CS51 Project Draft Specification

Solving Sliding Tile N-Puzzles With Genetic Algorithms and Other Search Heuristics

Eric Chan ericchan@college.harvard.edu
Liam Mulshine lmulshine@college.harvard.edu
Luca Schroeder schroeder@college.harvard.edu
Ezra Zigmond ezigmond@college.harvard.edu

10 April 2015

1 Brief Overview

In this project, we hope to gain an understanding of genetic algorithms by applying them so solve N-Puzzles. N-Puzzles (or sliding tile puzzles) are a popular way to compare and understand different search heuristics. Solving a planning problem like N-Puzzles using genetic algorithms presents an interesting challenge because each move significantly affects the valid moves at the next steps, and thus chromosomes in the algorithm must be interpreted in the context of the current state. We aim to implement a genetic algorithm for solving N-Puzzles and compare the results with implementations of against a more conventional approach to solving N-Puzzles using A* and search heuristics such as the Manhattan Distance and Pattern Database heuristics. The goal is to learn about genetic algorithms and search problems, and likely learn a new programming language along the way.

2 Feature List

2.1 Core Features For Minimally Complete Project

- **Generate solvable N-puzzles** (half of all possible N-puzzles are impossible to solve sufficient condition for solvability outlined in this paper)
- **Implement the genetic approach to solving N-puzzles outlined in this paper.** This uses indirect encoding for genes, meaning we conveniently do not have to use a match fitness function in our fitness evaluation.

- **Implement the Manhattan distance/taxicab geometry heuristic using an A*/IDA* algorithm** Manhattan distance is explained here, and this paper discusses possible implementations, including DFS/BFS search, best-first search and depth-first iterative deepening which combines all three.
- **Create graphs comparing the effectiveness of the algorithms** so that we can easily present the results that we find.

2.2 Cool Extension Ideas

- **Some sort of visualization** displaying a randomly generated N-puzzle and then displaying an animation of the solutions given by different algorithms along with a scoreboard of their times. Could even have the user try (unsuccessfully most likely) to beat the solutions generated by different heuristic techniques!
- **Implementing the Linear Conflict Heuristic outlined here.**
- **Incorporating a recursive planning heuristic** into the genetic algorithm as outlined in this paper for dramatic improvements to performance and the success of the genetic algorithm in solving larger N-puzzlesi.e. for a 15-puzzle (4x4), solve the fourth column and fourth row (now reducing the problem to a 8-puzzle), then solve the remaining third column and third row tiles, etc. Implementing this recursive planning heuristic would also implement the Pattern Database heuristic introduced in this paper.

3 Technical Specification

Right now, it made the most sense to us to take an object-oriented approach and therefore divide the different parts of the problem into classes. However, we realize that the genetic solver will need to keep around an immutable copy of the original puzzle, so it may ultimately be better to implement the algorithm functionally so that we dont have to worry about the changing state of puzzles. Something else were still thinking about is whether it makes sense for a puzzle to compute its own fitness, so that we dont have to give outside access to the internal tile state, or whether the solver should be able to look directly into the puzzle and take care of fitness evaluation.

Below, we provide pseudo-ML style signatures for our classes and methods with high level explanations of the way the functions work.

```
(* gives an abstract way of representing directions to move *)
type Direction = Left | Right | Up | Down

(* Class for representing tile puzzle *)
class Puzzle
  instance variables:
```

```
  (*dimension of the board*)
  val size : int
  (* containing all of the tiles in order. We think that
   * using an int list will be easier to work with instead of an int
       list list*)
  val tiles : int list
  (* keeps track of where the empty tile is *)
  val empty: int
  (* keeps track of the last move made *)
  val last_move: direction
  (* keeps track of what the solved state of the puzzle should be *)
  val solution: int list

  public methods:

  (*takes in the size and returns a new puzzle*)
  initialize: int -> Puzzle
  (* takes in a list of ints and creates puzzle with specified tile
     order
   * will include check to see if created puzzle is solvable*)
  initialize: int list -> Puzzle
  (* mixes up the tiles randomly until a solvable permutation is found
     *)
  shuffle: void -> void
  (* slides the puzzle in the specified direction, if move is valid *)
  move: direction -> void
  (* returns true if current arrangement matches solution *)
  solved?:void -> bool
  (* checks if a list of moves will solve the puzzle *)
  check_solution: direction list -> bool
  (* returns a list of valid moves from current arrangment *)
  valid_moves: void -> direction list
  (* uses the provided heuristic function to return a
   fitness score representing how close the puzzle is to being solved
       *)
  fitness: (heuristic function) -> int

  private methods:
  (* determines if current permutation can be solved *)
  solvable?: void -> bool
  (* swaps tiles around (should only be called if direction is a valid
     move*)
  slide: direction -> void
end

class GeneticSolver
  (* Abstract type to represent an individual solution candidate *)
  type chromosome : float list
  (* Abstract type to represent a collection of solution candidate *)
  type population : (float list) list
```

```
  instance variables:

  (* initial puzzle we are solving *)
  val puzzle: Puzzle

  public methods:

  (* initializes a genetic solver with a given puzzle *)
  initialize: Puzzle -> GeneticSolver
  (* Runs genetic algorithms and returns a list of directions
   * to solve if a solution was found, None otherwise. *)
  solve: void -> (direction list) option

  private methods:
  (* takes in a population size and the length we want each chromosome
     to be
   * and randomly creates a starting population *)
  generate_population : int -> int -> population
  (* takes in two chromosomes and performs a crossover operation to
   * produce a new one *)
  crossover : chromosome -> chromosome -> chromosome
  (*  randomly alters a chromosome *)
  mutation : chromosome -> chromosome
  (* evaluates fitness of population, generates crossovers and
     mutations
   * for next generation *)
  selection : population -> population
  (* contextualizes chromosome for specific puzzle state, apply move
     list
   * to puzzle, evaluate the fitness of the resulting partial solution
      *)
  fitness : chromosome -> int
  (*takes in 4 parameters: number of phases to run, number of
     generations in
   * each phase, number of chromosomes in each population, and length
      of chromosome
   *. Will keep running phases until a solution is found or the phase
      count reaches 0 *)
  run_genetic_alg : int -> int -> int -> int -> (direction list) option
  (*  takes in the number of generations to run, returns a direction
     list if it finds
   * a solution, None otherwise *)
  run_phase : int -> (direction list) option

end

(* A* Implementation: *)
class AStarSolver

  Instance Variables:
```

```
  (* space containing the list of puzzles that are within one move of
     the
   * current state of the puzzle (discluding puzzle that would bring it
       back
   * to the previous state *)
  mutable open_space: Puzzle list
  (* moves made up to current state *)
    mutable moves : Direction list
    (* space containing the list of puzzles that correspond to the best
        choice
    * puzzle at each step in the solving process  (based on heuristic
       score) *)
    mutable closed_space: Puzzle list

  Public Methods
  method run_solver: puzzle -> Direction list

end
```

4 Next Steps

- **Think more about which language we will use.** Right now, we are still deciding if taking an object-oriented or a more purely functional approach to the problem would make more sense. If we decide to go for a primarily object oriented approach, we might use a high level language like Ruby or Python. If we instead decide to aim for a mostly functional approach, we might want to try out a new functional language like Clojure or Scheme. However, another interesting language we have considered is Scala, which (like OCaml) has good support for both functional and object oriented paradigms and would allow us to try to use as much functional code as possible. Choosing a language will likely involve setting up several different environments and playing around to see what we like.

- **Clarify our understanding of the algorithm.** Although we mostly understand the genetic approach that we plan to implement, there are still . We will try to seek out additional resources to clarify our understanding, or perhaps try to go over parts of the paper with our advising TF to see if he might be able to clarify some of our questions.

- **Think about what graphing library we will use.** We already know that Python, through Matplotlib and NumPy has very good support for graphing, but when we decide on a language, we will investigate if there are good plotting libraries for that language. Otherwise, we could certainly output our results as test and feed them into a Python program to plot the results.