AI ASSIGNMENT 1

Artificial Intelligence DNET4

OCTOBER 27, 2015

REN KAI TAM
R00095982

Contents

I, the undersigned, declare the following to be 100% original and my own work, except where indicated otherwise. Signed: _____Kai Tam_____".

Assignment 1

Before implementing the code for Hill Climbing and Simulated Algorithms I created a Component class called “Components”. This class contains a list of methods / components that’s is required to construct both algorithms. The access modifier of these methods and variables within this class is set to “protected”, the reason for that is because it allows the Hill Climbing class and Simulated Annealing class to use the method within the same package. Both algorithms class inherits “Components” class.

Java Classes Description

Components class

This class does all the preparation that needed for the algorithm to run. It contain methods:

- ***ReadFromFile*** – This read the text file cities into a 2 dimensional array. Call by constructor.
- ***CumulativeDistance*** – This takes in an array integer (list of cities). It finds the distance between 2 cities within the list and return the sum of all the distances.

```
int totalDistance = 0;
if(stateList != null)
    for(int i = 0; i < citiesMatrix_15.length  ; i++)
        if((i+1) < citiesMatrix_15.length ) // (i+1) is looking for the next cities
        {
            Get the distance from citiesMatrix and add into the total distance
            totalDistance+=citiesMatrix_15[stateList.get(i)][stateList.get(i+1)];
        }
```

- ***GenerateFutureState*** – This method use *generateNumber* method that takes in 4 numbers of cities and then randomize the number in the range of 4 digit numbers. This 4 digit number will determine the order of the cities that will be rearranged then it used *convertStateIndex* method.
- ***GenerateNumber*** – Takes on the size, max number and mine, then it generates the range number from min to max. Size will determine the number of times it has to generate to populate the list, then it returns an ArrayList.
- ***convertStateIndex*** – This method takes in an array of index generated by *generateNumber* and an integer index. This method then will populated into the *childState* array, this will be the future state.

```
// Change the number according to the stateIndexes
for(int i = 0; i < stateIndex.size(); i++)
    childState[x][x+i] = currentState.get(stateIndex.get(i));

//Populate the numbers that doesn't require to change in the position
for (int i = 0; i < xSize  ; i++)
    if(!(i > x-1 && i < x+stateChangeVariable))
        childState[x][i] = currentState.get(i);
```

- ***FutureStateCumulativeDistance*** – use *cumulativeDistance* method to calculated each future State cumulative distances and put them into an array called *childStateCumulativeDistances*. Then use the method *determinedFutureState*.

- ***DeterminedFutureState*** – Use the return value of *findBestChildState* (index of the childState Array). Then it compare the cumulative values to the parent’s cumulative value. If child cumulative value is smaller than the parent’s value. Then the current state will be replaced.

- ***GenerateInitialState*** – This contains 2 useful methods, *generateNumber* and *cumulativeDistance*. Using this 2 method to generate the initial state.

```
//Generate the first random state
protected void generateInitialState(String header){
    currentState = generateNumber(xSize  , xSize-1, 0);
    parentCMValue = cumulativeDistance(currentState, null); // cumulative Distance for Current State
    printArray(null, currentState, header);
}
```

Assignment 1 – Hill Climbing

Technical Description

The Hill Climbing class extends the component class to inherit all variables and methods. Calling super within the constructor to read the text file and load data into a 2 dimensional array. This class contains 2 methods, hillClimb, main and a contructor. HillClimb method contains the algorithm that runs hillClimbing.

Algorithm

```
private void hillClimb(){
    //matrixDebug(citiesMatrix_15);
    generateInitialState("First Random Generated State");
    boolean setTrue = true;
    while(setTrue)
    {
        if(bestChildState > parentCMValue)
        {
            bestPaths.add(bestChildState);
            generateInitialState("New Random Generated State");
            setTrue = false;
        }
        else
        {
            generateFutureState();
            futureStatesCumulativeDistance();
        }
    }
    printArray(null, bestPaths, "Best Paths");
    System.out.println("Best Path Length : " + bestPaths.size() + "\nBest Paths :" + bestPaths.get(findBestChildState(null, bestPaths))
            + "\n**********************************RANDOM RESTART**********************************");//findBestChildState
}
```

First in the *hillClimb* method call *generateInitialState* to generate a random current state using *generateNumber* method, then it calculated the cumulative distance using *cumulativeDistance* method. Using While loop to keep looping through the methods until the Boolean setTrue is equal to false to break out of the loop. Within the while loop contains in statement to check if future state's cumulative value is greater than parent's cumulative value. For loop is used in the constructor to loop through multiple times to get the best result.

Scenario 1

If it future state is not greater than the parent's cumulative distance then keep generating new future state and calculate those possible future state cumulative distances and compare then with parent.

Scenario 2

If it future state is greater than the parent's cumulative distance, then the best path will be recorded. Programs stops here when the cumulative distance is greater than the parent, so new state will be randomly generated again and Boolean set to false to break out of the while loop and prints out the best state.

Simulated Annealing

Technical Description

The Simulated Annealing class extends the component class to inherit all variables and methods. Calling super within the constructor to read the text file and load data into a 2 dimensional array. This class contains 3 methods, *simulateAnnealing*, *acceptMove*, main and a constructor. *SimulateAnnealing* method contains the algorithm that runs Simulated Annealing class. *AcceptMove* contains the maths calculation.

Method – Accept Move

```
private boolean acceptMove (int currentState, int proposedState, int temperature){
    Random r = new Random();
    if (proposedState < currentState)
        return true;
    else if(temperature == 0)
        return false;
    else
    {
        float  probOfMove = (float) Math.exp(-(proposedState - currentState) / temperature);
        float randNum = (r.nextInt((10 - 0) + 1 ) + 0)/10.0f;
        System.out.println("Probability of Move: " + probOfMove + "\t randNum :" + randNum);
        if (randNum < probOfMove)
            return true;
        else
        {
            bestPaths.add(bestChildState);
            temp = bestChildState;
            return false;
        }
    }
}
```

This method takes in the cumulative distance of current state and the propose state If the propose state is greater than the current state then do the calculation to find the probability of moving future state which future state.

Algorithm

```
private void simulateAnnealing(){
    generateInitialState("First Random Generated State");
    boolean setTrue = true;
    while(setTrue)
    {
        generateFutureState();
        futureStatesCumulativeDistance();

        if(acceptMove(parentCMValue, bestChildState, temp))
        {
            generateFutureState();
            futureStatesCumulativeDistance();
        }
        else
        {
            generateInitialState("New Random Generated State");
            setTrue = false;
        }
    }

    printArray(null, bestPaths, "Best Paths");
    System.out.println("Best Paths :" + bestPaths.get(findBestChildState(null, bestPaths))
            + "\n**********RANDOM RESTART**********");// findBestChildState

}
```

First in the method *simulateAnnealing* calls *generateInitialState* to create the first random state. Then it generates the future state and calculate each future state cumulative distances into an ArrayList. *AcceptMove* compares the cumulative value between parent and the best child state.

Scenario 1
If child's state is smaller than parent's state then it returns true to continue to generate future states.

Scenario 2
If temperature variable is equal to zero, it return false. Then it generate a new state and start the process again.

Scenario 3
If the child's state is greater than parent's, probOfMove = *Math. Exp (-(proposeState - currentState) / temp))* this equation gives us a number range between 0 and 1. Then generate the number then using that number to compare with probOfMove. If the random number is less than probOfMove it returns true, then it keeps generating new future state and the process start again.

Scenario 4
If the random number is greater than probOfMove it returns false, algorithm stops and new state is generated and temperature value is change to the new lower value.

How to select new state

Select a random Initial state
To select the initial state, I created a method that takes in 3 parameters.

- Size – Which determined how many random number this method should generate.
- Max, Min – Number generator will generate the number within this range of number.

First it takes in 15 numbers and maximum of 14 and minimum of 0, this will generate the number between 15 numbers from 0 to 14. Within a while loop, If the number is previously generated it will not put it into the list, and it will continuously loop to generate more number until the array is populated. Then cumulative distance is calculated using cumulativeDistance method.

Select a new state
To generate future state, there are several stage involve. First there is a variable called "*stateChangeVariable*", this variable determined how many number should be randomize within the current state and also how many number it should be generated by the random generated number.

```
for(int x = 0; x < xSize-stateChangeVariable+1; x++)
{
    ArrayList<Integer> stateIndex = generateNumber(stateChangeVariable, (x+stateChangeVariable-1), x); // State index
    //Possible Future state
    convertStateIndex(stateIndex, x);
}
```

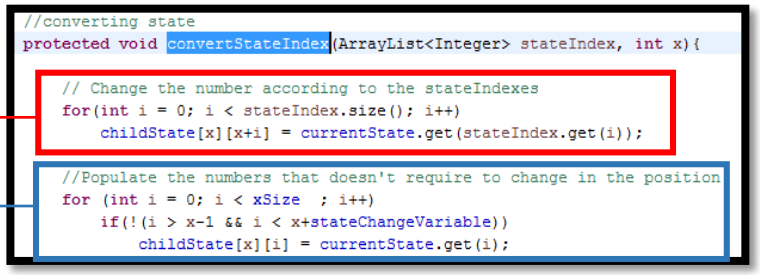
Step 1
Within for loop, using generateNumber method to generate 4 random numbers as shown above. For loop is used to give the position in the current state, so the random number can generate 4 numbers but in an increasing index.

[i.e.]

1. First time "x" is equal to 0, so it generate number between 0 to 3
2. The second time "x" is to 1, so it generate number between 1 to 4 and so on

This is important, because "x" will be the starting index for the next steps. After generated 4 numbers, it returns an array list of integer. This integer list contains the indexes, these indexes will passed into the method convertStateIndex.

Step 2

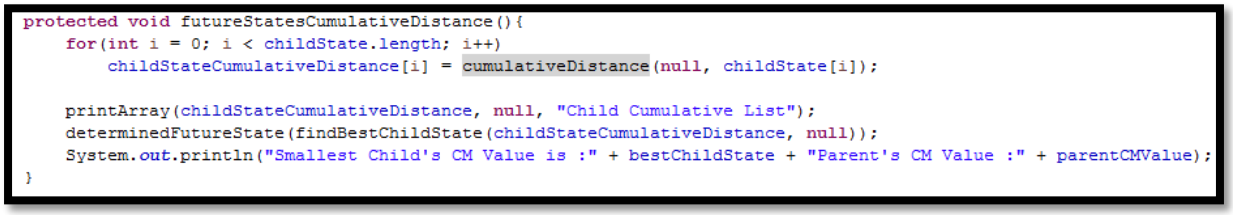


```
//converting state
protected void convertStateIndex(ArrayList<Integer> stateIndex, int x){

    // Change the number according to the stateIndexes
    for(int i = 0; i < stateIndex.size(); i++)
        childState[x][x+i] = currentState.get(stateIndex.get(i));

    //Populate the numbers that doesn't require to change in the position
    for (int i = 0; i < xSize  ; i++)
        if(!(i > x-1 && i < x+stateChangeVariable))
            childState[x][i] = currentState.get(i);
```

In this method first it will rearrange the number according to the staring index which is "*x*".
[i.e.] *x* = 0 | the StateIndex is (1 0 3 2) | currentState is (1, 10, 7, 5, 9, 11, 12, 4, 13, 6, 0, 3, 2, 14)

1. It gets the *currentState* value from the *startIndex* of 0 which has the value of 1. This value is an index. Using this index to rearrange the number on the currentState and put the number into childState.
2. ChildState is 2 dimensional array that contain lists of possible future state. So in this example:
    a. currentState.get(stateIndex.get(i)) = currentState.get(1).
    b. childState[x][x+i] = childState[0][0+0]
    c. so, childState[0][0] = 10
    d. Next, childState[0][0+1] = currentState.get(0)
    e. childState[0][1] = 1 and so on.
3. For loop will loop 4 times to finished populating the index numbers. Now the within childState[0] there are 4 numbers in the array.
    a. childState[0] = 10, 1, 5, 7, 0,0,0,0,0,0,0,0,0,0
4. The next for loop populate the same number into the childState.
    a. childState[0] = 10, 1, 5, 7, 9, 11, 12, 4, 13, 6, 0, 3, 2, 14
5. The method is finished and return back to generateFutureState method and the index "*x*" increase to 1 and the cycle starts again until all the possible childState are populated.
    a. childState[1] = 10, 9, 5, 1, 7, 11, 12, 4, 13, 6, 0, 3, 2, 14
6. This will continue until all childState array are populated.

Step 3



```
protected void futureStatesCumulativeDistance(){
    for(int i = 0; i < childState.length; i++)
        childStateCumulativeDistance[i] = cumulativeDistance(null, childState[i]);

    printArray(childStateCumulativeDistance, null, "Child Cumulative List");
    determinedFutureState(findBestChildState(childStateCumulativeDistance, null));
    System.out.println("Smallest Child's CM Value is :" + bestChildState + "Parent's CM Value :" + parentCMValue);
}
```

Each child state's Cumulative distance will be calculated using futureStateCumulativeDistance. FutureStateCumulativeDistance uses cumulativeDistance to calculate the distance of the children (CumulativeDistance method Explain at page 3 of the report).

Hill Climbing algorithm

In hill climbing algorithm, it will calculate the best cumulative distance using the method findBestChildState then it returns the index of the best childStateCumulativeDistance array. Then determinedFutureState used the index and determined if the child cumulative value is greater than or less than the parent's cumulative value. If bestChildState is less than parent's then the bestChildState will become the current state and it continue of looping through future state, otherwise it stop looking for new future state and generate new initial state again.

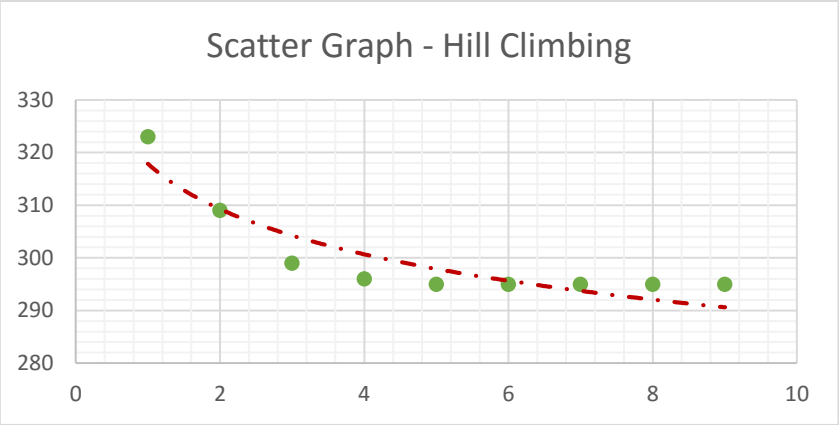
Simulated Annealing

In Simulated Annealing algorithm, the children cumulative distances are calculated. If the bestChildState is less than parent state, bestChildState becomes the current state then keep looping to find new future state, otherwise a heuristic value is calculated using this formula $exp(-(E_n - E_p)/temp)$. A random number is generated between 0 and 1, if the number is greater than this heuristic value then it start a new Initial state again, but id the number is less than heuristic value then it move into this state and find new future state.

Algorithms Evaluation

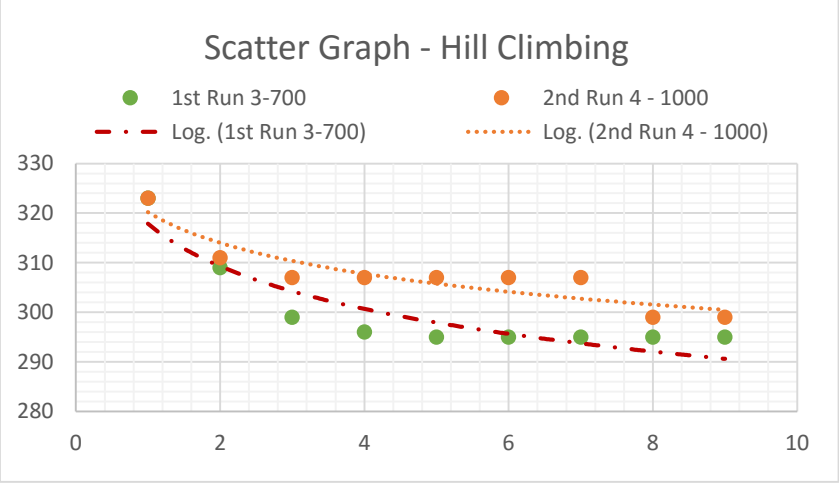
Hill Climbing

Result Set 1 – Graph and Evaluation



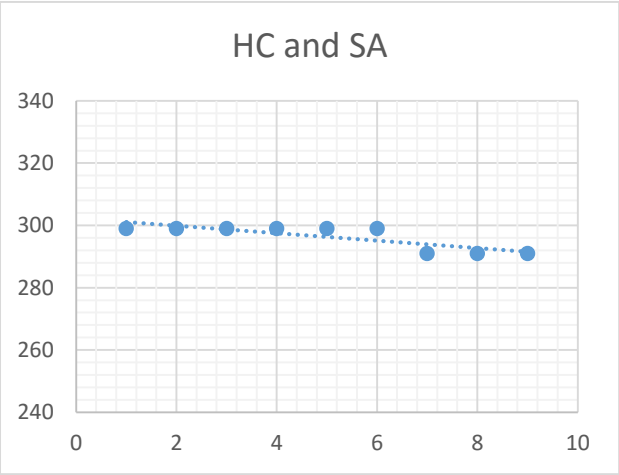
This result is produce by setting *stateChangeVariables* to 3. It take 9 times running at 700 time in a loop to produce average value of 300.2222. In my opinion this is a very good sets of result. The Closest value Hill Climbing can get is 295, it's only 4 unit away from the final solution.

Result set 2 – Graph and Evaluation

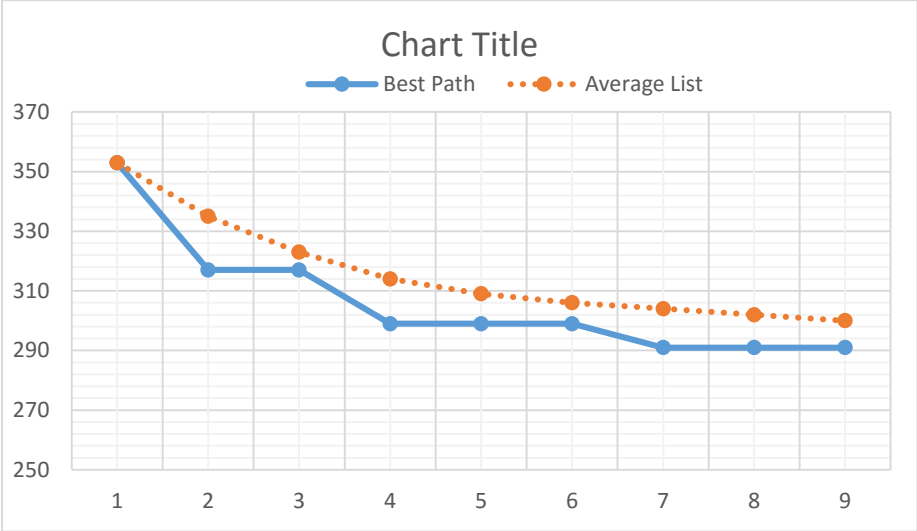


By increasing the stateChangeVariable to 4, it requires to run 1000 times in a loop to get almost as good result. Performance decreased as the stateChangeVariable increases and increasing the loop causes additional latency, it produced a less accurate result. The average unit for the 2nd run is 307.4444.

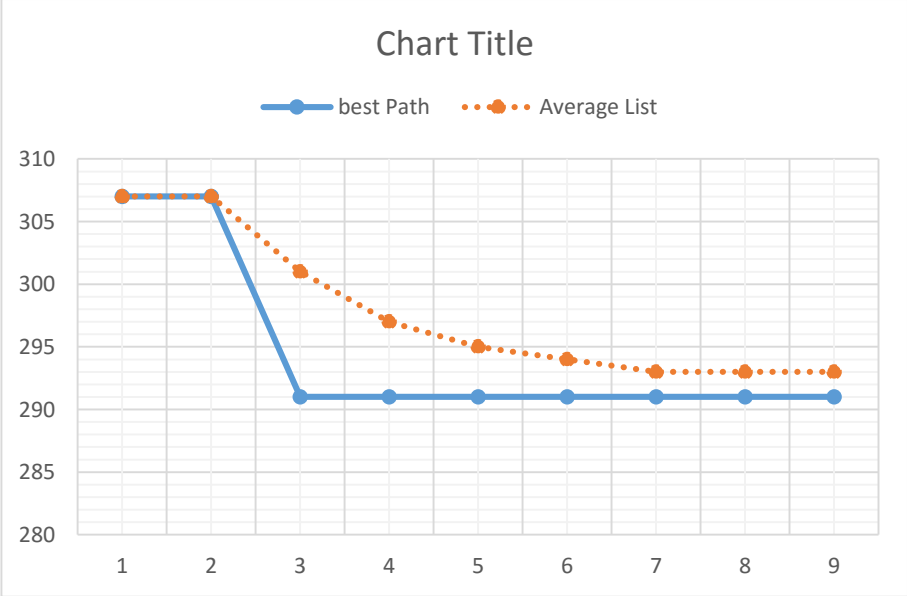
Hill Climbing & Simulated Annealing



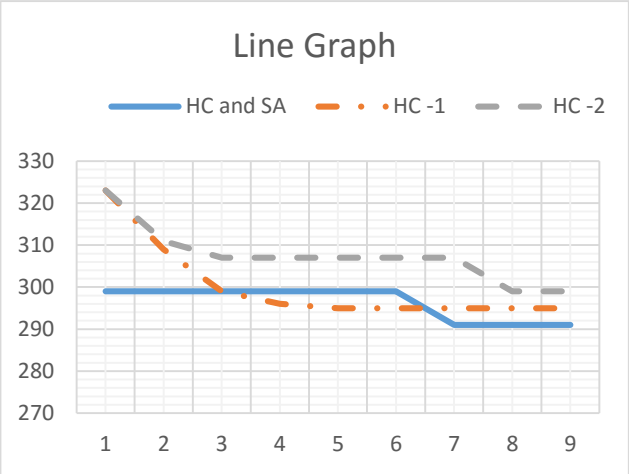
The above scatter graph shows the value that eventually reaches the final result (291 unit). Using the combination of this requires less processing power. Using stateChangeVariable (changing block of numbers) of 4, and run about 150 times in a loop for 9 times to get 9 results. The average number is 296.3333, it's significantly more accurate than just Hill Climbing alone.



In this graph, it graph the best path against the average list path. This is also using stateChangeVariable value of 4 and looping through 150 (x9) times.



This graph, stateChangeVariable increase to 4 and increase the loop count to 190 to get the best result overall.



In the line graph, it shows that the blue line gets significantly lower unit to begin with. Processing power requires a lot less, from the Hill Climbing result set 1 it requires to loop through 700 (x9) times to produce the output average of 300.2222 and the blue line which is using the combination of Hill Climbing and Simulated Annealing the it only looping through 150 (9) times and latency of producing the output is less than Hill Climbing. From this graph, show that This Hill Climbing and Simulated Annealing use less performance power it give better accuracy and less latency when producing output.

Improving performance

Performance can be improve my adjusting the stateChangeVariable and the loop count number. Increase or decrease the stateChangeVariable can determined by the number of child it produced, increase stateChangeVariable decrease number of child vice versa. Increasing the loop count will cause increase latency when outputting the result. Hill Climbing with Simulated Annealing help to drive down the loop count, this decrease the latency when outputting result.