**CS460 Project 3**

Juanchen Li

December 1, 2012

**Problem 1**

The static evaluation function that developed for the warfare game in state composed of the squares is as follows:

In which is the static evaluation function and is the evaluation function of a particular square. The function is defined as follows:

* If is occupied and all of its adjacent tiles are occupied:
  + If is occupied by MAX, return the value of .
  + If is occupied by MIN, return the value of .
* If is occupied, and not all of its adjacent tiles are occupied consider the subset of tiles within 2 tiles of :
  + If ’s opponent occupies no tiles in the subset, return if ’s owner is MAX, or if it is MIN.
  + If ’s opponent occupies tiles in the subset, let be the number of death blitz moves from by its owners it would need to capture all opposing tiles that could death blitz to . Then, let be the number of death blitz moves by ’s opponents it would need to take and all of the tiles that could death blitz in response.
  + When MAX owns
    - If is negative, return .
    - If is zero, return .
    - If is positive, return .
  + When MIN owns S, return the negative of what would be returned otherwise.
* If is unoccupied:
  + Let be the minimum of the amount of MAX’s tiles that could death blitz adjacent to and the amount of tiles adjacent to that MAX’s tiles can death blitz to. Let be the same as , but calculated in respect to MIN.
  + Return .

What this static evaluation functions does is try to estimate the value of any game state given the current board. When it sees that a square is already fully surrounded, it automatically credits the entire value of to its owner, since at that point it would be impossible to change owners.

In cases where is completely alone in a range of two blocks, or only has allies in those blocks, then ’s value is credited again to its owner. The entire value of is given to its owner even though at this point, it is not certain that will ultimately remain in the hands of the owner. However, if both players were both playing to gain control of only, then whoever owns in this case would indeed be able to maintain it in the end.

In cases where is occupied, but under risk of being death blitzed, the function tries to see who would be able to come out with in the end if both players only focused on taking , and tries to reward the points of to the player that would be able to retain . If it depends on which player is next to move, then the function would not award any points for that tile, since it only knows the current position of the board. The measures of testing how many death blitzes it would take each player to take or retain and remove any threats from was chosen against other possible measures because it proved the most consistent while still being simple and fast.

The static evaluation function also considers unoccupied tiles. It gives up to half of the value of the tile away, depending on which players have the most chances to death blitz that tile, should it be taken. This was simply an intuitive way to evaluate these tiles, since their states still do have an effect on the game state.

Although not perfect, this static evaluation function should preform pretty well. A minimax player using this function would value being able to secure its own high value tiles while setting up opportunities to blitz tiles that the opponents have. The player would spend its resources maximizing the tiles that they have completely secured and lowering the opponent’s defenses while being aware of the values of the tiles it has not secured yet, rather than simply trying to reach a local maximum score at every turn.

Right now, the weight for an occupied, but not yet completely secured tile is 1, and unoccupied tiles use 0.5. These weights were simply produced by think about the problem intuitively. However, they may be slightly off from the actual optimal weights for this problem, but to discover these weights would require further testing in which the game is implemented and played through minimax.

Here are some visual examples of how the function handles certain cases based on the rules given. The MAX player uses X and the MIN player uses N. For these examples, assume all tiles are worth 1 point.

N N N N

--- N-N --- ---

--X-- --X-- XXXXX NNXNN

--- --- XXX NNN

- - X N

In all of the above examples, the X in the middle of the grid would evaluate to zero. This is the intention of the function, that even though there are a variety of situations presented, the X in the middle can always be safely taken by MIN in a single turn, while MAX can always get rid of the rest of the N’s that threaten the middle X in a turn as well. This shows that the function considers the relative safety of the tile rather than simply how any other tiles are close by.

N N

--- ---

--X-- --X--

--- ---

X N

In the first above example, the function would give 1 to MAX. MIN would have to death blitz down twice in order to secure the middle, while MAX only has to death blitz up once. The second example would give –1 to MAX, since this time, it has two attackers that it must take care of separately, while MIN could take the middle tile in a single turn. Thus even though it is occupied by MAX, the score is credited towards MIN because of positioning.

- X -

X-- X-X -X-

----- X---X -X-X-

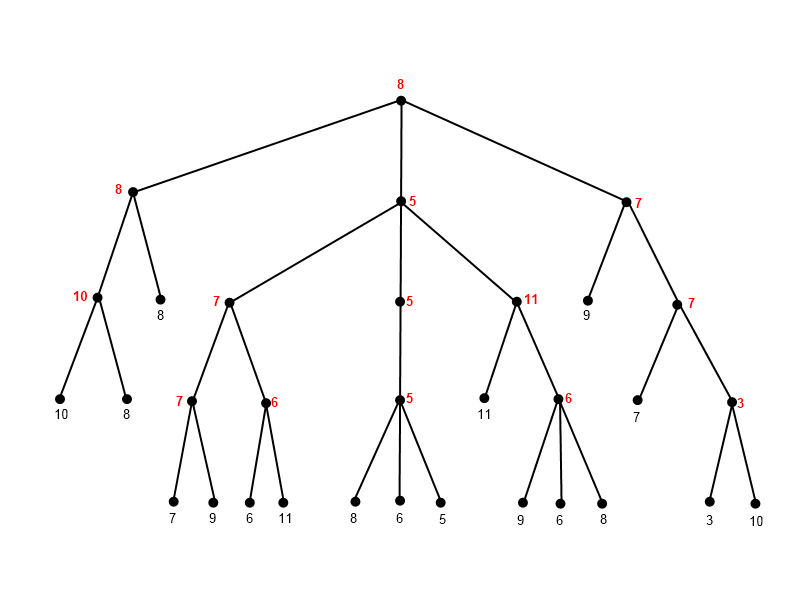
--- --- -X-

- - -

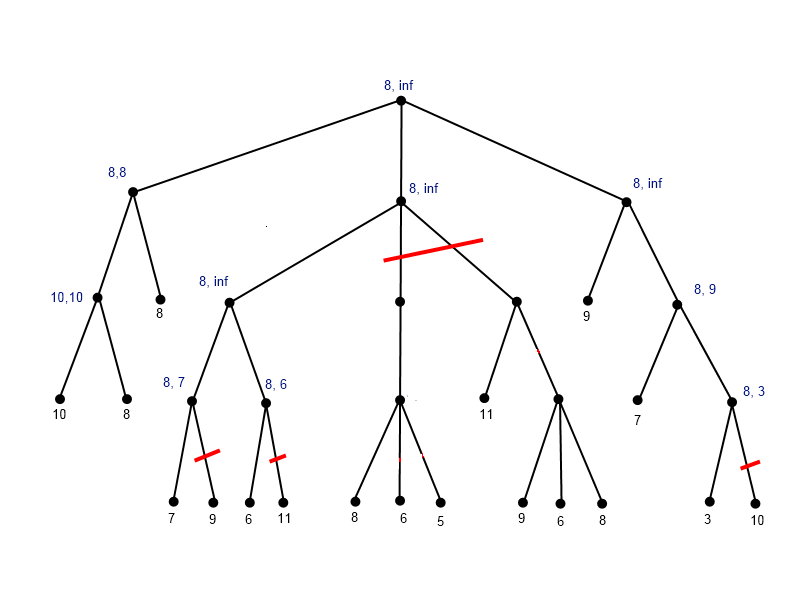
In the first example, MAX gets 0.125 for the tile, since it has one attacker even though it covers two spaces. In the second, max gets 0.375, since while there are now five attackers, it only covers two tiles. In the final example, no points are credited, since the point can only be captured by para drop. While the evaluation for empty spaces does not perfectly evaluate all situations, it provides a dependable estimate for most situations.

**Problem 2**

Minimax Search



Alpha Beta Pruning



The alpha and beta value shown are the last discovered ones for when a node is still relevant. If the alpha beta pruning realizes that a part of the tree no longer needs to be expanded, the alpha and beta values are left what they last were. These values are different from their minimax values. After the last node is cut, this alpha beta stops, but it could have propagated its final answer of 8 up, which would slightly change a few of the alpha beta values. I apologize for any inconvenience if that was the preferred method of reporting the values.

**Problem 3**

The original algorithm implemented to generate puzzles is a genetic algorithm:

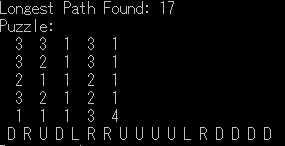
1. The algorithm takes in the parameters given by the user, and generates 8 completely random puzzles as its population. These 8 serve as the beginning population.
2. The algorithm finds how many moves it takes for every member of the population to be solved, and uses the number as a fitness measure to split the group up into the 4 fittest and the 4 least fit.
3. The 4 fittest members stay in the population, while the 4 least fit members are deleted.
4. From the 4 remaining members—which will be called A, B, C, D—4 children are produced, from AxB, BxC, CxD, and DxA.
   1. To produce a child, the algorithm first chooses a randomness of between 0% and 90%. This randomness dictates what percent of the children’s values are mutations rather than inherited from the parents.
   2. From all of the non-mutated values, they each have an equal chance to inherit the value from either parent.
5. The fitness of the new children is calculated by seeing how many steps it takes to solve the puzzle.
6. The total population is again split into two groups of 4 depending on their fitness values.
7. The algorithm then goes back to step 4, unless it realizes that it ran out of time.

When initially evaluating the problem, it appeared that it had many complexities, and that it would have been best to use a technique that would be able to find better puzzles while not necessarily needing to know particular strategies that create these paths. By using a genetic algorithm, it was anticipated that the paths that made certain puzzles hard to solve would be preserved by some children. By manually going and seeing the puzzles and their lengths at every generation, it does appear the genetic algorithm moves towards a maximum. The maximum length increases over the generations, and when it does, it was usually by one for higher levels.

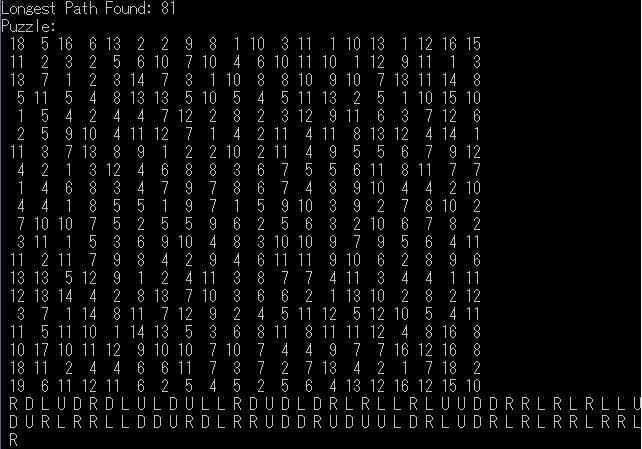
However, it seems that the genetic algorithm works at a slow rate for the larger puzzles and is somewhat luck based, where it became difficult for children to be better than their parents. This makes sense given the puzzle and algorithm, as most children will drastically change the paths taken by both parents, it seems that best optimization happens hear the start and end nodes. If the algorithm had ample time, it would still eventually find the global maximum, but can be underwhelming in short periods of time.

With this information in mind, it seemed like it would have been better to implement an algorithm that did hillclimbing by editing very few cells of the puzzle at each step, so that its main path would be primarily maintained between generations. Unfortunately, due to time constrains, I was not able to completely implement such an algorithm. However, by changing each child from using two parents (AxB) to a single parent (AxA) and lowering the randomness cap from 90% to 20%, it was possible to somewhat simulate such hillclimbing, which led to somewhat better results: around 90 for the 20x20 1—20 and 120 for 20x20 5—20. However it is still far from the ~200 that could be possible by forcing the player to zigzag 1’s like in the solution for the 5x5 puzzle. Nevertheless, the original genetic algorithm is submitted, since the logic in the algorithm is built around it. The puzzles below were generated by the genetic algorithm.

5x5 1—5



20x20 1—20



20x20 5—20

