University of Waterloo
CS 486
r2knowle: 2023-10-1

Assignment # 1

Q1a) We will expand the nodes in the following order:

$$s- > h- > k- > c- > a- > b- > d- > m- > e- > n- > g$$

Thus arriving from s to g.

Q1bi) This heuristic is admissible, to prove that this is the case we will use contradiction. Assume that the heuristic is not admissible this means that for some node n:

$$h(n) > h^*(n)$$

where $h^*(n)$ is the true cost of node n. This means that the true cost of n is actually shorter with a barrier, and since a barrier only every reduces the path size this is impossible. Therefore the heuristic is admissible.

Q1bi) Using depth first search with this heuristic gives us the nodes expanded in the following order:

$$s- > h- > k- > c- > a- > b- > d- > m- > g$$

Which gets us from s to g.

Q1bii) Using A^* gives us the nodes expanded in the following order:

$$s- > h- > k- > f- > p- > q- > a- > r- > t- > g$$

Which gets us from s to g.

Q2a) Yes, in the event that all the paths have an equal cost. To prove this lets use contradiction, assume that breath width search does not expand values in the same state as uniform-cost search does. That would imply that the node n's ancestors would be different in uniform search then breath first search. However if the ancestors are different then that would imply that breath first search is on an unoptimised path which is impossible as breath first search minimizes the number of nodes visited and since the path cost is constant this should minimize cost. Therefore because we have a contradiction breath width search is a special case of uniform-cost search.

Q2b) No, as there is no heuristic which guarantees us to always expand the child node (branch) consider the example in 1, we cant define a heuristic such that we will only expand the child node.

Q2c) Yes in the event that the heuristic function is 0, A* will preform exactly the same as uniform cost. This is because the cost of each node in A* is its uniform search cost plus the heuristic and therefore if the heuristic is zero they will be the same.

Q3a) We can represent the TSP by the following problem representation:

States = any combination of citites, where no city is repeated twice.

Representation = we can represent each state by the letter of the city in the order they are visited.

Inital State = the inital state is always city A.

Goal State = The goal state is when all the cities are visited, cost minimized.

Sucessor Function = The sucessor function is to move to a new state, where a new city letter is added to the end of our current state i.e we visit a new city.

Cost Function = The cost between any state is the minimum eucliean distance between all the cities they have in common and the one city they have different. The total cost of any state is the total eucledian distance between all the neighbouring cities in the representation.

Q3b) We can have 3 of the following heuristics:

h1(n): For this heuristic the cost for any city is equal to the minimum distance from this city to any other city that we have. This is admissible as the true cost of any node will always be greater then or equal to the minimum distance, if the true cost is less than it would imply that there is a new minimum path and is thus a contradiction.

h2(n): For this heuristic the cost for any city is equal to 0. This is always admissible as the actual cost of adding a city must be greater then 0. This heuristic will turn our A* algorithm into a uniform search algorithm.

h3(n): the cost for this city is equal to the distance from the current city. This ensures that we will always pick the city that is cheapest from our current node to the next node. This is admissible as there is no path that is cheaper between our current node and the next node as if there was we would use this node instead.

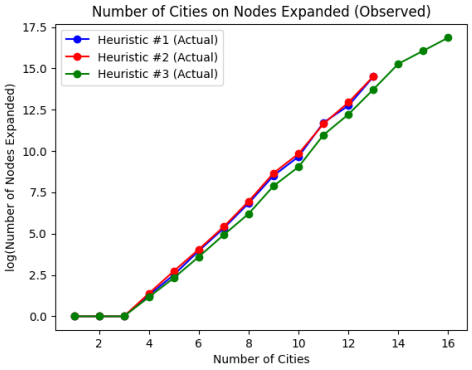
We should expect that h2 will get dominated by both h3 and h1, as h2 will not get rid of any states. For h3 we expect that it should dominate h1 as the actual smallest to our current node should cause a lot less errors then the entire graphs smallest.

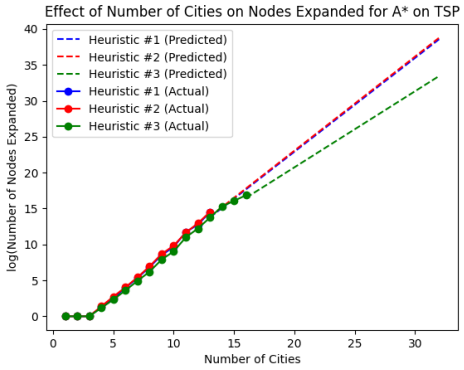
Q3c) On average (across all 10 instances of each city) we expect that each of the heuristics expand the following number of nodes for each of the cities:

		Number of Cities										
		1	2	3	4	5	6	7	8	9	10	11
Heuristic	h1	1	1	1	3.6	12.3	52.6	206.6	941.3	5002.8	15719.7	123394.6
	h2	1	1	1	4	15.4	57.4	225.9	1053.6	5832.2	18735.1	114533.8
	h3	1	1	1	3.2	10.2	36.7	138.1	497.5	2668.7	8452.2	8452.2

		Number of Cities				
		12	13	14	15	16
Heuristic	h1	343869.6	1982604.5	...	...	...
	h2	418689.3	2020802.0	...	...	...
	h3	203459.1	918482.0	4323873	9521119	21013284

Note that for h1 and h2, due to their slow nature it would take way too long to calculate for all 16 cities. Therefore as per the piazza post we only calucated the time it would take for the fastest heuristic (h3). Taking the log of these results we can then graph them to get the following graph shown on the left. By doing simple linear regression we can also predict the results for how many nodes would need to be expanded for up to the 32nd city which is seen on the right:





From the graph is evident that our 3rd heuristic dominates the 1st and second heuristic. Even with the relative decrease in number of nodes expanded, this heuristic would still be incredibly slow for TSP on 32 cities, infact each of the heuristics would need to expand the following number of nodes:

$$\text{Heuristic \#1} : 5.674124925210^{16} \text{ nodes expanded}$$
$$\text{Heuristic \#2} : 6.978236067710^{16} \text{ nodes expanded}$$
$$\text{Heuristic \#3} : 2.048667918710^{14} \text{ nodes expanded}$$

Thus it would appear out of all our heuristics h3 preforms the best, as seen by the graph and the predicted number of nodes to solve the 32 city problem.

Code for Q3) Here is all the code for my question 3, I exported the results to a separate python file which plotted everything in pyplot (which is not included in the below code):

```
import numpy as np
import heapq


cityMap = {}                                        # Store position of each city
stateCostMap = {}                                   # Store total cost of each state
cityMinCost = {}                                    # Store minimum cost of each city

states = []                                         # Stores our states to expand
citiesToVisit = []                                  # Stores the cities we haven't visited

# This function takes in two city and calculates the distance between them (l2 norm)
def calculateCost(city1, city2):
    city1cords = cityMap[city1]
    city2cords = cityMap[city2]

    return np.sqrt((city1cords[0] - city2cords[0]) ** 2 + (city1cords[1] - city2cords[1]) ** 2)


# This is the third heuristic, the cost of the state + the distance to each possible next city
def getCost3(state):
    return calculateCost(state[-2], state[-1]) + stateCostMap[state[0:-1]]

# This is the second heuristic, the cost of the state + 0
def getCost2(state):
    return stateCostMap[state[0:-1]]

# This is the first heuristic, the cost of the state + the minimum distance to the current state
def getCost1(state):
    return cityMinCost[city] + stateCostMap[state[0:-1]]


# db refers to the number of cities we need to loop through
for db in range(2, 14):
    numberOfStates = 0                                  # Stores total states for problem set
    for instance in range(1, 11):

        # Empty out our global variables
        cityMap = {}
        stateCostMap = {}
        cityMinCost = {}
        states = []
        citiesToVisit = []

        # Read in the file
        file = open("randTSP/" + str(db) + "/instance_" + str(instance) + ".txt", "r")
        numberOfCities = int(file.readline())

        states = []
        heapq.heapify(states)                           # States is a heap so we do our
            operations quickly
        finalGoalString = []

        # Convert the points in the text file into our dictionaries
```

```
    for i in range(0, numberOfCities):
        line = file.readline().split(" ")
        line[2] = line[2].split('\n')[0]

        if line[0] == 'A':
            stateCostMap['A'] = 0
        else:
            citiesToVisit.append(line[0])
        finalGoalString.append(line[0])
        cityMap[line[0]] = (int(line[1]), int(line[2]))

    listOfAllCites = citiesToVisit + ["A"]

    # Manually calculate the children from the first node
    for cityRef in citiesToVisit:
        minSize = -1
        for city in listOfAllCites:
            if (cityRef != city):
                currSize = calculateCost(city, cityRef)
                if (minSize == -1 or minSize > currSize):
                    minSize = currSize
        cityMinCost[cityRef] = minSize

    for city in citiesToVisit:
        newState = "A" + city
        # Add the distance + state representation onto the heap for later
        heapq.heappush(states, (getCost3(newState), newState))

    while True:
        numberOfStates += 1

        # Pop the closest state
        curPick = heapq.heappop(states)[1]

        count = 0
        # Add its children to the heap
        for city in citiesToVisit:
            if city not in curPick:
                actualCost = calculateCost(curPick[-1], city)
                stateCostMap[curPick] = actualCost + stateCostMap[curPick[0:-1]]
                newState = curPick + city
                heapq.heappush(states, (getCost3(newState), newState))
                count += 1

        if count <= 1:
            break

print("City " + str(db) +"#: " +str(numberOfStates/10))
```

Q4a) To begin we can define Sudoku as a CSP. Thus we will be having the following terms:

Variables : x = {1,...,9} and y = {1,...,9}. We will have x represent the current row and y represent the current column. Thus an element can be represented as $G_{x,y}$

Domain : Each $G_{x,y}$ will belong to {1,2,3,4,5,6,7,8,9}

Constraints : $\forall x \in \{1,..,9\}$ and $\forall y \in \{1,..,9\}$ where $x \neq y, \forall i \in \{1,..,9\}\ G_{i,x} \neq G_{i,y}$
$\forall x \in \{1,..,9\}$ and $\forall y \in \{1,..,9\}$ where $x \neq y, \forall i \in \{1,..,9\}\ G_{x,i} \neq G_{y,i}$
$\forall i1 \in \{1,2,3\}, \forall i2 \in \{1,2,3\}$ and $\forall x = \{1,2,3\}$ and $\forall y = \{1,2,3\}$:
$G_{[i1\times x, i2\times y]} = \{1,2,3,4,5,6,7,8,9\}$

These three constraints represent the each value being unique along the row and column. Moreover each 3x3 cube needs to have exactly $\{1,2,3,4,5,6,7,8,9\}$ with no repeats.

Q4b) We have the following solutions for the problems:

6	1	2	9	8	4	7	5	3
9	4	5	3	2	7	6	8	1
3	7	8	6	5	1	9	2	4
5	9	1	2	7	6	3	4	8
7	6	3	8	4	5	2	1	9
8	2	4	1	3	9	5	6	7
2	3	9	5	1	8	4	7	6
1	5	7	4	6	3	8	9	2
4	8	6	7	9	2	1	3	5

Figure 1: Easy Sudoku

3	4	8	1	6	7	9	2	5
6	5	7	3	9	2	1	8	4
2	1	9	5	8	4	3	7	6
7	8	1	2	3	6	4	5	9
4	2	5	7	1	9	6	3	8
9	3	6	4	5	8	2	1	7
1	9	3	6	7	5	8	4	2
5	6	2	8	4	1	7	9	3
8	7	4	9	2	3	5	6	1

Figure 2: Medium Sudoku

1	6	3	9	2	5	4	7	8
8	9	4	3	6	7	5	1	2
7	5	2	4	8	1	6	3	9
2	1	6	7	4	9	3	8	5
3	7	9	6	5	8	2	4	1
5	4	8	2	1	3	9	6	7
6	3	5	1	7	2	8	9	4
9	2	1	8	3	4	7	5	6
4	8	7	5	9	6	1	2	3

Figure 3: Hard Sudoku

3	4	1	9	7	8	6	5	2
5	9	2	1	6	3	7	4	8
7	6	8	4	5	2	3	9	1
6	1	9	8	2	7	4	3	5
8	2	3	5	9	4	1	7	6
4	5	7	6	3	1	8	2	9
9	3	6	7	1	5	2	8	4
2	8	5	3	4	6	9	1	7
1	7	4	2	8	9	5	6	3

Figure 4: Evil Sudoku

On the next page will be the timings and the discussion of the different methods of solving for Sudoku.

Q4c) Below in the tables for the preformance of the different version we used. Note that the means and standard deviation for both the nodes and timings are in their own tables:

	Easy Sudoku	Medium Sudoku	Hard Sudoku	Evil Sodoku
Version A	0.263372	18.2829	230.562	2033.96
Version B				
Version C				

Figure 5: Mean Run Time (Seconds)

	Easy Sudoku	Medium Sudoku	Hard Sudoku	Evil Sodoku
Version A	0.578759	23.8846	91.9113	1860.95
Version B				
Version C				

Figure 6: Standard Deviation for Run Time (Seconds)

	Easy Sudoku	Medium Sudoku	Hard Sudoku	Evil Sodoku
Version A	197364	1.29661×10^7	2.30562×10^8	1.51687×10^9
Version B				
Version C				

Figure 7: Mean Nodes Expanded

	Easy Sudoku	Medium Sudoku	Hard Sudoku	Evil Sodoku
Version A	434741	2.38846×10^7	9.19113×10^7	1.38905×10^9
Version B				
Version C				

Figure 8: Standard Deviation for Mean Nodes Expanded

From these results we can observe that version C and version B do far better then version A. This is because the runtime is directly tied to the number of nodes that we need to expand, and version A will expand the most nodes as it only terminates when it cant find any values in the domain to satisfy a variable. On the otherhand version B and version C can look ahead and see which variables will be impossible to satisfy and can avoid going all the way to the leaf before terminating.

Version C differs from version B as it uses 3 heuristics in order to improve its run time:

Heuristic #1 : Expand the most constrained variable first

Heuristic #2 : Use the least constraining values from the domain

Heuristic #3 : Expand the most constraing variable

In this order we will expand the variables, which as we saw decreased the run time and the number of nodes needed to expand from version B. All the code for each of the versions is provided on the following pages.

Code for Version A)

```
#include <iostream>
#include <string>
#include <chrono>
#include <vector>
#include <random>
#include <algorithm>
#include <numeric>

using namespace std::chrono;
using namespace std;

// Store the boards as strings
std::string easyBoard =
    "010900053040300681070050900590070040700805009020030067009010070157003090480002030";
std::string mediumBoard =
    "000160925007000180000080006780200000025010630000008017100070000062000700874023000";
std::string hardBoard =
    "103005400800000000000080600006049000079608240000210900005070000000000006007500103";
std::string evilBoard =
    "001970000090003008000402000610000405003000100407000029000705000200300010000089500";

// This counter will be used to keep track of the number of nodes expanded
int nodesExpanded = 0;


// Helper function to display the grid
void printGrid(const int map[81]) {
    for (int i = 0; i < 9; i++) {
        cout << "{";
        for (int x = 0; x<9; x++) {
            cout << map[i*9+x];
            if (x != 8) {
                cout << ", ";
            }
        }
        cout << "}," << endl;
    }
}


/*
 * This function is used to set the domain, the domain stores all the possible values at each
     index.
 *  Each time we add a value we need to update the domain accordingly, thus we take in the index we
 *  want to change
 */
bool setDomain(int map[81], int domain[][10], int idx) {
    int row = idx/9;
    int column = idx%9;

    for ( int i = 0; i < 9; i++ ) {
        domain[i+row*9][map[idx]] = 1;          // Update along the row
        domain[i*9 +column][map[idx]] = 1;      // Update along the column
    }

    int rowStart = row/3;                       // Start position of grid (row)
    int columnStart = column/3;                 // Start position of grid (column)
```

```
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            domain[columnStart*3 + (y+rowStart*3)*9 + x][map[idx]] = 1;
        }
    }
    return true;


}



/* We treat the original array like doing n insert, where n
 *  is the number of non zero elements in our array
 */
void init(int map[81], int takenValues[][10]) {
    for ( int y = 0; y < 9 ; y++ ) {
        for ( int x = 0; x < 9 ; x++ ) {
            if ( map[y*9+x] != 0 ) {
                setDomain(map, takenValues, y*9+x);
            }
        }
    }

}

/*
 * This is the main method for solving any sukodku problem, to begin we will pick a random
 *  index to loop through, after this we will we will check if there are any more values to read
 *  in. After this we will check our domain for that variable, and continue if theres a valid
     placement.
 *  We continue until we get a solution
 */
bool sukodku(const int board[81], const int domain[][10], const vector<int>* const valsToReplace,
    int pos) {

    int positionToCheck = valsToReplace->operator[](pos);
    pos++;

    if (valsToReplace->size() < pos) {
        printGrid(board);               // Print the solution!
        return true;
    }

    for (int idx = 1; idx < 10; idx++) {

        if (domain[positionToCheck][idx] != 1) { // Check if the domain is empty

            nodesExpanded++;                // Expand the nodes


            int newDomain[81][10];          // Make a deep copy of the domain
            int newMap[81];                 // Make a deep copy of the matrix

            for (int i = 0; i < 81; i++) {
                newMap[i] = board[i];
                for (int x = 0; x < 10; x++) {
                    newDomain[i][x] = domain[i][x];
```

```
            }
        }

        newMap[positionToCheck] = idx;
        setDomain(newMap, newDomain, positionToCheck);

        // Recurse until we can find the proper solution
        bool result = sukodku(newMap, newDomain, valsToReplace, pos);

        if (result) {
            return true;
        }
    }
}

    return false;
}
int main() {

    vector<int> timings;                        // Vector for storing the timings per run
    vector<int> nodes;                          // Vector for storing the number of nodes per run

    auto rng = std::default_random_engine{};    // Random needed for shuffle
    rng.seed(getpid());

    for (int runs = 0; runs < 50 ; runs++ ) {    // Number of runs to do

        std::string readin = evilBoard;         // Which board to use

        // This will convert the string into a 1D int array
        int board[81];
        for (int y = 0; y < 9; y++) {
            for (int x = 0; x < 9; x++) {
                board[y * 9 + x] = readin[y * 9 + x] - '0';
            }
        }

        int takenValues[81][10];

        for (int i = 0; i < 81; i++) {
            for (int x = 0; x < 10; x++) {
                takenValues[i][x] = 0;
            }
        }

        init(board, takenValues);
        // This will store all the indexes where we have a zero
        vector<int> valsToReplace;

        for (int i = 0; i < 81; i++) {
            if (board[i] == 0) {
                valsToReplace.push_back(i);
            }
        }

        // Shuffle the list of positions to check
        shuffle(valsToReplace.begin(), valsToReplace.end(), rng);

        nodesExpanded = 0;
```

```
        // Start the time
        auto start = high_resolution_clock::now();

        sukodku(board, takenValues, &valsToReplace, 0);
        // Finish the time
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(stop - start);

        timings.push_back(duration.count());
        nodes.push_back(nodesExpanded);

        std::cout << runs << endl;
    }


    // The below code will calculate the timiing and number of nodes for each run
    double sum = std::accumulate(timings.begin(), timings.end(), 0.0);
    double mean = sum / timings.size();

    double var = 0;
    for( int n = 0; n < timings.size(); n++ )
    {
        var += (timings[n] - mean) * (timings[n] - mean);
    }
    var /= timings.size();
    double stdev = sqrt(var);

    std::cout << "Timing average: " << mean << " | Timing std deviation: " << stdev << endl;

    sum = std::accumulate(nodes.begin(), nodes.end(), 0.0);
    mean = sum / nodes.size();



    var = 0;
    for( int n = 0; n < nodes.size(); n++ )
    {
        var += (nodes[n] - mean) * (nodes[n] - mean);
    }
    var /= nodes.size();
    stdev = sqrt(var);

    std::cout << "Node average: " << mean << " | Node std deviation: " << stdev << endl;

    return 0;
}
```