Assignment 1 Report
8 Queen Problem

Salma Abd Elaziz (28)

October 2017

Problem Statement

We would like to start from a random configuration of the 8 queens on the board and try to reach the goal state where there is no queen is attacking the another one.

- The state of a node would be the positions of the queens.
- The value of a node would be the number of attacks on the current board configuration.

Our goal here is to apply the local search methods to get the goal state.

Requirements

- Implement in your preferred programming language the Pseudo code of Hill Climbing Algorithm.
- Modify the Pseudo code of Hill Climbing to accommodate for
  - Random Restarts. Write your new pseudo code
  - Sideways moves. Write your new pseudo code Next, implement these pseudo codes and report in 100 experiment how much of the time you can get the goal in less than M steps.
- Write your pseudo codes and codes to implement the k-local beams algorithm. Where you start with K possible states and move to the best K states. So, a node should contain a set of K board configurations and K values. Report in 100 experiment how much of the time you can get the goal in less than M steps.

assumptions

- maximum number of looping in the random restart and sideways moves is 50 to not enter infinite loop.
- k is assumed to be 5 in the k-beam algorithm

Program Structure

- the Launcher class has an object of SearchMthods class to call the functions of the search
- each of the search method returns an object from class Node.
- Node class consists of an object from class State and an integer the is the value.
- state class consists of a 1-D array of Queen class, that is the index of the array is the row of the queen and the column is the value stored in the array.
- each Queen class object has two integer variables; column number and row number.

Hill Climbing

Algorithm:

```
function hillClimbing:
output : the node with the smallest attack numbers of
the initial state.
    current <-- the initial state
    loop
        neighbor <-- the best successor int the neighbor
        list
        if neighbor.value <= current.value then
            return current.state
        current <-- neighbor
```

Experiment:

- **m = 10**
  out of 100 experiment the number of the time I got the goal in less than 10 steps = 9
- **m = 20**
  out of 100 experiment the number of the time I got the goal in less than 30 steps = 10

- **m = 50**
  out of 100 experiment the number of the time I got the goal in less than 50 steps = 12

Random Restart

Algorithm:

```
function RandomRestart:
output : the node with the optimal solution if found and an
infinite loop didn't happened or smallest attack number
found during the iterations.
    loop until the maximum limit:
        result <— call to hillClimbing method
        if result is the optimal solution return it
        else keep track of the best state you reached to
            return it if an infinite loop happened.
    return the state you were tracking
```

Experiment:

- **m = 10**
  out of 100 experiment the number of the time I got the goal in less than 10 steps = 11

- **m = 20**
  out of 100 experiment the number of the time I got the goal in less than 20 steps = 12

- **m = 60**
  out of 100 experiment the number of the time I got the goal in less than 50 steps = 15

Random Restart

Algorithm:

```
function RandomRestart:
output : the node with the optimal solution if found and and
infinite loop didn't happened or smallest attack number
```

```
found during the iterations.
    loop until the maximum limit:
        result <— call to hillClimbing method
        if result is the optimal solution return it
        else keep track of the best state you reached to
            return it if an infinite loop happened.
    return the state you were tracking
```

Experiment:

- **m = 10**
  out of 100 experiment the number of the time I got the goal in less than 10 steps = 60
- **m = 20**
  out of 100 experiment the number of the time I got the goal in less than 20 steps = 81
- **m = 60**
  out of 100 experiment the number of the time I got the goal in less than 50 steps = 95

Sidewyas moves

Algorithm:

```
function hillClimbingWithSidewayMoves:
output : the node with the optimal solution if found and
infinite loop didn't happened or smallest attack number
found during the iterations.

    current <— the initial state
    noOfSideMoves <— 0
    loop
        neighbor <— the best successor int the neighbor
        list
        if current is the optimal solution return it.
        else if neighbor.value < current.value then
            return current.state
        if  neighbor.value = current.value then
```

```
            noOfSideMoves <— noOfSideMoves + 1
            if noOfSideMoves exceeds the limits
                return current.state
        else this means that we are improving the current
        state then noOfSideMoves <— 0
        current <— neighbor
```

Experiment:

- **m = 10**
  out of 100 experiment the number of the time I got the goal in less than 10 steps = 5
- **m = 20**
  out of 100 experiment the number of the time I got the goal in less than 20 steps = 5
- **m = 30**
  out of 100 experiment the number of the time I got the goal in less than 50 steps = 6

Sidewyas moves

Algorithm:

```
function hillClimbingWithSidewayMoves:
output : the best state that it can reach in its k beams.

    initialize k_States, nextGeneration as a sorted map of the
    number of attacks as the key and the state itself
    as the value.

    loop k times:
        state <— randomly generated state
        add state to k_States.

    loop:
        keys <— the keys in the k_State map.
        for each key in keys do:
            s <— find the state of this from the k_State map
```

```
        find the k neighbors of s.
        for each neighbor in neighbors of s:
            add it to nextGeneration map with the number
            of attacks as the key.

    if the first element in nextGeneration map(lowest key)
    is the optimal then
        return the first state in nextGeneration.
    If the best state in nextGeneration is worst than the
    best state in the current generation(K_state maap)
    then
        return the first state in k_Sate map.
    else
        reinitialize the k_State map.
        find the first k states in nextGeneration map
        and add them in the k_state map.
```

Experiment:

- **m = 10**
  out of 100 experiment the number of the time I got the goal in less than 10 steps = 9

- **m = 20**
  out of 100 experiment the number of the time I got the goal in less than 20 steps = 11

- **m = 30**
  out of 100 experiment the number of the time I got the goal in less than 50 steps = 16