Mini-Project: Finding the Minimum Cost Path in an $N \times N$ Grid

Evan Hataishi
ICS 674: Evolutionary Computation
Prof. L. Altenberg

February 18, 2020

1 Introduction

Minimum cost path is a classical problem in computer science. This problem has many variations but often uses a 2D matrix/grid representation: given an $M \times N$ matrix, source cell S, and destination cell D, find the shortest distance from S to D by moving up, down, left, and right to adjacent cells.

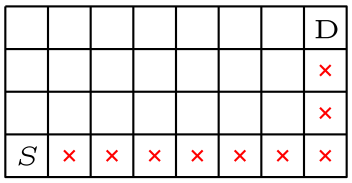

Figure 1: A 4×8 grid representation with 10 movements. $S = (0, 0)$ and $D = (7, 3)$.

A solution to the minimum cost problem is to simply apply either breadth-first or depth-first search from S. For any matrix or directed acyclic graph, finding D will trivially yield the shortest path as well. Although finding the minimum cost path for an $M \times N$ matrix can be found in $O(M \times N)$ time using a BFS, we would like to explore the application of an evolutionary algorithm to this problem.

1.1 Problem Statement

For the scope of this work, we add several constraints to the original minimum cost problem.

1. Consider only square matrices i.e. $N \times N$ matrices.
2. S and D are fixed at $(0, 0)$ and $(N - 1, N - 1)$ respectively.
3. Possible movements from one cell to the next are up, right, diagonal, and none.
4. All movements between adjacent (including diagonally adjacent) cells incur the same cost.

Constraint (2) implies that S will always be the bottom-leftmost cell and D will always be the upper-rightmost cell. Notice that for any $M \times N$ matrix, every path consisting of $M - 1$ movements up and $N - 1$ movements to the right will be a minimal path. In fact, if we are to consider only up and right movements between cells, every possible path will be the same length: $M + N - 2$. Therefore, constraint (3) is required to make this problem more interesting. By adding in a diagonal movement, we can reduce the cost of any up and right movement by 1. For any square matrix, the minimal path will always consist of exactly $N-1$ diagonal movements through every cell (i,i). The first diagonal movement goes to $(1,1)$, and the last diagonal movement goes to $(N-1,N-1)$. Figure 2 shows a square matrix with its optimal path in red.

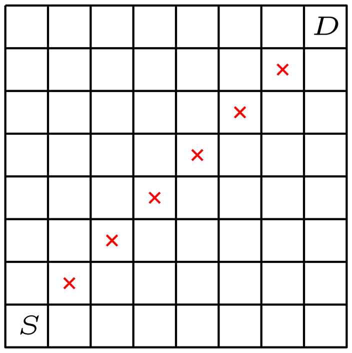

Figure 2: An 8×8 grid with a 7 move all-diagonal path.

Using an evolutionary algorithm, we would like to see a convergence to a path of all diagonals for any arbitrarily sized square matrix in order to solve this version of the minimum cost problem.

2 Evolutionary Algorithm

Search Space All valid [1] sequences of up, right, diagonal and none movements.

Representation A $2N-2$ length [2] vector containing elements "U", "R", "D", and "N".

Objective Function Let x be the sequence of $2N-2$ numbers representing a path's x-coordinate at every step, and y be the sequence of $2N-2$ numbers representing a path's y-coordinate at every step. Let $\hat{x}$ be the optimal sequence of $2N-2$ x-coordinates s.t. $\hat{x} = 1, 2, \ldots, N-1$ and $\hat{y}$ be the optimal sequence of $2N-2$ y-coordinates s.t. $\hat{y} = 1, 2, \ldots, N-1$.

$$f(x, y, \hat{x}, \hat{y}) = \sum_{i=1}^{2N-2} \sqrt{(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2}$$

Therefore, we aim to minimize the fitness value, with **zero** being the optimal fitness value.

Variation Operators Recombination is done by randomly picking a movement type between two paths at **every** step to form a single new path of length $2N-2$. Recombination is only done using parents in the top 50% of fitness. Mutation has a 10% chance of occurring on any path. The path is simply randomly generated from scratch if selected for mutation.

Selection Operator We use elitism to pick the top 30% of a generation to move on automatically to the next generation. The other 70% is formed from recombination.

All paths are generated randomly at the start. If a movement in a path will take the path outside the grid, we simply fill the rest of the path with up or right movements in the direction of D. For

[1] Paths cannot go outside a matrices dimensions

[2] A path will be exactly $2N-2$ moves when only upward and rightward movements are chosen. This implies that not all paths will reach the destination if a "none" step is used, and paths containing diagonals may finish early.

example, if a path reaches the top of the grid, we fill the rest of the path with right movements. If a path reaches the right edge of the grid, we fill the rest of the path with up movements. If a path reaches D prior to $2N-2$ movements, we fill the rest with no movements. Therefore, the optimal path will have $N-1$ "D" movements followed by $2N-2-N-1=N-3$ "N" movements.

3 Code

To avoid having code within this report, all code used for this project can be found on my GitHub in a public repository [3]: `https://github.com/evan-hataishi/ics-674/tree/master/mini_project`. `genome.py` contains a class `Genome`. All methods related to my genome representation are implemented within that class. `main.py` is a driver and runs the genetic algorithm. Within `main.py`, we first define some hyper-parameters: `GRID_SIZE=100, POPULATION=30, GENERATIONS=2000, SELECT_PERC=0.3`. The agents are then initialized and evolved within the `ga()` function.

3.1 Search Space Representation

`Genome` has an attribute called `__path`, which represents the sequence of up, down, diagonal, and none movements that genome makes. This path is initialized randomly within the genome constructor: `__init__(self, GRID_SIZE, path=None)`. After validating the path, the genome's fitness is also calculated by the `calc_fitness(self)` method.

3.2 Variation Operator Encoding

Crossover is implemented as a static method: `crossover(p1, p2)`. Given two parents, it produces a single child by picking each step randomly between p1 and p2's paths. Mutation is implemented in the `mutate(self)` method. It simply generates a new random path.

3.3 Selection Operator Encoding

Selection is done in `main.py`. All agents are sorted by fitness, and the top 30% are selected to move on to the next generation.

3.4 Termination Criterion

We only terminate when the optimal path (fitness=0) is reached. This does not typically occur within the first 2000 generations.

4 Results

Figure 3 shows the best, average, and worst genome fitness in the population over 2000 generations. Since the optimal fitness value is zero, and larger fitness values mean a "worse" genome, we negate every fitness value to obtain a more intuitive graph with zero at the top. Results seem fairly typical for a genetic algorithm. Average and worst fitness values are messy and have a high variance across generations. The best fitness values converge fairly quickly. After around 1000 generations, the best fitness is around -300. Improvements slow down after the first 1000 generations, but usually end up between 0 and -100 after the full 2000 generations. For a 100×100 grid, any fitness value

[3] All related code will not be modified after February 18, 2020 at 12:00 PM. This is verifiable via commit history.

under 100 is very good considering the worst fitness values may reach -10,000. This behaviour is fairly consisten over multiple executions of the evolutionary algorithm.

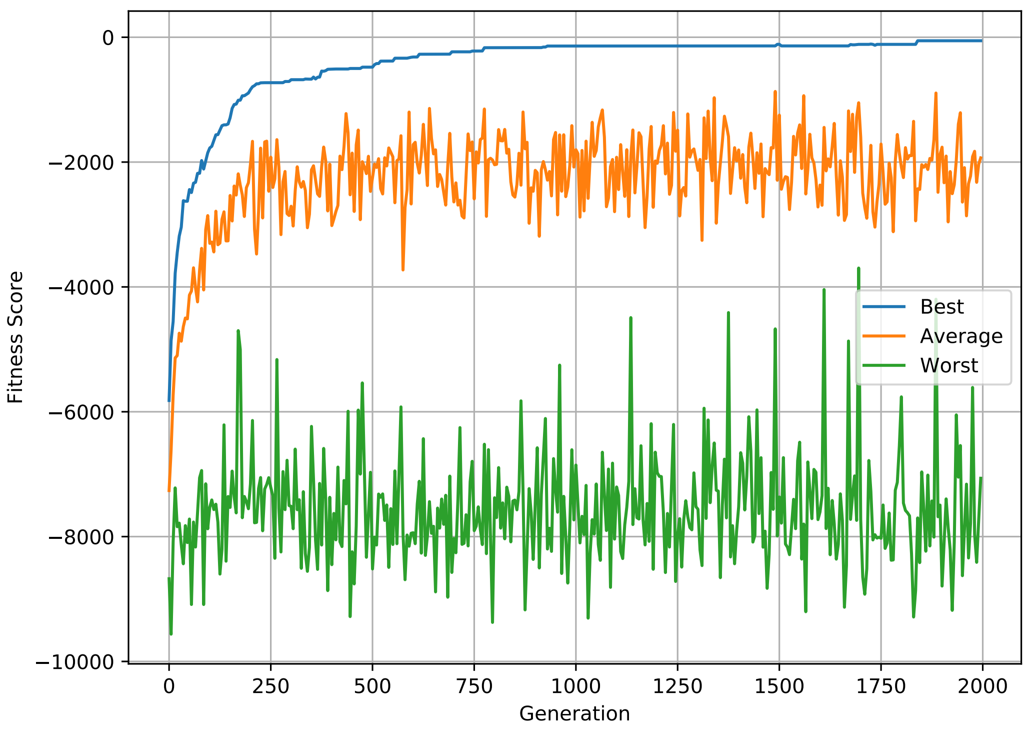

Figure 3: Fitness results over 2000 generations.