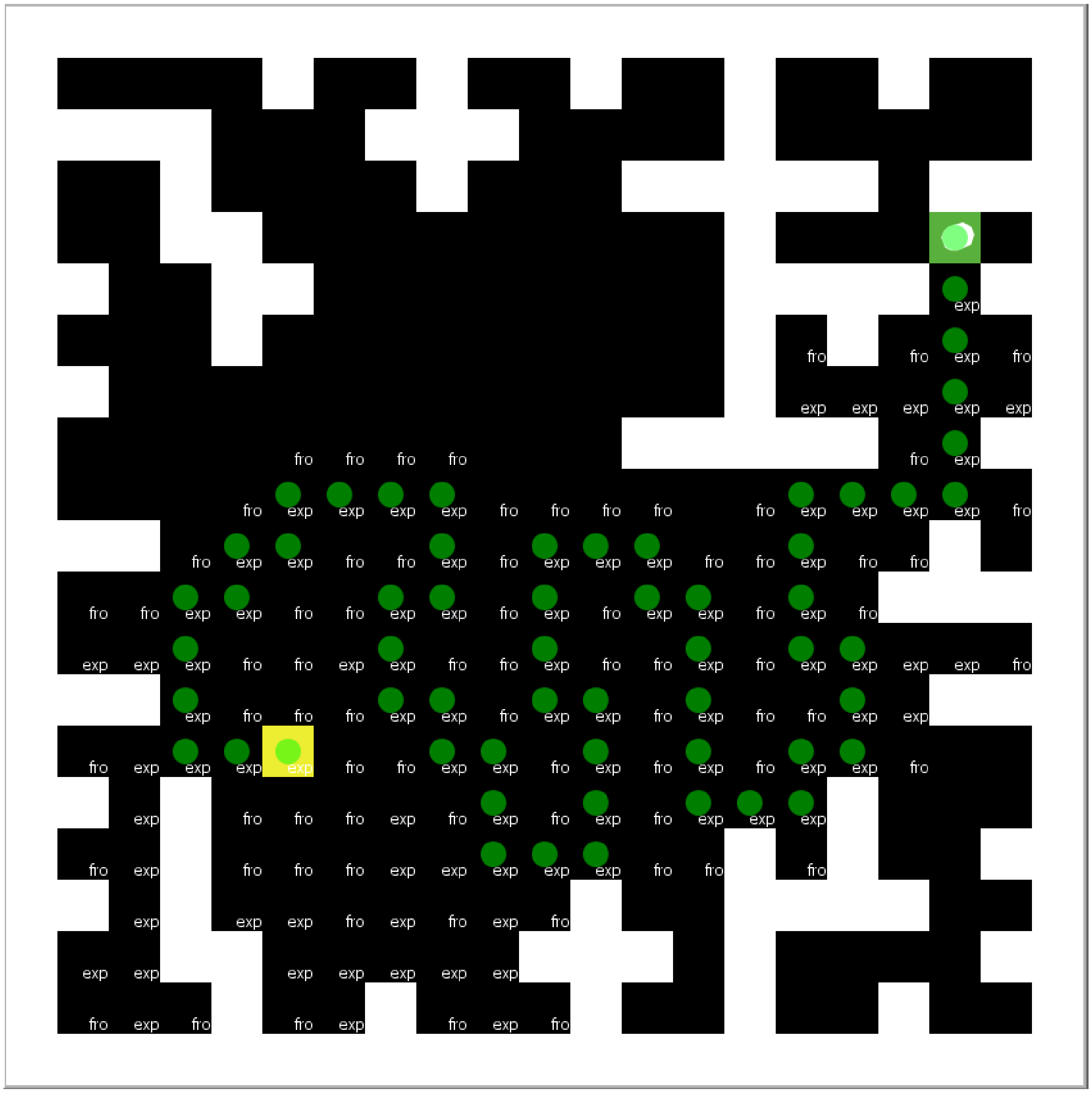
Both UCS, BFS, and A* achieve the minimum path cost which is 17, but the time is still a bit longer than GBFS due to their order of node expansion. UCS is essentially BFS in this case due to uniform path cost, and they have to consider a lot of nodes in the same depths before reaching the goal. A* also expands many surrounding nodes, but it is getting closer and closer to the goal due to the admissible heuristic.

The worst one is DFS because unfortunately in this map, the destination node is not too far from the starting point (the search tree is not too deep and the solution is rare) but the algorithm just keeps picking random paths to go around, leading to high path and time cost.

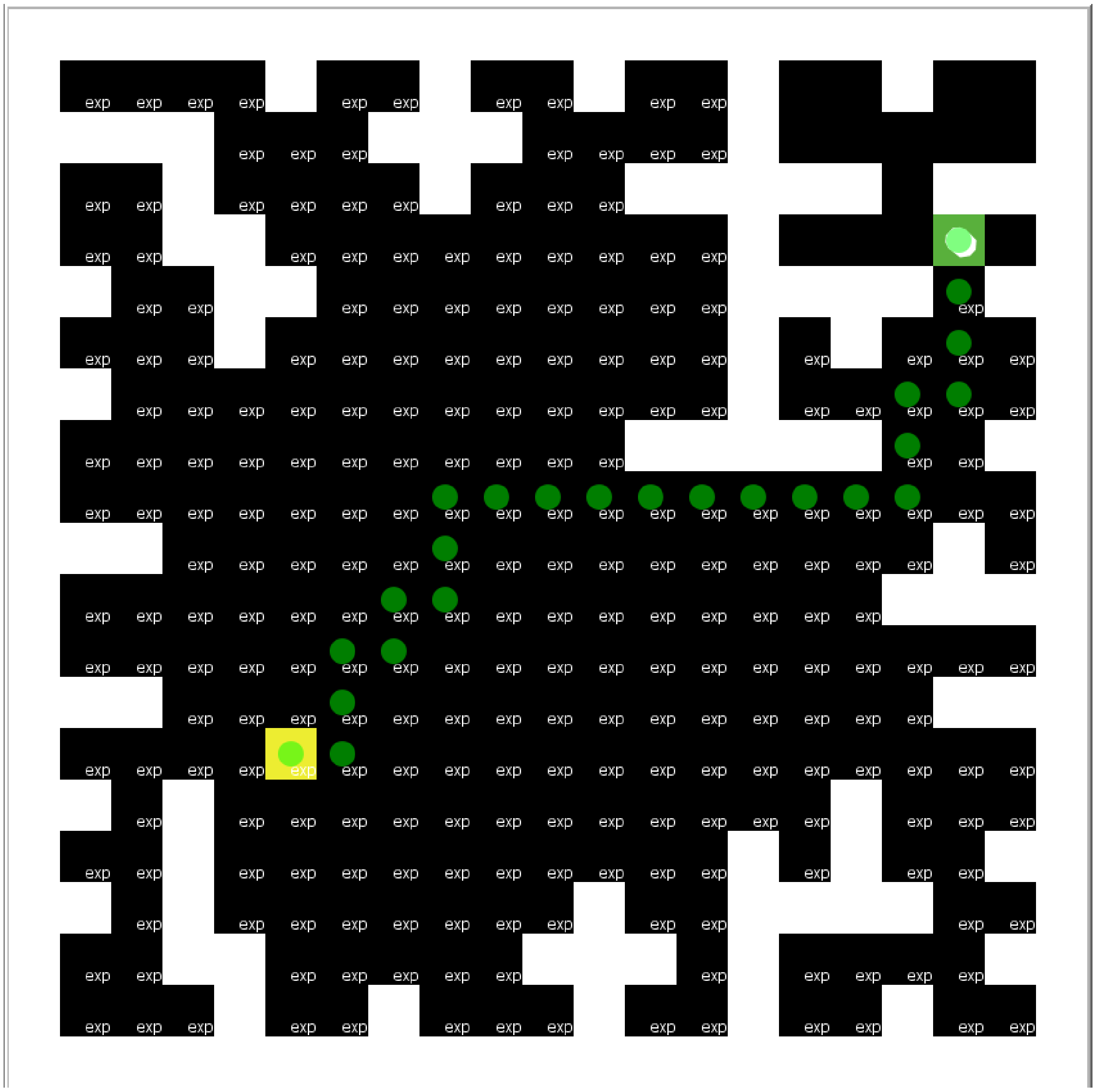
3.1.2. Map 2: Medium density of walls



This map is a half-maze, where walls come from all sides but do not extend fully enough to reach the center and form a complete maze. Agents are free to move in the middle of the map, but if the goal is closer to the wall, they will have a hard time finding it unless they try to look deep enough.

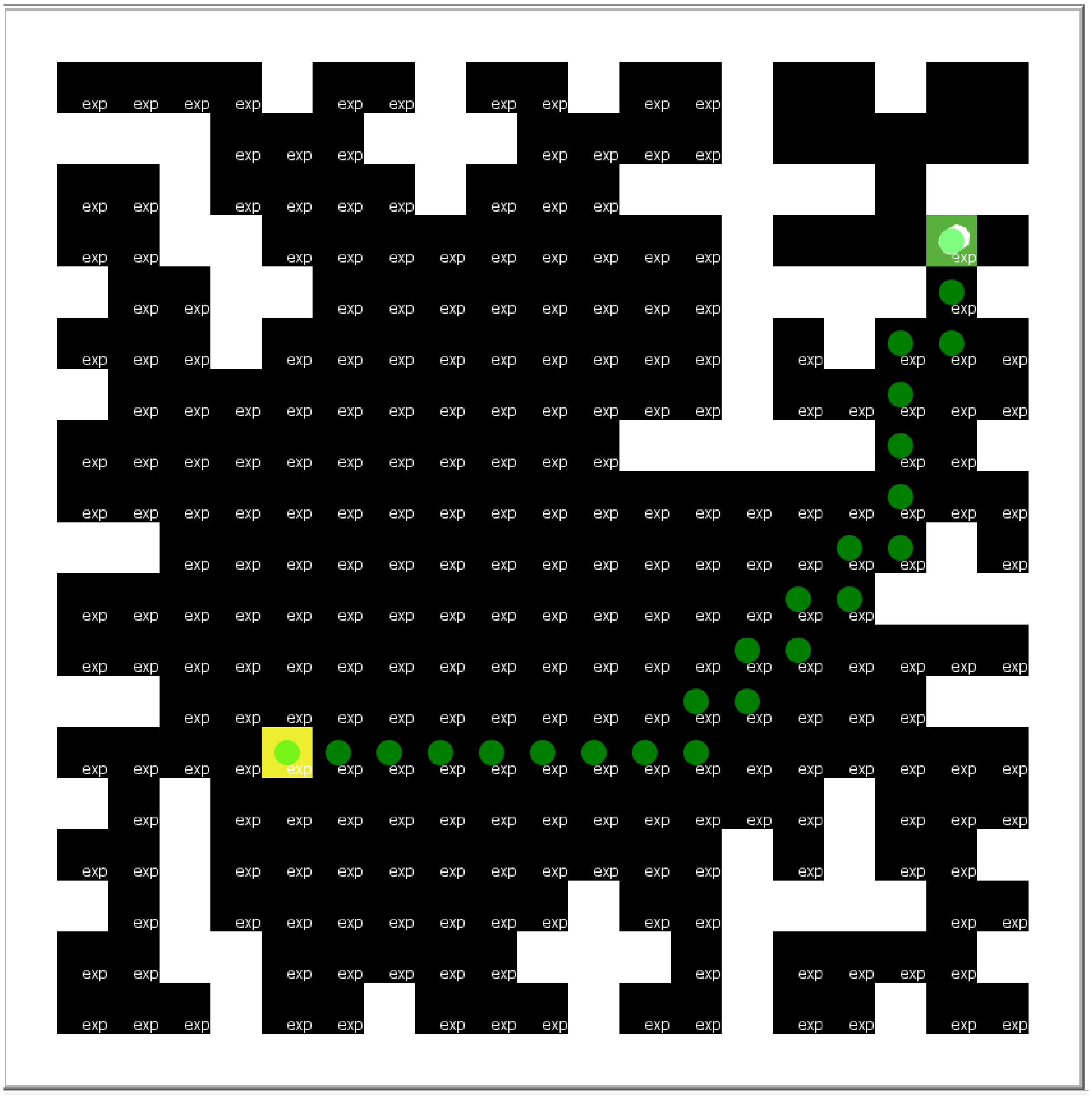

DFS: Path cost = 58, Time cost = 11.7ms

Once again, DFS loses track at the beginning because it is not informed of the direction of the goal. The search goes in a zigzag manner until it touches the goal, causing the path cost to rise significantly. However, this also leads to fewer expanding nodes and saves some time going deep instead of inspecting the middle part of the maze.

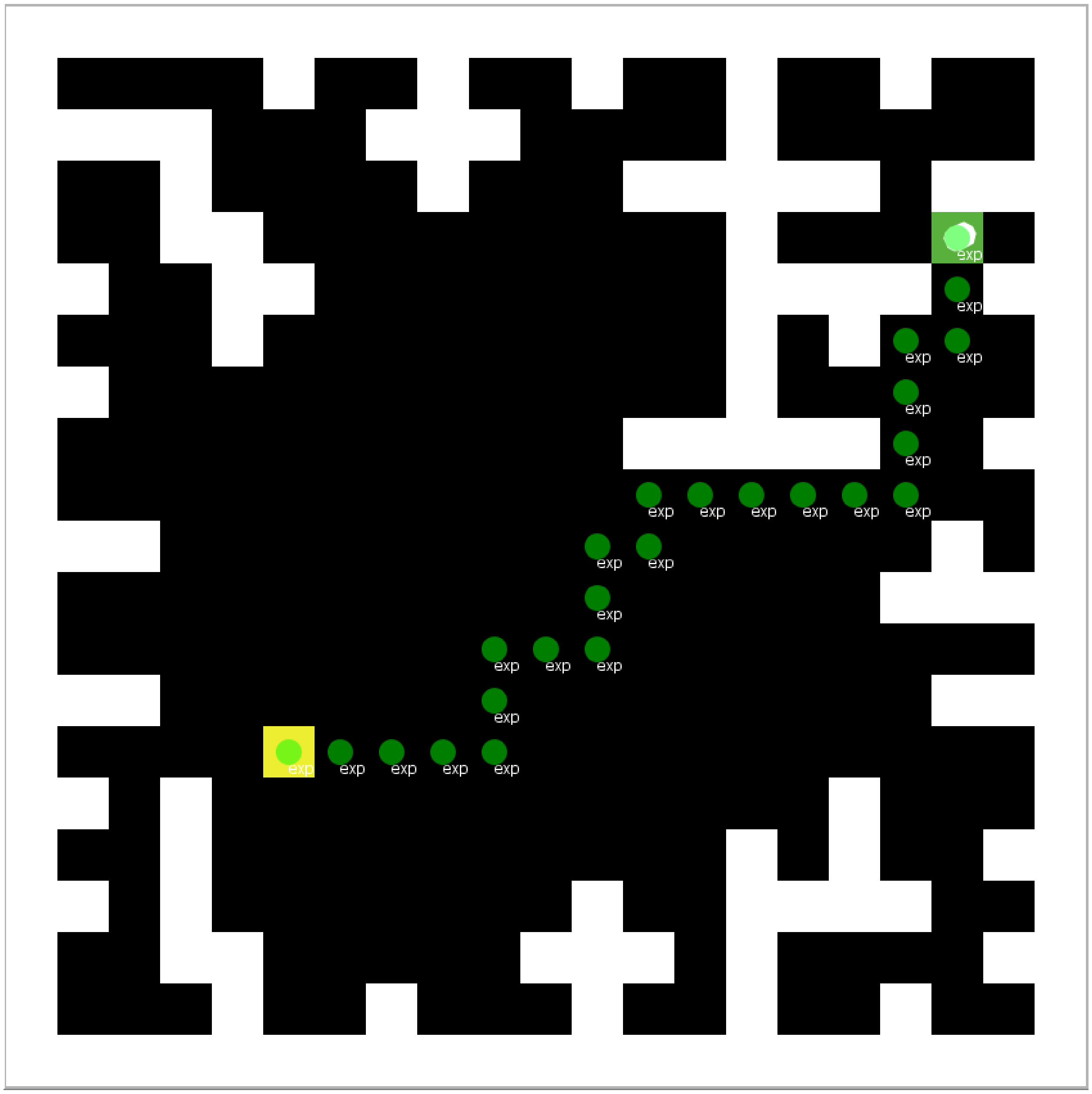


BFS: Path cost = 24, Time cost = 12.5ms

BFS outputs the minimum case since we have not considered different path costs. But clearly, from the look of this map, we see that BFS expands a lot of nodes (almost the whole map) just to find the goal, leading to a significant rise in algorithm time complexity.

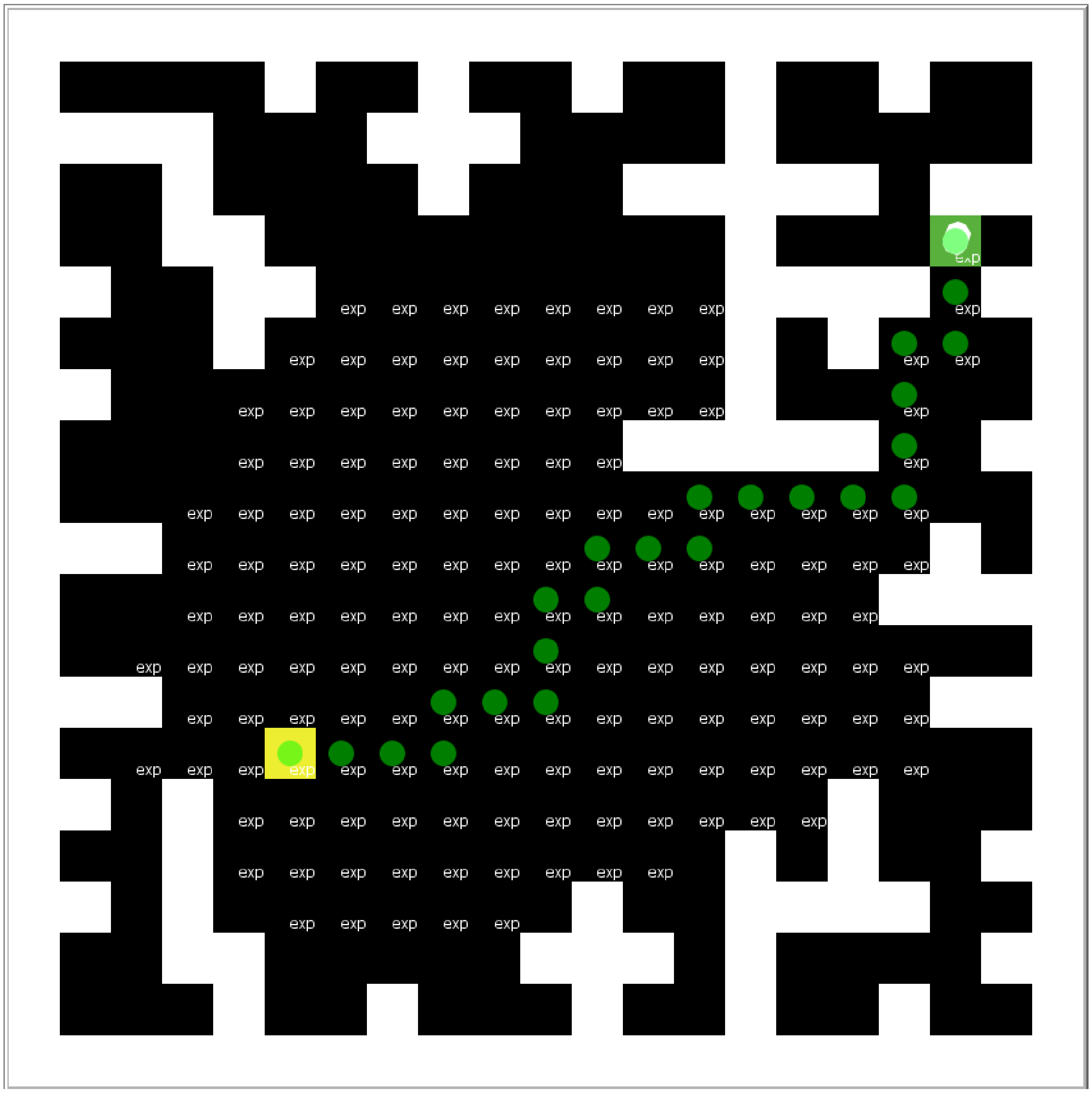

UCS: Path cost = 24, Time cost = 12.5ms

UCS is similar to BFS in this map, due to uniform path cost. They expand a lot of nodes but return the minimum path.



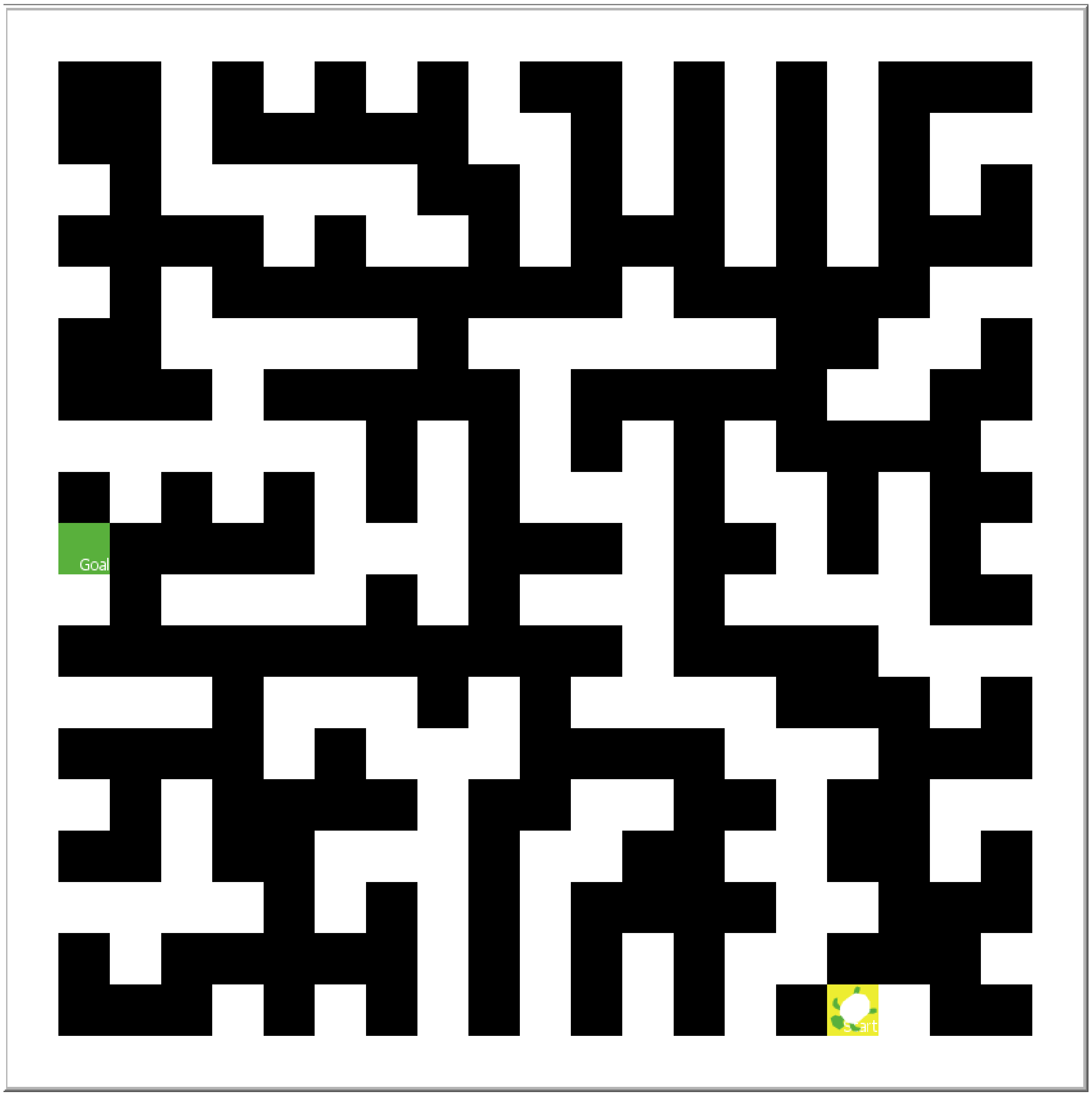
GBFS: Path cost = 24, Time cost = 1ms

GBFS is still the most effective solution because a greedy solution works best when the goal is not blocked by a lot of obstacles, causing the heuristic to closely predict the true path cost. In this case, GBFS outputs the minimum path in a very efficient timespan.

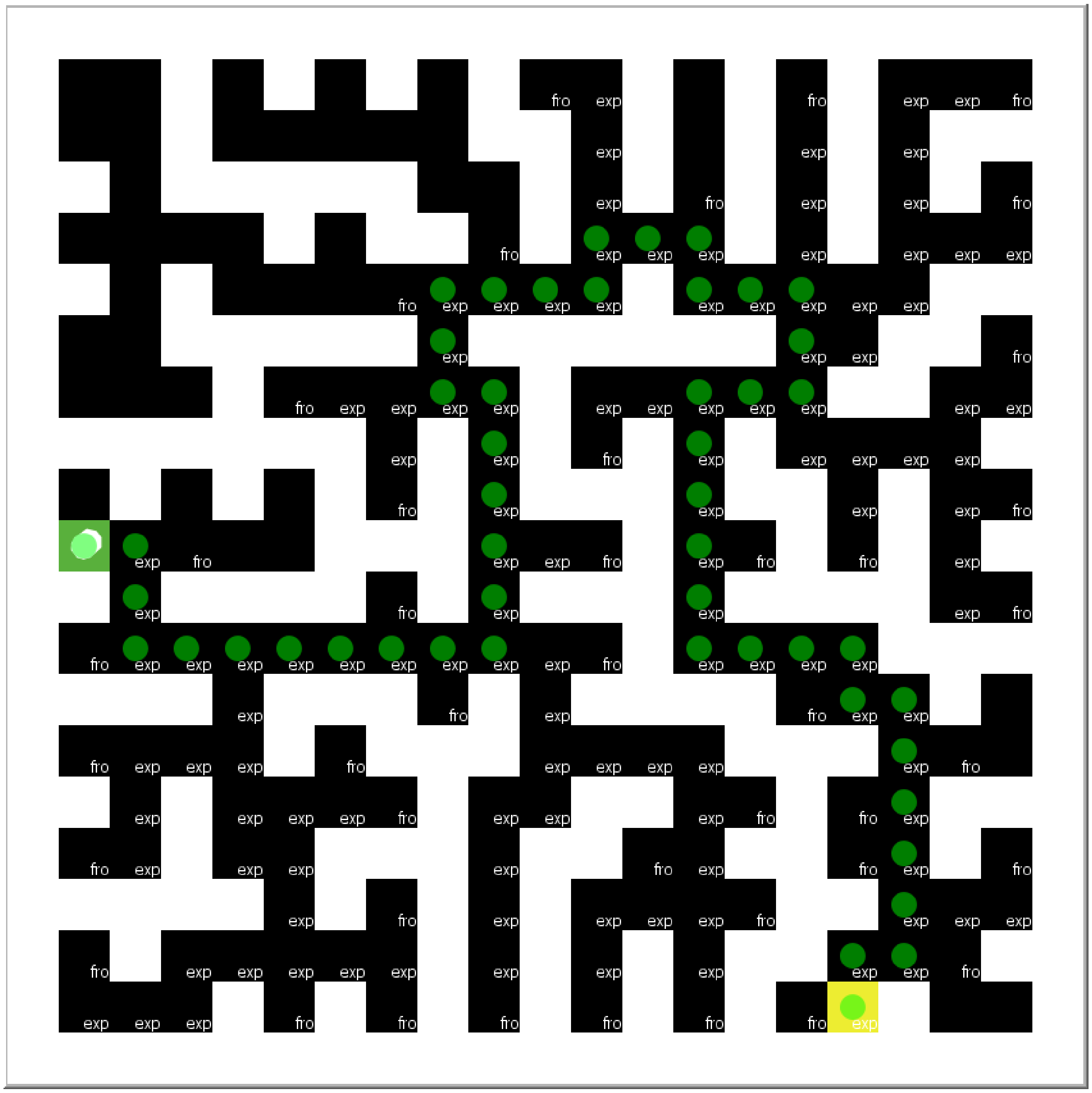

A: Path cost = 24, Time cost = 7.4ms*

Astar functions similarly to both UCS and GBFS in this case, but it also takes a bit more time due to a lot of nodes being expanded. Besides heuristic, Astar also has to consider the cost function and so it tries to deal with closer nodes also. However, we will see that Astar is both cost-effective and less time-consuming than other methods in later maps.

3.1.3. Map 3: Complete maze

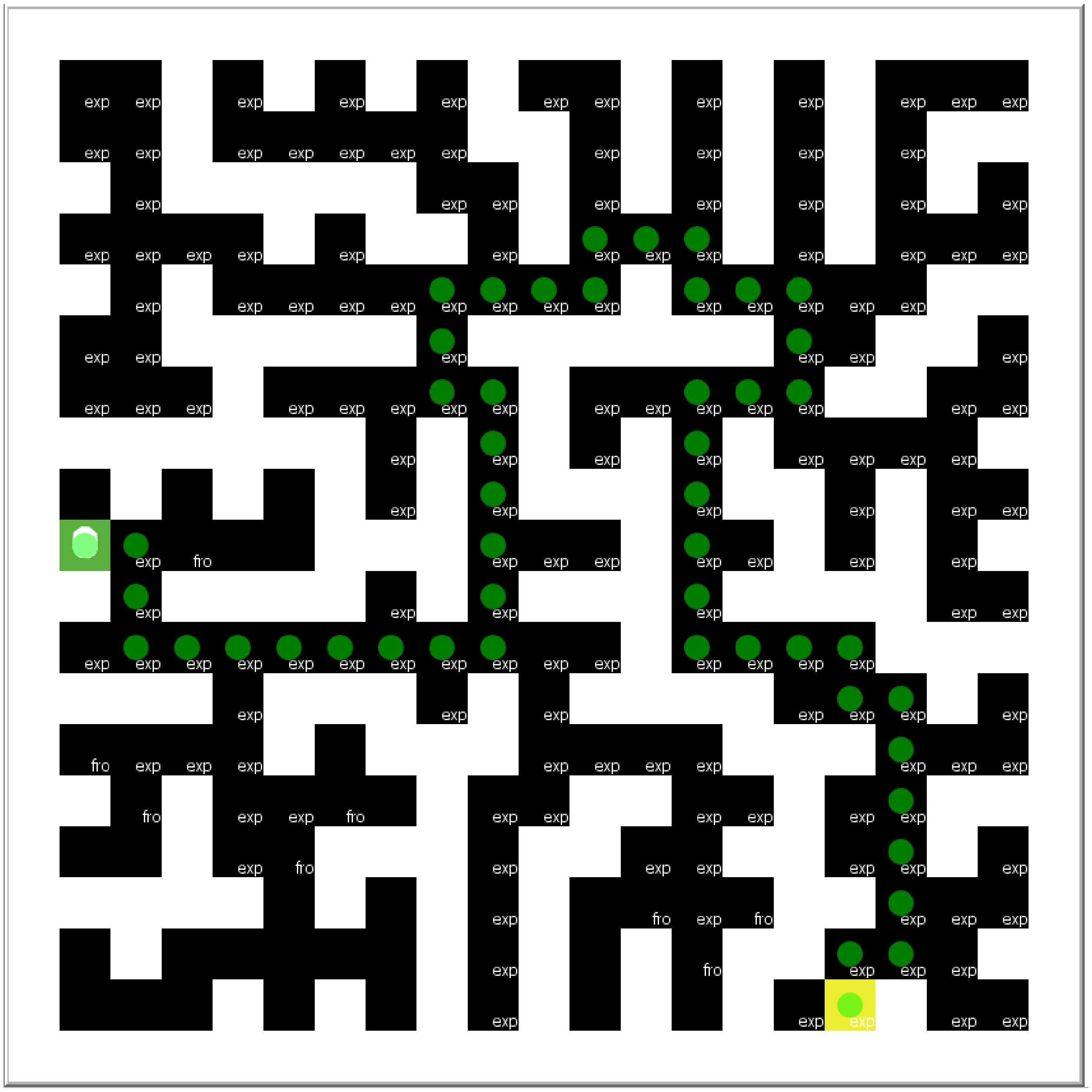


This map is a little special because every search algorithm will output one same path (except DFS). The path from the starting point to the goal is almost linear (there are small branches, but the algorithm will mostly take the path further to the left and further down.)



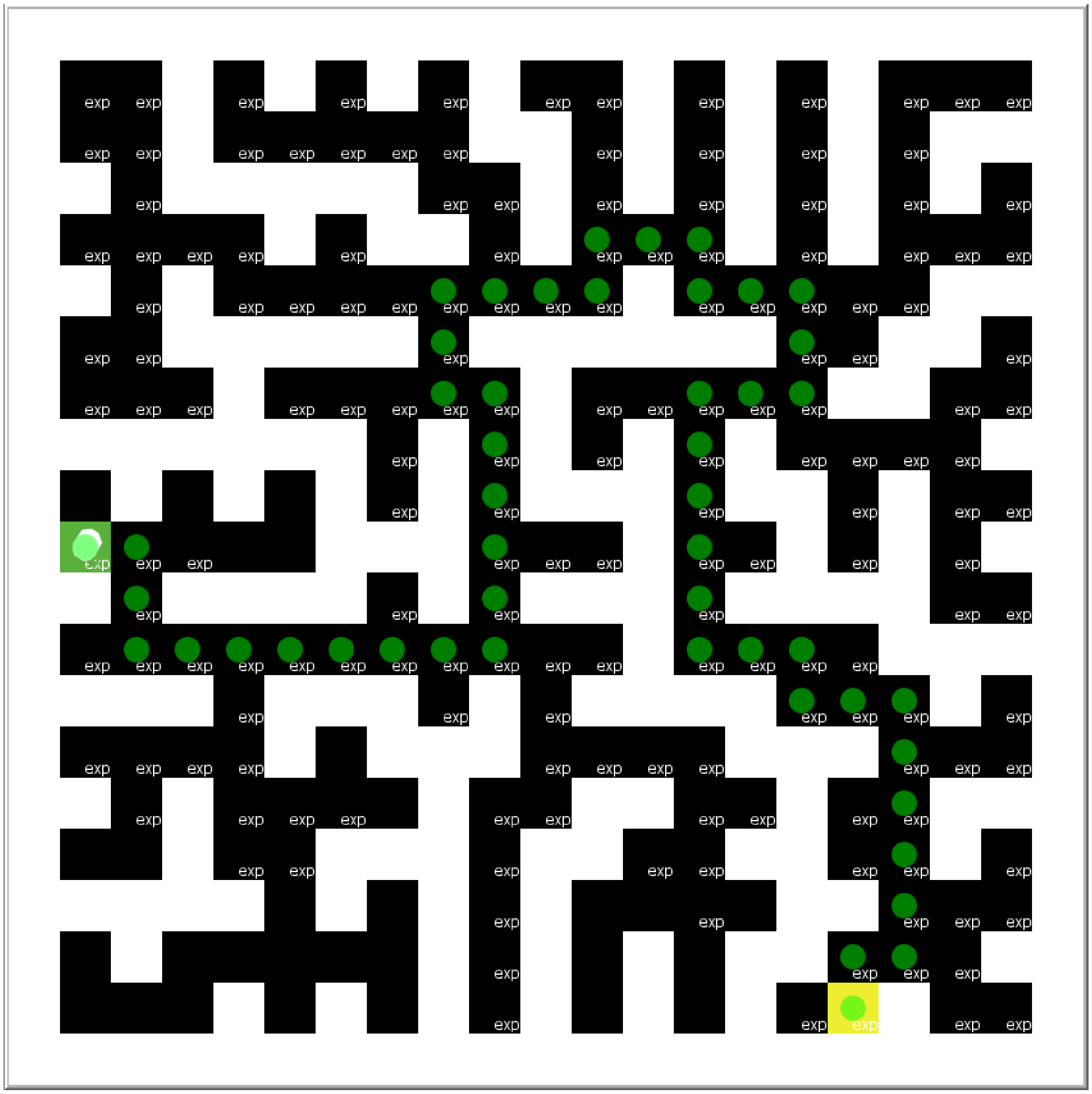
DFS: Path cost = 49, Time cost = 13.1ms

DFS is not so lucky because it still branches out a lot, albeit unnecessary ones. However, DFS returns the minimum path solution and that is commemorable. The more branches there are in the maze, the higher the risk DFS is going to go off the track.



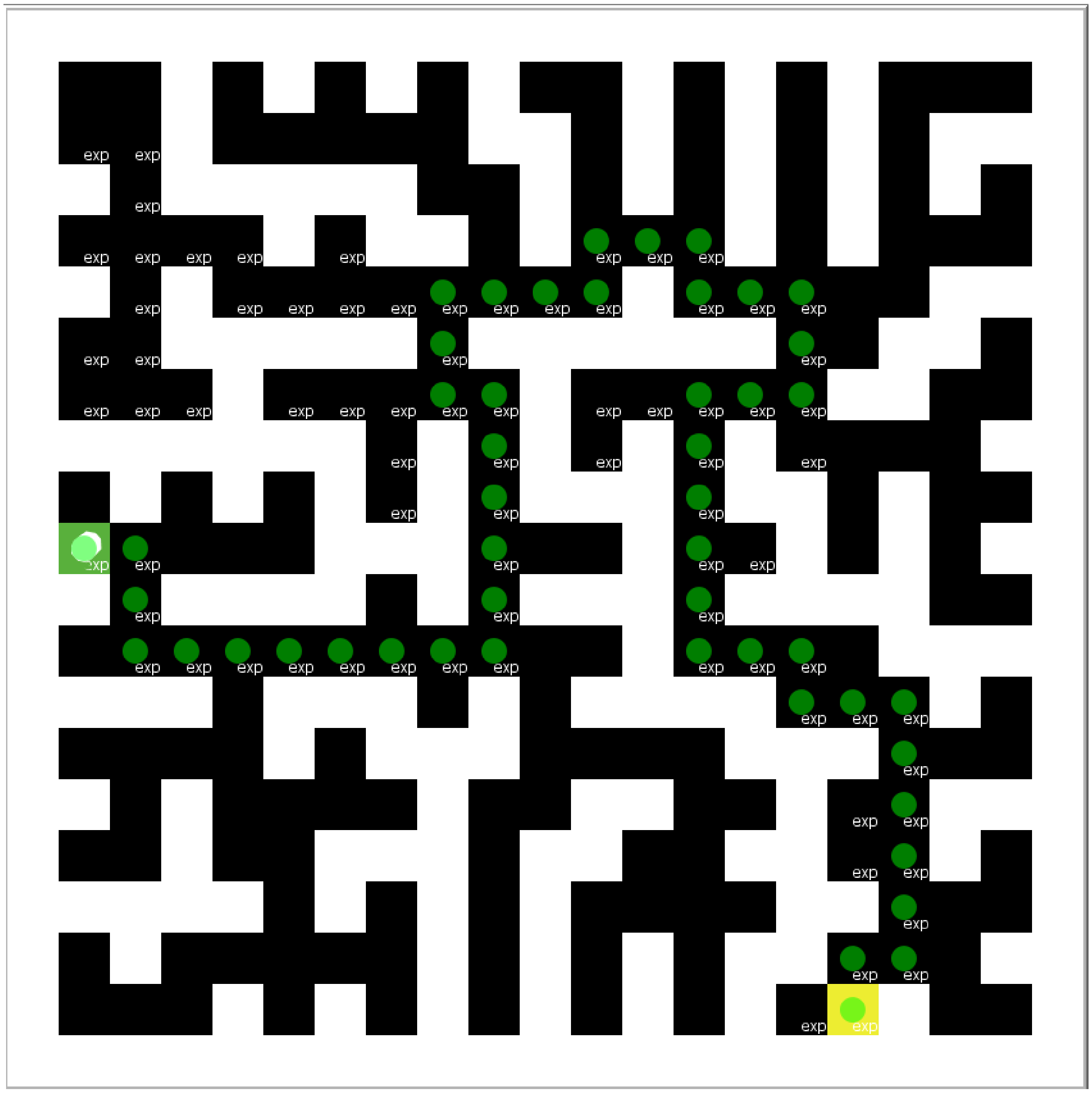
BFS: Path cost = 49, Time cost = 7.8ms

BFS outputs the minimum path but still suffers from expanding too many nodes, causing the algorithm to run a bit slowly. If the map was bigger, BFS's time complexity and space complexity will rise exponentially.



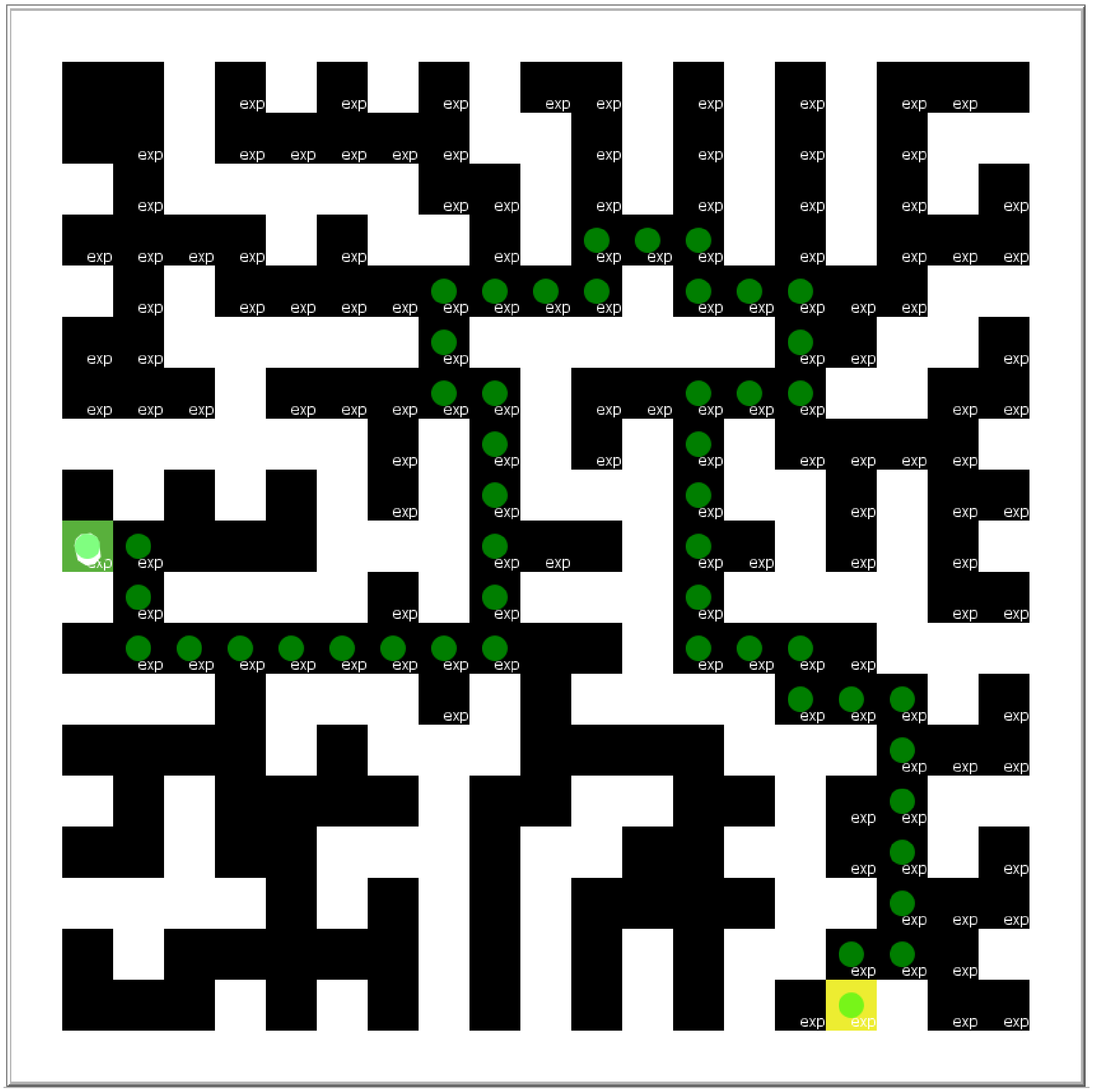
UCS: Path cost = 49, Time cost = 8ms

Again, under uniform cost path conditions, UCS is essentially the same as BFS.



GBFS: Path cost = 49, Time cost = 3.7ms

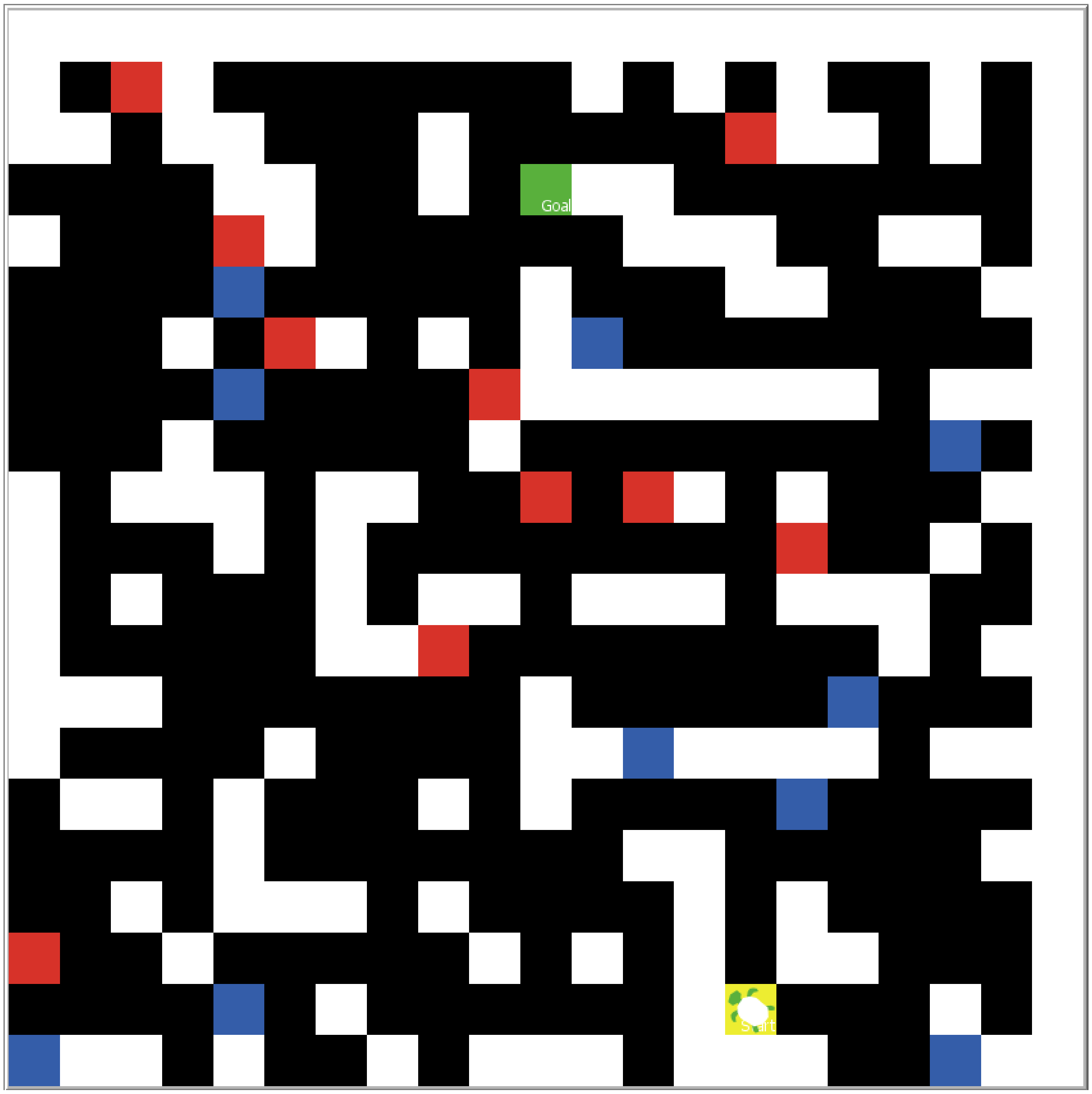
GBFS manages to return the minimum path solution also this time, and it does take a wrong turn when approaching the goal. The heuristic estimated it to be close to the goal, but there are walls blocking it, so it has to find other ways to get as close to the goal as possible. The time it takes to execute GBFS is still quite minimal and that is acceptable even if the algorithm took the wrong turn.



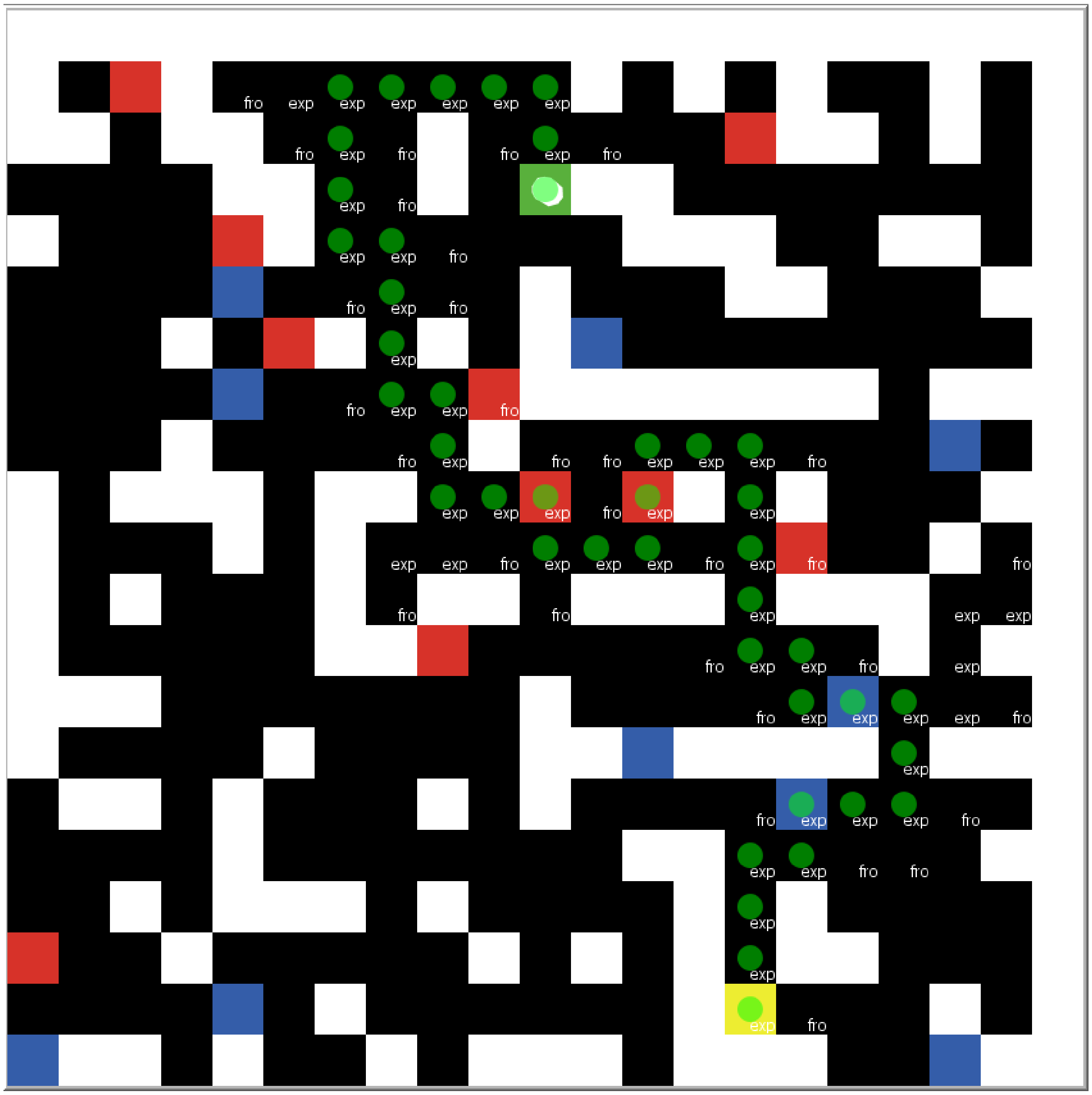
A: Path cost = 49, Time cost = 6.4ms*

Astar is basically GBFS in this map, but a lot of closer nodes are expanded to keep the f-cost as low as possible. Midway through the Astar algorithm also has to backtrack because some of the nodes it visited could have a lower cost function (g-value), causing it to explore a lot of nodes.

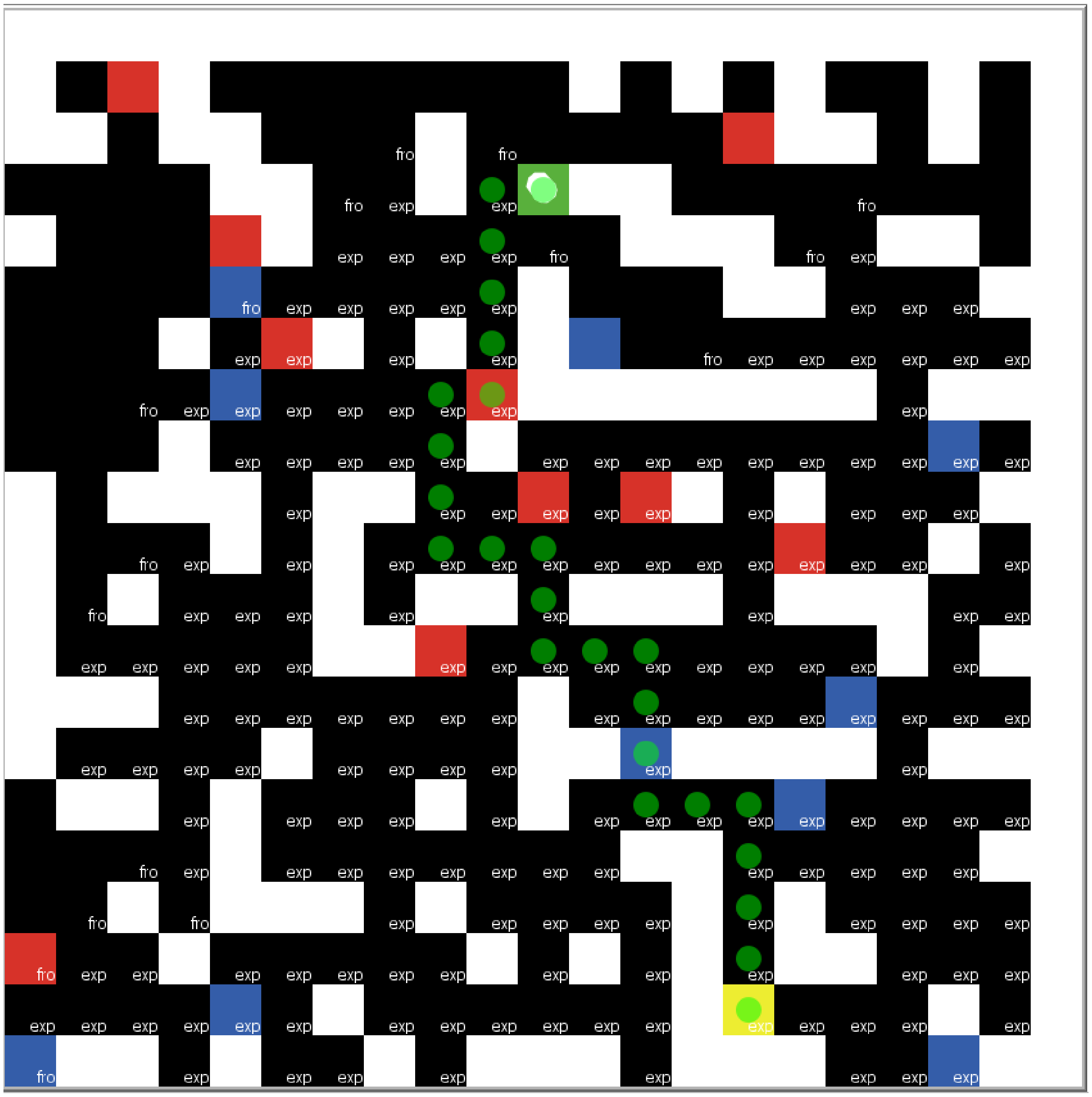
3.1.4. Map 4: Maze with few obstacles



In this map, we cut the maze a bit to leave space to add in several obstacles, including water (path cost = 5) and lava (path cost = 10). The agents have to consider the path cost much more seriously now, and the advantages of some algorithms will be easier seen here.

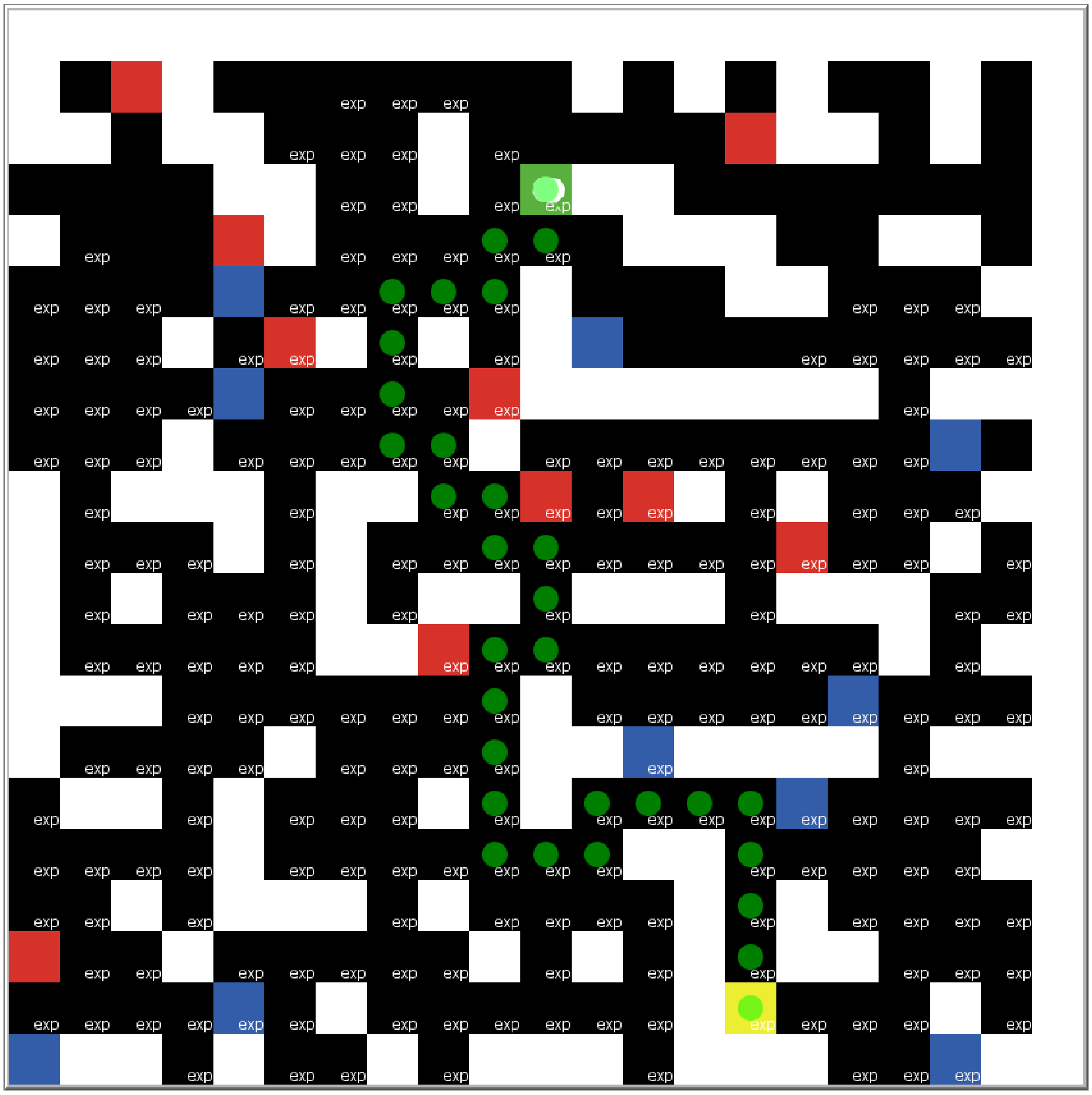

DFS: Path cost = 69, Time cost = 6ms

DFS is lucky in this case, it manages to choose the path that approaches the goal and reaches it in a very short time, with very few expanding nodes. However, the cost is far from the most efficient one.

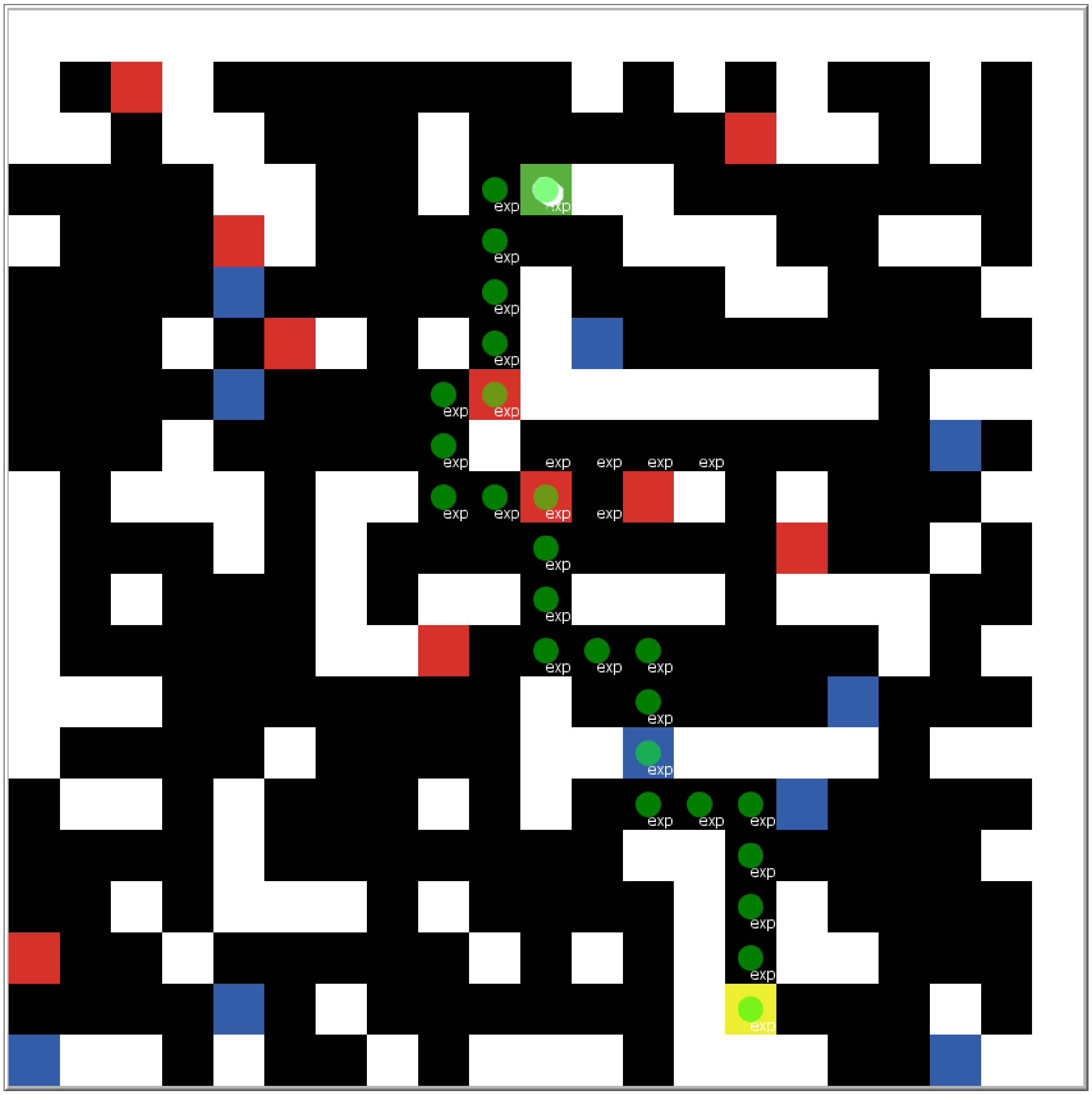


BFS: Path cost = 38, Time cost = 9.1ms

Unfortunately in this map, BFS fails to output the most efficient path. Each red node costs 10, and BFS only cares about the path that takes the least expanding nodes, so it suffers from an inefficient cost. Nevertheless, the time BFS takes is still too high because it worries too much about expanding every adjacent node.

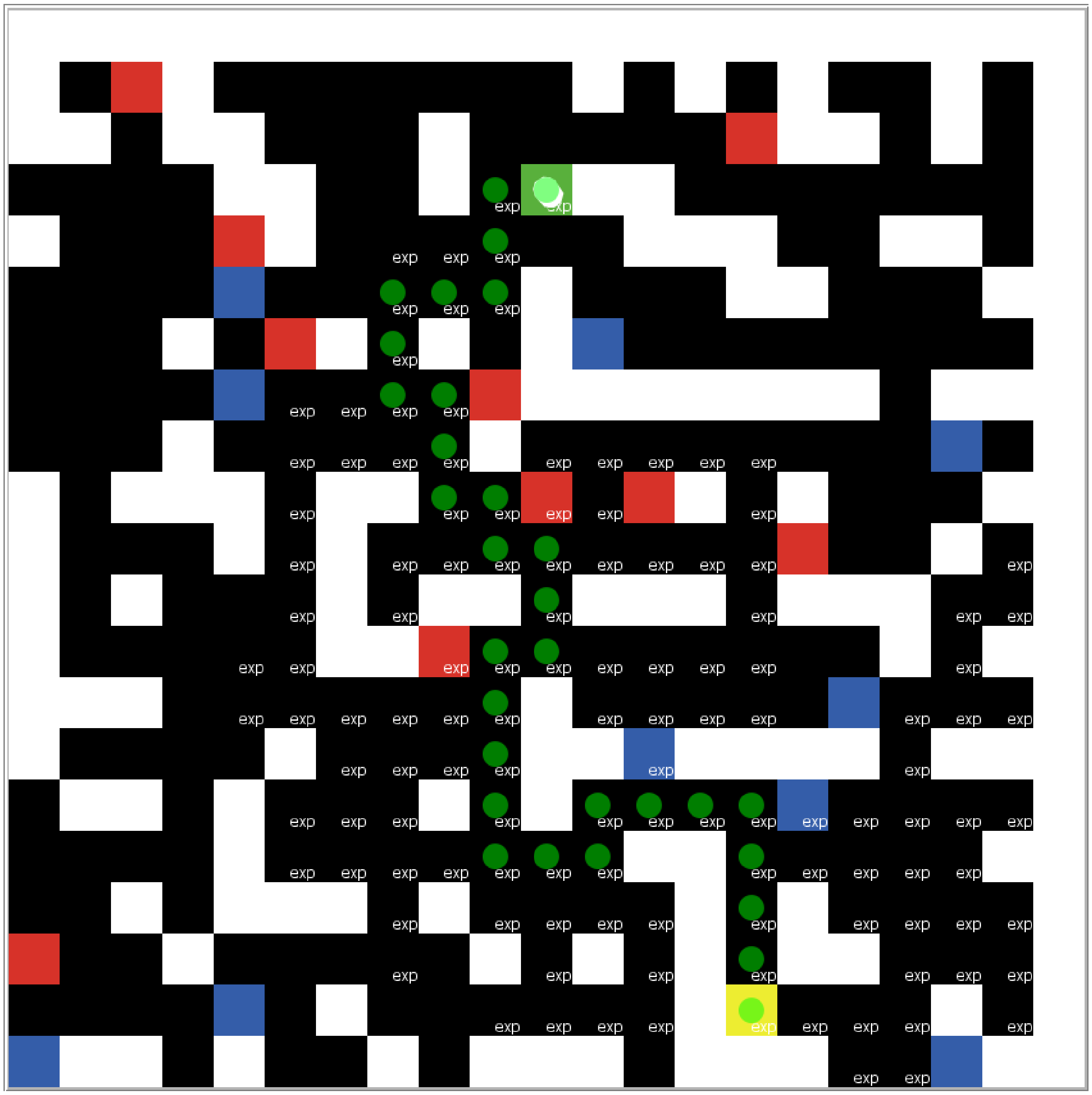

UCS: Path cost = 31, Time cost = 10.6ms

UCS performs significantly better than BFS in this case, with the path cost being the most optimized path it can be to reach the goal. It takes a lot more time than BFS due to expanding too many nodes (one red node costs 10 step and one blue node cost 5, so UCS will not touch them until the current minimum cost exceeds that).



GBFS: Path cost = 47, Time cost = 1.3ms

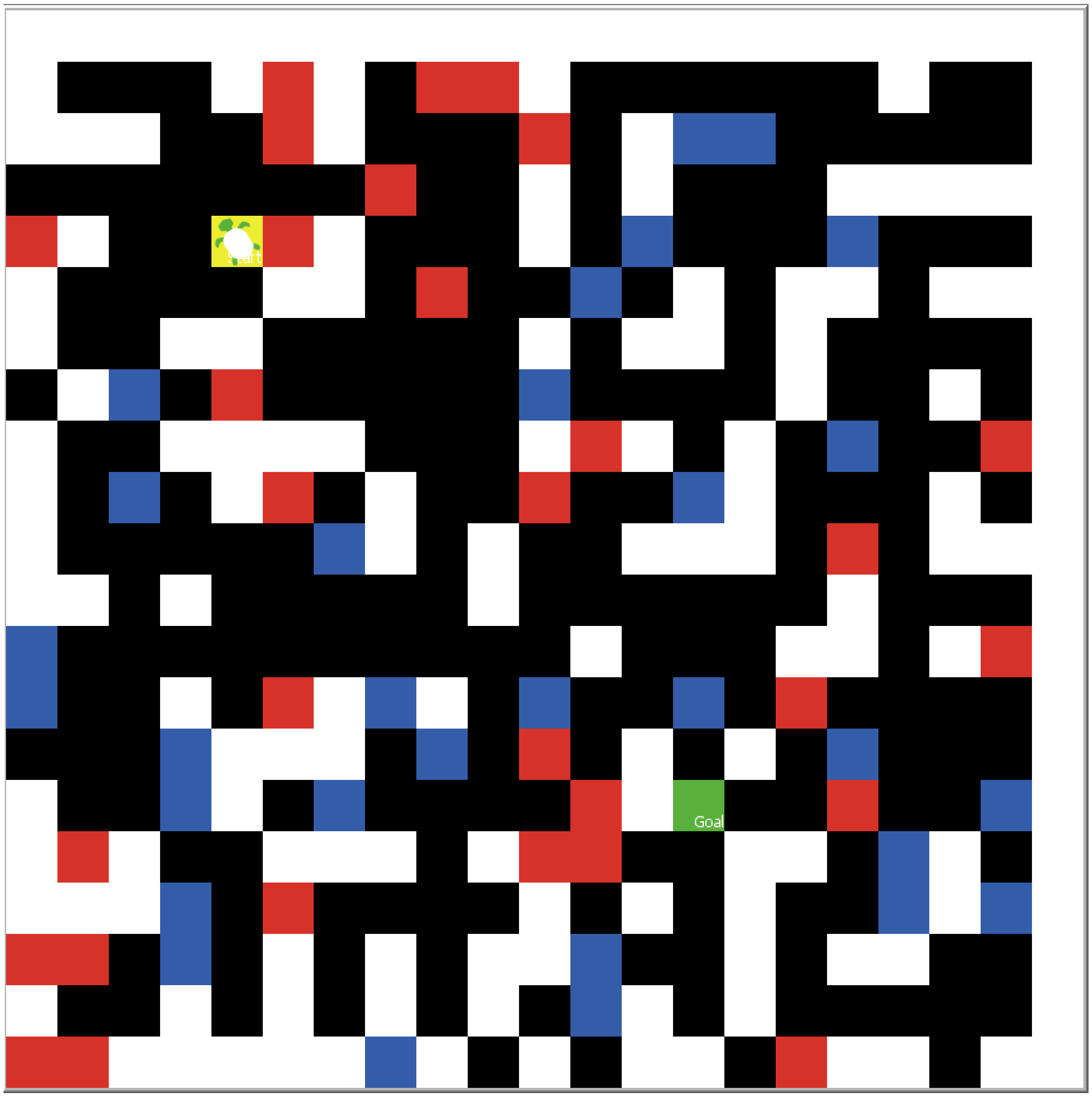
GBFS is still the most time-efficient algorithm, but now it fails to prove useful. The path it finds is far from optimized and hence cannot be used in reality circumstances. GBFS only worries about how close it is to the goal, regardless of whether it steps on a costly node or not.



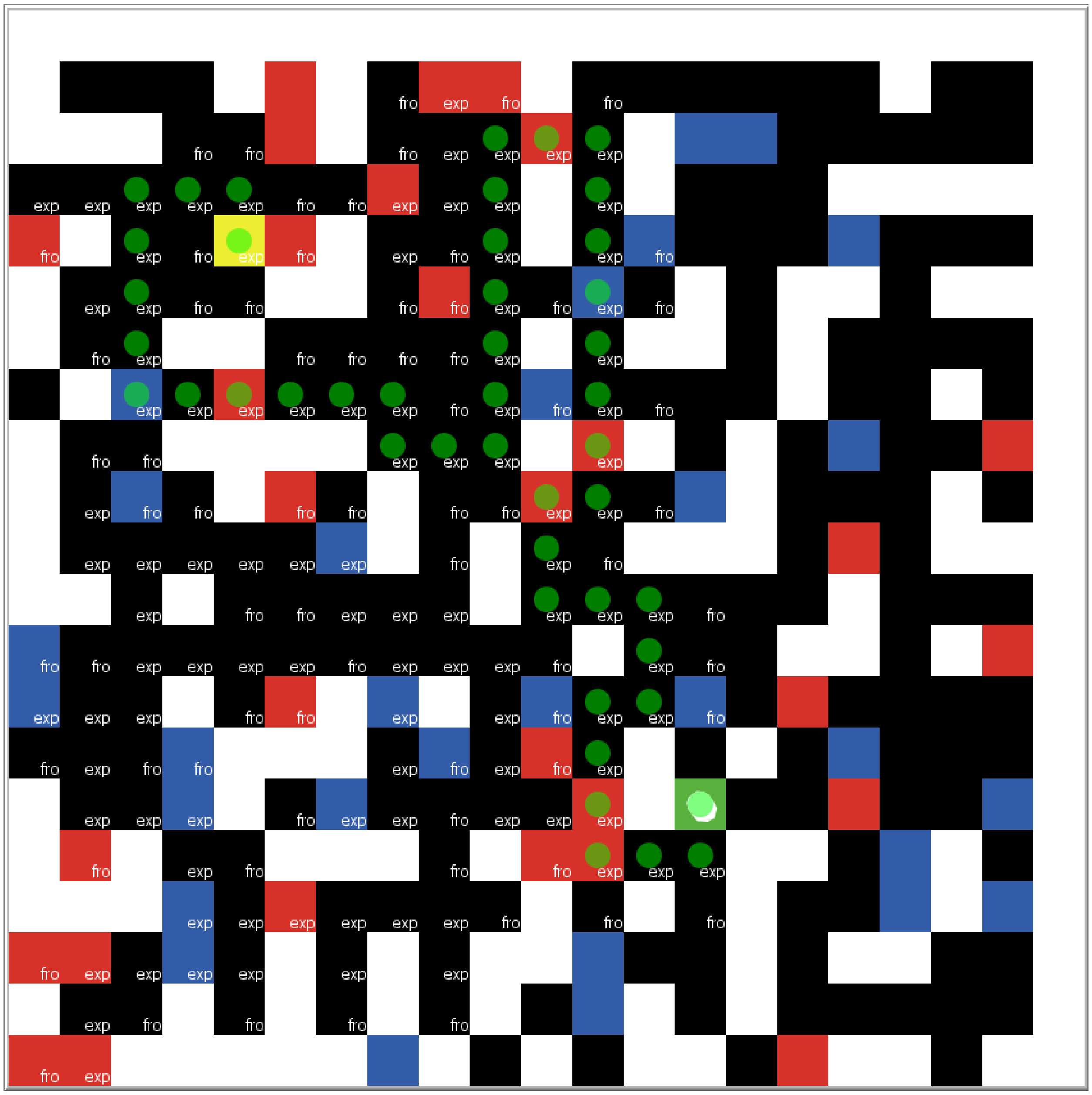
A: Path cost = 31, Time cost = 5.9ms*

The optimized path cost we found earlier with UCS is 31, and this time Astar outputs the same value, but with a much more optimized time! The algorithm considers both the distance from the current node to the goal (the heuristic) and the cost to reach the current node. It will not expand nodes too far from the current path because the cost function will keep it back.

3.1.5. Map 5: Maze with many obstacles

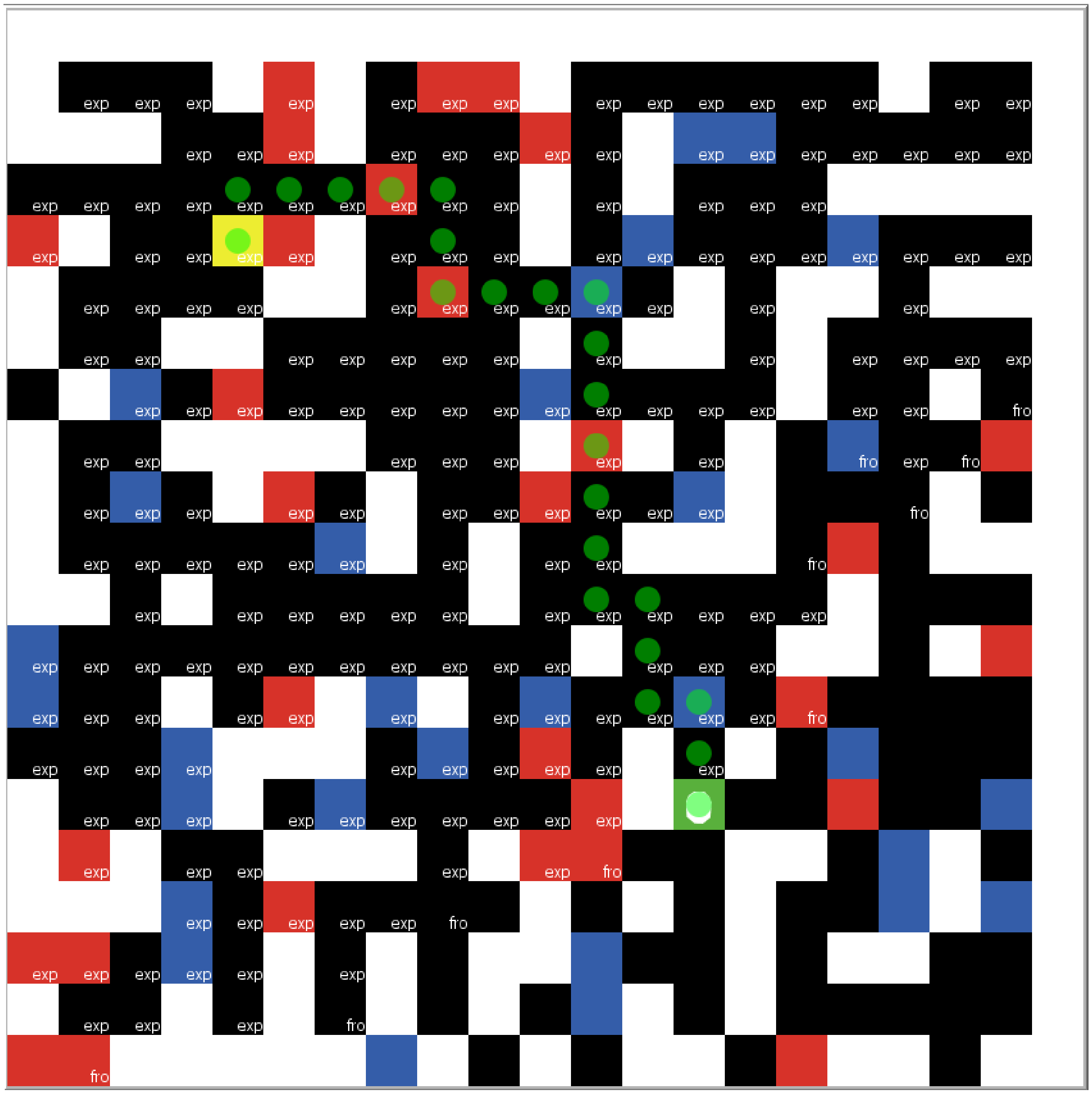


We move on to the last kind of map, where most of the maze is just obstacles, and the walls are now placed all over the place, causing the agents to make decisions consecutively.

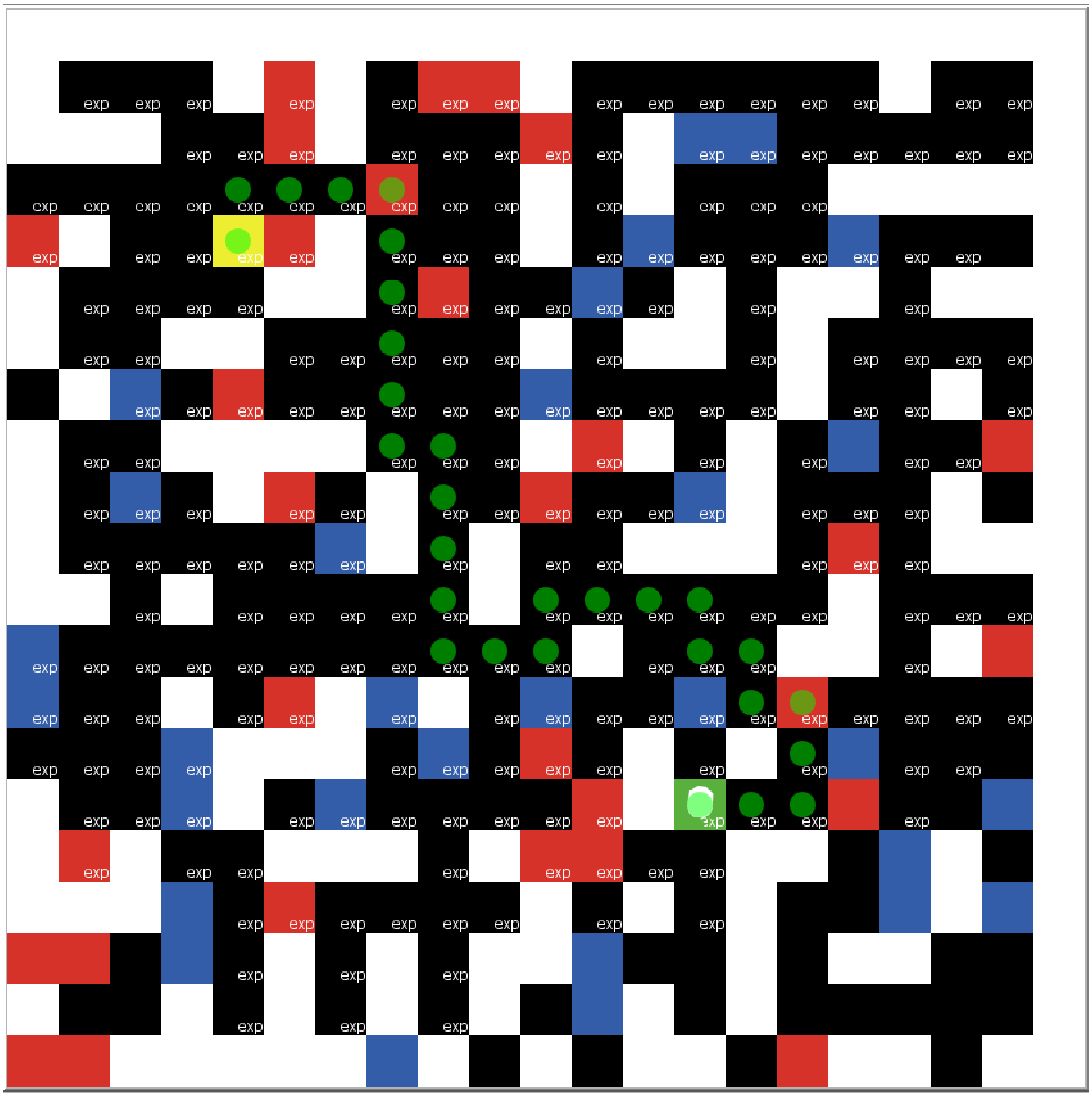


DFS: Path cost = 107, Time cost = 12.6ms

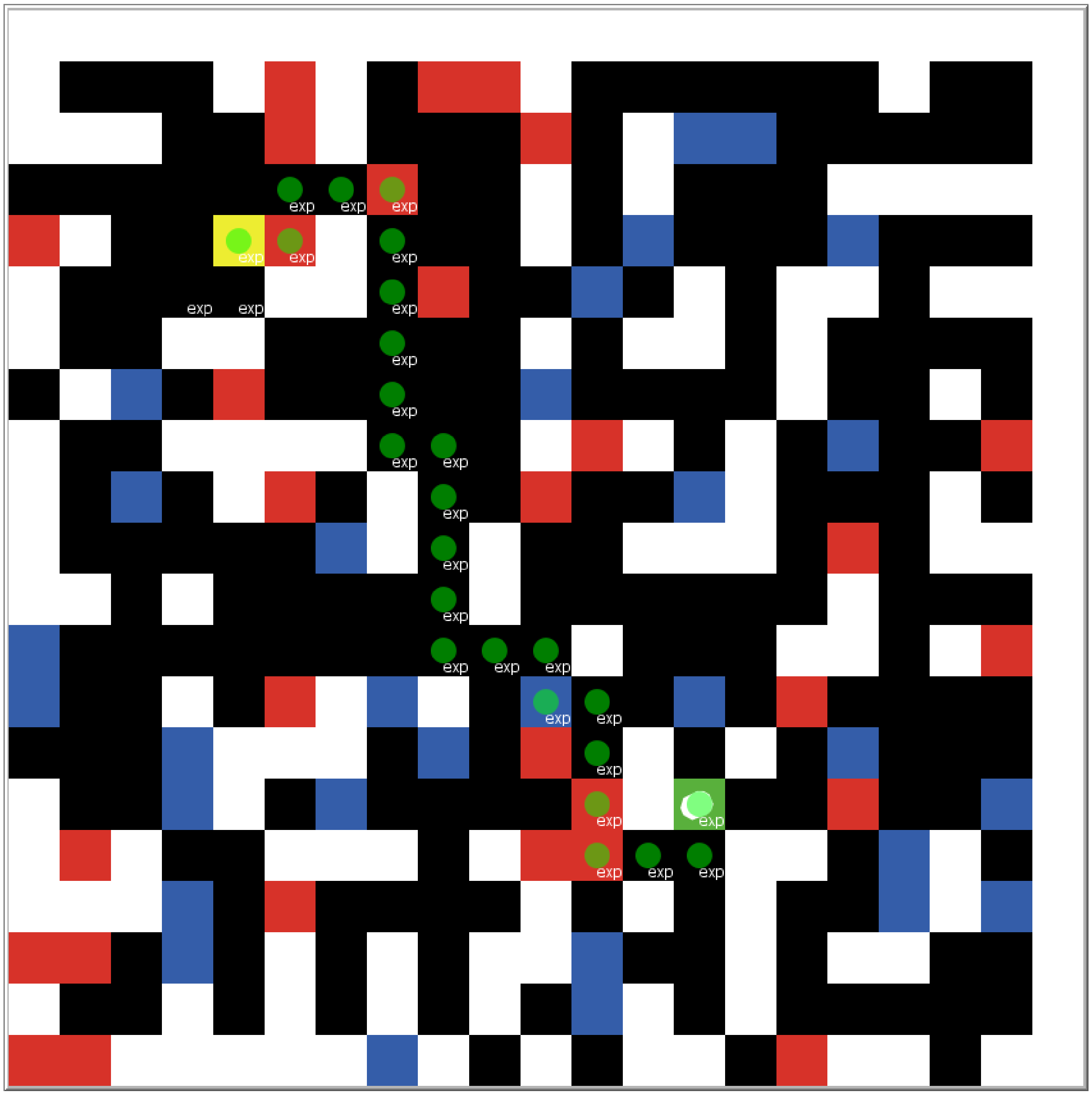
DFS proves to be unhelpful in complex situations like this, where the goal node is not too deep and the path cost varies, meaning that the more nodes they expand, the higher the risk they will walk into a path with very high cost. Both the path cost and the time cost are too expensive.


BFS: Path cost = 58, Time cost = 9.7ms

BFS outputs a better solution than DFS, but the path cost is of course, not the most optimized one (as we will see with UCS later).

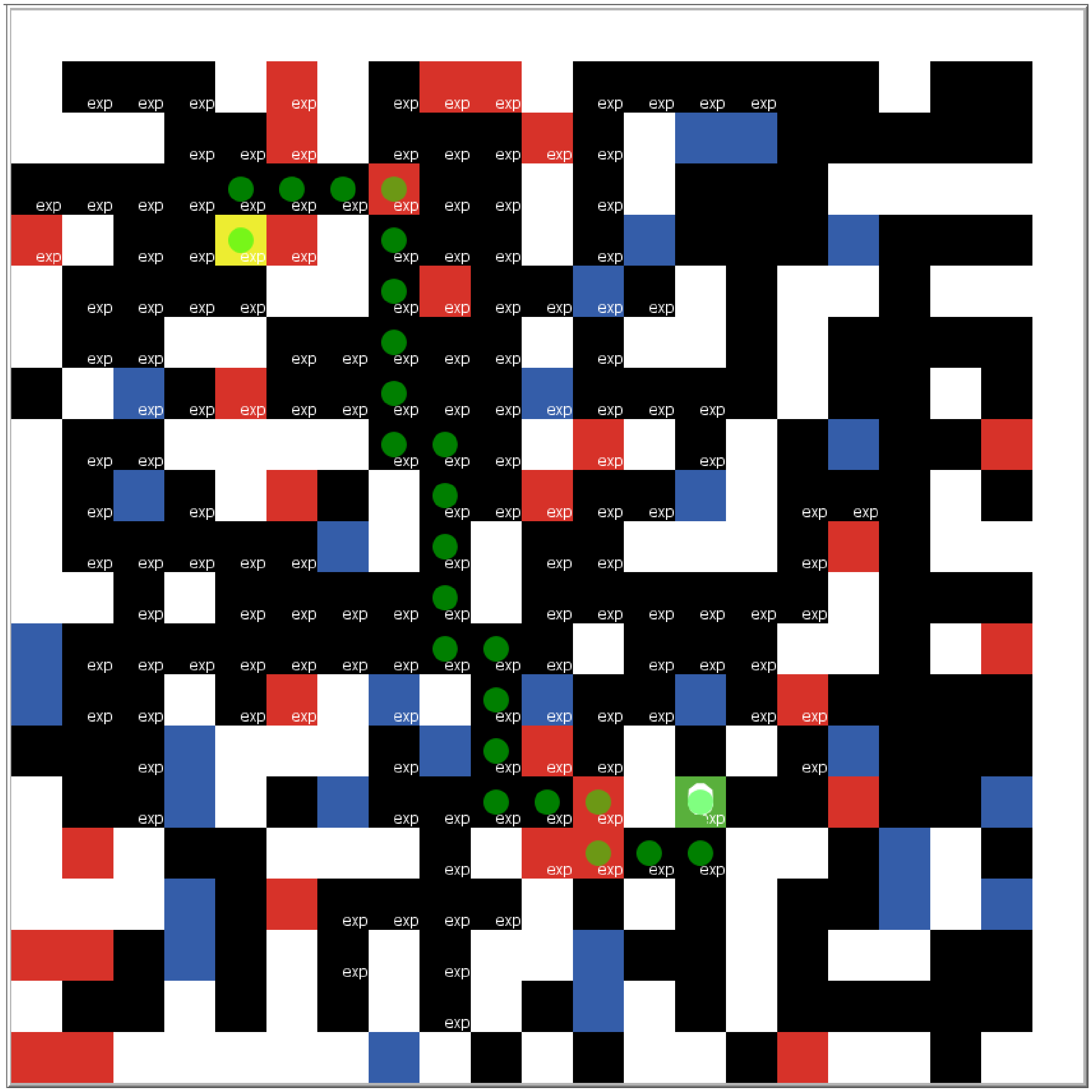

UCS: Path cost = 47, Time cost = 11ms

The minimum path cost is 47 and UCS needed to expand a large number of nodes to reach there. But luckily, UCS does not jump recklessly into choosing the deepest path or path with the least expansion. Instead, it always considers the minimum cost to reach all the nodes in its frontier, and once a node is discharged from the priority queue, we know that that node is already reached with the lowest cost possible.



GBFS: Path cost = 65, Time cost = 1.2ms

GBFS is very quick in this case but is not effective in finding solutions. It goes head-first into all traps possible surrounding the goal.

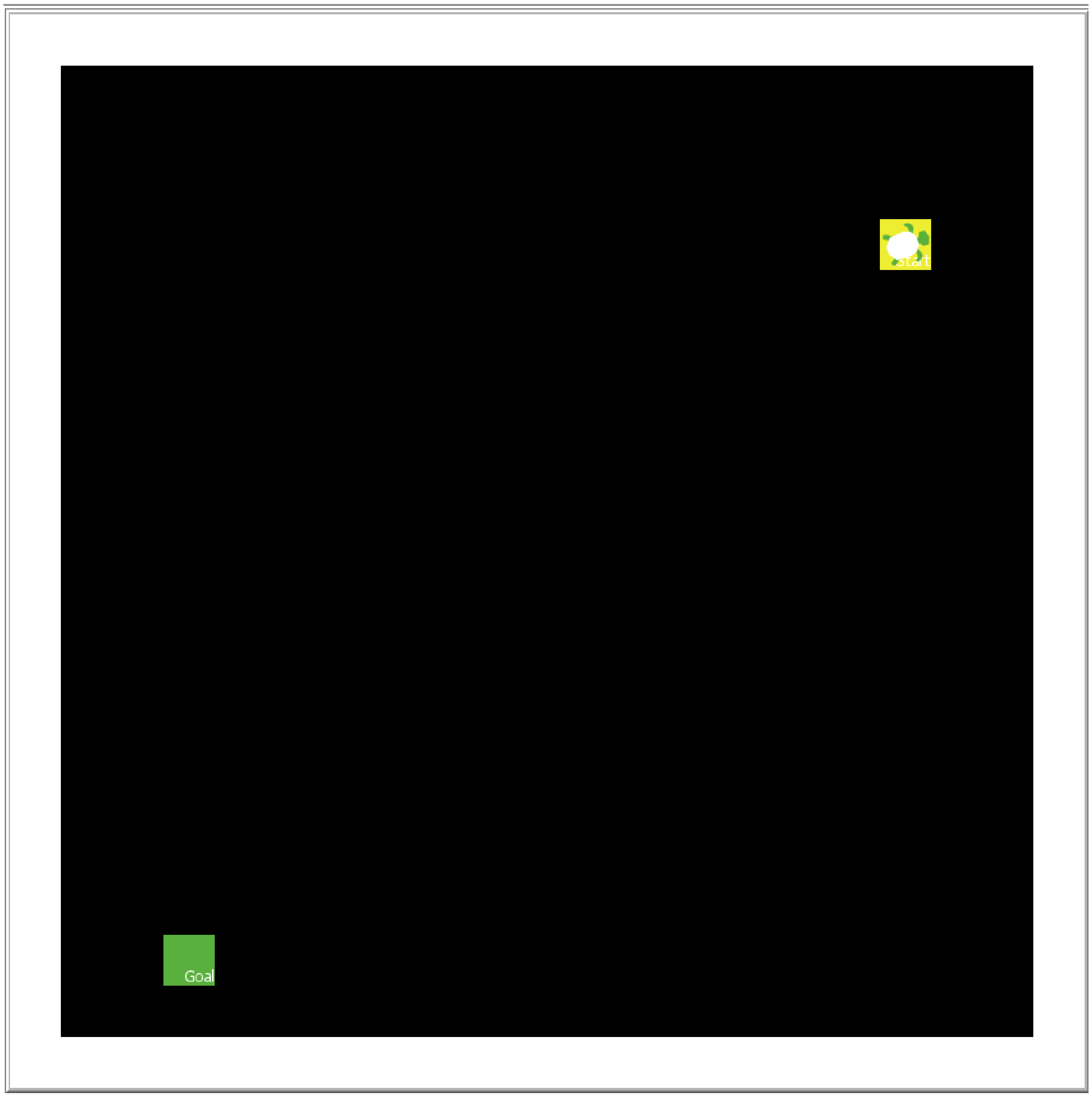

A: Path cost = 47, Time cost = 7.4ms*

Astar perfects UCS in every way possible, and we can see that by the fact that there are fewer nodes that Astar needed to expand, and the time it consumes to create this path is only 7ms, giving it the potential to become a prominent search algorithm.

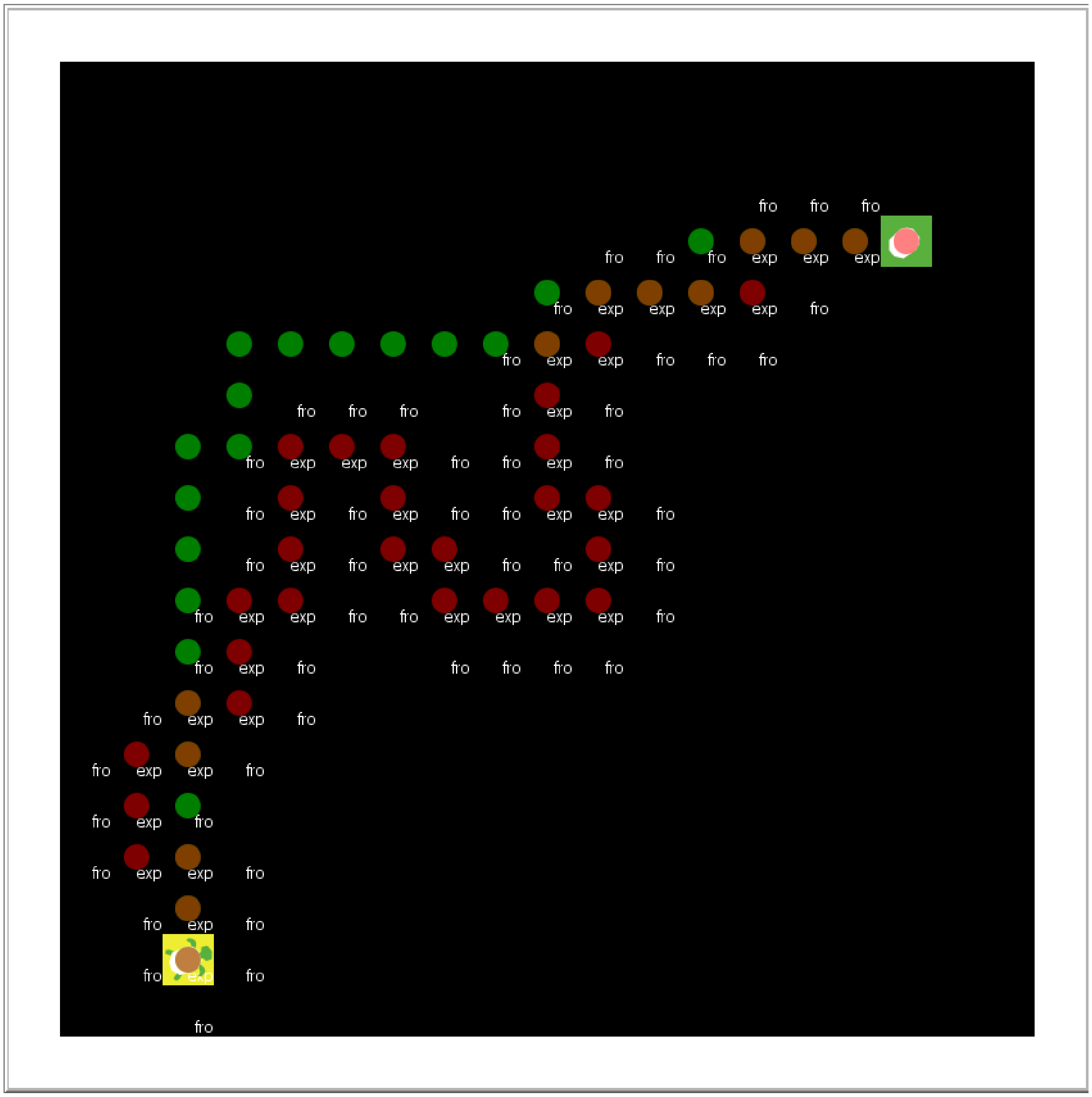
3.2. Level 2: Two algorithms, one agent, one start, two goals

The path from start to goal is marked using green circles, while the path from the goal back to start is marked using red circles. The yellow dots are the overlapping tile of the two (both the forward and back paths go through the same tile).

Map 1: No walls, no obstacles (except boundaries)



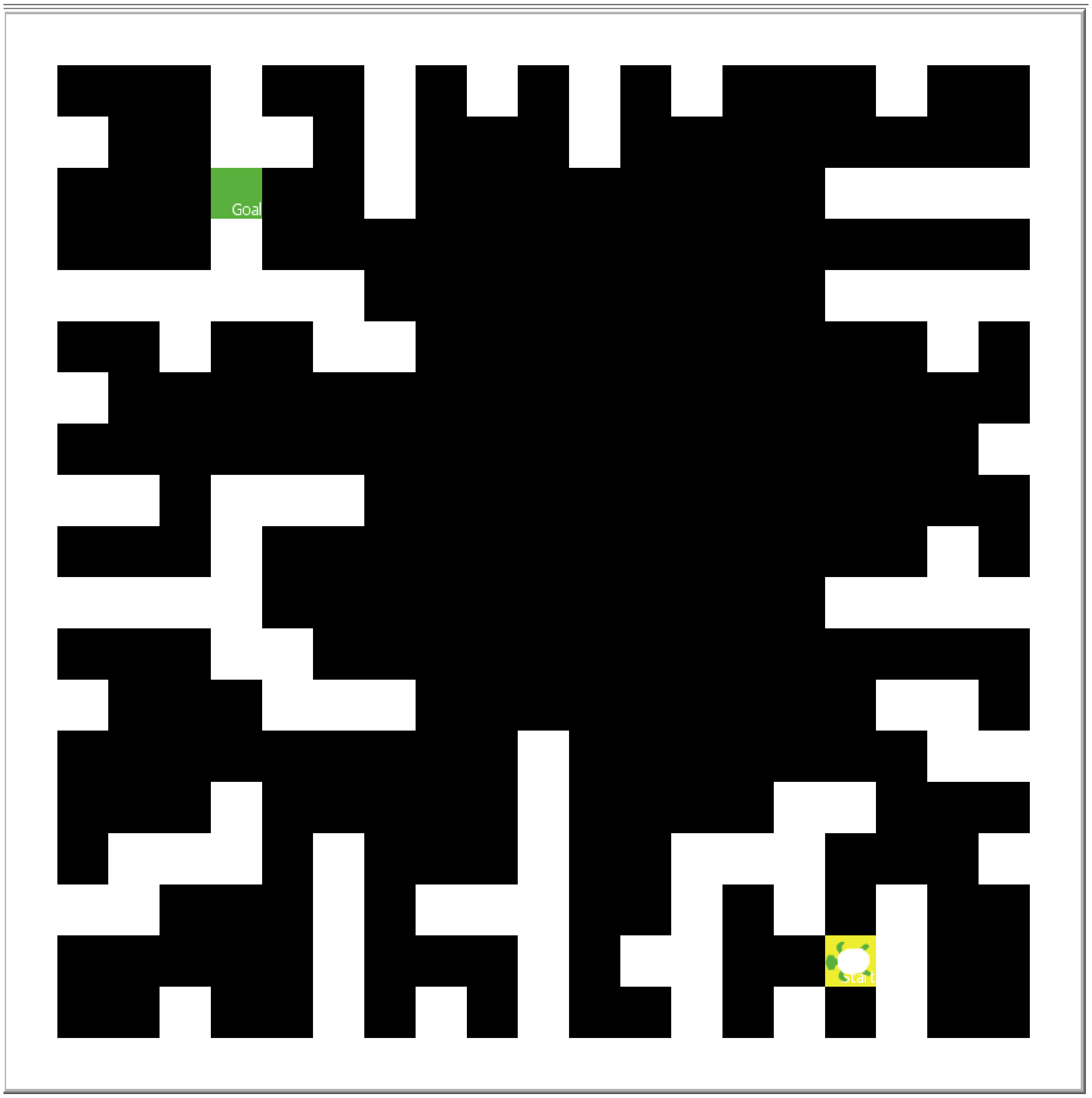
Since almost the whole level is empty, the path forward to the goal and backward to start do not change much. In some algorithms the path found will just be the same, with maybe upside down orientation.

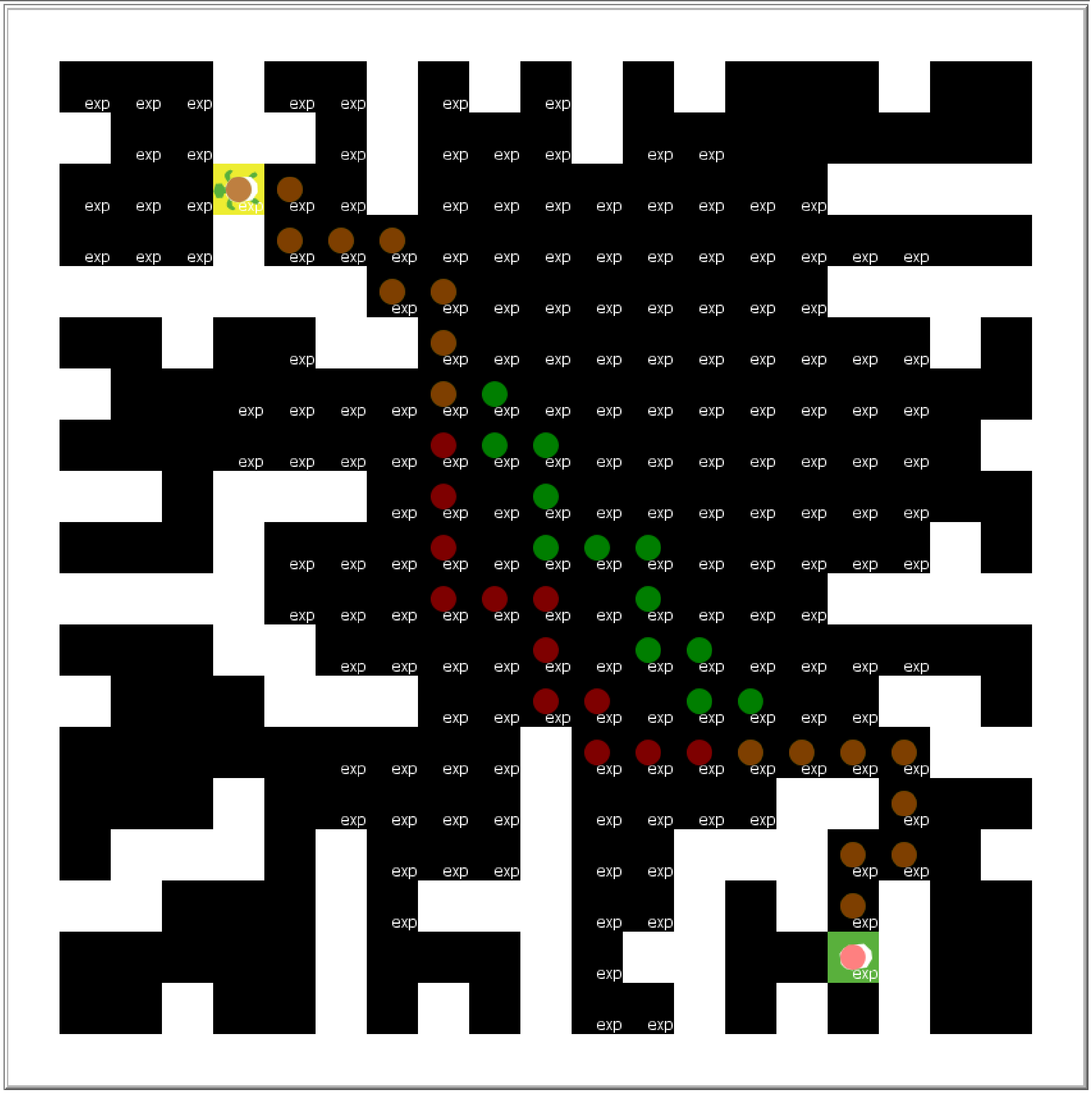


Start -> Goal: BFS (Path cost = 29, Time cost = 16.5ms)
Goal -> Start: DFS (Path cost = 39, Time cost = 6ms)

The forward algorithm is BFS, the minimum path for this path is 29. But the backward path is 39 (caused by DFS side-tracking). We see that in a wide open area (the goal is contained in the open area also). BFS should be a preferred choice.

Map 2: Half-maze

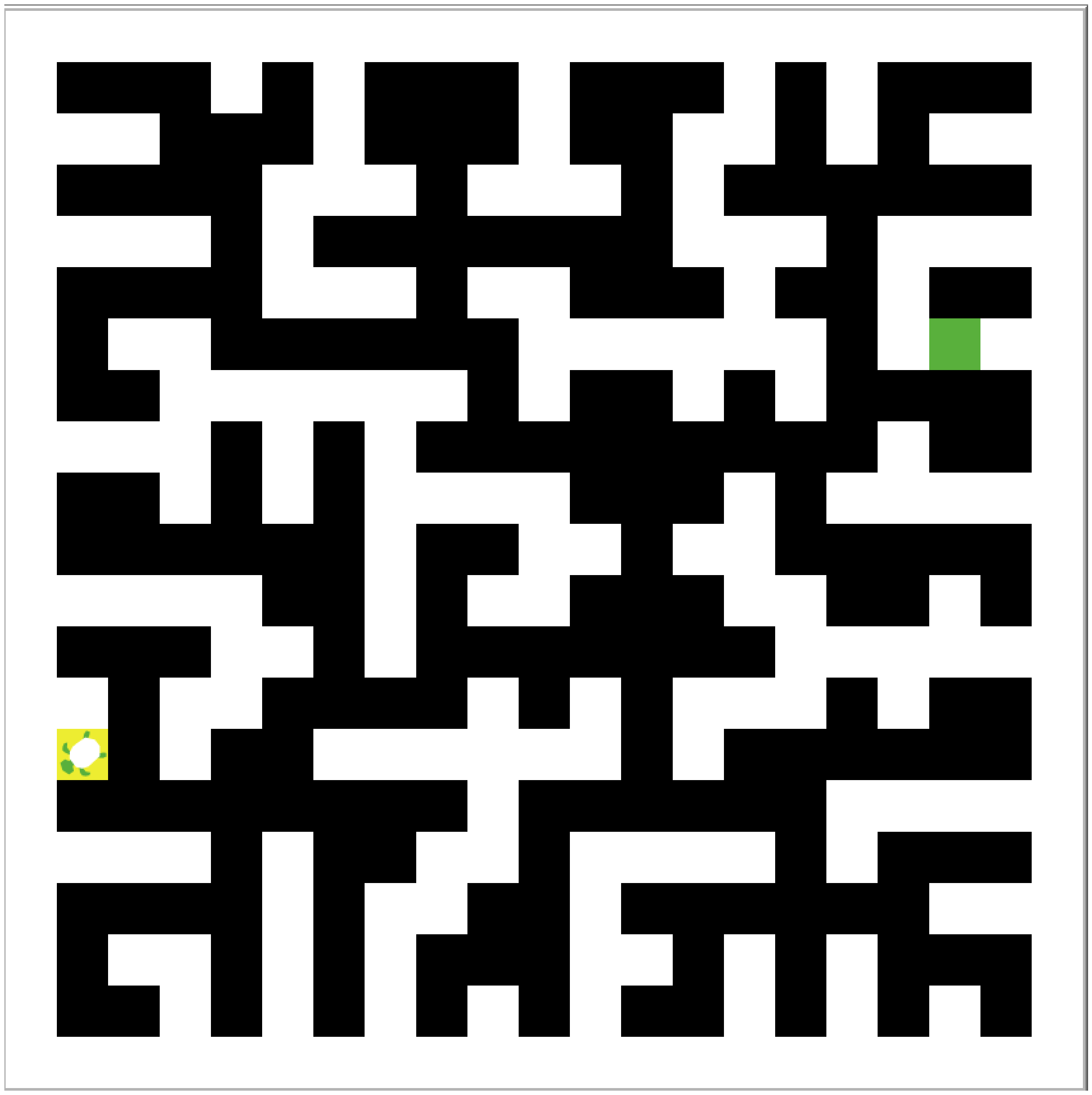


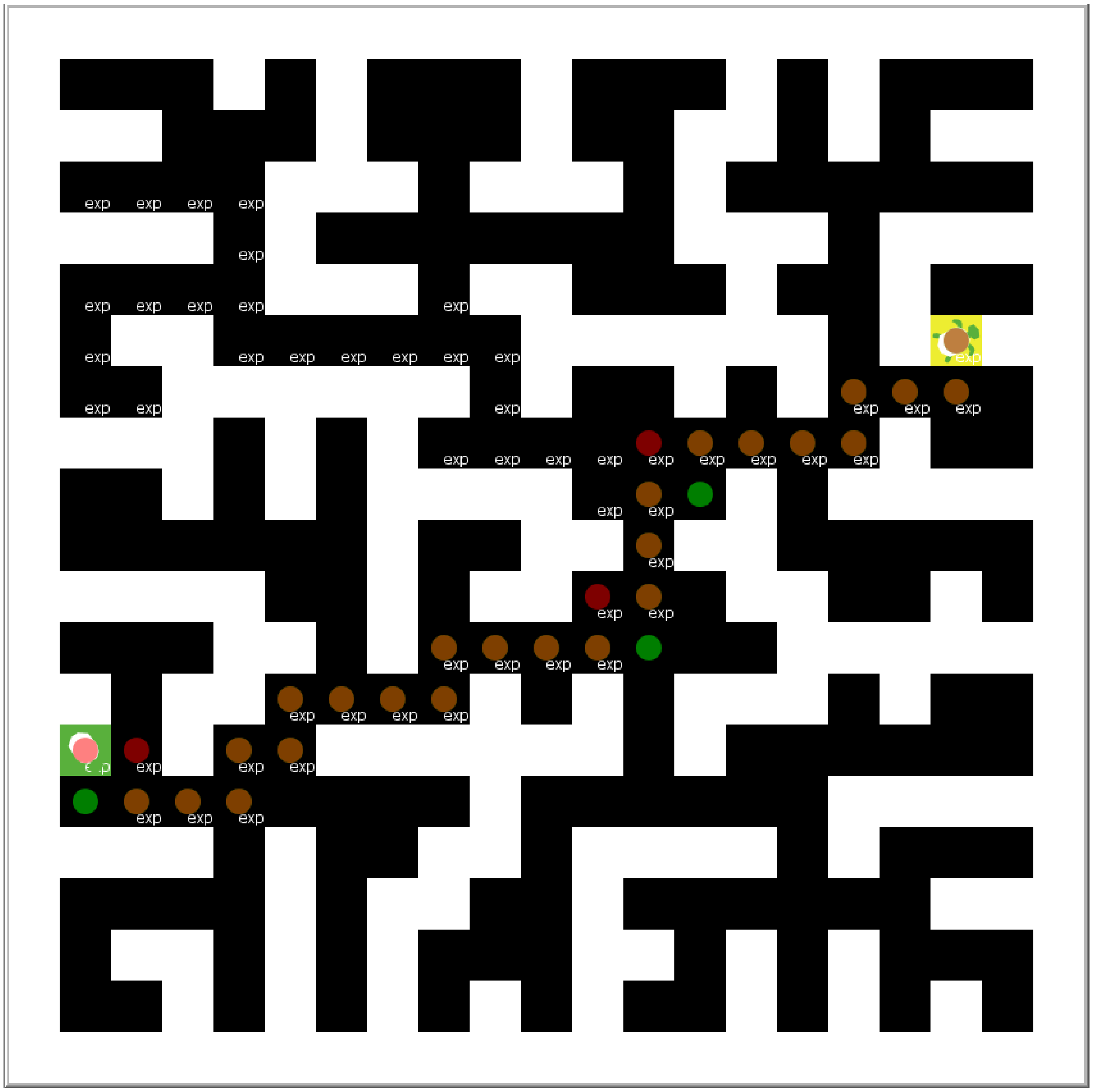


Start -> Goal: GBFS (Path cost = 30, Time cost = 1.9ms)
Goal -> Start Astar (Path cost = 30, Time = 8.6ms)

Both algorithms (forward and backward) give the same minimum path cost, with the time taken for GBFS much shorter than A*. However, from the look of it, we see that GBFS was a bit lucky as it has not hit any unsolvable wall just yet. On the other hand, Astar is just a bit bad with uniform cost search, since it considers way more nodes than necessary.

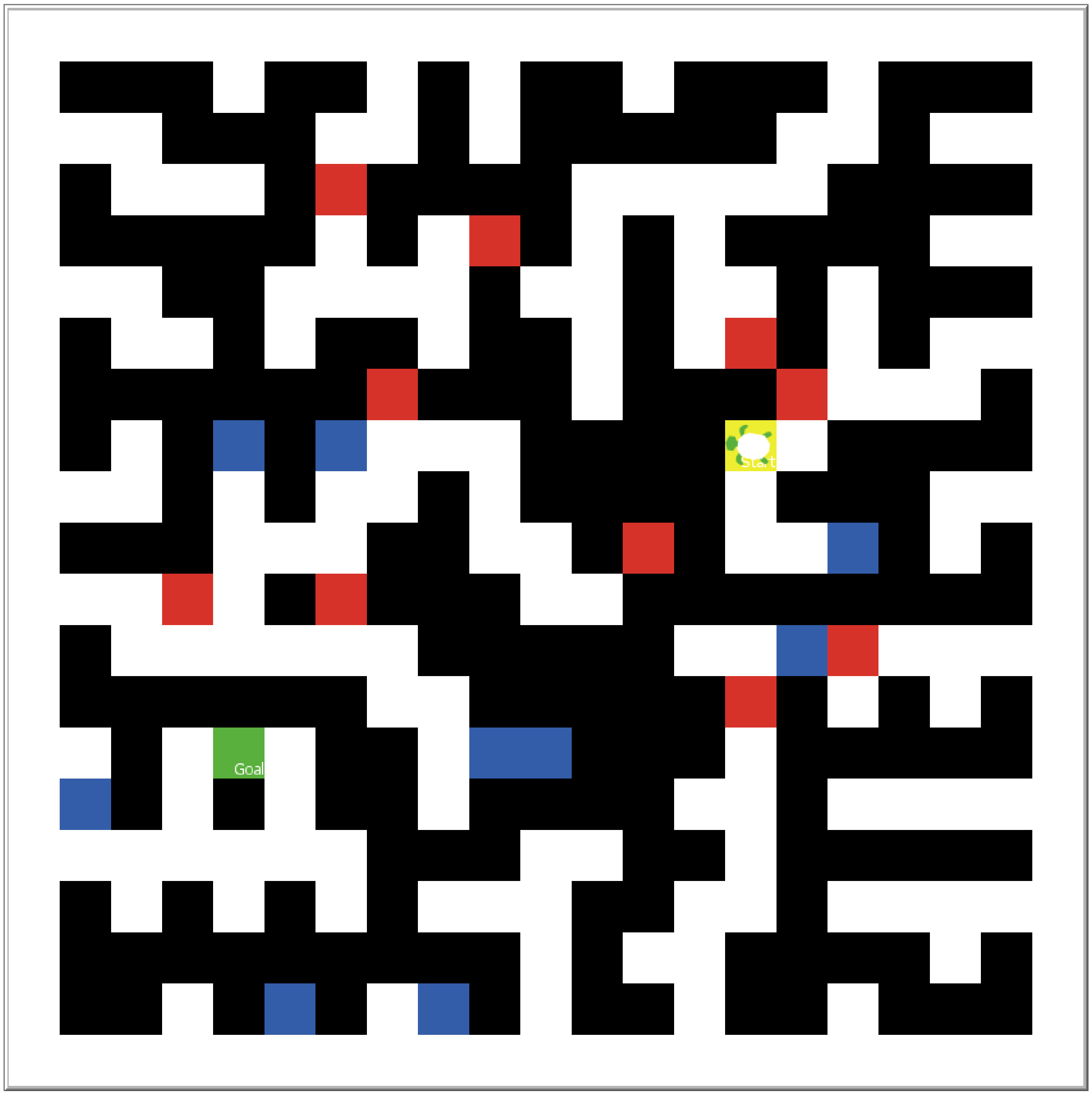
Map 3: Full maze

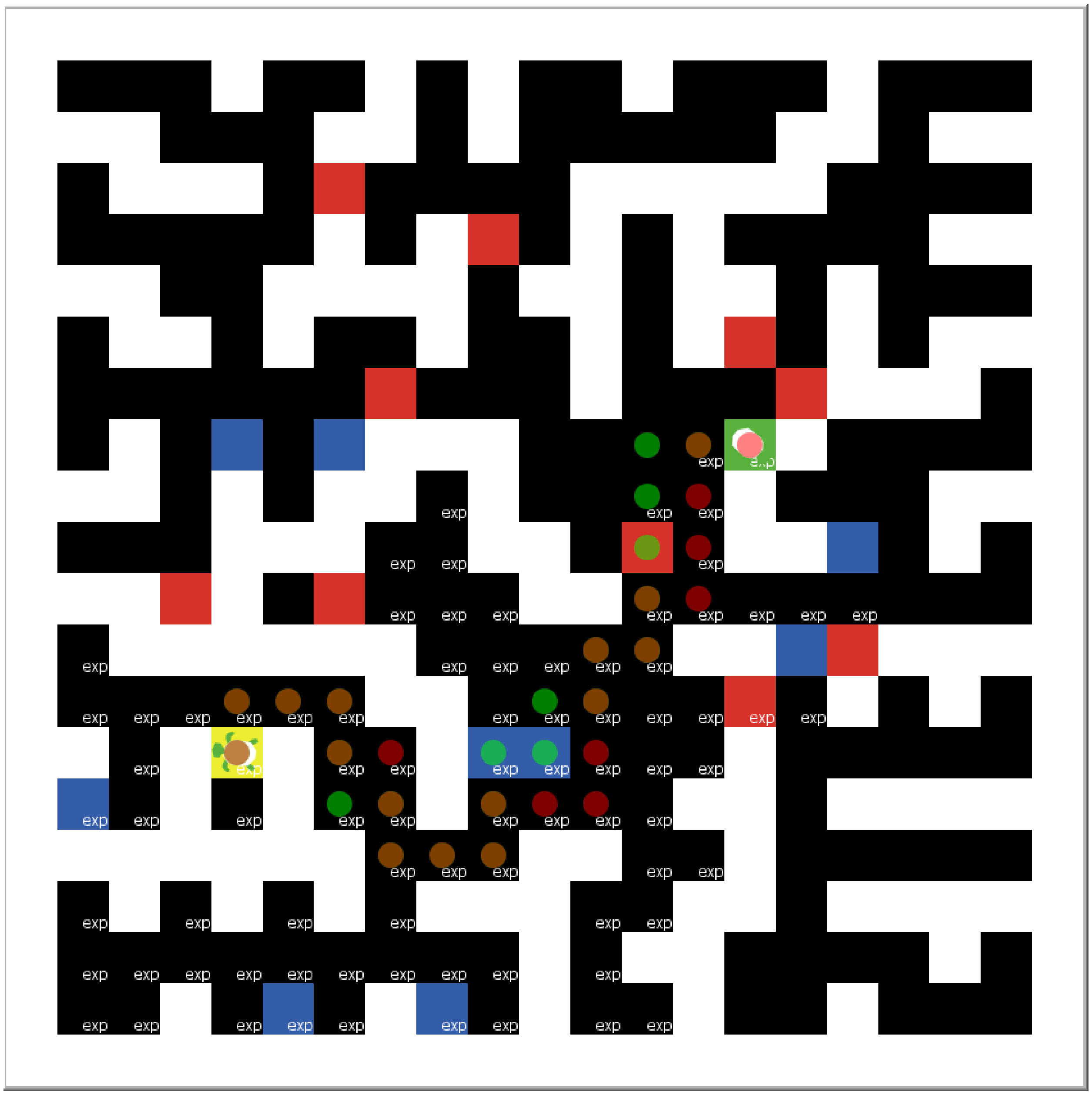

Start -> Goal: DFS (Path cost = 28, Time cost = 4.6ms)
Goal -> Start: GBFS (Path cost = 28, Time cost = 2.4ms)



Still in the uniform-cost environment, both DFS and GBFS give the same minimum path cost result, but DFS has a bit more expansion and thus lengthier. However in the end, the two paths were set on top of eachother. This means that in some cases when the straight path to the goal is blocked (the heuristic underestimates the cost), then BFS may be more preferred to deal with these situations.

Map 4: Maze with few obstacles

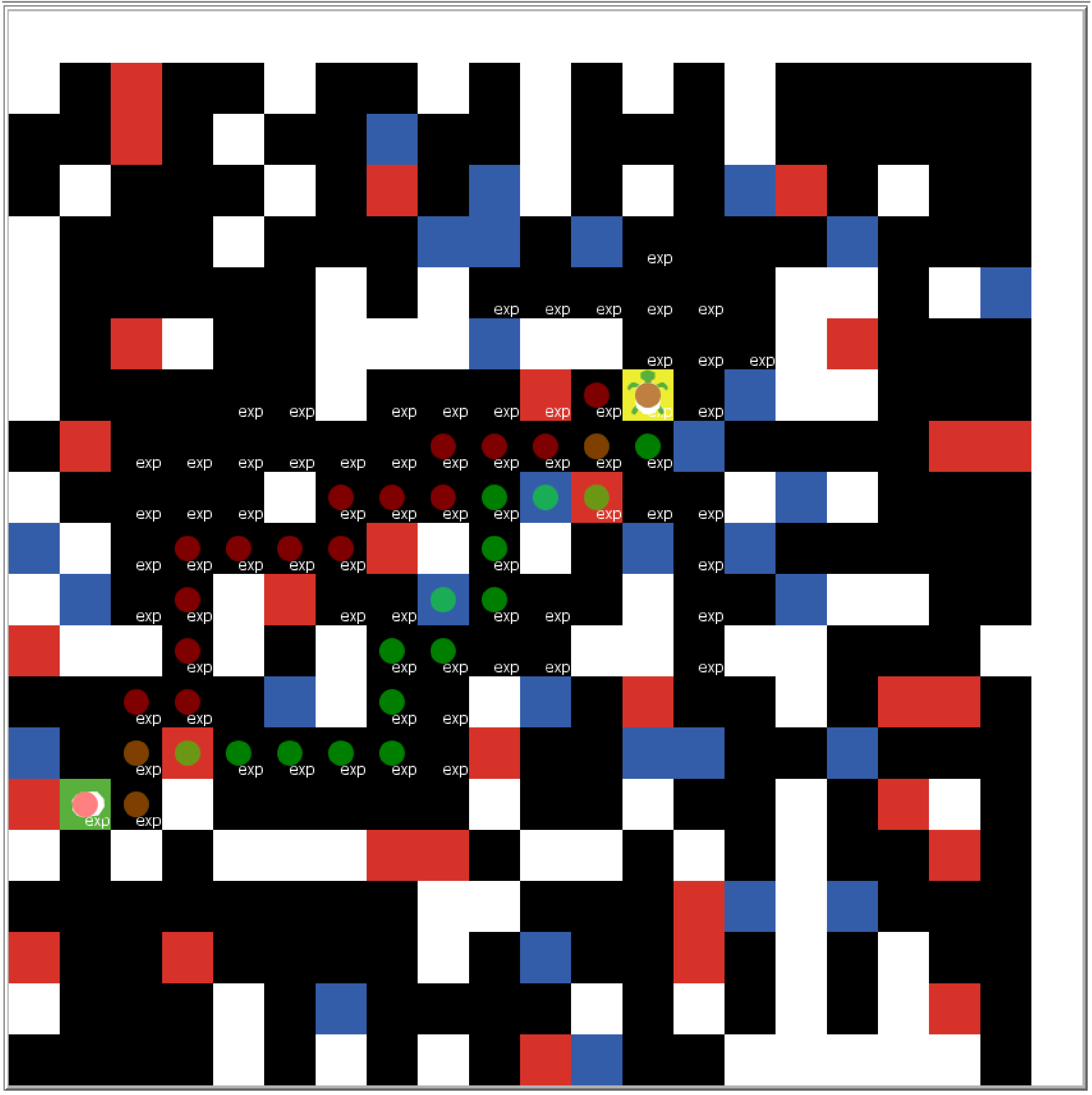




Start -> Goal: BFS (Path cost = 40, Time cost = 9.6ms)
Goal -> Start: UCS (Path cost = 23, Time cost = 3.9ms)

When different types of nodes appear with different cost functions, BFS begins to show its weak side. Both BFS and UCS try to expand in the most convenient manner possible (lowest cost, least number of expansion), but only UCS tries to account for the cost and BFS treats every of them equally. While UCS avoids high-cost nodes, BFS does not.

Map 5: Maze with many obstacles



Start -> Goal: GBFS (Path cost = 46, Time cost = 0.8ms)
Goal -> Start: Astar (Path cost = 20, Time cost = 3.3ms)

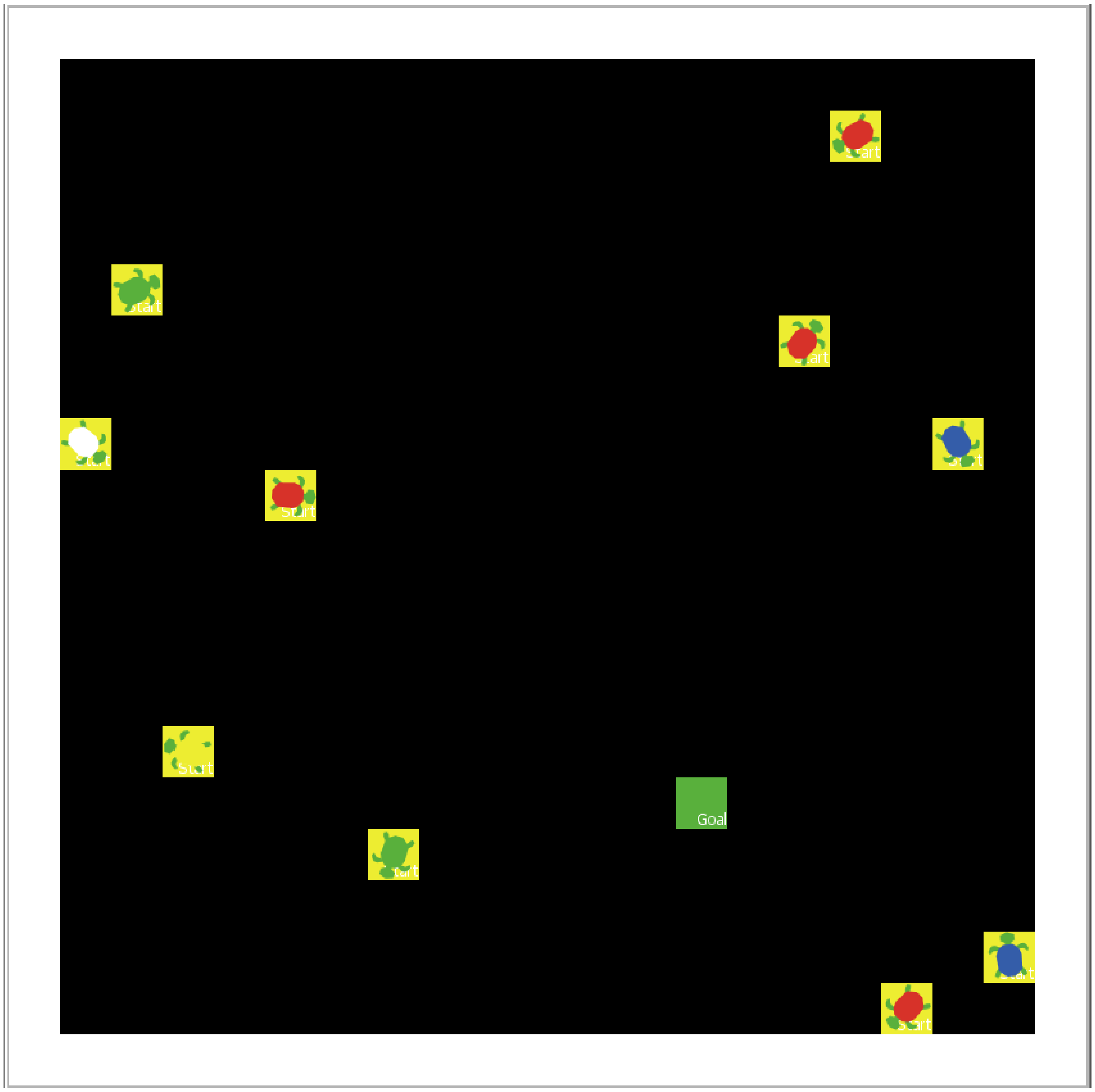
GBFS has always been much faster than Astar, memory-wise and performance-wise. But Astar always finds the optimum path while GBFS depends on luck. In case where there are too many elements to worry in the environment (too many obstacles and too many branches, especially video games), A* should be preferred over greedy algorithms.

3.3. Level 3: multiple agents, one goal, different starting points, and different algorithms

The color of the turtle's shell signifies the algorithm used:

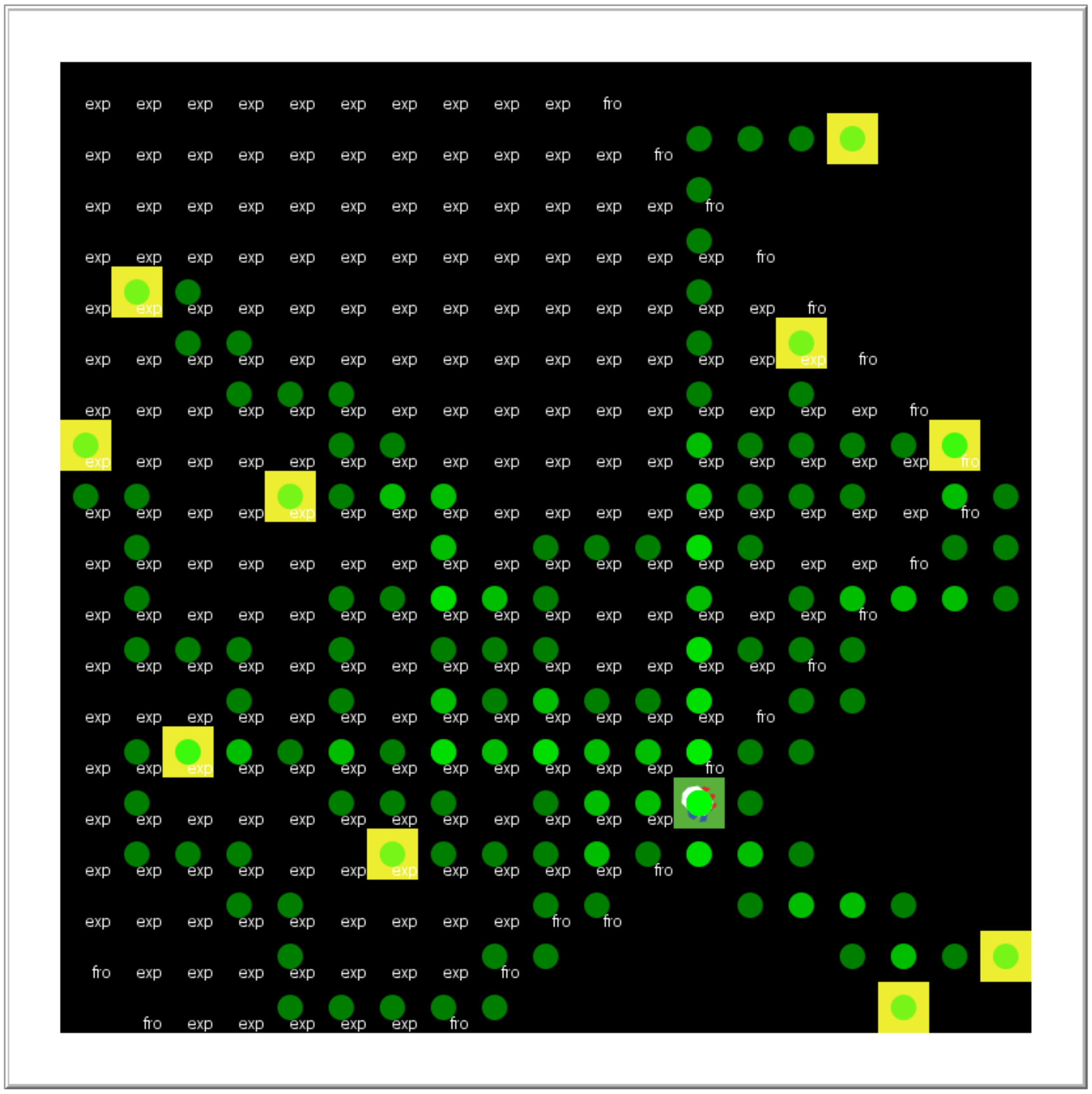
- White shell: DFS
- Red shell: BFS
- Yellow shell: UCS
- Blue shell: GBFS
- Green shell: A*

Map 1: No walls, no obstacles (except boundaries)



```
observer: "Path cost from DFS: 72"
observer: "Time cost: 4.535 milliseconds"
observer: "Path cost from GBFS: 10"
observer: "Time cost: 0.137 milliseconds"
observer: "Path cost from BFS: 17"
observer: "Time cost: 1.513 milliseconds"
observer: "Path cost from Astar: 22"
observer: "Time cost: 1.172 milliseconds"
observer: "Path cost from GBFS: 13"
observer: "Time cost: 0.123 milliseconds"
```

```
observer: "Path cost from Astar: 8"
observer: "Time cost: 0.081 milliseconds"
observer: "Path cost from BFS: 12"
observer: "Time cost: 1.129 milliseconds"
observer: "Path cost from BFS: 9"
observer: "Time cost: 0.232 milliseconds"
observer: "Path cost from UCS: 12"
observer: "Time cost: 1.111 milliseconds"
observer: "Path cost from BFS: 15"
observer: "Time cost: 1.178 milliseconds"
```

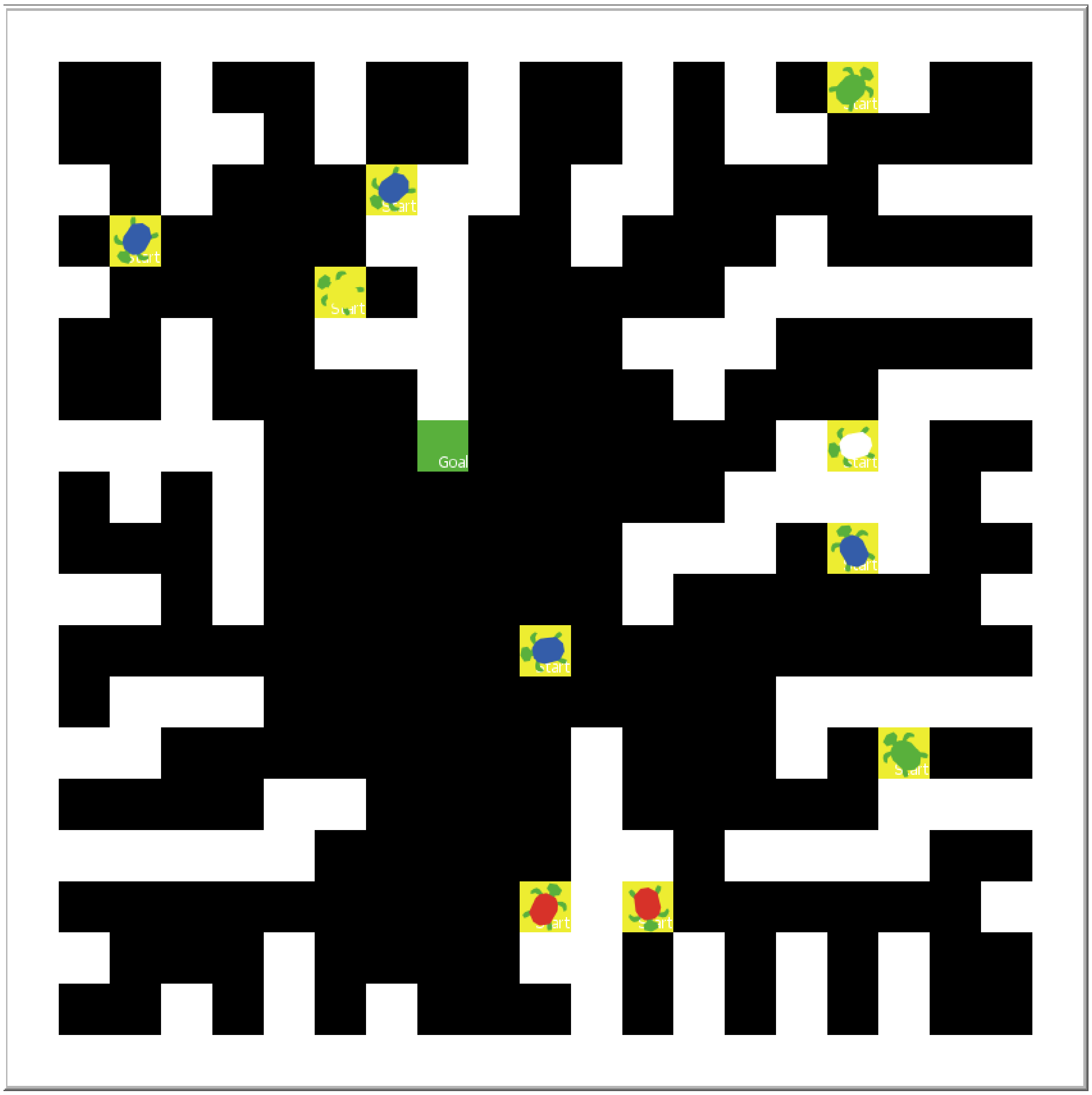


Commenting

- DFS gives the worst paths in an open environment.
- GBFS is always time-efficient no matter where it is. It is cost-efficient when in an open environment and/or the goal is not hindered or surrounded.
- BFS outputs the minimum path in a uniform cost environment, but it also expands the most nodes and causes performance drop.

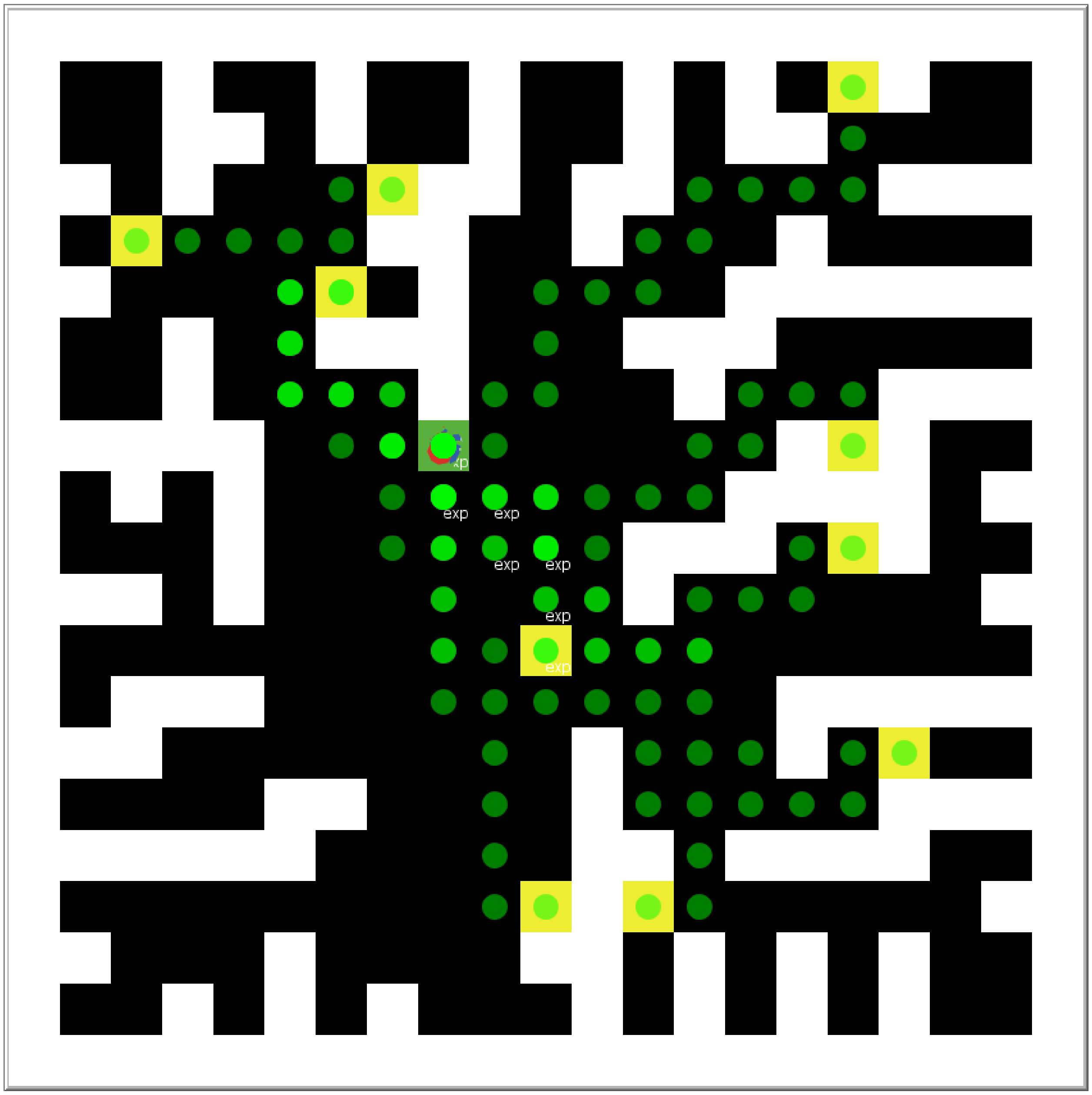
- UCS is similar to BFS in this case.
- Astar is a combination of UCS and GBFS, so it is a bit more efficient than UCS, but not as fast as GBFS (still, A* guarantees optimality more than the others).

Map 2: Half maze



observer: "Path cost from BFS: 12"
observer: "Time cost: 0.964 milliseconds"
observer: "Path cost from GBFS: 15"

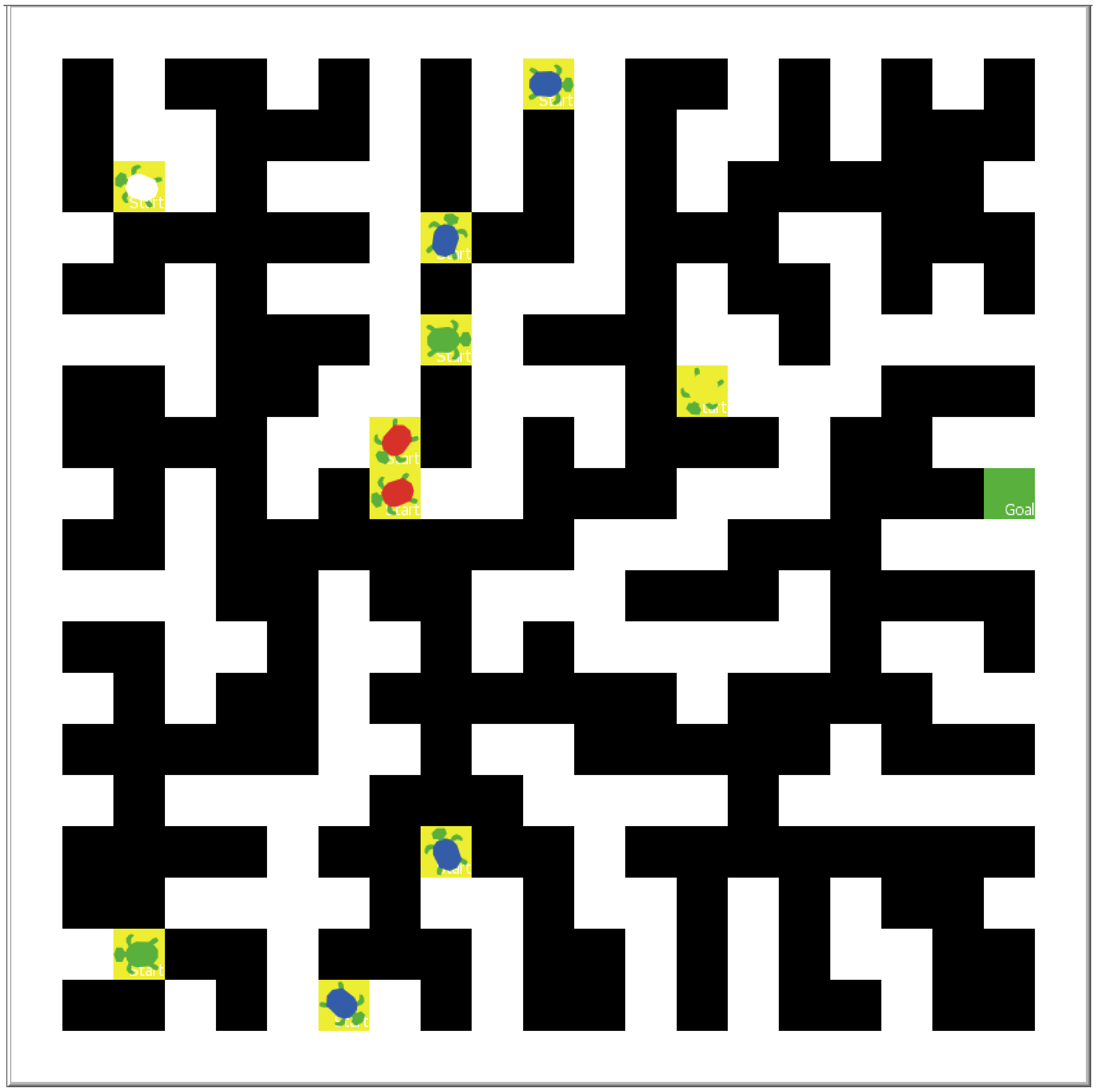
```
observer: "Time cost: 0.142 milliseconds"
observer: "Path cost from GBFS: 11"
observer: "Time cost: 0.057 milliseconds"
observer: "Path cost from DFS: 17"
observer: "Time cost: 3.125 milliseconds"
observer: "Path cost from Astar: 16"
observer: "Time cost: 0.262 milliseconds"
observer: "Path cost from Astar: 18"
observer: "Time cost: 0.592 milliseconds"
observer: "Path cost from BFS: 16"
observer: "Time cost: 1.171 milliseconds"
observer: "Path cost from GBFS: 11"
observer: "Time cost: 0.111 milliseconds"
observer: "Path cost from UCS: 8"
observer: "Time cost: 0.626 milliseconds"
observer: "Path cost from GBFS: 7"
observer: "Time cost: 0.09 milliseconds"
```



Commenting

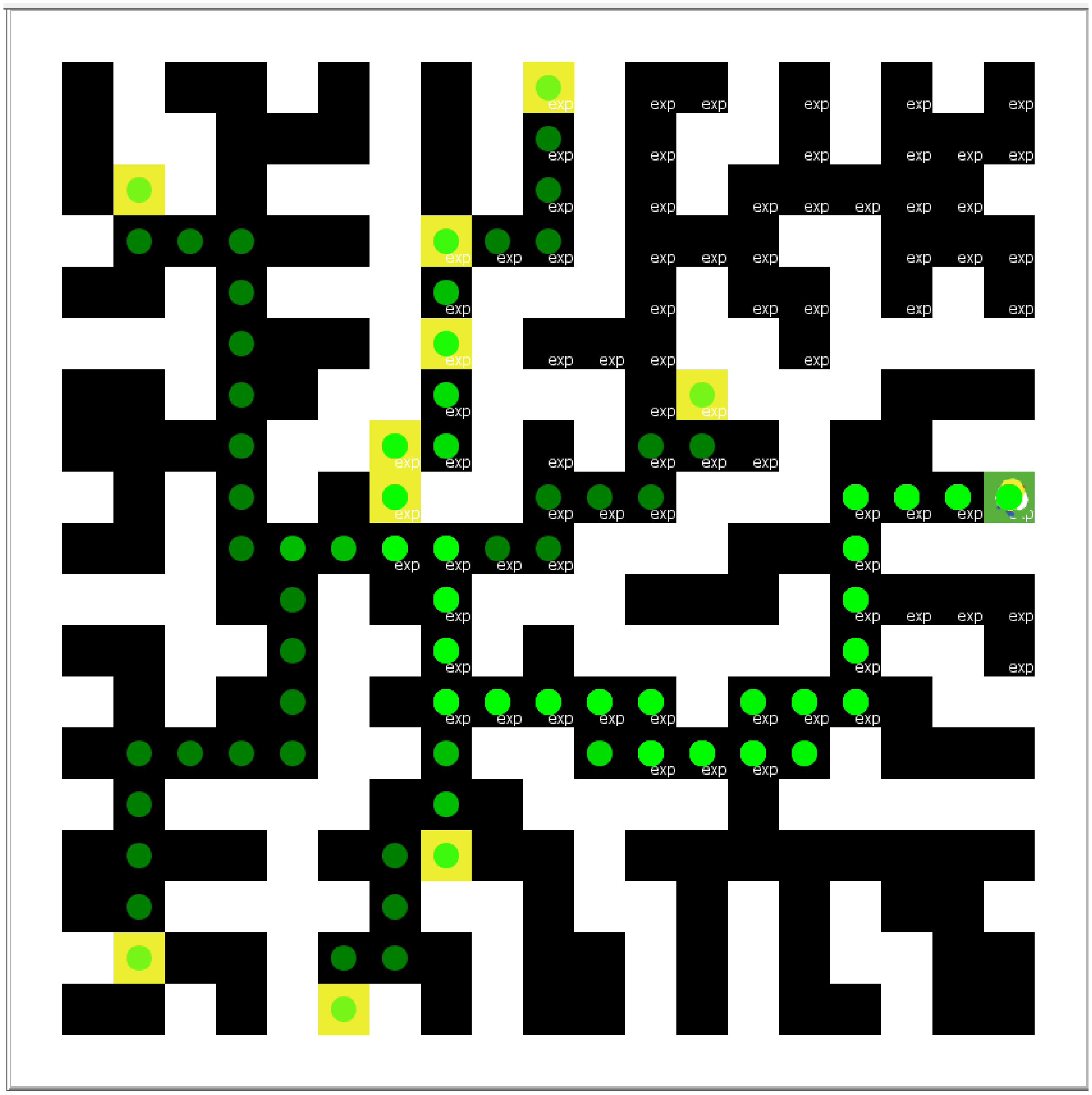
- DFS is a bit better in closed and cramped environments.
- GBFS is very quick, but only in this case where the goal is out in the open and not surrounded by a cage, for example.
- UCS is similar to BFS in this case. By the time they all expand all the necessary nodes, Astar and GBFS will have finished long ago.
- Astar's ability is still hindered because it makes no difference when dealing with uniform path cost.

Map 3: Full maze



```
observer: "Path cost from UCS: 29"
observer: "Time cost: 2.276 milliseconds"
observer: "Path cost from DFS: 34"
observer: "Time cost: 0.793 milliseconds"
observer: "Path cost from GBFS: 29"
observer: "Time cost: 0.483 milliseconds"
observer: "Path cost from BFS: 23"
observer: "Time cost: 1.973 milliseconds"
```

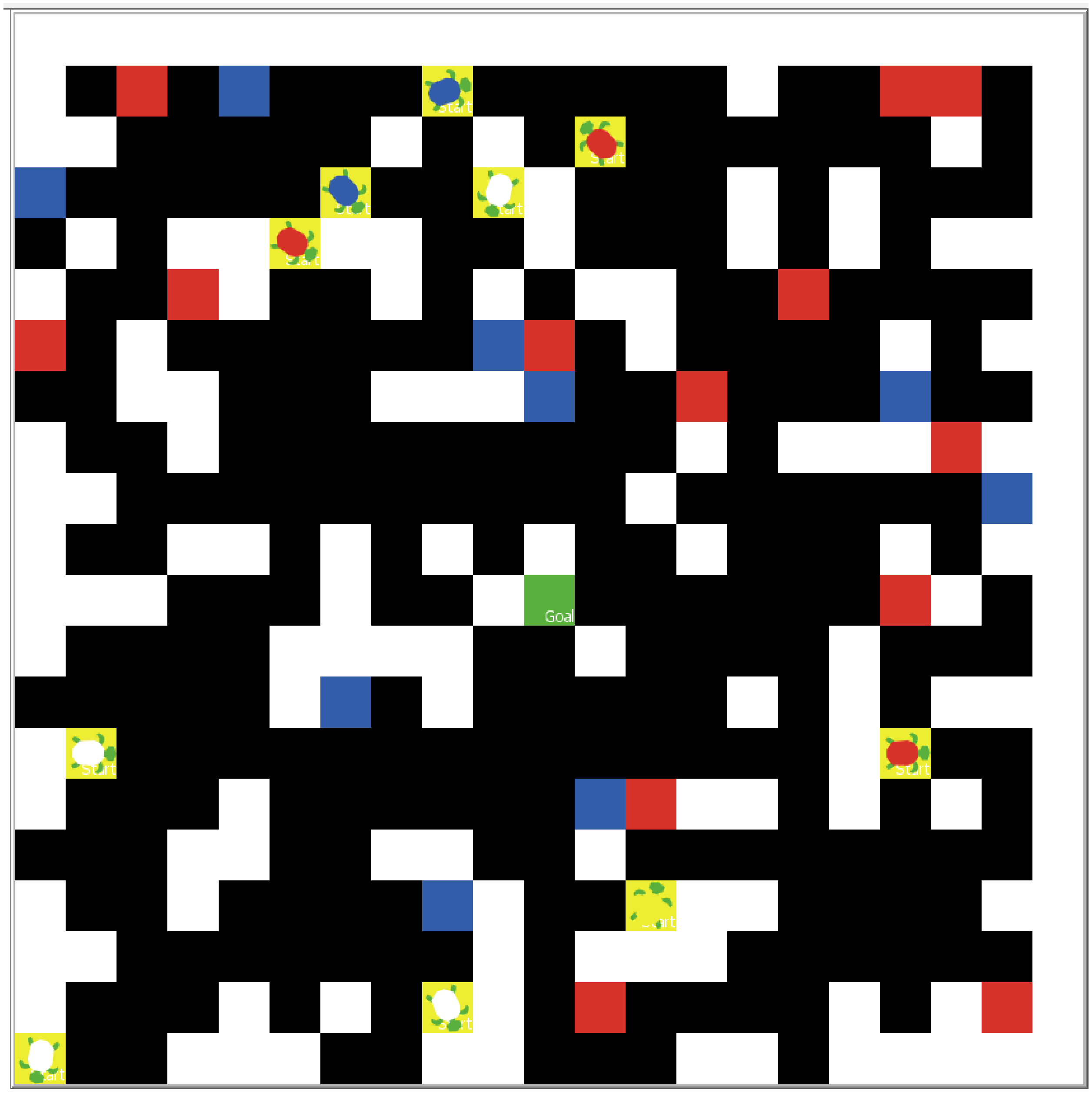
```
observer: "Path cost from Astar: 27"
observer: "Time cost: 1.002 milliseconds"
observer: "Path cost from GBFS: 21"
observer: "Time cost: 0.387 milliseconds"
observer: "Path cost from Astar: 35"
observer: "Time cost: 1.433 milliseconds"
observer: "Path cost from GBFS: 26"
observer: "Time cost: 0.145 milliseconds"
observer: "Path cost from BFS: 24"
observer: "Time cost: 0.789 milliseconds"
observer: "Path cost from GBFS: 34"
observer: "Time cost: 0.343 milliseconds"
```



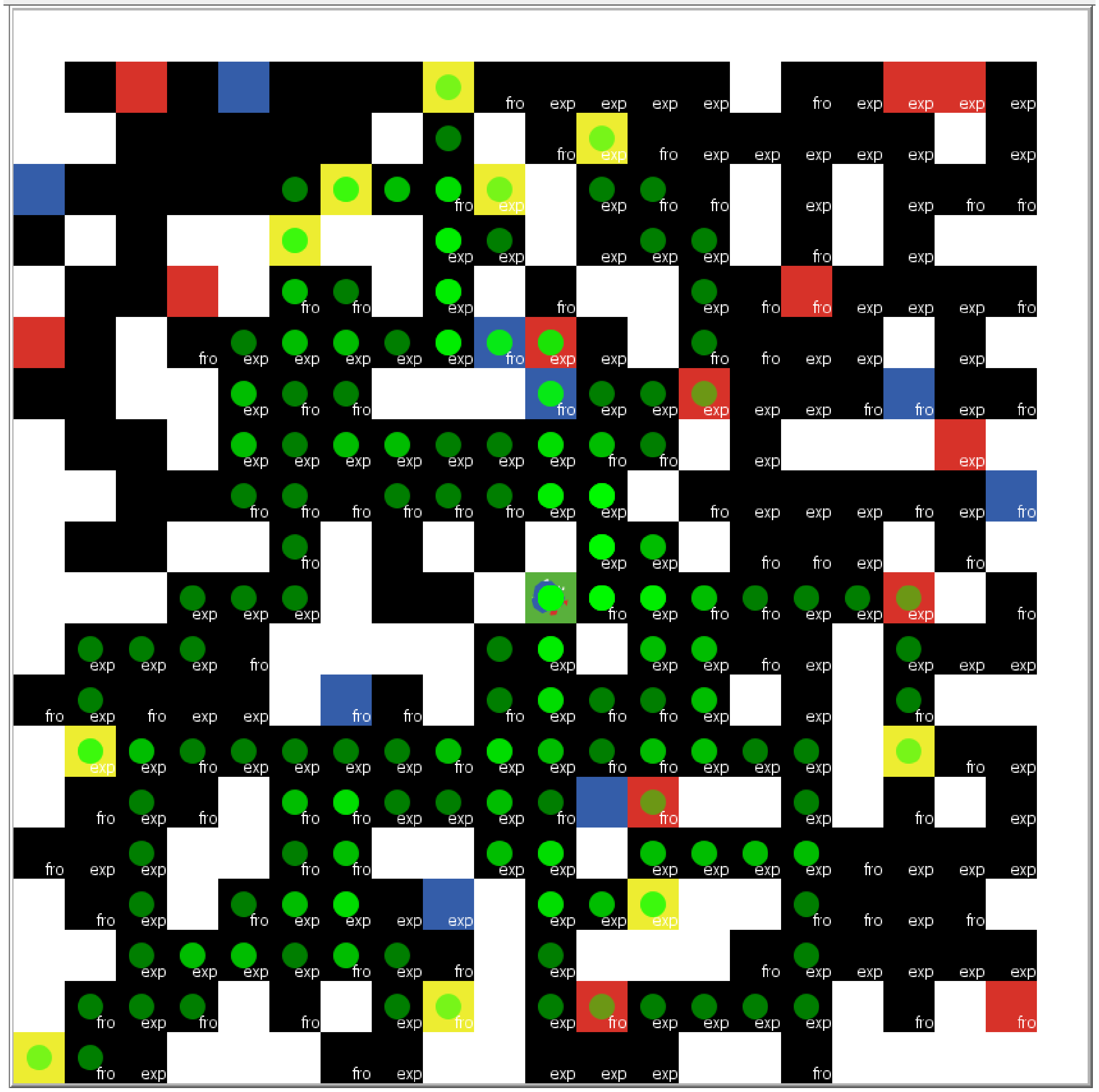
Commenting

- DFS is now nearly as quick as BFS and GBFS.
- GBFS is very quick, but only in this case where the goal is out in the open and not surrounded by a cage, for example.
- UCS is similar to BFS in this case. Even at a very near distance to the goal, they still fail to succeed GBFS in reaching there first.
- Astar's ability is still hindered because it makes no difference when dealing with uniform path cost.

Map 4: Maze with few obstacles



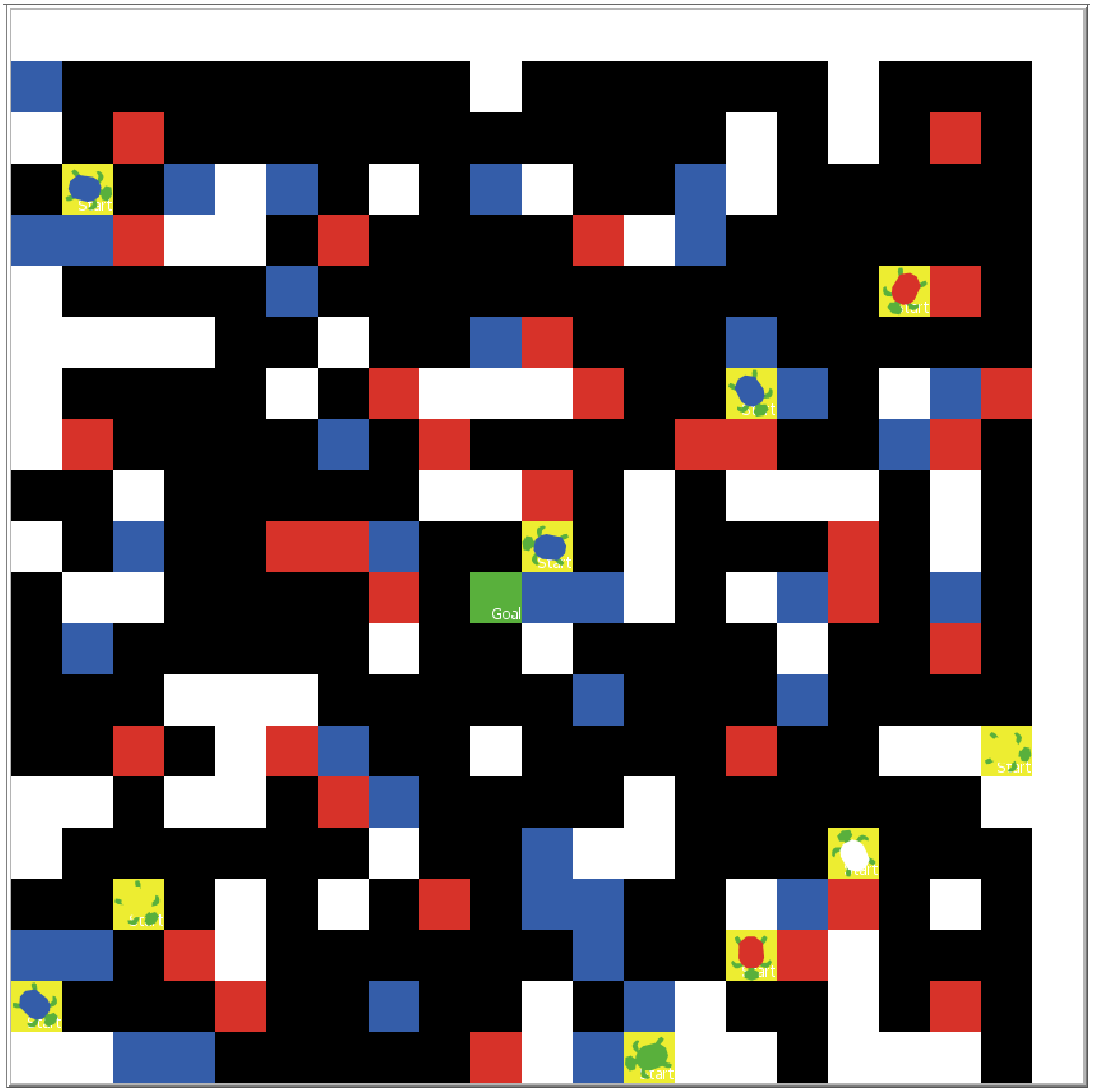
```
observer: "Path cost from Astar: 26"
observer: "Time cost: 0.38 milliseconds"
observer: "Path cost from UCS: 28"
observer: "Time cost: 0.364 milliseconds"
observer: "Path cost from GBFS: 33"
observer: "Time cost: 0.141 milliseconds"
observer: "Path cost from BFS: 26"
observer: "Time cost: 0.6 milliseconds"
observer: "Path cost from DFS: 21"
observer: "Time cost: 1.123 milliseconds"
observer: "Path cost from BFS: 30"
observer: "Time cost: 0.433 milliseconds"
observer: "Path cost from UCS: 27"
observer: "Time cost: 0.639 milliseconds"
observer: "Path cost from BFS: 29"
observer: "Time cost: 0.616 milliseconds"
observer: "Path cost from DFS: 22"
observer: "Time cost: 2.691 milliseconds"
observer: "Path cost from Astar: 19"
observer: "Time cost: 0.222 milliseconds"
```



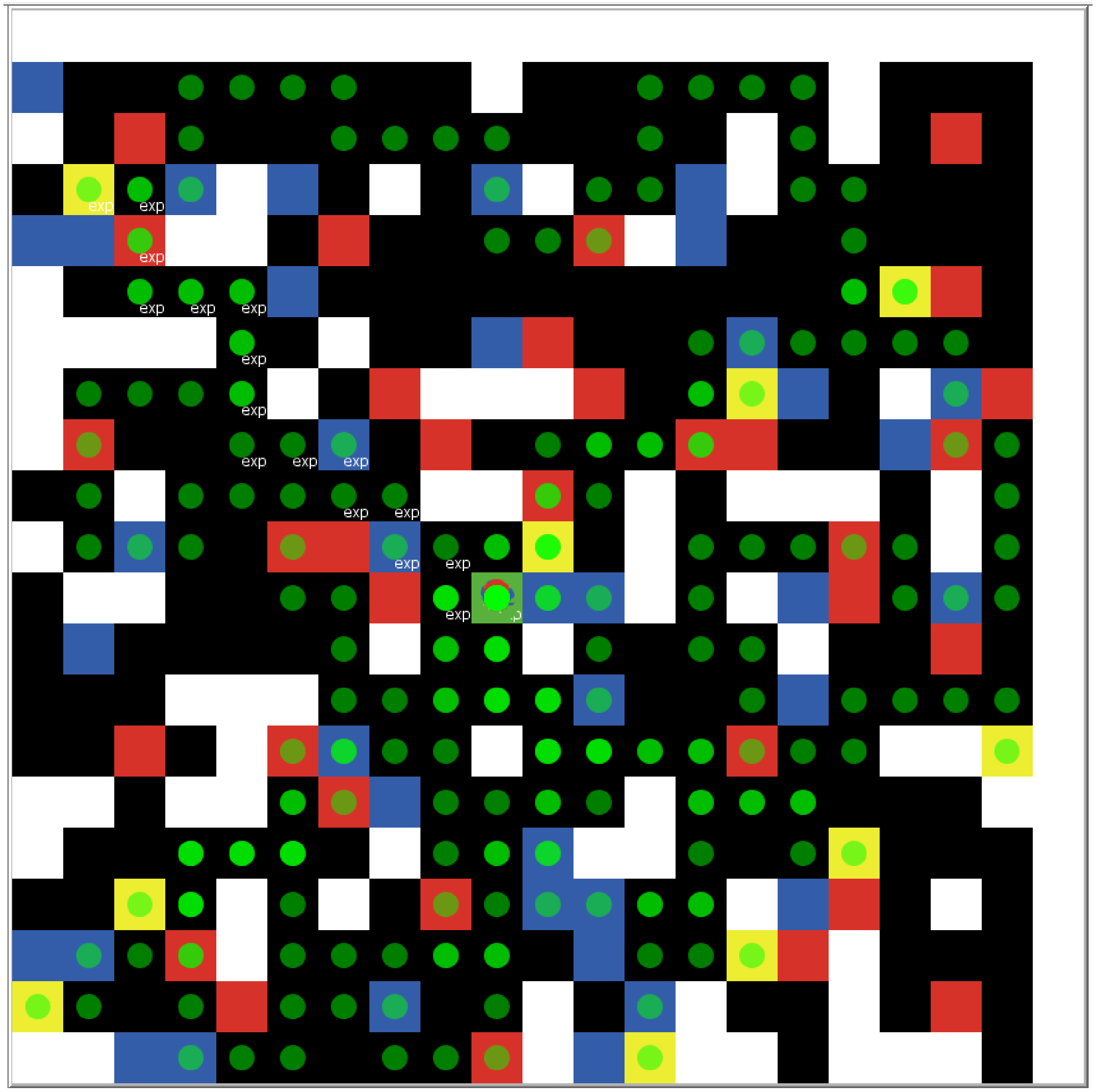
Commenting

- Astar, UCS and BFS prove to be more useful than GBFS and DFS right now.
- BFS is still stuck with expanding many nodes and ignoring their high cost.
- Comparing between Astar and UCS, we see that UCS is a bit slower, because it is more likely to expand circularly (the nodes closer to it, rather than to the goal, are considered first). Astar is the opposite, where nodes closer to the goal are more likely to be considered next.
- If we only relate time, then DFS is still quite useful as it beats GBFS when standing at a close distance to the goal.

Map 5: Maze with many obstacles



```
observer: "Path cost from BFS: 25"
observer: "Time cost: 1.069 milliseconds"
observer: "Path cost from GBFS: 28"
observer: "Time cost: 0.041 milliseconds"
observer: "Path cost from GBFS: 7"
observer: "Time cost: 0.017 milliseconds"
observer: "Path cost from Astar: 19"
observer: "Time cost: 0.129 milliseconds"
observer: "Path cost from GBFS: 44"
observer: "Time cost: 0.121 milliseconds"
observer: "Path cost from DFS: 254"
observer: "Time cost: 2.784 milliseconds"
observer: "Path cost from UCS: 20"
observer: "Time cost: 0.842 milliseconds"
observer: "Path cost from UCS: 18"
observer: "Time cost: 1.633 milliseconds"
observer: "Path cost from BFS: 37"
observer: "Time cost: 0.782 milliseconds"
observer: "Path cost from GBFS: 34"
observer: "Time cost: 0.066 milliseconds"
```



Commenting

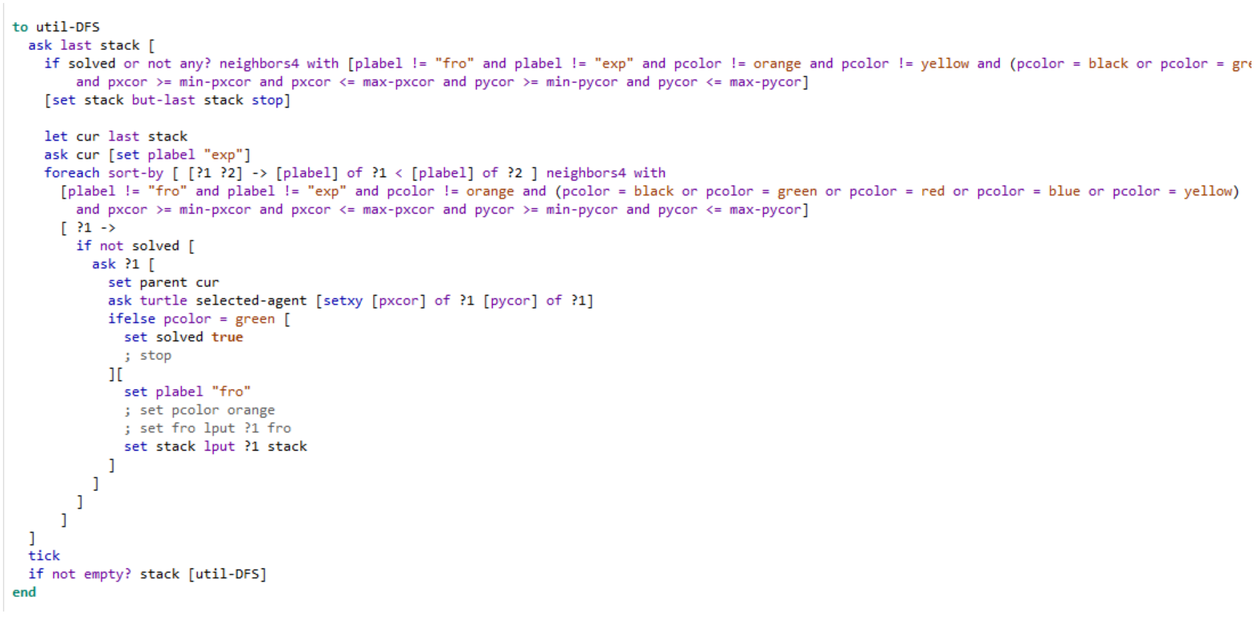
- With the f-value (a combined value of the cost function and the heuristic function), A* will most likely never fail to give the most optimum result, or no result at all.
- A* speeds also improved considerably now that there are many more traps in the environment.
- GBFS will keep getting trapped, so it is rendered useless regardless of how fast it processes the nodes.
- Only UCS and A* are helpful in this obstacle-filled environment, the rests are based on chance mostly.

4. Algorithms implementation

The sub-sections below provide a demo code in Netlogo.

4.1. Depth-first search

In this algorithm, the turtle will try to expand the path in some specific direction until it can no longer expand it, it will backtrack to the previous patch and try another path. It does not guarantee that we can find an optimal solution. However, if at least one solution exists it could always find it.



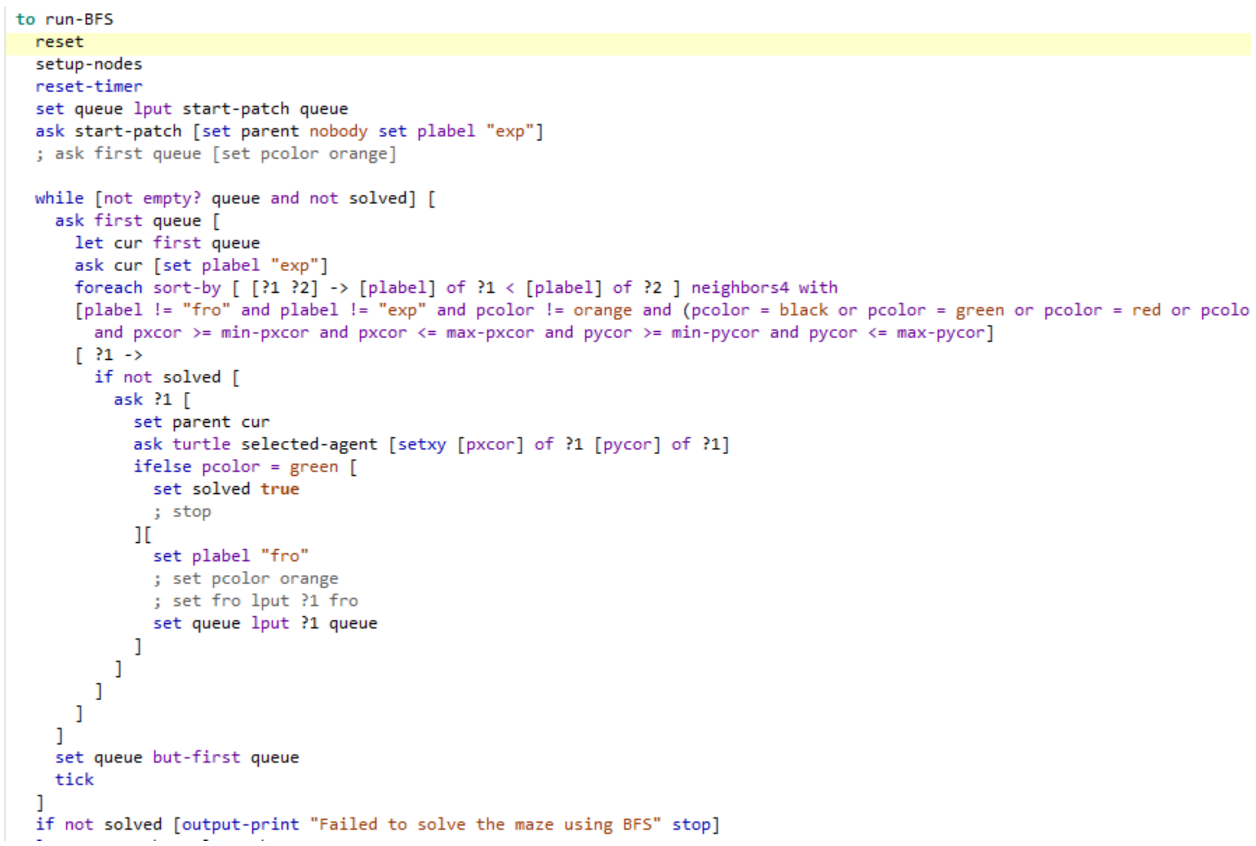
```
to util-DFS
  ask last stack [
    if solved or not any? neighbors4 with [plabel != "fro" and plabel != "exp" and pcolor != orange and pcolor != yellow and (pcolor = black or pcolor = gre
        and pxcor >= min-pxcor and pxcor <= max-pxcor and pycor >= min-pycor and pycor <= max-pycor]
    [set stack but-last stack stop]

    let cur last stack
    ask cur [set plabel "exp"]
    foreach sort-by [ [?1 ?2] -> [plabel] of ?1 < [plabel] of ?2 ] neighbors4 with
      [plabel != "fro" and plabel != "exp" and pcolor != orange and (pcolor = black or pcolor = green or pcolor = red or pcolor = blue or pcolor = yellow)
        and pxcor >= min-pxcor and pxcor <= max-pxcor and pycor >= min-pycor and pycor <= max-pycor]
      [ ?1 ->
        if not solved [
          ask ?1 [
            set parent cur
            ask turtle selected-agent [setxy [pxcor] of ?1 [pycor] of ?1]
            ifelse pcolor = green [
              set solved true
              ; stop
            ][
              set plabel "fro"
              ; set pcolor orange
              ; set fro lput ?1 fro
              set stack lput ?1 stack
            ]
          ]
        ]
      ]
  ]
  tick
  if not empty? stack [util-DFS]
end
```

4.2. Breadth-first search

In this algorithm, the turtle measures the priority of a node in terms of the number of edges to reach that node. And based on that priority, the turtle will look at the patches right next to the current patch first, and then the patches right next to all the patches that the turtle has looked at, and so on.

This algorithm is guaranteed to find the optimal solution if all the paths are of uniform cost. However, with weighted paths, the algorithm fails to find the optimal solution in some situations.



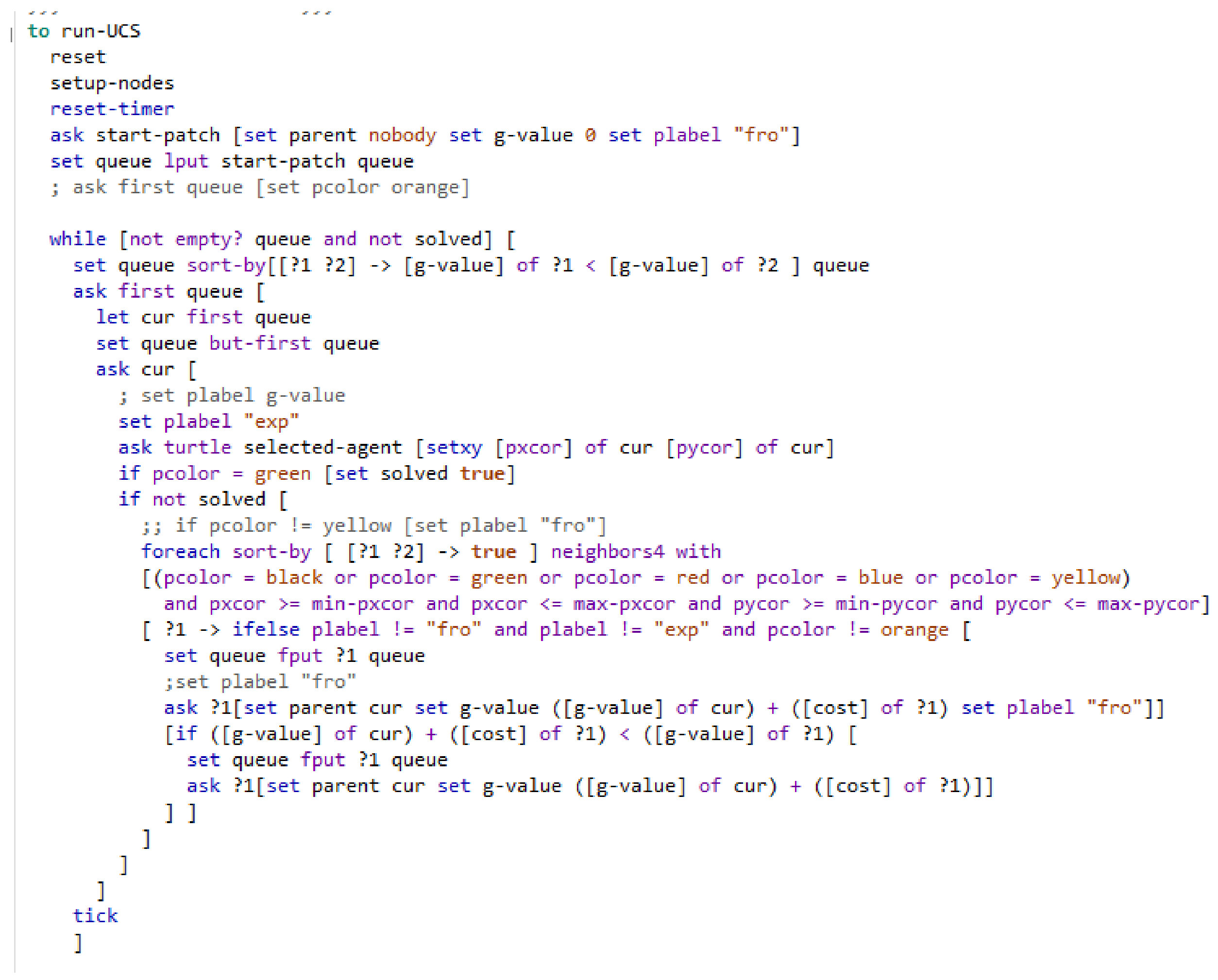
```
to run-BFS
  reset
  setup-nodes
  reset-timer
  set queue lput start-patch queue
  ask start-patch [set parent nobody set plabel "exp"]
  ; ask first queue [set pcolor orange]

  while [not empty? queue and not solved] [
    ask first queue [
      let cur first queue
      ask cur [set plabel "exp"]
      foreach sort-by [ [?1 ?2] -> [plabel] of ?1 < [plabel] of ?2 ] neighbors4 with
      [plabel != "fro" and plabel != "exp" and pcolor != orange and (pcolor = black or pcolor = green or pcolor = red or pcolo
        and pxcor >= min-pxcor and pxcor <= max-pxcor and pycor >= min-pycor and pycor <= max-pycor]
      [ ?1 ->
        if not solved [
          ask ?1 [
            set parent cur
            ask turtle selected-agent [setxy [pxcor] of ?1 [pycor] of ?1]
            ifelse pcolor = green [
              set solved true
              ; stop
            ][
              set plabel "fro"
              ; set pcolor orange
              ; set fro lput ?1 fro
              set queue lput ?1 queue
            ]
          ]
        ]
      ]
    ]
    set queue but-first queue
    tick
  ]
  if not solved [output-print "Failed to solve the maze using BFS" stop]
```

4.3. Uniform-cost search

This algorithm uses the weight of the patches to determine which patch to expand next. The patch with the lowest current distance from the starting position is always expanded first. The algorithm runs until the goal patch is expanded or there is no more patch to reach.

It could always find the optimal solution if it exists. If all patches are of uniform cost, then the algorithm behaves as if it is a breadth-first search algorithm.

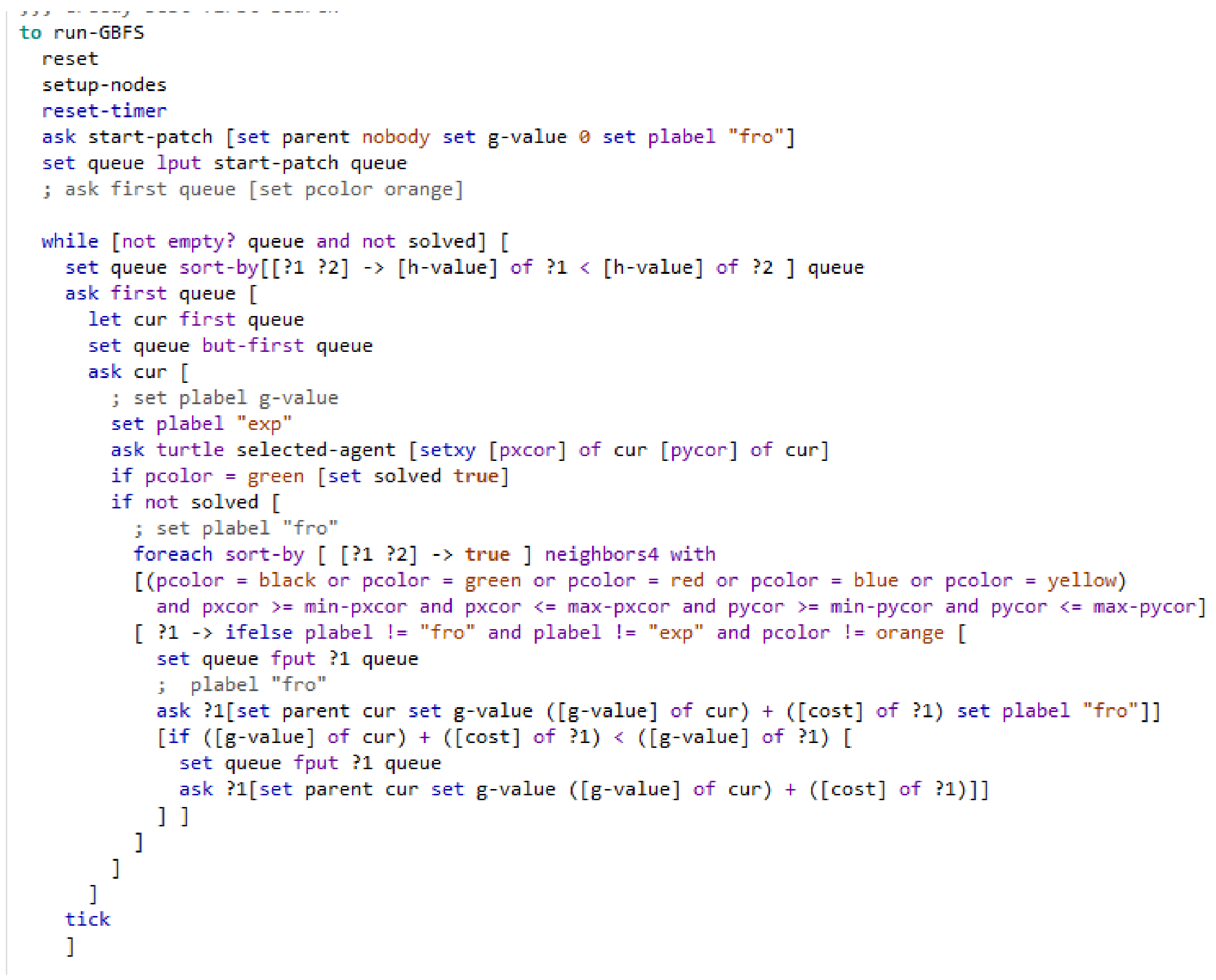


```
to run-UCS
  reset
  setup-nodes
  reset-timer
  ask start-patch [set parent nobody set g-value 0 set plabel "fro"]
  set queue lput start-patch queue
  ; ask first queue [set pcolor orange]

  while [not empty? queue and not solved] [
    set queue sort-by[[?1 ?2] -> [g-value] of ?1 < [g-value] of ?2 ] queue
    ask first queue [
      let cur first queue
      set queue but-first queue
      ask cur [
        ; set plabel g-value
        set plabel "exp"
        ask turtle selected-agent [setxy [pxcor] of cur [pycor] of cur]
        if pcolor = green [set solved true]
        if not solved [
          ;; if pcolor != yellow [set plabel "fro"]
          foreach sort-by [ [?1 ?2] -> true ] neighbors4 with
          [(pcolor = black or pcolor = green or pcolor = red or pcolor = blue or pcolor = yellow)
            and pxcor >= min-pxcor and pxcor <= max-pxcor and pycor >= min-pycor and pycor <= max-pycor]
          [ ?1 -> ifelse plabel != "fro" and plabel != "exp" and pcolor != orange [
            set queue fput ?1 queue
            ;set plabel "fro"
            ask ?1[set parent cur set g-value ([g-value] of cur) + ([cost] of ?1) set plabel "fro"]]
            [if ([g-value] of cur) + ([cost] of ?1) < ([g-value] of ?1) [
              set queue fput ?1 queue
              ask ?1[set parent cur set g-value ([g-value] of cur) + ([cost] of ?1)]]
            ] ]
          ]
        ]
      ]
    tick
    ]
```

4.4. Greedy best-first search

This algorithm tries to expand the patches with the best value of the heuristic function until it reaches the goal or cannot expand any paths anymore. In our code, we use the Manhattan distance from the current patch to the goal as our heuristic.



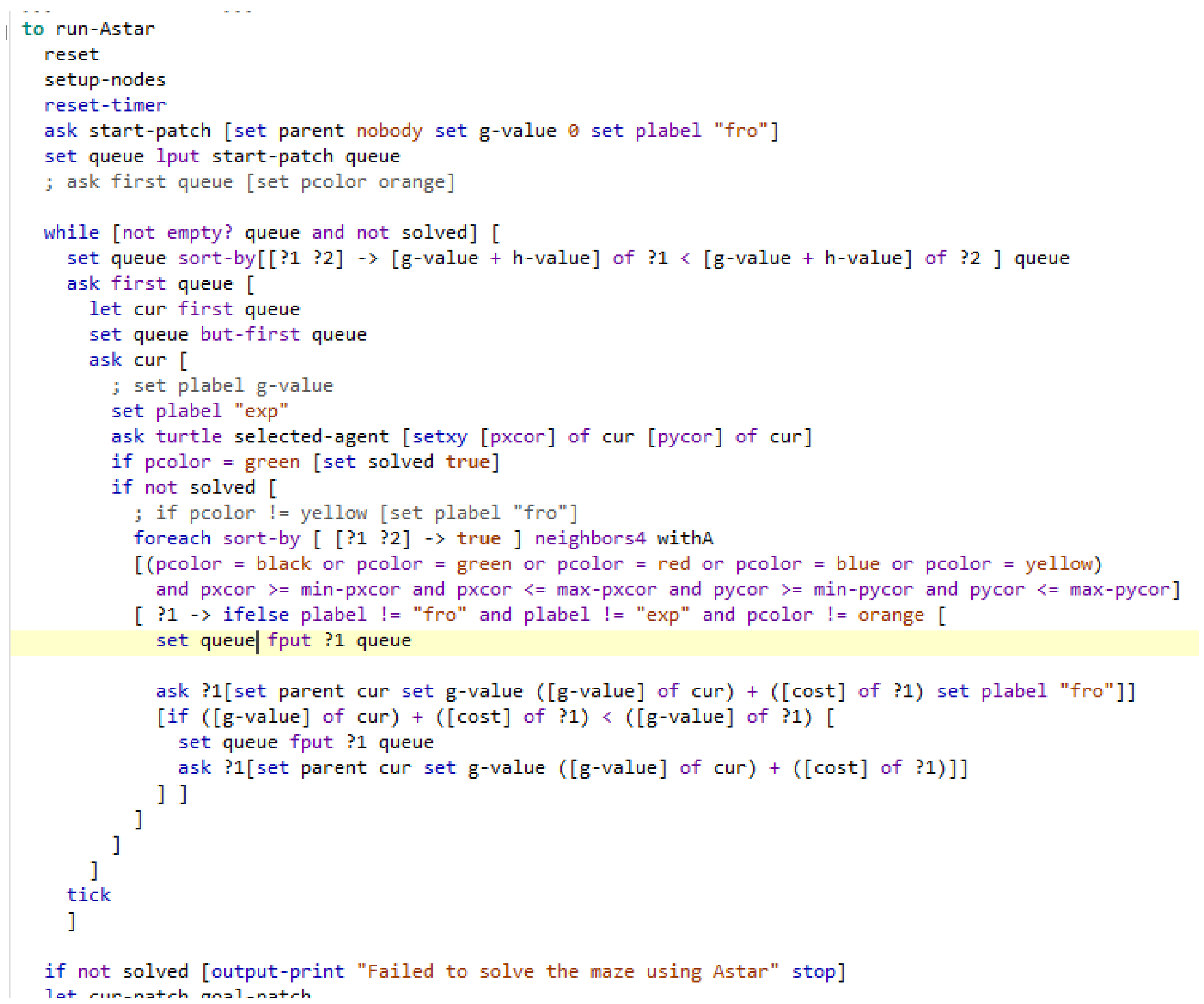
```
to run-GBFS
  reset
  setup-nodes
  reset-timer
  ask start-patch [set parent nobody set g-value 0 set plabel "fro"]
  set queue lput start-patch queue
  ; ask first queue [set pcolor orange]

  while [not empty? queue and not solved] [
    set queue sort-by[[?1 ?2] -> [h-value] of ?1 < [h-value] of ?2 ] queue
    ask first queue [
      let cur first queue
      set queue but-first queue
      ask cur [
        ; set plabel g-value
        set plabel "exp"
        ask turtle selected-agent [setxy [pxcor] of cur [pycor] of cur]
        if pcolor = green [set solved true]
        if not solved [
          ; set plabel "fro"
          foreach sort-by [ [?1 ?2] -> true ] neighbors4 with
          [(pcolor = black or pcolor = green or pcolor = red or pcolor = blue or pcolor = yellow)
            and pxcor >= min-pxcor and pxcor <= max-pxcor and pycor >= min-pycor and pycor <= max-pycor]
          [ ?1 -> ifelse plabel != "fro" and plabel != "exp" and pcolor != orange [
            set queue fput ?1 queue
            ;  plabel "fro"
            ask ?1[set parent cur set g-value ([g-value] of cur) + ([cost] of ?1) set plabel "fro"]]
            [if ([g-value] of cur) + ([cost] of ?1) < ([g-value] of ?1) [
              set queue fput ?1 queue
              ask ?1[set parent cur set g-value ([g-value] of cur) + ([cost] of ?1)]]
            ] ]
          ]
        ]
      ]
    tick
    ]
```

4.5. A* search

This algorithm uses an evaluation function to judge which patch to expand next. This evaluation function is the sum of the currently found distance to the patch and the value of the heuristic function of that patch. The heuristic function, again, is calculated as the Manhattan distance from the current patch to the goal patch.

It could be proved that the Manhattan distance in this search problem is a consistent heuristic, and thus the A star search algorithm could always find the optimal solution.



```
to run-Astar
  reset
  setup-nodes
  reset-timer
  ask start-patch [set parent nobody set g-value 0 set plabel "fro"]
  set queue lput start-patch queue
  ; ask first queue [set pcolor orange]

  while [not empty? queue and not solved] [
    set queue sort-by[[?1 ?2] -> [g-value + h-value] of ?1 < [g-value + h-value] of ?2 ] queue
    ask first queue [
      let cur first queue
      set queue but-first queue
      ask cur [
        ; set plabel g-value
        set plabel "exp"
        ask turtle selected-agent [setxy [pxcor] of cur [pycor] of cur]
        if pcolor = green [set solved true]
        if not solved [
          ; if pcolor != yellow [set plabel "fro"]
          foreach sort-by [ [?1 ?2] -> true ] neighbors4 withA
          [(pcolor = black or pcolor = green or pcolor = red or pcolor = blue or pcolor = yellow)
            and pxcor >= min-pxcor and pxcor <= max-pxcor and pycor >= min-pycor and pycor <= max-pycor]
          [ ?1 -> ifelse plabel != "fro" and plabel != "exp" and pcolor != orange [
            set queue fput ?1 queue

            ask ?1[set parent cur set g-value ([g-value] of cur) + ([cost] of ?1) set plabel "fro"]]
            [if ([g-value] of cur) + ([cost] of ?1) < ([g-value] of ?1) [
              set queue fput ?1 queue
              ask ?1[set parent cur set g-value ([g-value] of cur) + ([cost] of ?1)]]
            ] ]
          ]
        ]
      ]
    tick
    ]

  if not solved [output-print "Failed to solve the maze using Astar" stop]
  let cur-patch goal-patch
```

5. Self-assessment

No.	Criteria	Scores
1	Requirement 1	20%
2	Requirement 2: Level 1	15%
3	Requirement 2: Level 2	15%
4	Requirement 2: Level 3	15%
5	Generate at least 5 test cases for each level with different attributes. Describe them in the experiment section of your report. Videos to demonstrate.	15%
6	Report your algorithms, experiments with some reflection or comments.	20%
Total		100%

6. References

Graph Search DFS and BFS in NetLogo
BAM Maze Generator 2 in NetLogo