Project Report

Search

Members:

1. Student Name – Student Code – Grade

2. ...

3. …

(Grade là điểm các thành viên trong nhóm tự đánh giá lẫn nhau dựa trên mức độ đóng góp, theo thang điểm 0-10)

**Question 1: Depth First Search**

* Idea:

Return list actions to goal

Use data structure in util.py for autograde

Depth first search use stack data structure

State: coordinate (x, y)

Stack push, pop (self, item), argument item

getSuccessor return list (successor, action, stepCost), so node should contain item(state, list actions)

* pseudo code:

Push (start state, empty list) to stack

While stack not empty

pop item get state, actions

if state visited

continue to next item in stack

mark state visited

if current item is goal

return list actions

for successor in item’s successors

if successor not in visited

push item(successor, actions to successor) to stack

* Result: 3/3 tests

**Question 2: Breadth First Search**

* Idea: Similar to question 1, but using queue for breadth first search, queue push and pop similar to stack push pop
* pseudo code:

Push (start state, empty list) to queue

While queue not empty

pop item get state, actions

if state visited

continue to next item in queue

mark state visited

if current item is goal

return list actions

for successor in item’s successors

if successor not in visited

push item(successor, actions to successor) to queue

* Result: 3/3 tests

**Question 3:** uniform-cost graph search

* Idea: Similar to Q1 and Q2, but use priority queue to prioritize smallest cost

PriorityQueue push(self, item, priority). argument: item, priority

getSuccessor return list (successor, action, stepCost), a node should contain item(state, list actions), cost

* pseudo code:

Push ((start state, empty list), cost=0) to priorityqueue

While priorityqueue not empty

pop node get state, actions,cost

if state visited

continue to next item in priorityqueue

mark state visited

if current item is goal

return list actions

for successor in item’s successors

if successor not in visited

push item(successor, actions to successor, cost to successor) to priorityqueue

* Result: 3/3 tests

**Question 4: A\* search**

* Idea: Using evaluation function f(n) = g(n) + h(n)
  + g(n) is the cost from initial node to node n
  + h(n) is estimated cost of cheapest path from n to goal
  + **f(n) = h(n) + g(n) is the value that using as the priority of PriorityQueue**
* pseudo code:
  + Init open\_set is PriorityQueue, close\_set = []
  + close\_set.add(start\_node)
  + while !goal:
    - get list successor
    - for successor in list successor
      * set visited = false
      * calculate g(successor)
      * for successor in close\_set:
        + if(higher cost than previos) set visited = True
      * if (visited = false)
        + open\_set.push(successor (with f(successor))
        + close\_set.push(successor (with g(successor))
    - open\_set.pop()
  + return action( direction )

**Result will give a list of action to goal**!

* Result: 3/3 tests

**Question 5: Finding All the Corners**

* Idea: function getSuccessors return list successors (state, action, 1)

state[0] contain coordinate

state[1] contain visited corner

state = (coordinate, visitedcorner)

getSuccessors check 4 direction next 1 move and check if hit wall, if hit corner or not handle and append valid successor to successors

function \_\_init\_\_ dont need edit

function getstartstate return start state

function isgoalstate check if this state is unvisited corner add to visited corner and check if all 4 corner visited or not return true else false

* pseudo code:

function getstartstate(self)

return (start position, empty list for visited corner)

function isgoalstate(self)

get position and visitedcorner from state

if position is corner

if position not in visitedcorner

add position to visitedcorner

return true if visitedcorner=4 else false

else return false

function getsuccessors

for action in 4 direction

nextnode = current coordinate + action coordinate

if not hit wall

successorvisitedcorner = visitedcorner

if nextnode is corner

if next node not in successorvisitedcorner

add nextnode to successorvisitedcorner

successor = ((nextnode, successorvisitedcorner), action, 1)

add successor to successors

* Result: 3/3 tests

**Question 6: Corners Problem: Heuristic**

* Idea: heuristic use manhattandistance from current position to corner, prioritize closest corner

manhattandistance in util.py: manhattanDistance( xy1, xy2 )

function cornersheuristic return number < shortest path

* pseudo code:

check 4 corner

if corner[i] not in visitedcorner

add corner[i] to unvisitedcorner

while exist unvisitedcorner

check manhattandistance from currentposition to all unvisitedcorner and select min of them

heuristicsum += minmanhattandistance

currentposition = corner

remove corner from unvisited corner

return heuristicsum

* Result: 3/3 tests

**Question 7: Eating All The Dots**

* Idea:
* pseudo code:
* Result: 0/3 tests

**Question 8: Suboptimal Search**

* Idea:
* pseudo code:
* Result: 0/3 tests