Report for CS440 Assignment 1

Group member: Xuan Wang (xwang182), Shuo Feng(sfeng15), Yuchen Qian(yqian14)

Part 1

In the part 1.1, we implemented DFS, BFS, Greedy best-first search and A\* algorithm.

We created our own data structure to store the original graph. Each symbol is a corresponding node in our new structure. Then, we run these algorithms to find the optimal path for each maze.

Pseudo code:

DFS:

stack s = {}

push the start node into the stack

while true

node v = stack.top()

stack.pop()

if v is the end node: break

if v is not visited: mark it as visited

for each unvisited neighbor w of v

set the v to be the previous node of w

push w into S

end

end

cost:

open maze : 372

medium maze: 50

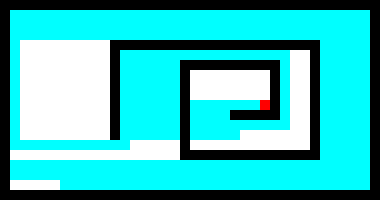
big maze : 94

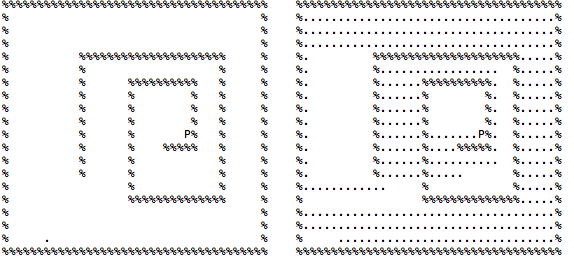
expanded nodes:

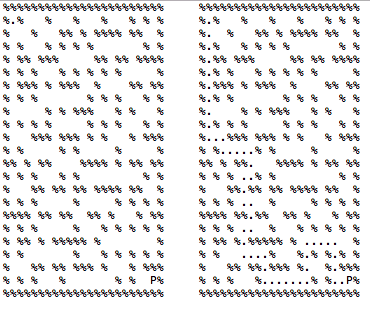
open maze: 750

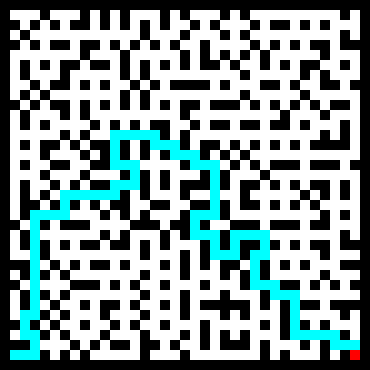
medium maze: 87

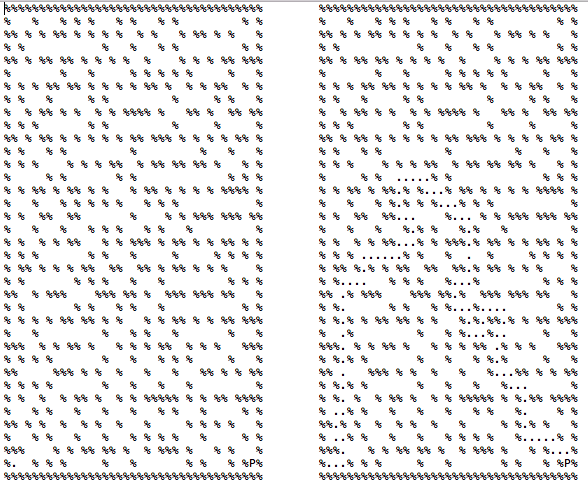
big maze: 860











BFS

queue q = {}

add the start node v into the queue

while q is not empty

v = q.dequeue()

for each node n that is adjacent to v

if n is not out of bound and does not hit the wall

set the v to be the previous node of u

q.enqueue(n);

cost:

open maze : 54

medium maze: 42

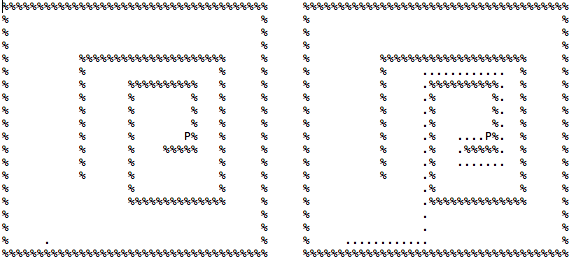
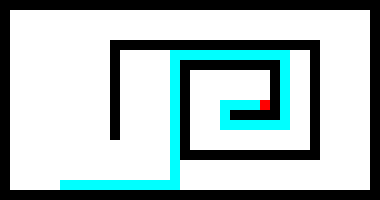
big maze : 62

expanded nodes:

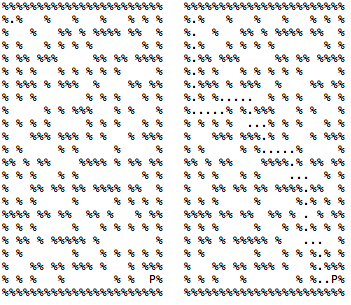
open maze: 301

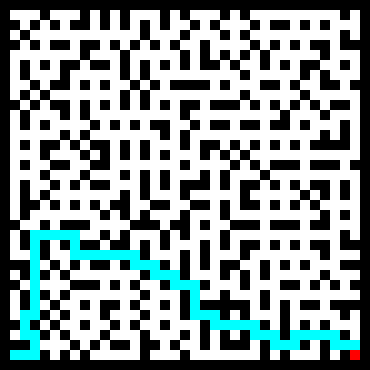
medium maze: 228

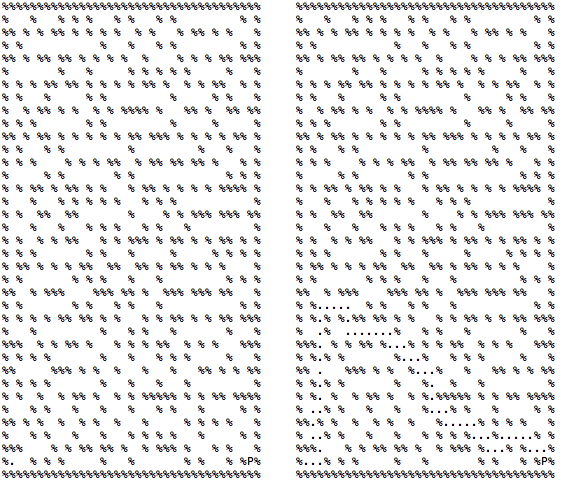
big maze: 737











Greedy best-first search:

(smallest heuristic = smallest manhattan distance)

Priorityqueue pq = {}

Pq.enqueue(startnode v)

While pq is not empty

c = pq.getSmallestHeursticNode

if c is the goal

break

else

foreach neighbor n of c

if n is unvisited

mark n visited

pq.enqueue(n)

cost:

open maze : 58

medium maze: 56

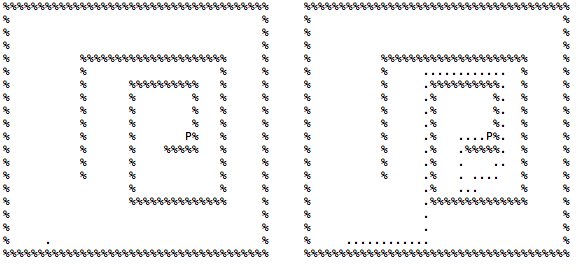
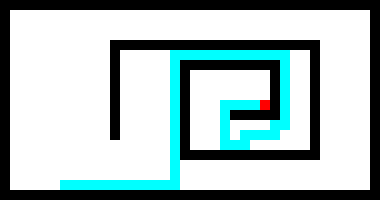
big maze : 76

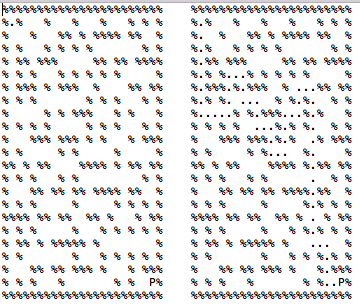
expanded nodes:

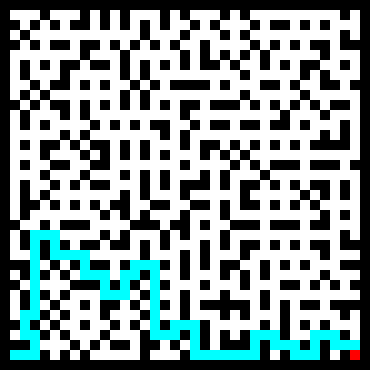
open maze: 152

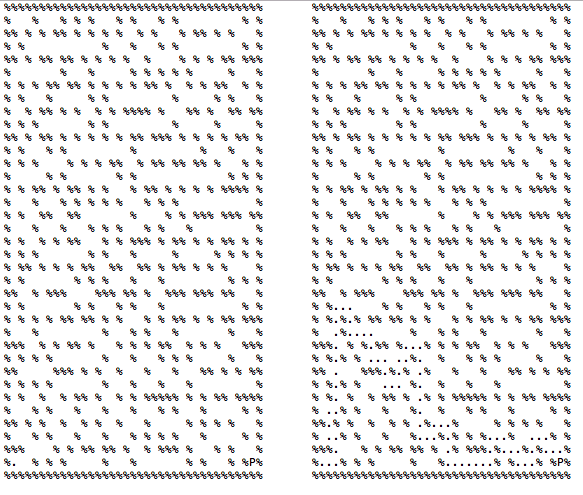
medium maze: 101

big maze: 159









A\*:

(smallest heuristic = smallest manhattan distance + the cost)

Priorityqueue pq = {}

Pq.enqueue(startnode v)

While pq is not empty

c = pq.getSmallestHeursticNode

if c is the goal

break

else

foreach neighbor n of c

if n is unvisited

mark n visited

pq.enqueue(n)

cost:

open maze : 54

medium maze: 42

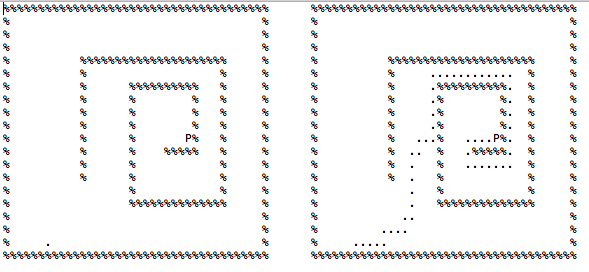
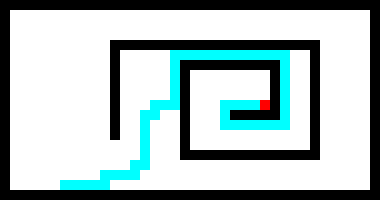
big maze : 64

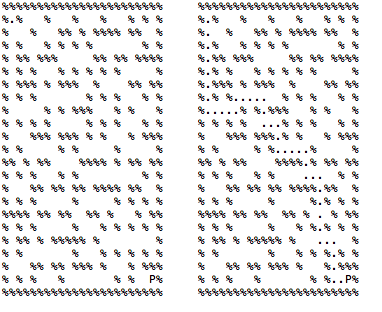
expanded nodes:

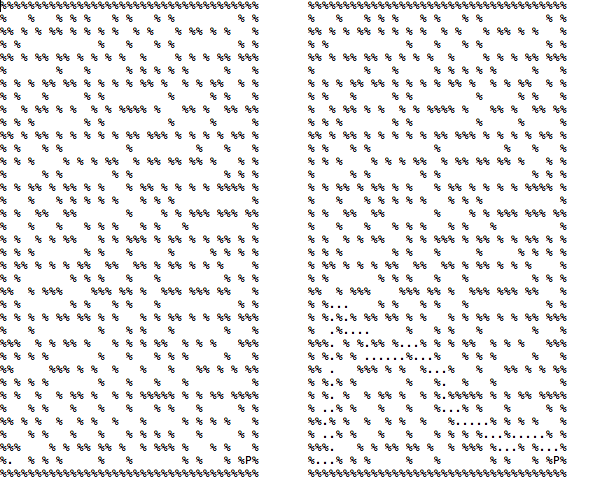
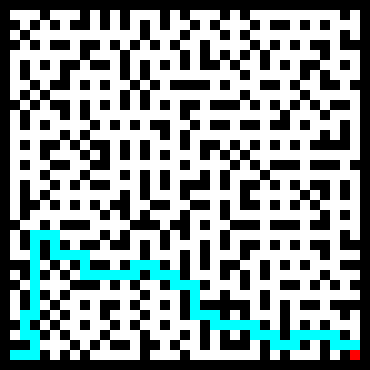
open maze: 205

medium maze: 120

big maze: 302







One of the interesting things behind our implement is that we overloaded the “<” operator in order to implement our priority queue. We set the highest priority node to be the one which has the lowest heuristic value. Also, we keep track of the path based on remembering the previous node of the current node. Therefore, we find the path from the end of the node and keep tracking its previous node.

Part 1.2

In this part, we modified our operator overload function in order to change the cost of moving.

forward movement has cost 2 and any turn has cost 1;

    bool operator<(const node& right) const {

     return (this->dist\*2+this->cost > right.dist\*2 + right.cost);

}

Down in the algorithm, we keep track our direction and at first, we add all cost by 3.

Whenever we find a node moves within it’s previous direction, we reduce the cost by 1.

Otherwise we don’t subtract the cost.

cost:

small turn: 120

big turn: 133

expanded nodes:

small turn: 150

big turn: 320

forward movement has cost 1 and any turn has cost 2;

     bool operator<(const node& right) const {

      return (this->dist+this->cost > right.dist + right.cost);

 }

Down in the algorithm, we keep track our direction and at first, we add all cost by 3.

Whenever we find a node moves within it’s previous direction, we reduce the cost by 2.

Otherwise we don’t subtract the cost.

cost:

small turn: 84

big turn: 92

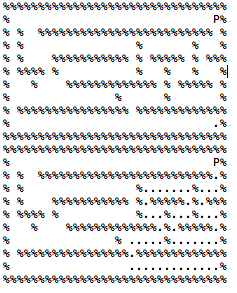
expanded nodes:

small turn: 162

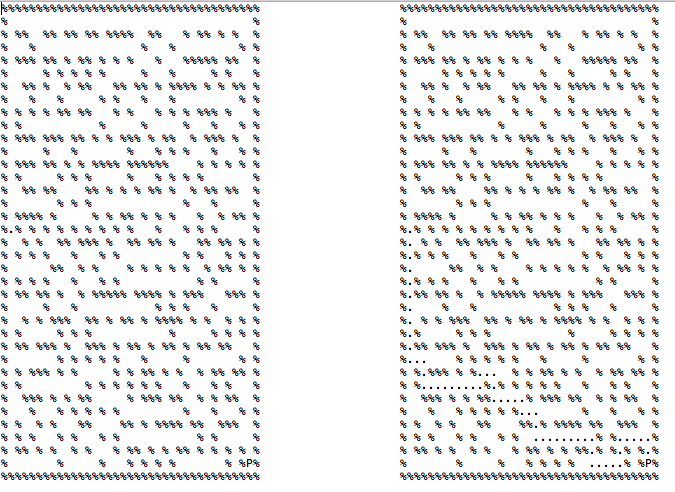
big turn: 366

Here is our graph as well as the output in the terminal

1.2.1 (forward 2 turn 1)small maze:

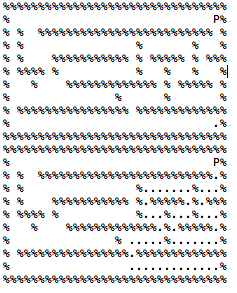


big maze:

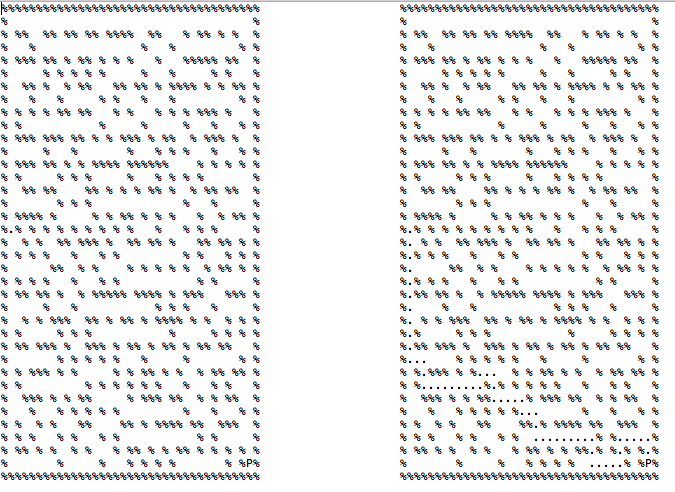




1.2.2 (forward 1 turn 2)small maze:



big maze:





The interesting aspects behind there is that we treat the cost equally and then subtract by their direction. This is greatly different than the normal way to add the turn cost.

When we calculate the cost, we check how many turns did we make in order to update our total cost by adding the number of turns (times 1 or times 2 depends on the cost of a turn)