Project 1

Justin Pham

SID 862114467

Email [jpham079@ucr.edu](mailto:jpham079@ucr.edu)

Date Nov 5 2021

To complete this assignment, I consulted:

* Blind Search and Heuristic Search slides
* Notes take during lecture
* C++ documentation
  + <https://www.cplusplus.com>
* The following YouTube video to understand what counts as one “move”
  + <https://www.youtube.com/watch?v=o4ZDw9oFlP8>
* My fellow students for idea
  + No code was exchanged. No student had access to another student’s work in any way. We discussed possible heuristics and strategies to solve the problem.

All code is original except for subroutines for **vector**, **priority\_queue, unordered\_set**, and **string** functions

* **vector** and **priority\_queue** is used to handle queuing nodes
* **unordered\_set** is used to hash previous nodes, to keep a history of sorts
* **string** is used to represent the nodes.
  + subroutines **compare** and **find** and used to compare and find elements inside the string

Outline:

* Cover Page (this page)
* My report (pages 2 - 6)
* Sample trace on an easy problem (page 7)
* Sample trace on a hard problem (page 8)
* My Code (page 9 - 16)
  + Github Link: <https://github.com/Pjsrcool/CS-170-Project-1>

CS 170: Project 1 Extra Credit: Nine Men in a Trench

Justin Pham, SID: 862114467, Nov 8 2021

**Introduction**

Search algorithms can be very useful on puzzles with a grid. Sliding puzzles such as Nine Men in a Trench are a great example. Not only can we use a search algorithm to traverse the puzzle board, we can use search to solve the puzzle itself. The premise of Nine Men in a Trench, is to move the sergeant, represented as the number 1, to the left hand side of the trench. There are recesses and other men in the trench, meaning we need to move men into and out of the trench to relocate the sergeant to the left hand side. The puzzle is solved when the sergeant is on the left hand side and the other men are in their proper positions.

This report details my findings from completing Project 1 of Professor Eamonn Keogh’s Introduction to AI course at UC Riverside, Fall 2021. The following heuristics will be explored: no heuristic, misplaced tiles, Manhattan on the sergeant, obstructing men, left man. The code is written in C++.

**The Heuristics**

The heuristics are: no heuristic, misplaced tiles, Manhattan distance on the sergeant, obstructing men, left man.

For clarity, cost, weight and f(n) will mean the same thing. f(n) = g(n) + h(n)

g(n) will represent the depth of the node

h(n) will represent he the heuristic

In the example boards, ‘ - ‘ will represent an inaccessible space, ‘0’ will represent an open space, ‘1’ will represent the sergeant, and numbers will represent men.

No Heuristic

By using no heuristic, essentially perform a Uniform Cost Search. When performing a Uniform Cost Search with my code, the heuristic function will simply return h(n) = 0 each time a heuristic is requested. Since h(n) = 0, we do not look into the future, but only the past. We essentially prioritize searching nodes with the lowest depth, a Breadth First Search. The following heuristics will look at both the past (g(n)) and the future (h(n)).

Misplaced Tiles

This the probably the best heuristic we have. For Misplaced Tiles, we simply count the number of men who are not in their proper positions. It ignores empty space and inaccessible space. The heuristic function will return h(n) = {number of men who are not in the correct position}. For example, the following board have a h(n) = 3 because the sergeant, man 2, and man 3 are not in the proper positions.

[-, -,0,-,1,-,-]

[3,2,0,4,0,0]

Manhattan on the Sergeant

The Manhattan distance is the distance we travel by moving vertically or horizontally on a grid. For this heuristic, we use the Manhattan distance between the sergeant and the his goal position. The fact that we use distance means that the Manhattan on the Sergeant heuristic is not admissible. For example, if we have the following board:

[-, -,0,-,0,-,-]

[0,0,0,0,0,1]

This board is solvable in one move: move the Sergeant straight to the left. But heuristic returns h(n) = 5. Since, h(n) is higher than the actual solution, it not admissible. As a result, the use of Manhattan on the Sergeant heuristic will often return a solution with a depth 1 or 2 higher than the depth of the optimal solution.

Obstructing Men

The Obstructing Men heuristic is similar to the Misplaced Tile heuristic in that it looks at men who are no in the proper position. In the Obstructing Men heuristic, we simply count the number of men between the sergeant and his destination. We return that count as our heuristic. The intuition is that we much move each obstructing man to make way for the sergeant. Each man we move is one cost. Technically, we should multiply the by 2, since we need to move the obstructing men back, but it would be pointless because it does not improve the search in any way. Consider the following for as an example:

[-,-,-,0,-,1,-,-]

[0,3,0,5,0,2,0,4]

Here, man 2, man 3 and man 5 are between the sergeant and his destination. That is 3 men so h(n) = 3.

Left Man

The Left Man heuristic works by checking each man that is in the proper position. If a man is the proper position, check if the man on his left is in the proper position. If he is not, then we increment the heuristic value. The main weakness of this heuristic is that it relies on men being in the proper position. If no man is in the proper position, then h(n) = 0. Since this happens more frequently than not, Left Man is only marginally faster then No Heuristic, Uniform Cost Search. To illustrate an example, we have the following board:

[-,-,-,1,-,0,-,-,]

[4,2,5,3,0,0,0]

Only man 2 is in the correct position, and the man to his left (man 4) is in the wrong position. So h(n) = 1.

**Performance**

Below are charts comparing the performance of each heuristic. For comparison of each heuristic, we will be using the following 2 test cases:

Easy Case:

[-,-,0,-,0,-,-]

[2,3,4,1,0,0,0]

Hard Case:

[-,-,-,0,-,0,-,0,-,-]

[0,2,3,4,5,6,7,8,9,1]

Figure 1

Figure 2

As we can see in Figure 1 and Figure 2, the number of nodes expanded is much lowest on Misplaced Tile heuristic. It is, therefore, the most space efficient heuristic by far. For the Easy Case, the Left Man heuristic performs the same as No Heuristic so it is basically pointless. But for the Hard Case, both take too long to run, and I run out of memory to complete the runs. Seeing that on the Easy Case, Misplaced Tile expanded about 4 times less nodes than No Heuristic and Left Man, we can guess that the Hard Cases’s No Heuristic and Left Man would expand at least 7100000. But knowing that the branching factor is not linear, the number of nodes expanded would realistically be O(d^b) = ~28^4 or as high as ~28^8 nodes.

Figure 3

Figure 4

Now looking at the execution time shown in Figure 3 and Figure 4, the Easy Case further shows how pointless the Left Man heuristic is; both have similar execution times. We also see that the Misplaced Tiles Heuristic is much faster due to needing to expand less nodes.

As for the Manhattan Distance, recall that the it is not admissible. Here we prove that it can still be fast despite the fact that the Manhattan Distance on the Sergeant heuristic does not provide an optimal solution.

One thing to note, these tests are performed on a laptop at different times of the day. So the number of nodes expanded will stay the same, but the execution times will fluctuate. The numbers in Figure 3 and Figure 4 are acquired by selecting the median of the fluctuations.

**Conclusion**

Here is a summary of my findings relating to the 5 searches I investigated:

* Using No Heuristic is extremely slow and a memory intensive. The Left Man heuristic often return h(n) = 0, meaning that it provides negligible improvement over not using any heuristic.
* The Misplaced Tile heuristic and the Obstructing Men heuristic are very similar in approach, but the Misplaced heuristic performs better. This is because the Misplaced Tile heuristic looks at ALL misplaced men, while the Obstructing Men heuristic only looks at men that are between the sergeant and his destination. In other words, the Misplaced Tile heuristic considers the entire board while the Obstructing Men heuristic considers a portion of the board. This makes the Misplaced Tile heuristic much more reliable.
* Any distance heuristic is useless in Nine Men in a Trench puzzle. This is because the puzzle’s search tree is weighted by number of moves, not by tile distance. As a result, using the Manhattan Distance on the Sergeant heuristic will often overestimate

\*Note: the code is run with the following command:

./solve <top row of problem state> <bottom row of problem state> <top row of goal state> <bottom row of goal state>

**Sample Trace of an Easy Problem**

./solve "--0-0--" "2341000" "--0-0--" "1234000"

Welcome to 9 Men in a Trench solver

Select the your prefered search type:

1. Uniform Cost Search

2. Misplaced Tiles Heuristic

3. Manhattan Heuristic with Sergeant

4. Count Obstructing Men Heuristic

5. Check Left Man of each man in correct place

2

node size: 1

--0-0--

2341000 -->COST: 0 -->Depth: 0

node size: 7

--0-0--

2341000 -->COST: 5 -->Depth: 1

node size: 6

--4-0--

2301000 -->COST: 5 -->Depth: 1

node size: 15

--0-1--

2340000 -->COST: 5 -->Depth: 1

.

.

. // I removed 139 pages of the trace to save space

.

.

node size: 4294

--3-0--

0200401 -->COST: 8 -->Depth: 5

node size: 4293

--0-0--

0032401 -->COST: 8 -->Depth: 5

node size: 4301

--0-0--

0230140 -->COST: 8 -->Depth: 6

node size: 4300

--0-0--

1234000 -->COST: 8 -->Depth: 8

nodes expanded: 542

---FINAL ANSWER---

--0-0--

1234000 -->COST: 8 -->Depth: 8

--> Run time: 0.109223 seconds

\*Note: the code is run with the following command:

./solve <top row of problem state> <bottom row of problem state> <top row of goal state> <bottom row of goal state>

**Sample Trace of an Hard Problem**

./solve "---0-0-0--" "0234567891" "---0-0-0--" "1234567890"

Welcome to 9 Men in a Trench solver

Select the your prefered search type:

1. Uniform Cost Search

2. Misplaced Tiles Heuristic

3. Manhattan Heuristic with Sergeant

4. Count Obstructing Men Heuristic

5. Check Left Man of each man in correct place

2

node size: 1

---0-0-0--

0234567891 -->COST: 0 -->Depth: 0

node size: 8

---0-0-0--

0234567891 -->COST: 2 -->Depth: 1

node size: 7

---0-0-0--

0234567891 -->COST: 2 -->Depth: 1

node size: 6

---0-0-0--

0234567891 -->COST: 2 -->Depth: 1

node size: 5

---0-0-0--

0234567891 -->COST: 2 -->Depth: 1

.

.

. // I removed

.

.

**Code Listing**

// link to my code is <https://github.com/Pjsrcool/CS-170-Project-1>

**header.h**

#ifndef HEADER\_H\_

#define HEADER\_H\_

#include <iostream>

#include <iomanip>

#include <queue>

#include <vector>

#include <queue>

#include <string>

#include <unordered\_set>

#include <chrono>

#include <ctime>

using std::vector;

using std::queue;

using std::priority\_queue;

using std::unordered\_set;

using std::string;

using std::cout;

using std::cin;

using std::endl;

using std::to\_string;

using std::chrono::time\_point;

using std::chrono::system\_clock;

using std::chrono::duration;

// we use an enum to decide which type of search to perform

enum SearchType {

UniformCost,

MisplacedTiles,

Manhattan\_On\_Sergeant,

Count\_Obstructing\_Men,

Check\_Left\_Man,

};

// this is how we will represent the node

struct Node {

// this is the current configuration of the board

// in this program, we will limit the board to 2 row

// state[0] will be the row with recesses

// state[1] will be either empty space, or men

// '-' --> inaccessable

// '0' --> empty space

// any int --> man

vector<string> state;

// current depth of the node

int depth;

// prev cost + heuristic

int cost;

// sets the configuration of the recess row and state row

void setState(string r, string s) {

state.clear();

state.push\_back(r);

state.push\_back(s);

}

// sets the depth of the node

void setDepth(long d) {

depth = d;

}

// sets the depth of the node

void setCost(long c) {

cost = c;

}

// prints the state of the node and its current depth

void print() {

cout << state[0] << endl;

cout << state[1] << " -->COST: " << cost << " -->Depth: " << depth << endl;

}

};

// Comparison object used to order

// Nodes that enter the queue,

// lowest cost on top of queue

class SmallerCost {

public:

bool operator() (Node l, Node r) {

return l.cost > r.cost;

}

};

// the generic search function

Node Search(Node initState, SearchType S, const Node& goalState);

#endif /\*HEADER\_H\*/

**Search.cpp**

#include "header.h"

unordered\_set<string> history; // prevents expansion of a previously visited node

unordered\_set<string> movement; // prevents circular movement when expanding a node

Node goal; // the goal state of the solution

// we make it global to reduce overhead

long nodesExpanded = 0;

// function to check if the node is a goal state

bool isGoalState(Node n) {

if (n.state[1].compare(goal.state[1]) == 0)

return true;

return false;

}

// function that returns the result of a heuristic calculation

int heuristic (SearchType S, Node& node) {

int h; // the heurisitc value

switch(S) {

// normal uniform cost search

case UniformCost:

h = 0;

break;

// Count and return number of misplaced tiles

case MisplacedTiles:

h = 0;

for (int i = 0; i < node.state[0].length(); ++i)

if (node.state[0][i] != '0' && goal.state[0][i] != node.state[0][i])

h++;

for (int i = 0; i < node.state[1].length(); ++i)

if (node.state[1][i] != '0' && goal.state[1][i] != node.state[1][i])

h++;

break;

// find the manhattan distance between

// the sergeant and position 0

case Manhattan\_On\_Sergeant:

{

h = 0;

int temp = node.state[0].find('1');

if (temp < node.state[0].length())

h = temp + 1;

else

h = node.state[1].find('1');

}

break;

// count the number of men between the sergeant

// and position 0

case Count\_Obstructing\_Men:

{

h = 0;

int sergeant = node.state[0].find('1');

if (sergeant >= node.state[0].length())

sergeant = node.state[1].find('1');

for (int i = 0; i < sergeant; ++i)

if (node.state[1][i] != '0')

h++;

}

break;

// check each man that is in the correct position

// if the man to his left is in the wrong position,

// increase the heuristic value

case Check\_Left\_Man:

h = 0;

for (int i = 0; i < node.state.size(); ++i)

for (int j = 1; j < node.state[i].length(); ++j)

if (node.state[i][j] != '-' && node.state[i][j] != '0' &&

goal.state[i][j] == node.state[i][j] && node.state[i][j-1] != goal.state[i][j-1])

h++;

break;

}

// return the heuristic value

return h;

}

// helper function for ExpandNodes()

// this function find alls possible moves a man at coordinate (i,j) can make using recursion

void ExpandNodeHelper(const int i, const int j, Node node, priority\_queue<Node, vector<Node>, SmallerCost> & children, SearchType & S) {

Node temp;

string r, s;

// move left

if (j > 0 && node.state[i][j-1] == '0') {

r = node.state[0];

s = node.state[1];

temp.setState(r,s);

temp.state[i][j-1] = temp.state[i][j];

temp.state[i][j] = '0';

temp.setDepth(node.depth);

temp.setCost (node.depth + heuristic(S, temp));

// temp.setCost (node.cost + heuristic(S, temp));

if (movement.insert(temp.state[0] + temp.state[1]).second) {

children.push(temp);

ExpandNodeHelper(i, j-1, temp, children, S);

}

}

// move right

if (j < node.state[i].length() - 1 && node.state[i][j+1] == '0') {

r = node.state[0];

s = node.state[1];

temp.setState(r,s);

temp.state[i][j+1] = temp.state[i][j];

temp.state[i][j] = '0';

temp.setDepth(node.depth);

temp.setCost (node.depth + heuristic(S, temp));

// temp.setCost (node.cost + heuristic(S, temp));

if (movement.insert(temp.state[0] + temp.state[1]).second) {

children.push(temp);

ExpandNodeHelper(i, j+1, temp, children, S);

}

}

// move up

if (i > 0 && node.state[i-1][j] == '0') {

r = node.state[0];

s = node.state[1];

r[j] = s[j];

s[j] = '0';

temp.setState(r,s);

temp.setDepth(node.depth);

temp.setCost (node.depth + heuristic(S, temp));

// temp.setCost (node.cost + heuristic(S, temp));

if (movement.insert(temp.state[0] + temp.state[1]).second) {

children.push(temp);

ExpandNodeHelper(i-1, j, temp, children, S);

}

}

// move down

if (i < node.state.size() - 1 && node.state[i+1][j] == '0') {

r = node.state[0];

s = node.state[1];

s[j] = r[j];

r[j] = '0';

temp.setState(r,s);

temp.setDepth(node.depth);

temp.setCost (node.depth + heuristic(S, temp));

// temp.setCost (node.cost + heuristic(S, temp));

if (movement.insert(temp.state[0] + temp.state[1]).second) {

children.push(temp);

ExpandNodeHelper(i+1, j, temp, children, S);

}

}

}

// function that expands a node and puts them into the queue

void ExpandNode(Node node, priority\_queue<Node, vector<Node>, SmallerCost> & children, SearchType & S) {

nodesExpanded++;

for (int i = node.state.size() - 1; i >= 0; --i)

for (int j = node.state[i].size() - 1; j >= 0; --j)

if (node.state[i][j] != '-' && node.state[i][j] != '0') {

Node temp;

string r, s;

r = node.state[0];

s = node.state[1];

temp.setState(r,s);

temp.setDepth(node.depth + 1);

temp.setCost(node.cost + 1);

movement.clear();

ExpandNodeHelper(i, j, temp, children, S);

}

}

// the generic search function

Node Search(Node initState, SearchType search, const Node& goalState) {

priority\_queue<Node, vector<Node>, SmallerCost> nodes; // queue of nodes to check

Node answer; // use this to store the answer

goal.setState(goalState.state[0], goalState.state[1]);

nodes.push(initState); // initialize the queue

// initialize answer to all blanks

// we return all blanks if no goal

// state is found

answer.setState("no solution", "no solution");

answer.setCost(-1);

while (!nodes.empty()) {

// print the current queue size

cout << "node size: " << nodes.size() << endl;

// we grab and print the top node

Node temp = nodes.top();

temp.print();

// check if it is the goal state

// if it is, then return it

if (isGoalState(temp)) {

answer.setState(temp.state[0], temp.state[1]);

answer.setDepth(temp.depth);

answer.setCost(temp.cost);

cout << endl << "nodes expanded: " << nodesExpanded << endl;

return answer;

}

// node was not a goal state. so we

// check if we have expanded it in the past

// if not, we expand it, then remove it

// otherwise, we simply remove it

if (history.insert(temp.state[0] + temp.state[1]).second)

ExpandNode(nodes.top(), nodes, search);

nodes.pop();

}

// we return our results

cout << "nodes expanded: " << nodesExpanded << endl;

cout << "faoweiuhfap;wioefh;awiefa;wieofha;weif" << endl;

return answer;

}

**main.cpp**

#include "header.h"

int main(int argc, char\*\* argv) {

Node InitialState; // inital state of the problem

Node goal; // goal state of the problem

time\_point<std::chrono::system\_clock> start, end; // we use this to find the run time

Node answer; // used to store the final state of the solution

int preferedSearch = -1; // the type of search we want to perform

// initialize the Initial state of the problem with

// commandline arguments

InitialState.setState(argv[1], argv[2]);

InitialState.setCost(0);

InitialState.setDepth(0);

// initialize the goal state with

// commandline argumetns

goal.setState(argv[3], argv[4]);

// promp the user to select which search to use

cout << "Welcome to 9 Men in a Trench solver\n";

cout << "Select the your prefered search type:\n";

cout << "1. Uniform Cost Search" << endl

<< "2. Misplaced Tiles Heuristic" << endl

<< "3. Manhattan Heuristic with Sergeant" << endl

<< "4. Count Obstructing Men Heuristic" << endl

<< "5. Check Left Man of each man in correct place" << endl;

do {

cin >> preferedSearch;

} while (preferedSearch > 5 || preferedSearch < 1);

preferedSearch--;

// record the current time and begin the search

start = system\_clock::now();

switch (preferedSearch) {

case UniformCost :

answer = Search(InitialState, UniformCost, goal);

break;

case MisplacedTiles :

answer = Search(InitialState, MisplacedTiles, goal);

break;

case Manhattan\_On\_Sergeant :

answer = Search(InitialState, Manhattan\_On\_Sergeant, goal);

break;

case Count\_Obstructing\_Men :

answer = Search(InitialState, Count\_Obstructing\_Men, goal);

break;

case Check\_Left\_Man :

answer = Search(InitialState, Check\_Left\_Man, goal);

break;

}

end = system\_clock::now();

// after the search we record the time

// we substract the times to find how many

// seconds it took the search to complete

duration<double> runTime = end - start;

// display results

cout << endl << "---FINAL ANSWER---" << endl;

answer.print();

cout << "--> Run time: " << runTime.count() << " seconds" << endl;

return 0;

}