Artificial Intelligence Homework

Archer Problem

Buzdugă Ionuţ Gabriel
Group CEN 2.1B
Year 2
Computer and Information Technology-English

1 Problem Statement

Archer arrangement problem

Let us suppose the k ×k grid presented in Figure 2. The grid is configured with a pattern of walls. You are required to place n archers on this grid such that they cannot shoot each other. An archer can shoot up, down, left, right and also diagonally and its shoot can reach at most w locations in all directions, up to the grid edges.

i)Write a detailed formulation for this search problem.

ii)Identify a search algorithm for this task and explain your choice.

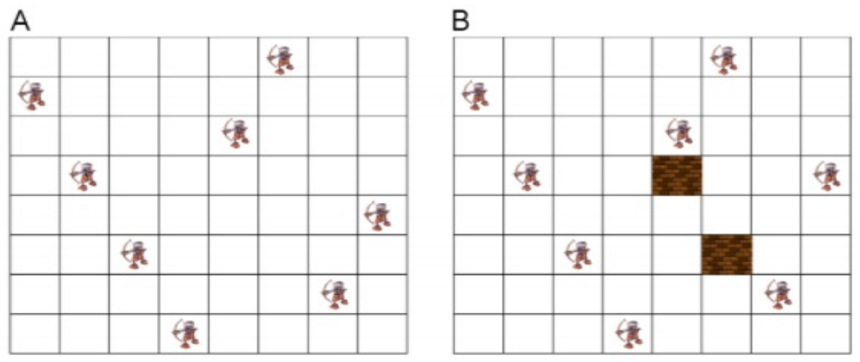


Figure 2: Valid arrangements of archers so that they cannot shoot each other. (A) no walls added. (B) two walls added such that the archer in the last col- umn cannot shoot the archer in the second or fourth column. Note that these solutions do not contain any two archers on the same row, column and diagonal of the grid so they are valid for whatever value of w.

2 Problem Requirements

1. **Write a detailed formulation for this search problem.**

   In this problem our goal is to manage to fit a number of n archers on a grid with the dimensions k xk without them attacking each other.

   The solution that I found in order to solve this search problem follows the model presented in Chapter 5 of the Artificial Intelligence course.

   - States:Any arrangement of 0 to n Archers on the board
   - Initial state:No Archer on the board.
   - Succsesor Function:Add an Archer to an empty field on the board.
   - Goal test:n Archers on the board such that no Archer attacks another
   - Path Costs:0 (we are only interested in the solution).

2. **Identify a search algorithm for this task and explain your choice.**

   For this task I chose to use the Depth First Search(DFS) algorithm in order to implement the problem.

3 Pseudocode

In this section it will be presented the Search Algorithm that I used to solve the problem

The Depth first Search Algorithm is used for finding the solutions of arrangements for the archers and walls on the grid.It is a uninformed search strategy.Tha main idea is to use this algorithms because it explores each path possible(each arrangement of the board) until there are no more succesor nodes(no more solutions available).

3.1 Depth First Tree Search Pseudocode

```
DEPTH_FIRST_TREE_SEARCH($problem, n, R$)
1  while $frontier$
2          $node = frontier.pop()$
3          if $problem.goal\_test(node.state)$
4                  Print the grid solution
5          $frontier.extend(node.expand(problem))$
6  return None
```

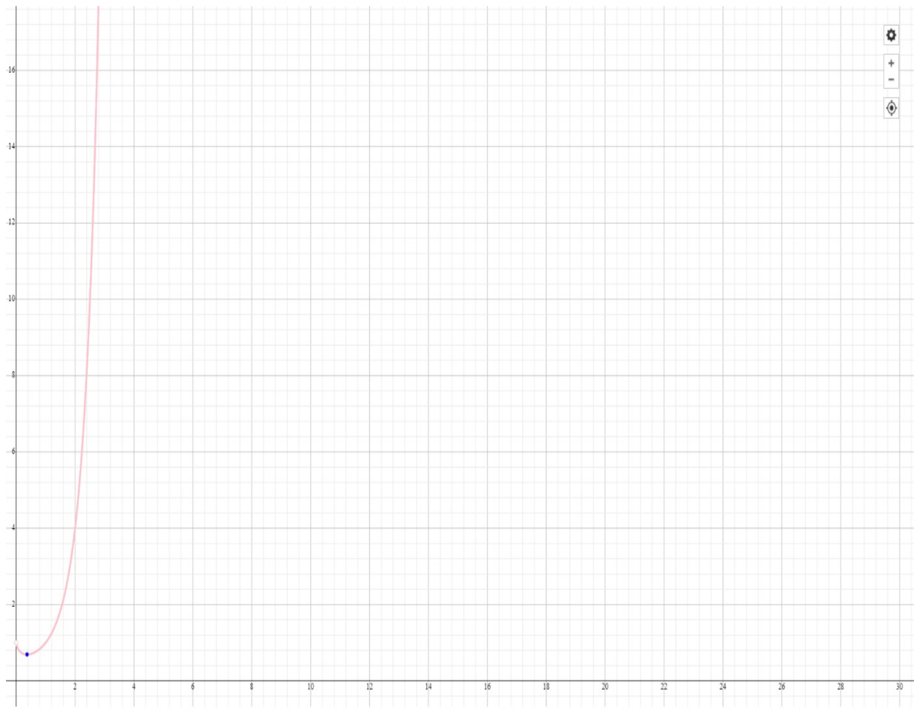
With this search algorithm I was able to find multiple arrangements of archers and walls for given the maximum number of archers that we can put on a grid.

Here we can observe how the time complexity behaves in our algorithm.

Search Space – Search space of our problem consists of a total of N^N states, corresponding to all possible configurations of the N Archers on board.

Therefore, the worst-case time complexity of our algorithm is O(N^N). But, this worst-case occurs rarely in practice and thus we can safely consider our solution as a optimal one.

This is the graph which shows how the O(N^N) behaves.

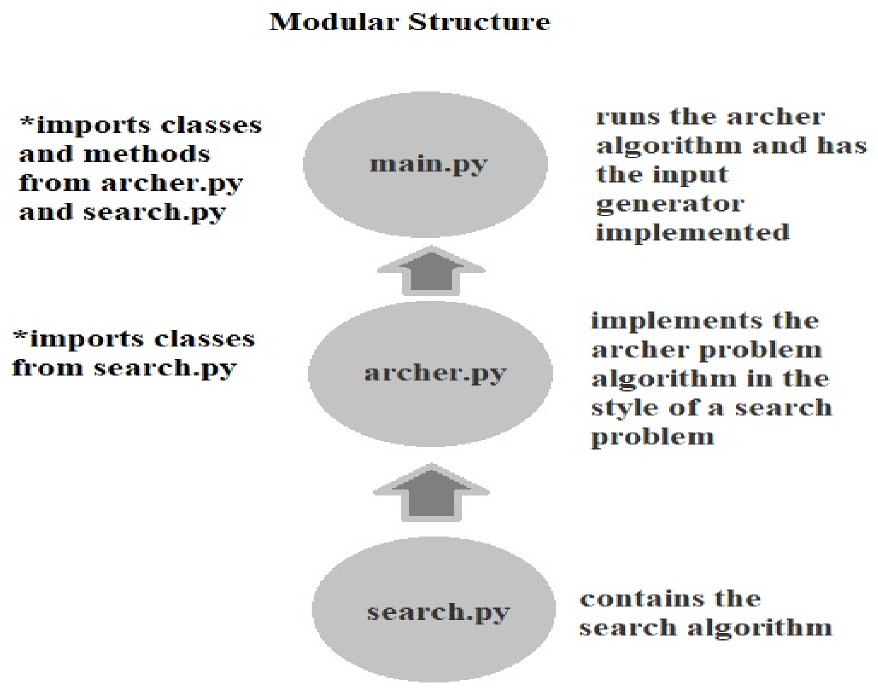


N^N Time Complexity Graph

4 Application Outline

4.1 High Level Application Structure

In this section we review the modular structure of our application on a high level hierarchy.


Modular Structure
*imports classes and methods from archer.py and search.py
main.py
runs the archer algorithm and has the input generator implemented
*imports classes from search.py
archer.py
implements the archer problem algorithm in the style of a search problem
search.py
contains the search algorithm

Our program uses a modular structure so that it can be easily divided in different work files.

All the ".py" files are part of the project structure and are interconnected by importing the classes and methods needed between the files as you can see in the above picture.

4.2 Description of input data

Our input data is relevant to what the algorithm needs to work and contains:

- An integer **N** which represents the number of archers That need to fit on the board.
- An integer **w** which represents the range of the archers.
- A list variable named **restrict** which contains on what positions will the walls that block the archers be placed.
- We also got a list variable named **placement** which initializes the empty board which will be used to place the archers onto.

4.3 Description of output data/results

In the results section it is presented how the output data sets of the problem are being made.

The output data is consists of:

- The number of walls that will be placed on the grid.
- Then for each correct arrangement of archers and walls it will be printed the **pattern** list which holds the position of each archer then the position where the walls take place and then a representation of a grid with the archers and the walls placed on it.The program will display arrangements until there are no more solution to be found.

4.4 List of Application modules

1. The search.py module

   Here we have implemented the all the neccesary classes and methods for the search algorithm.

   The module consists of two classes and two other functions:

   - Class **Problem** (The abstract class for the formal problem.Which will be subclassed and have it's methods implemented.)
   - Class **Node** (Which is a node in a search tree. Contains a pointer to the parent (the node that this is a successor of) and to the actual state for this node.).
   - Function **depth_first_tree_search**(Searches the deepest nodes in the search tree first. Search through the successors of a problem to find a goal.It also acts as our output of data function.)
   - Function **find_walls**(which is a function that helps to identify the positions of the walls in the **restrict** list and is used in the previous function.)

2. The archer.py module:

   In this module we have implemented the algorithm for the search problem.

   The module consists of one Class:

   - Class **ArcherProblem** (The problem of placing N archers on an k xk grid with none attacking each other.)

3. The main.py module

   This module has two purposes:

   - Prepares the input data used for the problem(both manually and random generated)
   - Runs the algorithm function in order to solve the Archer problem.

4.5 List of all classes/procedures/functions used in the application

Some of the parameters listed that are not described here,will be described using comments on the source code

1. search.py

   - Class **Problem(object)**
     The abstract class for a formal problem.

     This Class ia used as a template for the methods we need to create the problem algorithms. **Parameters:**

     – The only parameter is *object* (which is a new style way in python to define a class)

     Methods:

     – $\_\_init\_\_(self, initial, goal = None) : s)$ (The constructor specifies the initial state, and possibly a goal state, if there is a unique goal.)

     – actions(self, state):(Return the actions that can be executed in the given state.)

     – result(self, state, action):(Return the state that results from executing the given action in the given state.)

  - $goal\_test(self, state)$:Return True if the state is a goal. The default method compares the state to self.goal or checks for state in self.goal if it is a list
  - $path\_cost(self, c, state1, action, state2)$:(Return the cost of a solution path that arrives at state2 from state1 via action)
  - value(self, state):(For optimization problems, each state has a value.)

- Class **Node**
  A node in a search tree. Contains a pointer to the parent (the node that this is a successor of) and to the actual state for this node.
  Methods:
  - $\_\_init\_\_(self, state, parent = None, action = None, path\_cost = 0)$:
    Create a search tree Node, derived from a parent by an action
  - expand(self, problem):
    List the nodes reachable in one step from this node.
  - $child\_node(self, problem, action)$:
    Returns the child of the current node

- Function $depth\_first\_tree\_search(problem, archerNumber, restrict)$:
  Searches the deepest nodes in the search tree first. Searches through the successors of a problem to find a goal.
  This is where we also print our output data for each correct arrangement found by going through each node.

- $find\_walls(restrict, archer)$:
  The parameter restrict is a list and it is used to mark where the walls are going to be placed,while the archer parameter it is used to verify if an Archer on the position archer is on the same position as a wall.
  The functions either returns True or False.

2. archer.py

   This module only contains one Class:

   - Class ArcherProblem(Problem):
     The problem of placing N Archers on an kxk grid with none attacking each other. A state is represented as an N-element array, where a value of r in the c-th entry means there is a queen at column c, row r, and a value of -1 means that the c-th column has not been filled in yet. We fill in columns left to right.
     The Class only takes one parameter which is the Class Problem(implemented before in search.py)
     The Methods of class ArcherProblem are:(the parameters and the return values of each method will be described in the source code as comments)
     Methods:
     – $\_\_init\_\_(self, N, w, placement)$:
       it always gets called when a new ArcherProblem object is created
     – actions(self, state):
       In the leftmost empty column, try all nonconflicting rows.

- result(self, state, row):
  Place the next Archer at the given row.
- conflicted(self, state, row, col):
  Would placing an archer at (row, col) conflict with anything?
  this is the first conflicting condition
- conflicted1(self, state, row, col):
  this is the second conflicting condition
- conflict(self, row1, col1, row2, col2):
  it is used in the conflicted method
  Check if no conflicts.
- conflict1(self, row1, col1, row2, col2):
  it is used in the conflicted1 method
  Check if no conflicts.
- $goal\_test(self, state)$:
  Check if all columns filled, no conflicts.

3. main.py
   This is the main module that acts as a start up file,imports classes and functions from the other two modules and has the **input generator** implemented in this way:
   k(the size of the grid is randomly generated between 4 and 10 size[because there are too few solutions for k less than 4)
   w(the range of archers also gets a random int between 0,k-1)
   W(is the number of walls is also randomly generated between 0,k-1)
   we construct the restrict list which holds the position of each wall is placed

Conclusions

Working on this problem helped me understand more about the course and the importance of search problems.

There are multiple new things that I learned while working on this project:

1. First of all,it helped me to improve my coding ability by interacting with new types of ways to solve problems.

   The Artificial Intelligence approach to solving problems is very interesting because it mimics the human thinking and applies it to our algorithms.

   That is because search problems are designed to reach a goal state by modifying,checking,and keeping count of other states.

2. Secondly,I believe another key factor that has improved my ability to solve problems is having to make a report about the problem I am working with.

   Being able to enunciate a problem in a formal way can undoubtedly improve our knowledge about the problem,but also about programming in general.

   The main factors of this are in my opinion The application outline and the experimental data sections.

   The first one has a big role in helping how to structure the algorithm in from modules interactions to classes,functions and other similar things.

   Meanwhile,the latter helps to verify that our problem works on multiple sets of data and the output data is clearly developed to both me and the reader.

4.6 References

1. GeeksforGeeks.com
2. StackOverflow.com
3. The chapters and laboratories of the Artificial Intelligence course .