Name: Wenhao Li ID:#14328112

Introduction:

Every character in this report and code is written ONLY by myself. The programming language I am using is C++. The code is run and tested on my own computer(Windows x64 platform). I only used standard C++ library, so the code should be able to compile on any platform.

How to compile:

The code directory is as follows

./HW2.sln …Visual studio 2019 solution file

./HW2 …The source codes (Timer.h Map.h Map.cpp main.cpp)

./HW2/Debug …The executable file

If you are on Windows and have Visual studio, open the .sln solution file and hit ctrl+f5 to run or run the executable directly. If you are on Unix, use g++ to compile, no extra flags are needed.

Algorithm description:

1. Cost Function:

As given in the homework description:

+4 for cleaned up

-1 for move ←

-1.1 for move →

-1.2 for move ↑

-1.3 for move ↓

-0.2 for suck

0 for NoOp

The smallest cost is the one with highest performance points.

1. Uniform cost tree search:

We construct a fringe as a normal array. First, expand the initial state and push the result into the fringe. And we enter in a loop. Each time, only the smallest cost node in the fringe is expanded and push the expanded nodes into the fringe. The expanded nodes are removed from the fringe and pushed into the result. Each expansion is counted as 1 more depth. If the fringe is empty before expansion, the search will ended as failure.

1. Uniform cost graph search:

Similar to uniform cost tree search stated above. The difference is that we created a new array called ‘visited’. And every time we pick a node in the fringe, we first compare it with all visited nodes and check whether the node has been visited before. If it is, delete it from the fringe to avoid revisiting.

1. Depth-limited depth-first tree search:

We construct the fringe as a stack (LIFO). First, expand the initial state and push the one of the resulting nodes randomly into the fringe. And we enter in a loop. Each time, we expand the node at the top of the fringe, choose one of the resulting nodes randomly (to break the tie of the nodes at the same depth) and push it into the fringe. Each expansion is counted as 1 depth. If no more expansion is possible, we pop off this node from the fringe.

1. Depth-limited depth-first tree

Similar to depth-limited depth-first tree stated above, the only difference is we created a new array called ‘visited’. Each time, we choose the node at the top of the fringe and we compare the resulting expanded nodes with all visited nodes to check whether the node has been visited before. If it is, delete it from the resulting nodes to avoid revisiting.

Result: