

## Project 3: *Solving the 8-Puzzle with Genetic Programming*

---

**Assigned: Tuesday, March 7**  
**Part I Due: Wednesday, March 15, 23:59:59 (i.e., before midnight)**  
**Part II Due: Thursday, March 30, 13:00:00 (i.e., 1:00 PM)**

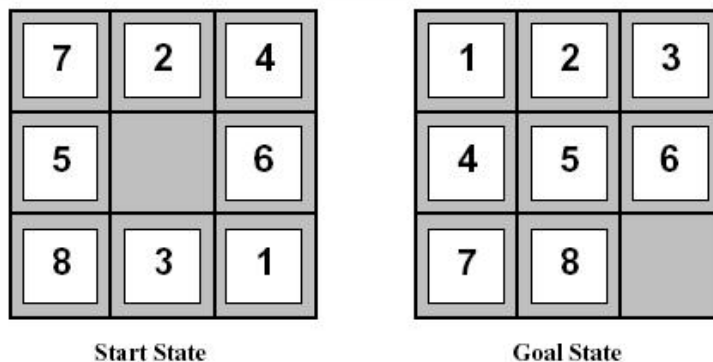
**Note on teaming:** For this project, you are allowed (and encouraged) to work in teams of **up to 3 students**. (But, you are allowed work alone if you prefer.) All students on a team will receive the same grade, so it is up to you to choose your teammates wisely, and to ensure that each team member contributes as equally as possible to the project. The team will turn in 1 paper and 1 set of software. If a team includes both graduate and undergraduate students, the project will be graded at the graduate level. It is expected that the quality of the team project should be higher than that of a team size of 1. Also, if any member of the team has already used their late lab, then this current project can't be used as a late lab for any member of that team. Part I of your project requires that you declare your team members and describe your design; you are not allowed to change your team composition after the Part I due date.

**Note on two parts:** As you've seen, this project has two parts, each with a separate due date (see header for dates). Part I counts as 25% of your project grade, with Part II counting 75%. See the instructions below for what is due for each part.

### Introduction

In this project, you will write a genetic programming program (called *GP-generate*) that learns how to solve the 8-Puzzle. You probably are familiar with the 8-puzzle. This puzzle is a "toy problem" in Artificial Intelligence, meaning that it isn't a real-world problem with deep importance, and it is much simpler than most "real life" problems because all the factors that affect the problem are known in advance (i.e., no uncertainty or unpredictability). But toy problems are useful for learning because it allows us to explore AI and machine learning approaches on a well-defined problem, and lets us test out new ideas for search and learning algorithms.

The 8-puzzle is a small board game for a single player, consisting of 8 square tiles numbered 1 through 8 and one blank space in a 3 x 3 grid. Moves of the puzzle are made by sliding an adjacent tile into the position occupied by the blank space. This move has the effect of exchanging the positions of the tile and the blank space. Only tiles that are horizontally and vertically adjacent to the blank space can be moved. While the goal state can be defined as any arbitrary sequence, it is usually defined as all numbers being in order, starting from the left top. The figure below shows an arbitrary start state and the goal state we will use for the 8-puzzle:



Here is a URL for an applet for the NxN-sliding tiles-puzzle, if you want to study this problem by experience: <http://www.cut-the-knot.org/pythagoras/SLIDER.shtml>. (The default is a 15-puzzle (i.e., 4x4); you can change the size as one of the options.)

### Some suggestions for how to formulate the solution

One way of thinking about the actions in this puzzle is in terms of where the blank square should move. In the above start state, if you were to move square 2 down, the result is that the blank square moves up (effectively exchanging positions with square 2). So, instead of viewing this action as “move 2 down” you can instead call the action “up”, indicating where the blank square is going. This allows your actions to be independent of the specific tile being moved. Other examples in the above start state: “move 5 right” becomes “left”, “move 3 up” becomes “down”, and “move 6 left” becomes “right”.

In this puzzle, you can also consider treating an “illegal” move as a NOP (No Operation), rather than returning failure. For example, in the above start state, if you gave the actions “up up”, the 2<sup>nd</sup> “up” would have no effect, since the blank square can’t move outside of the puzzle.

So, how exactly do you solve the 8-puzzle? Lots of people have studied this problem; the typical approach includes the definition of heuristics that measure the “quality” of a puzzle state, loosely defined as how close a particular puzzle state is to the goal state. If you can measure the quality of a puzzle state, and compare it to the quality of the puzzle state that results after you perform a particular tile move, you can determine whether that tile move is a good action to take or not.

A paper by Finkelstein and Markovitch (referred to later as F&M) (called “A Selective Macro-learning Algorithm and its Application to the NxN Sliding-Tile Puzzle”, in *Journal of Artificial Intelligence Research*, volume 8, 1998, pages 223-263, available online here: <http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume8/finkelstein98a.pdf>) discusses a macro-learning algorithm for the NxN sliding-tile puzzle<sup>1</sup>. They found the following metrics to be useful in the formation of heuristics for these puzzles (pages 236-237, 256-257 of F&M):

- *Placed(s)*: For a particular board configuration  $s$ , starting from the upper left corner, count tiles until you find a tile out of place. *Placed(s)* is how far you get in your counting before you found a misplaced tile. For example, in the above start state, the value of *Placed(s)* is 0, because the very first tile (7) is not in the correct place.
- *Misplaced(s)*: This metric equals  $9 - \text{Placed}(s)$ . For example, in the above start state, the value of the *Misplaced* metric is 9.
- *ManhattanOfFirstMisplacedTile(s)*: Manhattan distance of the first tile that is not in place in board configuration  $s$ . For example, in the above start state, “1” is the first misplaced tile, and it is a Manhattan distance of  $2 + 2 = 4$  from its goal position, so the value of this metric is 4. If all tiles are in place, the value of this metric is 0.
- *ManhattanOfFirstFromBlank(s)*: Manhattan distance of the above tile from the blank space. For example, above, the Manhattan distance of “1” from the blank space is  $1 + 1 = 2$ , so the value of this metric is 2. If all tiles are in place, the value of this metric is 0.

Based on these metrics, Finkelstein and Markovitch found a good heuristic function,  $h(s)$ , that allows us to measure the quality of a particular board configuration  $s$ , relative to a given goal configuration (page 257 of F&M). I’m simplifying it a bit here for a board size of  $3 \times 3$ , and assuming the standard goal configuration given in the example on page 1:

$$h(s) = 36 \cdot \text{Misplaced}(s) + 18 \cdot \text{ManhattanOfFirstMisplacedTile}(s) + \text{ManhattanOfFirstFromBlank}(s)$$

With this heuristic, lower values are better. Let’s apply it to the two board configurations on page 1 and see what we get. The start state has an  $h(s)$  value of  $36 \cdot 9 + 18 \cdot 4 + 2 = 398$ . The goal state has an  $h(s)$  value of  $36 \cdot 0 + 18 \cdot 0 + 0 = 0$ . Whew! At least this is consistent with what we would expect.

Finkelstein and Markovitch defined a basic set of operators, which they called  $B$ , corresponding to the moves of the blank space (i.e., up, down, left, right), as follows:

$$B = \{u, d, l, r\}$$

Finkelstein and Markovitch also found a set of “macros” (see page 257 of F&M), which they found useful for solving these puzzles. Here is the set of macros that they defined, which they called  $M$ :

---

<sup>1</sup> It isn’t necessary to look up this paper or read it. I’m just providing the reference here, in case you are curious and want to study these ideas further.

$M = \{lur, rul, uld, ruuld, dllur, drrul, urrdluld, urrdluld, uuldrdluurd, lurrdluld, urdrrullldrrur, urdrrullldrrurd, llurdrrullldrrurd, uldllurdrullldrrurd\}$

For example, the first macro, *lur*, refers to a sequence of moves consisting of “left”, then “up”, then “right”.

Finally, Finkelstein and Markovitch proved (page 256 and following) that it is always possible to find an operator  $o$  from either  $M$  or  $B$  that reduces the heuristic function value. That is, you can always find operator  $o \in B \cup M$ , applied to board configuration  $s$ , such that  $h(o(s)) < h(s)$ .

### What does this all mean for your GP task?

If you have another idea for how to solve this puzzle, you may certainly go in a different direction. Using the theory of Finkelstein and Markovitch is just a suggestion. But, based on the theory of Finkelstein and Markovitch, it should be possible to create a human-engineered program that solves the 8-puzzle by doing the following:

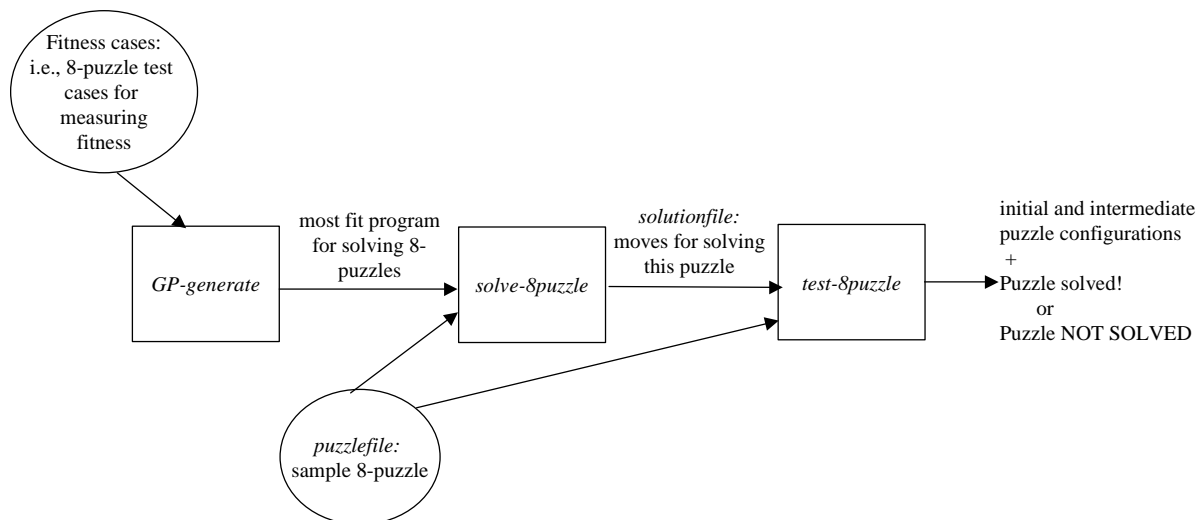
For a starting board configuration  $s$ , calculate the quality of this configuration,  $h(s)$ . Then, for each operator in  $M$  and  $B$ , calculate the quality of the resulting board configuration that would result from applying that operator. Select an operator that reduces the heuristic measure of the board. Repeat until the puzzle is solved.

So, your task is to formulate a GP learning algorithm that discovers such a program (or perhaps discovers another approach). As you know, for any genetic programming task, you must define the following:

- Terminal set (Your terminal set will probably at least include “u”, “d”, “l”, and “r”, plus perhaps terminals corresponding to the Finkelstein and Markovitch macros in  $M$ .)
- Primitive function set (The earlier discussions should give you ideas, although you should feel free to take another approach if you have a different idea on how to solve the puzzle. Be sure to encapsulate as much functionality into each primitive function as possible, to help reduce the search space of your problem.)
- Fitness cases (i.e., the test cases you will use to judge the fitness of your hypotheses. This will probably be a set of randomly generated board configurations. *But see caveat below in section called “Generating test cases”.*)
- Fitness function (i.e., the function that will measure the fitness of an individual hypothesis)
- Method of probabilistic selection (e.g., fitness proportionate selection)
- Parameters: population size, fraction of population to be replaced by crossover, maximum depth of members in initial population, maximum depth for trees created by crossover, probability of mutation, and perhaps other design-specific parameters.

### Programming Steps

This project has several pieces, as illustrated in the figure below:



In this project, you will create solutions for all of these pieces, except for the *test-8puzzle* program, which is provided for you (see description below). In particular, the two programs you will write are as follows:

1. *GP-generate*: A program that uses genetic programming to generate a program that solves 8-puzzle problems. You will be testing the fitness of the individuals in the program population using the fitness cases that you generate. The resulting “best fitness” individual program should work for any valid 8-puzzle, not just a single given puzzle. This program outputs (in any form you like), the best individual program for solving 8-puzzles.
2. *solve-8puzzle*: A stand-alone program that inputs the best fitness program (generated by your *GP-generate* program), and “runs” it on the sample 8-puzzle (provided in file *puzzlefile*) to generate a series of moves (output in the file *solutionfile*) that (hopefully) solves the sample puzzle. Note that “running” your best fitness program simply means that you parse/interpret the solution tree according to the meanings you place on your primitive functions and terminals. The output of “running” your best-fit solution tree is a series of moves that are represented by the characters “u”, “d”, “l”, “r”, consistent with the description of *solutionfile* described for the test-8-puzzle utility below. See description below for more details on the formats of the *puzzlefile* and the *solutionfile*. [Note: you will need similar functionality for your *GP-generate* program, since you’ll need to test the fitness of the individuals of your population. This stand-alone version is so that your resulting “most fit” function has life beyond your *GP-generate* run, and can be tested on specific 8-puzzle examples later.]

### Generating test cases

Theory on the 8-puzzle has proven that the only puzzles that are solvable are those with an even number of transpositions (if you’re interested, the earlier referenced URL has a short discussion on this point). A transposition is a swapping of 2 adjacent elements. When counting transpositions for the 8-puzzle, you check to see how many transpositions (one after the other) are needed to get the puzzle to the goal configuration. [Note: a transposition is not a “move”. In a transposition, we just magically swap 2 adjacent elements, and keep swapping until everything is in order.] In the example state given on page 1, the number of transpositions needed is 16. (Check it out for yourself; I believe you should get 4 transpositions to move “1” to the upper left, 1 transposition to move “2” to the 2<sup>nd</sup> position (working from the result of the 1<sup>st</sup> transposition), 3 transpositions to get “3” to the correct spot, 3 transpositions for the “4”, 2 transpositions for the “5”, 1 transposition for the “6”, 2 transpositions for the “7”, and 0 for the “8” and the blank.).

So, when you generate test cases, you need to be sure you don’t generate an impossible puzzle. An example impossible puzzle would be the goal configuration with exactly two of the tiles swapped (for example, the 7 and the 8). Such a puzzle would have 1 transposition, which obviously isn’t even, and thus isn’t solvable. The *test-8puzzle* program provided for you has code to generate a valid random puzzle that is guaranteed to be solvable. The way it works is to start from the goal state and make a series of random valid moves to shuffle the board to some other configuration. Since all these moves could be reversed, we know the resulting puzzle must be solvable. You may reuse this code directly in your project, if you like.

### Provided for you: *test-8puzzle* program

A program called “test-8puzzle”, including the source code (*test-8puzzle.cpp*) and a header file (*puzzle.h*), is provided for you in directory `~parker/courses/cs594ml/Project3`. Some sample puzzle and solution files are also provided (one pair that works: “works-puzfile2” and “works-solfile2”, and one pair that doesn’t work: “doesntwork-puzfile1” and “doesntwork-solfile1”).

This program takes an example 8-puzzle configuration (either randomly generated or provided in a file) and runs a specified solution to determine if it is correct. Here are the particulars:

```
linux> test-8puzzle [-s] [-p puzzlefile] solutionfile
```

Option “-p” allows the user to specify a specific input puzzle file (*puzzlefile*). If this option isn’t given, the code will randomly generate a puzzle to solve. The format of the puzzle file is a series of 9 digits, numbered 0 through 8, separated by blanks. The order of the integers represents the order of the squares in the puzzle, going left to right, starting in the top left corner. The 0 indicates the location of the blank square.

As an example, the start state on page 1 would be represented as: 7 2 4 5 0 6 8 3 1

*solutionfile* gives a series of moves of the blank square. The format of this file is a series of characters “u”, “d”, “l”, “r”, representing moves of the blank square up, down, left, right, respectively. The characters are separated by blanks, and are all on one line.

As an example, the moves “u l d” on the start state on page 1 would result in the following puzzle configuration:

```
5 7 4
0 2 6
8 3 1
```

Option “-s” is the “short” option, which is less verbose on the output. It just outputs the original puzzle and a statement on whether the puzzle has been solved (“Puzzle solved!” or “Puzzle NOT SOLVED”). If the `-s` option is not specified, the output of `test-8-puzzle` is the starting puzzle, then a listing of each move specified followed by its intermediate result, repeated until all moves have been executed. A final statement on whether the puzzle has been solved is also given.

As we’ve already noted, you will need similar functionality within your *GP-generate*, in order to test the fitness of an individual parse tree. If you like, you can reuse and/or adapt the provided code for this purpose.

### Data to Gather and Results to Report

Your results should include:

- A graph that plots (on a single graph) (1) the average fitness of the population, and (2) the fitness of the best individual in the population, as a function of the generation number.
- A graph that plots (on a single graph) (1) the average structural complexity of the population, and (2) the structural complexity of the best individual in the population, as a function of the generation number.
- A graph that plots the percentage variety of the population, as a function of the generation number. This just shows what percentage of the population is unique (and should start at 100%, since duplicates are removed when the population is initially formed).

Your reported results should include the program that represents the best individual found by your learning system. Your discussion should analyze this best individual, in terms of how it goes about solving the problem, and any other interesting observations you make about your solution.

### Part I Content and Grading

Part I is your team’s design for this program. It should provide enough information so that you can immediately begin programming your project without needing to make additional design decisions. This part does not require any implementation – it is focused on your overall approach to the problem. The section below on “What to turn in” for Part I gives the specific items that should be included in this design.

Part I will be graded on the thoroughness of your design, and evidence that you have thought through the various issues and design choices needed to begin programming the project. *Note, however, that you will not be restricted to this design in your final Part II solution.* You are free to change the design later if you discover flaws during your implementation and testing. The primary purpose of Part I is to ensure that you solidify your teams quickly, begin work right away on the project, and thoroughly consider your approach before you go headlong into programming.

### Part II Paper Guidelines

As always, as part of your completed (Part II) project, you must prepare a paper (3-6 pages) describing your project. Your paper should be formatted using common word processing software (such as LaTeX or Word), and should include the following:

- An abstract of 200 to 300 words summarizing your findings.

- An introduction describing the learning task and your formulation of the problem, including the definition of all the genetic programming design items required in Part I. (Again, it is not required that this design match your Part I design.)
- A detailed description of your experiments, with enough information that would enable someone to recreate your experiments.
- An explanation of the results, including the best program learned by your learning system. Include the graphs mentioned earlier in “Data to Gather and Results to Report”. Use additional figures, graphs, and tables where appropriate. Your results should make it clear that the genetic programming approach has in fact learned.
- A discussion of the significance of the results.

## Undergraduate Grading for Part II

Your grade will be based primarily on the quality of your project implementation and your description of your findings in the paper writeup. You should have a “working” software implementation, meaning that the learning algorithm is implemented in software, it runs without crashing, performs learning iterations as appropriate for a genetic programming system, and is well-documented.

If you have difficulties in getting the learning to work, you will still receive significant credit if you turn in a thorough paper that is readable and clearly outlines your experiments and their results, along with a discussion on why you think you obtained the results you did (or failed to obtain the results you were hoping for). That is, if your genetic programming approach does not successfully learn to solve the 8-puzzle, your work and results should clearly show that you spent considerable time trying to find the correct parameters, function and terminal sets, etc. (As we’ve said before, keep in mind that you illustrate this scientifically in the form of research results – figures or graphs that show the results of using different parameters. You *do not* illustrate this by handwaving, making excuses, or saying you tried something for  $x$  hours. Instead, you show quantitative results of those experiments.)

Additionally, you must proofread your paper, ensuring no spelling or grammatical errors (such errors will reduce your grade). Figures and graphs should be clear and readable, with axes labeled and captions that describe what each figure/graph illustrates.

## Graduate Grading for Part II

Graduate students will be graded more strictly on quality of the research and paper presentation. I expect a more thorough analysis of your results, and (hopefully) a working learning system (i.e., meaning the system actually learns). Your analysis should include a discussion (and perhaps results) on the following points (in addition to the points previously noted above):

- (If relevant) other function and/or terminal sets that you experimented with before you got a working system
- Effect of system parameters on results (such as population size, percentage for selection, percentage for crossover, etc.)
- Analysis of structural complexity and population variation, as compared to fitness, and whether it sheds any insight on the learning solution quality or convergence
- Future work that you believe would improve the learning
- Any other insightful observations you’d like to make

*You should not address these points in a bullet-type fashion, but instead work the answers into your paper in a discussion-style format. The paper should have the “look and feel” of a technical conference paper, with logical flow, good grammar, sound arguments, illustrative figures, etc. Graduate students are expected to format their paper in standard IEEE conference format (see <http://www.ieee.org/portal/pages/pubs/transactions/stylesheets.html> for style files).*

*However, even with this additional information, your paper must not exceed 6 pages.*

---

## WHAT TO TURN IN:

### Part I: Turning in your plan (due Wednesday, March 15, 23:59:59) [Counts 25% of your project grade.]

Part I of your project is your program design. It must contain the following information:

- Names of team members (**again, note that you are not allowed to change your team composition after the Part I due date**).
- Your definitions of the following for your genetic programming approach to the 8-puzzle:
  - Terminal set [Give the names and types]
  - Primitive function set [Give the list of functions, the number of operators, the types of the operators, and a brief description of exactly what function will be calculated by each primitive]
  - Fitness cases [Give pseudocode for how you will generate the fitness cases]
  - Fitness function [Give the function]
  - Method of probabilistic selection [Give the function]
  - Parameters: population size, fraction of population to be replaced by crossover, maximum depth of members in initial population, maximum depth for trees created by crossover, probability of mutation, and perhaps other design-specific parameters. [Give your starting point for these parameters, including a brief statement of the reasoning behind your starting choices.]
- Pseudocode for the main loop in the *GP-generate* function.
- Pseudocode for the main loop in the *solve8-puzzle* program.

You must prepare this information in a document and submit in pdf format. The document DOES NOT need to be a complete paper with flowing discussion, but should include enough explanation to make it clear how you are defining your approach to this problem. Part I should include sufficient detail that a person knowledgeable in genetic programming could take your design and implement it without unanswered questions. Email this pdf file to the instructor ([parker@cs.utk.edu](mailto:parker@cs.utk.edu)) by the deadline. Name this document as follows:

- Individual projects: Name this file *Yourlastname-Design-3.pdf*.
- Team projects: Concatenate your last names in alphabetical order and use the above naming convention. For example, if Smith, Jones, and Wesson are teammates, then your paper would be named *Jones-Smith-Wesson-Design-3.pdf*.

### Part II: Turning in your project (due Thursday, March 30, 13:00:00) [Counts 75% of your project grade.]

You should email BOTH your project and paper to BOTH the instructor AND the TA ([parker@cs.utk.edu](mailto:parker@cs.utk.edu); [m Bailey@cs.utk.edu](mailto:m Bailey@cs.utk.edu)) by the deadline.

Your submission should be in 2 parts (i.e., submitted in exactly 2 files):

1. Paper (in pdf format).
  - Individual projects: Name this file *Yourlastname-Paper-3.pdf*.
  - Team projects: Concatenate your last names in alphabetical order and use the above naming convention. For example, if Smith, Jones, and Wesson are teammates, then your paper would be named *Jones-Smith-Wesson-Paper-3.pdf*.
2. Tar or zip file (OK if it's compressed) of the programs and data files needed to run your learning algorithm, including a README file that gives instructions on how to run your code, and (if relevant), a makefile for creating the executable of your code.
  - Individual projects: Name this file *Yourlastname-Project-3.tar* (or *.zip*, or *.tar.gz*, etc.).
  - Team projects: Team projects: Concatenate your last names in alphabetical order and use the above naming convention. For example, if Smith, Jones, and Wesson are teammates, then your paper would be named *Jones-Smith-Wesson-Project-3.pdf*.