Solving Sokoban with Genetic Algorithms

Dylan Bauer - bauer898, Nick Karlovich - karlo015

December 18, 2019

Abstract

Sokoban is a puzzle video game created by Hiroyuki Imabayashi in 1981. In this game, there is a player that pushes boxes around a map. The player is trying to get each box onto specific goal locations. Sokoban gained popularity at its release with puzzle solvers, and it piqued the interest of many mathematicians and computer scientists. The rules of Sokoban somewhat realistically represent how moving storage containers around a warehouse is done in a real world scenario. This makes Sokoban a good candidate for academic research. While it has not gained as much attention from the academic world as Chess, Go, or Jeopardy, there has still been a sizeable amount of research dedicated to solving Sokoban puzzles. Most Sokoban solvers that are used in academics rely on A* and problem optimizations to solve the puzzle. We did not find many other solvers that used local searches such as genetic algorithms, hill climbing, and local beam search. We decided to use a genetic algorithm to solve Sokoban since we found academic papers that showed genetic algorithms being viable in solving maze-like problems like the Zen Garden puzzle. We wanted to see how effective a genetic algorithm would be on Sokoban puzzles. In our paper, we implement and modify a genetic algorithm that will try to solve Sokoban puzzles. To test our program, we ran our genetic algorithm on 6 problems of varying difficulty and size. With our baseline, we also compared how certain variables such as solution length, mutation rate, population size, and box on goal rewards changed the effectiveness of the genetic algorithm. We also tried using different heuristic functions to optimize our algorithm. After analyzing what we determined to be the best configuration of variables for solving Sokoban puzzles, we ran our genetic algorithm against XSokoban, a rigorous 90-puzzle Sokoban puzzle set that is typically used to compare the score of Sokoban solvers. Finally, we discuss the outcome of our genetic algorithm and what other optimizations and features could be added to our program in order to build a higher quality solver.

1 Introduction

"Sokoban" is a Japanese word that means "warehouse keeper". It is also the name of a game with a single agent in a maze-like environment where the agent can push boxes. The goal of the game is to push the boxes onto the predetermined goal tiles around the environment. Once all of the boxes are placed on the goal tiles, the puzzle is finished. A puzzles' difficulty starts off easy with one or two boxes in a room, but scales incredibly quickly as the number of boxes and goal states increases (Figure 1).

The design of the Sokoban game lends itself to a specific type of problem in computer science called a transportation puzzle. Some examples of other transportation puzzles are mazes, sliding puzzles, river crossing puzzles, train shunting puzzles, and rush hour puzzles[13]. Transportation puzzles are useful to us because we can apply the methods and optimizations from the puzzles to similar problems in the real world. Although Sokoban doesn't directly translate to real world warehouse box solving (Sokoban can only push elements, most warehouses have multiple moving "agents", and warehouses are designed to be easy to navigate while Sokoban maps are intentionally convoluted and difficult), they still further the research and exploration of ideas in transportation puzzles and thus allow researchers to create algorithms and optimizations for Sokoban and other similar games that can then be utilized in real world scenarios.

For our project, we are working on building a genetic algorithm that can solve Sokoban puzzles. We will be focusing on how different parameters such as sequence length, scoring parameters, heuristic functions, mutation rate, and population size affect the effectiveness of a Genetic Algorithm Sokoban solver. We will

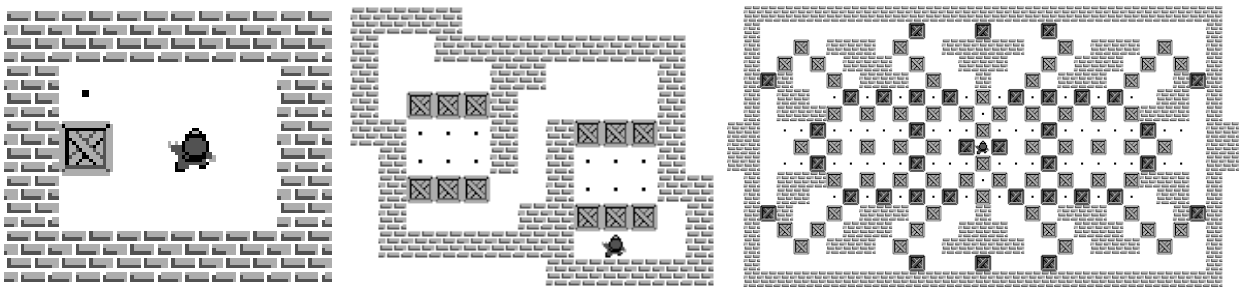

Figure 1: A trivially easy puzzle, a hard yet solvable puzzle, and an incredibly difficult puzzle.

also be looking into if and how Sokoban-specific optimizations can be implemented to further improve a genetic algorithm based solver.

The difficulty of this problem stems from the large sample space of possible moves and long solution lengths. Computing heuristics for Sokoban is also difficult because obtaining a solution is "all or nothing", which can easily lead to local heuristic minimums. For example, consider the situation where the solver gets 9 of the 10 boxes into the goal states but a key move to make this "solution" work requires pushing another box into a deadlock[1] state. In general, sub-goals and problems in Sokoban are interrelated and thus cannot be solved independently. Because of this, many optimizations have been created to mitigate these issues such as deadlock detection, zero-sum move detection, tunneling, and random restart.

When attempting to solve Sokoban, many academic papers [4][15] use the XSokoban testing set[2][11]. In our project, our goal is to make our genetic algorithm intelligent enough that it can begin solving these problems. The XSokoban set is a data set of difficult Sokoban puzzles, so to start building our genetic algorithm, we will use some example puzzles that were provided with the Sokoban implementation we chose to use. To build our genetic algorithm, we will choose six puzzles (two easy, two moderate, two hard) from the initially provided puzzles and test each one with variations of our genetic algorithm. After testing the validity and success of our genetic algorithm on those six puzzles, we will run our algorithm against XSokoban.

For our problem, we decided on 4 variables that we will test: number of moves, which is calculated by multiplying a constant by the length of a known solution, mutation rate of the genetic algorithm, box on goal reward, which increases the score of individuals with boxes on the goal states, and the population size of each genetic epoch. Each run of the genetic algorithm will have 5 minutes to find a solution. Then, through our initial testing and our innate understanding of the problem, we decided on default values for the variables and ran a baseline. Our baseline set the number of moves to 5 times the known solution length, the mutation rate to 75 percent, the box on goal reward to 0, and the population size to 100. Then, we analyzed how each genetic algorithm ran after we changed one variable in the genetic algorithm. Each version of the genetic algorithm ran 5 trials, each trial lasting for a maximum of 5 minutes. When the genetic algorithm finds a solution, it will exit early and return its fitness score with the time it took to solve the puzzle. Our genetic algorithm does not intend to discover an optimal solution, just one that exists.

After running a total of 16 different variations of our genetic algorithm across the 6 different puzzles with a Manhattan heuristic guiding its search, we ran it again with a more accurate and admissible heuristic based on BFS. Finally, after we determine what we deem to be the "optimal settings", we will run our genetic algorithm against XSokoban and see how it performs.

[1] a.k.a. A state where the box cannot be pushed out, for example a corner where there is no way for the agent to get behind the box as it is surrounded by walls on two sides

[2] `https://www.cs.cornell.edu/andru/xsokoban.html`

2 Related Work

In the following sections, we describe the complexities of Sokoban that make it such a difficult problem. We then describe the history of Genetic Algorithms and their use in search puzzles, and we quickly look at the history of research that has been performed on Sokoban.

2.1 Complexity of Sokoban

Sokoban is a very complex game to solve. It is considered to be a NP-Hard Problem[4]. According to [7], there are some reasons as to why the complexity of Sokoban is so high. First, it has a complex lower-bound estimation, which means that it is difficult to compute heuristics for Sokoban. The branching factor of Sokoban is also incredibly large and can have search spaces estimated to be at 10^{98} [9]. Also, some Sokoban puzzles have solutions that take over 500 moves to solve optimally, which is significant considering a branching factor varying between 0 and 136[9]. Finally, due to the rules of Sokoban, it is possible to put a game of Sokoban into a provably unsolvable state called a deadlock. This means there are moves that, although aren't illegal, will cause the solver to fail, which adds more computational complexity to the problem.

Along with many other NP-Complete puzzles, Sokoban is considered to be PSPACE-complete[2][14][4]. This means that the problem can be solved with an amount of space that is polynomial in size relative to the size of the input, and that same space can be used to solve other problems of the same size.

2.2 History of Genetic Algorithm Searches

Genetic algorithm searches are a form of local search that attempts to generate random sequences of actions and assess the fitness of each of these random sequences. Different genetic algorithms can choose to have different implementations to decide the next generation. Typically, the best individual is carried into the next generation, and the individuals are recombined based on their fitness. Individuals that perform the worst will breed the least, and they are more likely to be removed from the population. Individuals with the best fitness will breed more, and they are less likely to be removed. There is also a chance to have a random mutation in offspring, which hopefully prevents the sequence pool from converging to a non-optimal solution.

Genetic algorithms have been applied to maze-like problems before. In [1], they discussed using a genetic algorithm on the Zen Puzzle garden problem. The Zen Puzzle garden, much like the Sokoban puzzle, involves a maze with walls and an agent trying to traverse it through a sequence of moves. However, the goal state of a Zen Puzzle is slightly different from Sokoban. In Zen Puzzle, the goal is to visit every square without touching the same square again. In Sokoban, the goal is to push boxes into goal locations. Zen Puzzle solutions are invalid if they revisit a square. Sokoban solutions are invalid if they reach a deadlock state, which is harder to detect. The genetic algorithm performed better than informed search methods in Zen Puzzle Garden, and seemed like it could be applicable to other maze-solving applications.

2.3 History of Sokoban in Academia

Sokoban has been a heavily researched game in Academia. Specifically, many academics have been trying their hand at creating a Sokoban solver that can solve as many puzzles in the XSokoban[11] set as possible. The earliest mention of a Sokoban solver dates back to 1993 with Mark James' Sokoban solver, which was able to solve 1 of the 90 problems in XSokobon in 2 hours [5]. After M. James, a large portion of research on Sokoban solvers occurred in the late 1990 and early 2000's. The first "real" Sokoban solver named "Rolling Stone", was introduced in A. Junghanns and J. Schaeffer's 1997 paper[6]. Rolling Stone was their first attempt at creating a solver that could solve Sokoban problems. It used IDA* and a few basic optimizations such as deadlock tables and move ordering to optimize its solution. In the end, it was able to solve 12 of the 90 puzzles in XSokoban. In Junganns' thesis on Sokoban in 1999, he mentions other researchers that created Sokoban solvers. In 1999, Stefan Edelkamp, the Meji University Sokoban Laboratory, and an author named

Deepgreen were able to get their Sokoban solvers to complete 13, 55, and 62 of the problems in XSokoban respectively [8]. A point of note is that the 90 problems in XSokoban are not in increasing difficulty. Thus, the 13 problems that Stefan Edelkamp's solver could solve may not have been in the set of the 55 solved problems that the Meji University's Sokoban Laboratory's solver could solve.

In the last 20 years since the creation of Rolling Stone, many more solvers have been created. In Timo-Vrikkla's master's thesis[15], he covers many of the known Sokoban optimizations in 2011. Some of the optimizations reported by him include bi-directional solving, multi-agent searching, problem abstraction, the Van Lishout solving method, and PI-corral pruning.[15] In his paper, he reports that when these optimizations are built into solvers, algorithms such as "Rolling Stone" were able to solve 59 of the problems in XSokoban compared to the 12 it was originally able to solve when it was first created.[15]

Since the creation of Rolling Stone, other solvers have also prevailed that are able to perform similarly or slightly better than Rolling Stone such as "Talking Stone", which was able to solve 61 of the 90 XSokoban problems[3]. In recent years (2016), a paper was published by the University of Alberta where a group of researchers were able to create an improved heuristic for Sokoban[12]. With the improved heuristic, the team at Alberta found that their solver could now compute solutions to 86 of the 90 problems found in XSokoban.

This shows that there has been a lot of work done in the field of path planning and specifically, a lot of work has been done in optimizing and finding problem specific optimizations for Sokoban, which has allowed the number of XSokobon solutions to go from 1/90 in 1993 all the way to 86/90 in 2016.

3 Experiment Design

In order to build our genetic algorithm, we will need code from the AIMA genetic algorithm and code from an implemented Sokoban game. Then, we will need to write code to interface between the AIMA interface and the Sokoban game, and we will need to implement a genetic algorithm. Getting the AIMA code will be easy since it was provided to us earlier in the semester. Finding the Sokoban implementation will be a bit more difficult because we want the Sokoban code to be written in Python so that it can be easily translated and interfaced between AIMA.

We will test our program against our base case and compare it to a trial that has one of four parameters changed. The parameters we will change include solution length, mutation rate, box on goal reward, and population size. Each trial will have 5 minutes to solve the Sokoban puzzle. If the genetic algorithm does not solve the puzzle, it will be considered a failure. Due to the randomness involved in a genetic algorithm, we will test each trial five times to get a more accurate sampling of how these changes impact the solving rate. There are 3 difficulties of puzzles that we will test it against. The difficulties are easy, moderate, and hard. Each of these will have two puzzles. The easy puzzles will have relatively straightforward maneuvers and low box counts. The moderate puzzles will have more difficult maneuvers, but the box count will still be low. The hard puzzles will have difficult maneuvers as well as many boxes. The first test will be ran using a genetic algorithm that uses the Manhattan distance as a heuristic. We will then run it again with a more accurate heuristic. Instead of using the Manhattan distance, we will create a 2D distance array with BFS to find the shortest distance from a goal to any point in the map. This will allow our heuristic to determine the actual push-distance a box is from a goal instead of relying on an approximation that doesn't take walls into consideration when determining distances to goals.

3.1 Genetic Algorithm Design

First, we will decide how to implement our genetic algorithm. The basic design we've come up with is based around creating a "gene" that is represented as a string of moves, 'u': up, 'd': down, 'l': left, 'r': right. This will tell the Sokoban game which direction to move the player.[3] In order to generate our "genes", the solver will take in a "solution length", which is how many moves we would allow our gene to make before giving up on the solution. The solution length will be determined by multiplying the known solution length

[3]As a note, some informed search algorithms use upper case versions of all the letters which represent when a box was pushed, but after some consideration we decided that implementing it would be unwieldy and difficult to use in a genetic algorithm.

by some constant. We will then randomly create a permutation of up, down, left, and right moves until the gene reaches the "solution length". Then, we will feed the Sokoban game the "gene" and then score how close that gene makes it to the solution. The scoring of a solution occurs once all the instructions in the gene have executed or once a goal state is reached. The scoring mechanism for our genetic algorithm will be explained below, but for now, we will look at how our genetic algorithm did its mutations and population generation.

Through testing our code with the AIMA implementation of a genetic algorithm, we began to run into problems with AIMA's requirements. After struggling with integrating the two systems together, we ended up rebuilding many of the existing genetic algorithm functions from scratch. Our method of running the genetic algorithm is very similar in concept to how AIMA does its genetic algorithm, but, in the end, we used little to no actual AIMA source code.

To start our genetic algorithm, we run every gene through a game of Sokoban. Once every gene in the population is "run" through Sokoban, it is scored by our scoring algorithm. Each (gene, score) pair is stored and sorted via the score it achieved. Then, a new array, which represents the probability of choosing each gene, is filled by this formula.

$$P(gene[i]) = \frac{gene[i].score}{\sum_{k=0}^{PopulationLength} gene[k].score}$$

Then, each value of the array will have the previous value added to itself such that the probability of the final entry will always be 100. This is also known as a cumulative distribution function. If, for whatever reason, the total score of the entire population set is less than or equal to zero, meaning that no gene made any forward progress, the entire population will be re-initialized with random new values. Otherwise, we will then begin recombining and mutating existing genes.

3.2 Mutation and Recombination

First, we will take the gene with the highest overall score and carry it over into the next population. With Sokoban, we decided that it was beneficial to ensure through every population that we never accidentally went backwards in progress. We will account for this by always carrying over the highest scoring gene. For the remaining genes in the population, they will be recombined and possibly mutated.

We start this process by choosing two random genes to mutate. We do this via our previously determined cumulative distribution array.[4] Once we have recombined the two genes we chose, we will then attempt to apply a mutation to it. The chance that any given gene is mutated is based off of the genetic algorithm's mutation rate. The higher the mutation rate, the more likely the gene is to be mutated. If the mutation occurs, the mutation function randomly chooses one "input" (left-right-up-down) in the gene and randomly reassigns it to another input (Appendix 8.8). This reconstructed gene is then placed back into the population. This mutation and recombination is done for every gene in the population. Once this is done, the genetic algorithm starts running the genes through the puzzle again and continues this procedure until either the time on the simulated search runs out or until it finds a goal.

This method of mutation and recombining is one of the ways that distinctly differentiates a genetic algorithm from other informed/uninformed searches such as BFS, DFS, UCS, or A*. By randomly recombining those two genes together and then randomly mutating the combined "gene", it is almost guaranteed that invalid moves will be introduced into the newly created genes. This is because Sokoban is a heavily sequential game rather than an episodic game. This feature of the genetic algorithm is a double-edged sword though due to its ability to destroy sequences yet avoid local maximums.

[4]The cumulative distribution array and the formula above ensure that the probability of a gene being picked is relative to its score.

3.3 Scoring Method

Since we were the first people we know of to create a genetic algorithm for Sokoban, we didn't really have much of a starting ground for scoring the game and creating accurate heuristics so we came up with our own. (lines 27-31 in Appendix 8.5)

```
1       if(total > MAX_BOARD_VAL and correct_counter < 1):
2           total = 0 # no progress made on puzzle
3       else:
4           total = SOL_LENGTH - num_of_moves + MAX_BOARD_VAL - total
5           total = total + (correct_counter * BOX_ON_GOAL_REWARD)
```

When computing the score, the first thing we compute is the MAX_BOARD_VAL. The MAX_BOARD_VAL is computed by the function in Appendix 8.2 Computing the Max Fitness. This function computes the score of the initial state via the given heuristic function. A lower score in the heuristic is better.

In our genetic algorithm, we used the two heuristics, the Manhattan distance[5] and a BFS-based path distance. Both of these heuristics are admissible, but the BFS gives a better estimation than the Manhattan distance which can go through objects while the BFS cant. Computing the Manhattan is done in real time every time the heuristic is needed. The BFS heuristic is computed by creating a dictionary of 2D arrays at the very beginning of genetic algorithm, where each 2D array is filled with the BFS distance away from one of the goals, while following the geometry of the map (going around walls, rather than through them as the Manhattan does).[6] When we compute the heuristics, we create a different 2D array where we compare the distance (via Manhattan or BFS) between every goal and every box. We then use a Python package called Munkres that runs a combinatorial optimization algorithm called the Hungarian Algorithm.[7] After the Hungarian algorithm is run, we get back the optimal (estimated) way to push all the boxes to different goals, and we return that total distance.

For our scoring algorithm, we decided that if the solution outputted by a specific gene has pushed the boxes further away from the goal state (i.e. total (the output of the heuristic) is less than MAX_BOARD_VAL (line 1 above)) and no boxes are on the goal, then we don't want to consider that gene for recombination and we set its "recombination probability" to 0. If the "gene" has a lower total (distance from the goal), then we go into lines 4-5. On line 4, the expression (SOL_LENGTH - number_of_moves) rewards solving the problem in less moves than the entire solution length, which is a small push to optimize answers that are more optimal. Notice that num_of_moves will always equal SOL_LENGTH when the "gene" doesn't find a solution, and less than SOL_LENGTH when it does, so we will never get a negative score. On line 4, the expression (MAX_BOARD_VAL - total) rewards each "gene" for pushing the boxes closer to the goal state. Finally, on line 5, we reward each gene a set constant number of points for each box that they place on the goal.

Obviously, this method of computing heuristics has some very serious edge cases where it fails to perform well, such as a case where the agent needs to push the box away from the goal in order to eventually push it into the goal state, a case where a box starts on a goal state but a second box is inaccessible unless the first box is moved off the goal state temporarily, or a case where the solution is very close, except for a box that was placed in deadlock.

3.4 Existing Sokoban Game

After some searching online, we found a Python implementation of Sokoban that was used as part of an assignment in an AI course at Colombia University.[10] The code used in this implementation was meant for use in informed search algorithms such as UCS, BFS, DFS, and A*. This meant that a fair amount of refactoring was needed in order to allow us to input our moves into the Sokoban game using a genetic algorithm rather than an informed search. The biggest difference between their informed search

[5] https://en.wikipedia.org/wiki/Taxicab_geometry

[6] The BFS heuristic for one specific goal does not take into consideration other goals and where other boxes may currently be located on the map, including on the goal itself.

[7] https://en.wikipedia.org/wiki/Hungarian_algorithm

implementation and our genetic algorithm implementation was how certain moves would be created and sent to the game. In the informed search, they would check what moves were valid from their current state and pick one from there. Comparatively, a genetic algorithm, which is designed to have mutations and re-combinations, allows for invalid moves to be attempted (but will not successfully complete).

The six puzzles that we tested with are shown in Figure 2, Figure 3, and Figure 4.

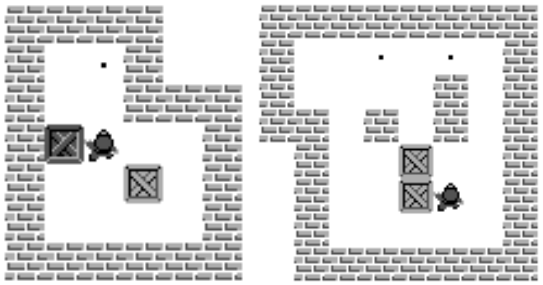

Figure 2: The easy puzzles, easy2.txt and easy4.txt respectively

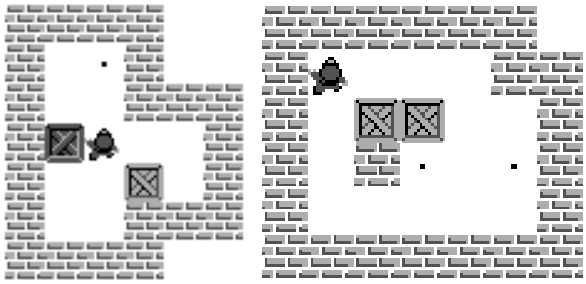

Figure 3: The moderate puzzles, mod1.txt and mod3.txt respectively

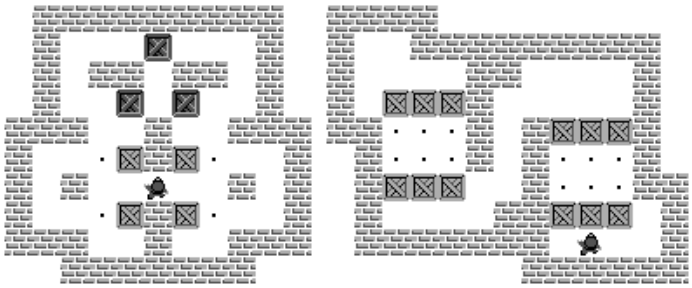

Figure 4: The hard puzzles, hard1.txt and hard2.txt respectively

4 Results

The results of the testing are shown in Figure 5, Figure 6, and Table 1. Figure 5 compares run-times of the Manhattan and BFS heuristics. Figure 6 compares the success rate of the Manhattan and BFS heuristics on each of the 6 puzzles. Table 1 shows how changing the solution length, mutation rate, box on goal reward, and population size affects the number of solves and the average time to solve the puzzle. Since we were unable to solve any of the hard puzzles, the hard puzzles were not included in the table.

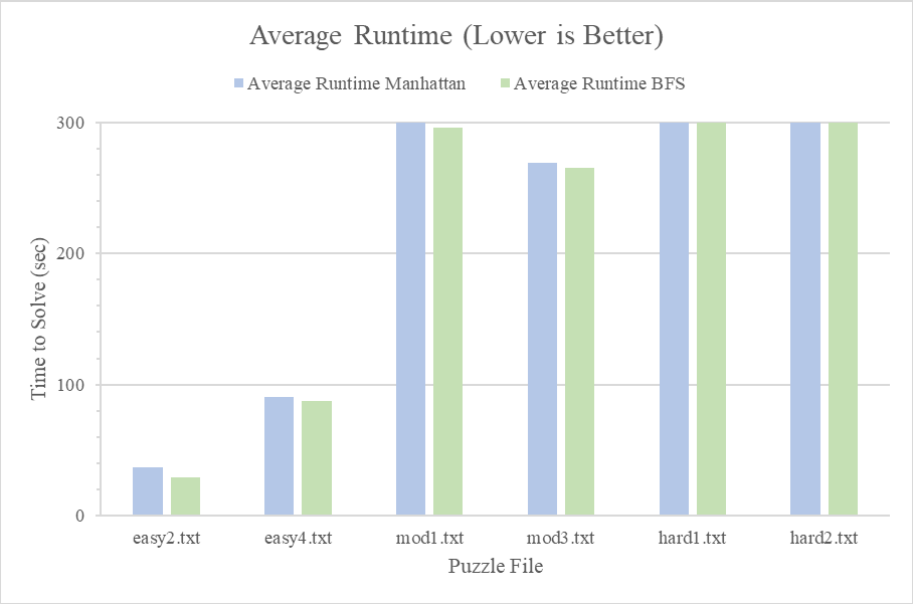

Figure 5: The average run-time for the genetic algorithm with both heuristics over all six puzzles.

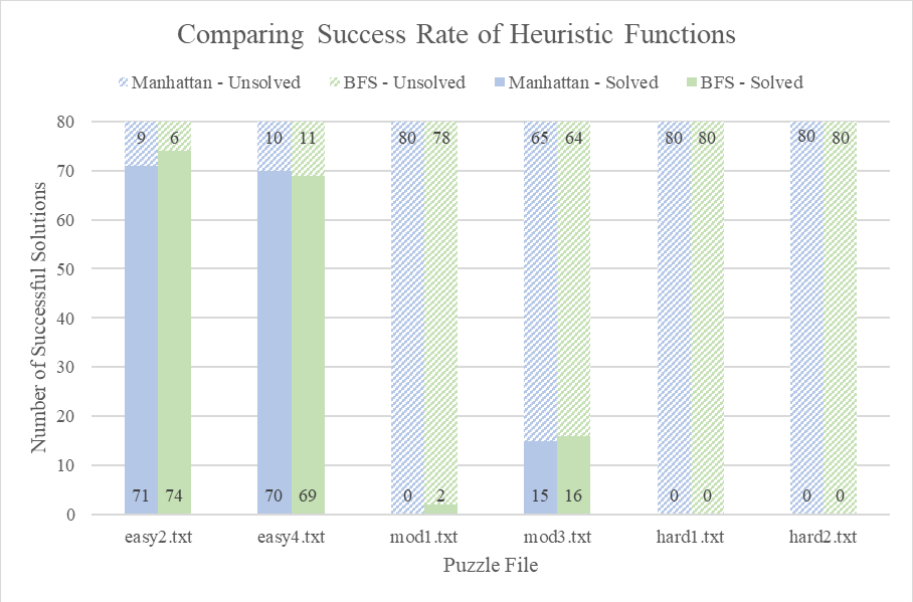

Figure 6: The number of puzzles the genetic algorithm solved with both heuristics over all six puzzles.

5 Analysis

5.1 Tuning the Genetic Algorithm

Based on the results in Figure 5 and Figure 6, it would appear that using the BFS-based heuristic instead of the Manhattan distance yielded a slight but noticeable improvement in each of the test puzzles. Based on this, we decided we would use our BFS heuristic for our XSokoban test rather than the Manhattan distance.

Table 1: The effect of variables on the output.

	Easy Puzzles		Moderate Puzzles	
Solution Length	Solves	Avg Time	Solves	Avg Time
2x	5	247.788	0	300
5x (Default)	18	36.300	2	281.5
10x	20	8.906	4	272.541
20x	20	28.022	3	272.955
	Easy Puzzles		Moderate Puzzles	
Mutation Rate	Solves	Avg Time	Solves	Avg Time
0.25	16	82.745	0	300
0.5	18	63.233	2	287.449
0.75 (Default)	18	36.300	2	281.5
1	20	24.117	1	289.885
	Easy Puzzles		Moderate Puzzles	
Box on Goal	Solves	Avg Time	Solves	Avg Time
0 (Default)	18	36.300	2	281.5
1	20	25.472	3	286.697
2	20	28.388	5	237.995
5	20	18.525	5	248.231
10	20	15.235	2	276.419
20	20	13.297	1	290.649
	Easy Puzzles		Moderate Puzzles	
Population Size	Solves	Avg Time	Solves	Avg Time
10	18	62.463	1	288.568
50	18	38.456	2	285.963
100 (Default)	18	36.300	2	281.5
500	18	77.616	1	292.7885
1000	14	170.219	1	287.475

Likewise, a solution length multiplier of 10 seemed to solve faster and more consistently than any other multiplier, as illustrated by Table 1. We also found that a mutation rate of 100 percent allowed the program to solve faster and more frequently. It appeared that a mutation rate between 75 percent and 100 percent would be ideal, but we chose 100 percent because we needed to ensure our individuals did not get stuck at local maximums and would change its sequence frequently. A box on goal reward helped the solution solve as well. A reward of 2 seemed to best solve the problem, as having less points awarded seemed to not direct the algorithm enough towards a solution, and having more points seemed to make the algorithm too unwilling to move boxes off of goal locations. This especially helped in solving moderate difficulty puzzles. The population size also had a drastic effect on solving the problem. Having a population size of 10 did not offer enough genetic diversity to allow the population to solve consistently. On the other hand, having a population size of 500 or 1000 seemed to have too many individuals to consider, making the algorithm slow to run. Having 50 to 100 individuals seemed to be ideal. We decided on 100 individuals to increase genetic diversity, as having 100 individuals had a slight edge over 50 individuals as shown in Table 1.

It was interesting to see how each variable affected the puzzle. For the solution length, it seemed to require a large number of moves over the actual solution length, which suggests that there is room for improvement in our individual's choice of moves. The mutation rate had the most success at a very high percentage. That seems to suggest that our genetic algorithm is having difficulties converging to a solution. This is probably due to getting stuck at local maximums and needing lots of variation in the input sequences to be able to move off of those local maximums. It also means that our heuristic could be improved so that

less drastic mutations are needed to converge onto a solution. Box on goal rewards were also very critical for solving puzzles faster. Having a slight reward seemed to converge the solutions faster. It was especially noticeable in the easy puzzles. However, in the moderate puzzles, having a large reward actually made it more difficult to solve. This is most likely because it was unwilling to move balls off of goal states, which is necessary in the more difficult puzzles. Population size also seemed to be relatively important, as having too small of a population and having too large of a population both caused the puzzle to solve less consistently. It seemed that providing a better heuristic had a very noticeable impact on how the program solved, with most puzzles solving faster and more frequently. Ideally, if we had a better heuristic, we could improve our genetic algorithm enough to start solving more difficult puzzles. Improving our heuristic could probably result in the most success of our algorithm, but more experiments would have to be done. One important thing to note is that our results had a large degree of error associated with them. This was because genetic algorithms depend on randomness a lot, so replicating the results could vary a lot. Running more trials would allow us to have more accurate results and allow us to make stronger conclusions on our data.

5.2 Running Against XSokoban

Having used what we decided to be the optimal settings for our genetic algorithm, we ran our algorithm against the 90 XSokoban puzzles. Unfortunately, our genetic algorithm failed and did not solve any of the XSokoban puzzles. Based on the best individuals that the algorithm was returning, it appeared that our individuals were making progress on the puzzle. However, they were not finishing the puzzle and were likely getting stuck on local maximums. If we were to try to implement more improvements to our algorithm, our program might eventually be able to solve some of XSokoban's puzzles.

There are several factors that we believe contributed to the failure of our genetic algorithm. One factor is the large branching factor of Sokoban[9]. Since most Sokoban puzzles have solutions between 100 and 1000 moves, the solution space to consider is enormous.

Another failure of our algorithm was deadlocks. Our genetic algorithm did not have any way to detect deadlocks due to the complexity of the implementation. This caused many genes that resulted in deadlock states to be considered in recombining genes since they still reached local maximums even though they weren't good solutions.

Cycles and unchanging board states also caused problems for our algorithm in solving the puzzle. We found that a solution length multiplier of 10 gave our algorithm the best results. This meant that one in every 10 moves made a meaningful contribution to the problem. The inefficiency of our genetic algorithm was in part caused by invalid moves (such as the player walking into a wall) and move cycles (making valid moves, but not changing the board state). Because our player was not always making moves that made sense, our solution length was longer and the time it took to solve the problem increased.

5.3 Improving our Approach

Due to the failure of our implementation to handle the difficulty of the XSokoban tests, there are several ways forward to improve our genetic algorithm. One such way is to fine-tune our solution more. Fine-tuning our genetic algorithm was difficult, as we were trying to simulate test conditions by allowing our genetic algorithm 5 minutes to solve. If we had more time and better tools to approximate the best settings for our genetic algorithm, we could adjust our parameters so that we could achieve the best success rate possible. This would allow the algorithm to converge to a solution better and possibly result in more solved puzzles. It could even allow our algorithm to start solving more difficult puzzles. Adjusting the parameters to a genetic algorithm is known to be especially finicky, which made it difficult to do in the span of this project. Another way we could fine-tune our project is to try changing multiple variables at once. This makes optimizing our algorithm even more difficult as the number of configurations increases drastically. However, it could be possible that changing one variable could influence other variables. This would make testing especially difficult as many iterations of testing would have to be done to find an optimal setting. We attempted to use a control group and measure each variable individually, which made the most sense given the span of

the project. Given more time, it would be interesting to see the relationship between each of these variables and what performance benefits they might yield.

There are also features that we could add to our project to make it more successful. Most notably, deadlock detection could greatly improve our algorithm. This was implemented in [6] and expanded upon in [9] and was shown to increase the success rate of their algorithm. By using deadlock detection, our genetic algorithm could better deal with individuals in the population that were approaching deadlock states. If we created deadlock tables to identify when an individual pushed a box into a square that is known to cause a deadlock, then we could either remove the individual from our population or try to adjust the gene so that it was no longer in deadlock. Both of these actions have costs, as it would take computation time to modify the gene, but conserves the gene's moves. Alternatively, it would be easier to just remove the individual from the population, but we would lose its information, which would be detrimental if the gene was close to a solution.

Another implementation to possibly try is to vary the parameters of our genetic algorithm as it runs or to refresh the population more frequently with random restarts. It is very likely that the individuals in our genetic algorithm are getting stuck on local maximums. To prevent this, we could try to vary the mutation rate or other parameters. For example, starting with a larger population and then reducing the population size as the algorithm runs could allow for a more diverse starting population. Then, reducing the population size would prevent the algorithm from becoming extremely slow. Reducing the box on goal reward might also make solutions that push boxes off of goal states more likely, which could allow the solution to solve better in some cases. Another improvement would be random restarts. Currently, we have random restarts that occur when the individuals in our population make no forward progress on a puzzle. We could set limits to our genetic algorithm so that after a certain period of time, or after a certain number of generations, the population is reset.

Another optimization that could yield improvements to our algorithm would be to filter moves. Since our genes are randomly generated, there are often invalid moves that do not change the board state. For example, the sequence "rrrr" when the player is already on the left side of the map would cause the player to run into a wall. This would not change the board state since the moves will be read as invalid, which increases the solution length necessary for our genetic algorithm to solve the puzzle. To fix this issue, we could iterate through each of our individuals to ensure that their moves make sense. However, the computation time for recombining and mutating individuals would increase considerably with this approach. Alternatively, we could partially enforce this by generating our initial population with random depth-limited searches to ensure that they explore the map sufficiently. This could also help the children of the initial population converge on a solution faster since the children would already know how to traverse the maze. Another inefficiency that occurs in our genes are "move cycles". "Move cycles" are where a sequence of moves are all valid, but the sequence brings the board back into a previous state. For example, the moves "lrlr" would be considered a "move cycle" because even if the first two moves changed the board state, the next two moves will bring the board back to the same state as if only the sequence "lr" was executed. It is easier to detect move cycles because we do not need to simulate the Sokoban game to detect these. However, they do not ensure that all moves are valid as accurately as iterating through the entire individual's sequence. This would take less time to compute computationally, and it would be easier to implement. Ideally, either of these two changes would reduce the solution length of our individuals, making the problem easier to solve.

An investigation into better heuristics for our fitness function algorithms might also improve our genetic algorithm. To determine the fitness of an individual, we used the Hungarian algorithm, which considered the distance a box was from every goal and used each distance to compute the fitness. This algorithm is optimal, but the effectiveness of the Hungarian algorithm is determined by the heuristic we use. We noticed that using a better heuristic, namely using a BFS to generate distances instead of the Manhattan distance, improved the convergence of our solutions. Using a more accurate heuristic that better approximates how far the individual is from the solution would give our algorithm a better chance at solving more complex Sokoban puzzles.

6 Conclusion

Our genetic algorithm was not effective enough to solve XSokoban puzzles, but it proved to be successful in smaller-scale puzzles. Once the solution space of the puzzle gets sufficiently large, the solution no longer converges quickly to a solution, so we will need to implement other features to overcome this barrier.

There are several ideas that could be implemented into our genetic algorithm to make it efficient enough to solve XSokoban puzzles, such as deadlock detection, filtering moves, modifying variables when the algorithm gets stuck on local maximums as it runs, improving fitness functions, and optimizing our parameters. Each of these ideas comes at the cost of computation time, so future experiments determining the actual benefits of these features will be necessary.

7 Contributions

Our work was as follows: Nick Karlovich - karlo015

- Ran tests
- Proofread document
- Reworked downloaded Sokoban implementation to accept a genetic algorithm
- Implemented initial genetic algorithm
- Made Figures
- Wrote multiple sections of the final paper which include: Abstract, Introduction, Complexity of Sokoban, History of Sokoban in Academia, Experiment Design, Appendix

Dylan Bauer - bauer898

- Ran tests
- Proofread document, added commas where Nick forgot
- Refactored genetic algorithm
- Implemented an interface to test the code
- Wrote code for BFS heuristic functions
- Wrote multiple sections of the final paper which include: Introduction, History of Genetic Algorithm Searches, Experiment Design, Analysis, Conclusions, Appendix

References

[1] M. Amos and J. Coldridge. A genetic algorithm for the zen puzzle garden game. *Natural Computing*, 11(3):353–359, 2012.

[2] J. Culberson. Sokoban is pspace-complete. 1997.

[3] J.-N. Demaret, F. Van Lishout, and P. Gribomont. Hierarchical planning and learning for automatic solving of sokoban problems. 2008.

[4] D. Dor and U. Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[5] M. James and B. MacDonald. Optimal tunneling: A heuristic for learning macros. 1993.

[6] A. Junghanns and J. Schaeffer. Sokoban: A challenging single-agent search problem. In *In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch*. Citeseer, 1997.

[7] A. Junghanns and J. Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 1–15. Springer, 1998.

[8] A. Junghanns and J. Schaeffer. *Pushing the limits: new developments in single-agent search.* University of Alberta, 2000.

[9] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

[10] K. McKeown. Assignment 2: Sokoban search algorithm, Oct 2013.

[11] A. Meyers. Xsokoban.

[12] A. G. Pereira, R. Holte, J. Schaeffer, L. S. Buriol, and M. Ritt. Improved heuristic and tie-breaking for optimally solving sokoban. In *IJCAI*, pages 662–668, 2016.

[13] J. O. Postma. Generic puzzle level generation for deterministic transportation puzzles. Master's thesis, 2016.

[14] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192, 1970.

[15] T. Virkkala. *Solving sokoban.* PhD thesis, Ph. D. dissertation, Masters thesis, University Of Helsinki, 2011.

8 Appendix

8.1 Making Dictionary of Distance Arrays for BFS

```
def makeDistanceDictionary():
    # opens and reads Sokoban puzzle into a 2D array
    fd = open(name of Sokoban file)
    grid = fd.read().split("\n")
    for i in range(grid):
        grid[i] = list(grid[i])

    goals = []
    distance_array = [[]] * len(grid)
    # checks for goals in the grid and initializes distance 2D array
    for y in range(len(grid)):
        for x in range(len(grid[i])):
            if grid[y][x] == "." or grid[y][x] == '*':  # if coordinate in grid is a goal
                goals.append((x,y))
            distance_array[y].append(float('inf'))      # add infinity as distance to 2D array

    # make a dictionary that returns a 2D array of distances when given goal coordinates
    distance_matrix_dict = {}
    for goal in goals:
        array_copy = copy.deepcopy(distance_array)
        distance_dict = {}
        q = [goal]      # queue used to create distance array
        dist = 0        # distance from goal
```

```
        while q != []:
            for i in range(len(q)):
                cur = q[0]
                del q[0]        # remove coordinate from queue
                # finds left, right, up, down from current coordinate
                moves = [(cur[0]-1, cur[1]), (cur[0]+1, cur[1]),
                         (cur[0], cur[1]-1), (cur[0], cur[1]+1)]
                for (x,y) in moves:
                    # if move has not been found and move does not go into a wall, add to queue
                    if dist < grid_lst_copy[y][x] and grid[y][x] != "#":
                        q.append(move)
            dist += 1
        # add 2D distance array to dictionary
        distance_matrix_dict[goal] = grid_lst_copy
    return distance_matrix_dict
```

8.2 Computing the Max Fitness

```
def compute_max_fitness():
    payoff_mtx = []
    # BOARD is the initial Sokoban puzzle board
    goals = BOARD.goals
    boxes = BOARD.boxes

    # for each goal, get heuristic distance from each box
    for goal in goals:
        box_line = []
        for box in boxes:
            box_line.append(HEURISTIC(goal.x, goal.y, box.x, box.y))
        payoff_mtx.append(box_line)

    m = Munkres()                   # a.k.a. The Hungarian Algorithm
    indexes = m.compute(payoff_mtx)
    total = 0
    # sum values obtained from Munkres into a total score
    for (row, column) in indexes:
        value = payoff_mtx[row][column]
        total += value
    return total
```

8.3 Determining the final board state an individual finds

```
def search(board, sequence):
    if board.is_win():
        return (board, 0)      # (the board, the number of moves it took)

    # for each character in sequence
    for char_idx in range(len(sequence)):
        next_move = convert_string_to_direction(sequence[char_idx])
        if board.is_valid(next_move): #check if move valid, if not, ignore move.
            board.move(convert_string_to_direction(sequence[char_idx]))
        else:
            if board.is_win():
                return (board, char_idx) # (the board, the number of moves it took)
    return (board, len(sequence))
```

8.4 Making Dictionary of Distance Arrays for BFS

```
def find_distances(goal_list, box_list, idx, payoff_mtx):
    # for every goal, get distance
    for goal in goal_list:
        box_line = []
        for box in box_list:
            box_line.append(HEURISTIC(goal.x, goal.y, box.x, box.y))
        payoff_mtx.append(box_line)

    # solve matrix
    m = Munkres() # a.k.a The Hungarian Algorithm
    return (m.compute(payoff_mtx), payoff_mtx)
```

8.5 Compute Fitness of an Individual By Using the Hungarian Algorithm

```
def compute_fitness(pop_arr, fitness_dict):
    total_fitness = 0
    board_arr = []
    for i in range(0, POP_NUMBER):
        board_arr.append(new_board(BOARD.filename))
        for gene_idx in range(0,len(pop_arr)):
            (board_arr[gene_idx], num_of_moves) = search(board_arr[gene_idx], pop_arr[gene_idx])
            correct_counter = 0
            payoff_mtx = []
            # Does Hungarian algorithm to find the cheapest distance
            # to all goals from all boxes.
            (indexes, payoff_mtx) = find_distances(board_arr[gene_idx].goals,
                                board_arr[gene_idx].boxes, gene_idx, payoff_mtx)
            total = 0
            # SOL_LENGTH is the max number of moves made before the gene is out of moves.
            # If a puzzle can be solved in 10, a good SOL_LENGTH might be 10 * 10, or 100.

            # if you use all the moves, then total will be 0 at this point.
            for row, column in indexes:
                value = payoff_mtx[row][column]      # minimum values of Hungarian
                total += value
                if value == 0:                       # a box is on the goal state
                    correct_counter += 1             # add points to heuristic function
                # MAX_BOARD_VAL is the maximum fitness
                # if the total Hungarian distance of all boxes from the goal is greater
                # AND no boxes are in goal states, then this individual should have heuristic 0
                if(total > MAX_BOARD_VAL and correct_counter < 1):
                    total = 0 # no progress made on puzzle
                else:
                    total = SOL_LENGTH - num_of_moves + MAX_BOARD_VAL - total
                    total = total + (correct_counter * BOX_ON_GOAL_REWARD)
                fitness_dict[''.join(pop_arr[gene_idx])] = total
                total_fitness = total_fitness + total
            return total_fitness
```

8.6 Manhattan Distance Heuristic

```
def manhattan(goal_x, goal_y, box_x, box_y):
    return abs(goal_x - box_x) + abs(goal_y - box_y)
```

8.7 BFS Heuristic

```
def BFS(goal_x, goal_y, box_x, box_y):
    # distanceDictionary is initialized after makeDistanceDictionary is called
    return distanceDictionary[(goal_x, goal_y)][box_y][box_x]
```

8.8 Genetic Algorithm

```
def genetic_algorithm(population, fitness_dict, pmut):
    gene_pool=['u','d','l','r']         # possible moves
    new_best_score = ("",0)             # starts with a blank individual
    start = time.time()                 # creates start time
    # runs algorithm until there is a winner or 5 minutes has passed
    while (time.time() - start < TIME_TO_SOLVE and
          not winner(new_board(BOARD.filename), new_best_score[0])):
        seq_fitness_percent = []
        total_points = compute_fitness(population, fitness_dict)
        if total_points > 0:
            best_score = ("",0)
            # finds best gene in population and appends
            # sequence, fitness, and percent to seq_fitness_percent
            for gene1 in population:
                gene = "".join(gene1)
                # this is the score we're trying to maximize
                score = fitness_dict[gene]
                if score > best_score[1]:
                    best_score = (gene, score)
                seq_fitness_percent.append((gene, score, (score / total_points)))
            a = sorted(seq_fitness_percent, key = lambda tup: tup[2])
            new_best_score = best_score
            # keeping the best one from each generation of genes
            population[0] = best_score[0]
            for j in range(1, len(population)):
                #sort tuple by last value, smallest earliest on.
                for i in range(len(seq_fitness_percent)):
                    if i != 0:
                        temp = seq_fitness_percent[i]
                        temp_prev = seq_fitness_percent[i - 1]
                        # add the previous value to the next percentage
                        seq_fitness_percent[i] = (temp[0], temp[1], temp_prev[2] + temp[2])
                percentage = random.random()
                #default first values
                permute1 = seq_fitness_percent[0][0]
                permute2 = seq_fitness_percent[1][0]
                for i in range(len(seq_fitness_percent)):
                    if percentage < seq_fitness_percent[i][2]:
                        permute1 = seq_fitness_percent[i][0]
                        break
                percentage = random.random()
                for i in range(len(seq_fitness_percent)):
                    if percentage < seq_fitness_percent[i][2]:
                        permute2 = seq_fitness_percent[i][0]
                        break
                population[j] = soko_recombine(permute1, permute2)
                population[j] = soko_mutate(list(population[j]), gene_pool, pmut)
        else:       #if the total score is 0 reset the population.
            population = init_population(POP_NUMBER, ['u','d','l','r'], SOL_LENGTH)
    end = time.time()
    total = end - start
    compute_fitness(population, fitness_dict)
```

```
    m = 0
    for x in range(len(population)):
        m1 = (fitness_fn(fitness_dict, population[x]))
        if m1 >= m:
            m = m1
            best = population[x]
    return total, best
```

```
def init_population(population_number, gene_pool, solution_length):
    # population_number : Number of indiviuals in the population_number
    # gene_pool         : A list of the options that the genetic algorithm can choose from.
    # solution_length   : The number of moves we allow a test to take to get to a solution.
    g = len(gene_pool)
    population = []
    # creates individuals for the population
    for i in range(population_number):
        new_individual = [gene_pool[random.randrange(0,g)] for j in range(solution_length)]
        population.append(new_individual)
    return population
```

```
def soko_recombine(x, y):
    c = random.randrange(0, SOL_LENGTH)
    return x[:c] + y[c:]
```

```
def soko_mutate(x, gene_pool, mutate_rate):
    # no mutation occurs
    if random.uniform(0,1) >= mutate_rate:
        return x
    else:
        # mutation occurs
        g = len(gene_pool)
        c = random.randrange(0, SOL_LENGTH)
        r = random.randrange(0, g)
        new_gene = gene_pool[r]
        return x[:c] + [new_gene] + x[c + 1:]
```

9 Code Used

AIMA Python:
https://github.com/aimacode/aima-python

Python Implementation of Sokoban from Colombia:
https://github.com/janecakemaster/sokoban/tree/ef5825801c4fdb853c2079cd322811a158def681

Python Munkres:
https://pypi.org/project/munkres/