A Simulated Annealing solution to the Tetravex game

Abstract This report shows my solution to the Tetravex game. As directed the program makes use of the Simulated Annealing algorithm to find its solution. This report shall first present the Simulated Annealing algorithm, then it shall present some notes on the implementation process of this algorithm into the final program. Then the following section shall present some of the optimizations that were considered and implemented to improve the execution time. Then finally the report shall present the final results, some possible improvements, and shall end with a conclusion.

Contents

1 Introduction **1**
- 1.1 The Tetravex game . . . 1
- 1.2 Program compilation & execution . . . 1

2 Simulated Annealing **1**
- 2.1 Finite Makov Decision Process . . . 2
- 2.2 Simulated Annealing algorithm . . . 2

3 Optimizations **3**
- 3.1 Simulated Annealing optimization . . . 3
- 3.2 General program optimizations . . . 3

4 Results **4**
- 4.1 Execution time sampling . . . 4
- 4.2 Execution time analysis . . . 4

5 Improvements **4**

6 Conclusion **4**

Jose A. Henriquez Roa

February 6, 2022

1. Introduction

1.1. The Tetravex game

The tetravex game is an edge-matching puzzle inspired by some of the work presented by Knuth began in his book series The Art of Computer Programming [1]. The game itself was invented by Scott Ferguson for the Windows Entertainment Pack which was released in 1991. Since then an open source version has been made available in the GNOME Games collection under the name `gnome-tetravex`. This same executable was used to generate some solvable board examples that were used to test the presented program implementation.

The game itself is made up of a square grid board and a set of tiles. In the gnome-tetravex version of the game, the board size ranges from a 2x2 to a 6x6 grid. The titles themselves are made up of four numbers (usually 0 to 9) at each of side, and the total number of tiles matches the grid size. The goal of the game is to place all of the tiles in the grid as such that each side of each tile matches with the number of their four adjacent neighbors. A solved tetravex board example is given in the following figure 1.

```
+-----+-----+-----+
|  7  |  6  |  9  |
| 5 3 | 3 4 | 4 5 |
|  0  |  9  |  7  |
+-----+-----+-----+
|  0  |  9  |  7  |
| 1 5 | 5 8 | 8 1 |
|  1  |  5  |  4  |
+-----+-----+-----+
|  1  |  5  |  4  |
| 5 8 | 8 9 | 9 6 |
|  6  |  9  |  1  |
+-----+-----+-----+
```

Figure 1: 3x3 Solved board example

1.2. Program compilation & execution

Before moving on to the following section presenting the algorithm and the implementation we shall first give some few notes on the compilation and execution of the program.

There is a total of 3 files in the presented program, two source files and one header file. These are respectively `main.cc`, `tetravex.cc` and `tetravex.hh`. All of the logic related to loading input files, solving and writing out the output file is contained in the tetravex.cc source file.

As instructed the entire program can be executed by simply executing the following command the root directory:

```
$ g++ -std=c++17 *.cc
```

This will output an `a.out` binary file that should be executed through the following command:

```
$ ./a.out in.txt out.txt
```

Where `in.txt` should be in the following format:

```
7965\n
6655 @\n
3767\n
5649\n
```

```
+-----+-----+
|  7  |  6  |
| 9 6 | 6 5 |
|  5  |  5  |
+-----+-----+
|  3  |  5  |
| 7 6 | 6 4 |
|  7  |  9  |
+-----+-----+
```

Figure 2: 2x2 Board input file contents example (left) and associated board representation (right)

Where each line defines a tile in the format `<north><west><east><south>` in their respective initial position in the grid. The `@` symbol specifies that the tile is fixed in its initial position and shall not be moved while attempting to solve the board. This implementation handles input files without any fixed tile marker.

The output file shall present the solved board in the same format as the input file, but without the use of any fixed tile markers.

2. Simulated Annealing

This section shall present the algorithm and the main concepts that were used to solve the Tetravex problem in the presented program implementation. The algorithm that was used to solve the problem is the simulated annealing algorithm introduced in the 1983 paper “Optimization by Simulated Annealing”

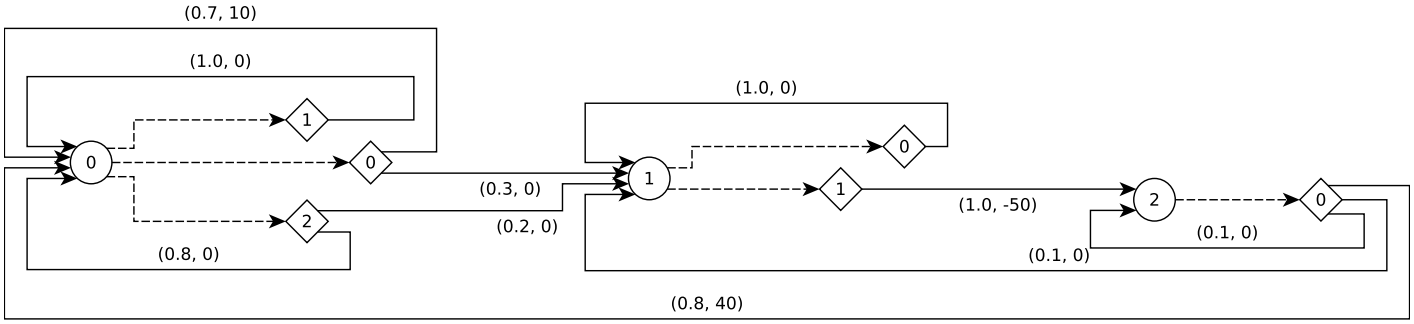

Figure 3: Markov Decision Process diagram. The circles represent the states and the diamonds the actions. In this diagram there is only one possible reward when passing to a state s' after choosing an action a, this is not always the case.

[3], this one is based on the Metropolis Hastings algorithm which itself was presented in the 1953 paper "Equation of State Calculations by Fast Computing Machines" [2].

This section shall first present some concepts of the Finite Markov Decision Process which are required before moving on to the following section on the Simulated Annealing algorithm.

2.1. Finite Makov Decision Process

An example of a Makov Decision Process (MDP) is given in figure 3. To present what an MDP is we shall use the agent-environment analogy. An MDP is a stochastic model that defines all of the possible interaction between an agent in a specific environment. At each state (circle) the agent is presented a set of actions (diamonds). When an action a is chosen by an agent in a given state s then the environment responds by changing the state to a new one s' and giving a reward r depending on a predefined (sometimes unknown) transition probability.

Do note however that the following algorithms use a special type of MDPs called Markov Chains. Which are essentially are MDPs with only one action and for which the rewards are neglected (e.i set to zero). The choice to present MDPs instead of directly presenting Markov Chains was to introduce the concept of *reward* which is used to simplify the presentation of the following algorithms.

2.2. Simulated Annealing algorithm

The Simulated Annealing algorithm is based on the Metropolis–Hastings algorithm, The later is a method of generating a Markov Chain from a probability distribution. And the former is a variant of this one used for optimization.

The simulated annealing algorithm is rather straightforward at each iteration the algorithm considers transitioning to a new state s' from the current state s. The probability of actually transitioning to the new state depends on two components. First a loss function, which defines a relative ordering in between the possible states, more attractive states will have a smaller loss. And a temperature, which regulates how likely the algorithm is to transition to less attractive states, with a higher temperature meaning that less attractive states are more likely to be considered.

There are many variants for the acceptance probability, however, the one that was considered for the presented implementation is:

$$p = e^{\frac{-L(s')-L(s)}{T}} \tag{1}$$

Where L is the loss function, T is the temperature, s' is the new state being considered and s is the current one.

The algorithm attempts to converge to an optimal solution by decreasing the temperature at each iteration, this has the effect of first allowing the algorithm to sufficiently explore the state space when the temperature is high and constraining it to choose the

Incl.	Self	Called	Function	Location
100.00	0.00	1	Tetravex::solve(int const&, ...	cmkv: tetravex.cc
100.00	0.00	393 157	chooseTilesToSwap(unsign...	cmkv: tetravex.cc
100.00	0.00	(0)	main	cmkv: main.cc
58.43	58.43	393 157	open	libc-2.31.so: open64.c
58.43	0.00	393 157	std::random_device::_M_ini...	libstdc++.so.6.0.28
58.43	0.00	393 157	std::random_device::rando...	cmkv: random.h, basic_string.h
28.56	28.56	393 157	read	libc-2.31.so: read.c
28.56	0.00	393 157	std::random_device::_M_ge...	libstdc++.so.6.0.28
28.56	0.00	393 157	std::random_device::operat...	cmkv: random.h
13.01	13.01	393 157	close	libc-2.31.so: close.c
13.01	0.00	393 157	std::random_device::_M_fini()	libstdc++.so.6.0.28
13.01	0.00	393 157	std::random_device::~rand...	cmkv: random.h
0.00	0.00	259 102	__expf_sse2	libm-2.31.so: e_expf.c, math_config.h

Figure 4: Code profile

most attractive states when the temperature is low.

In regards to the loss function we note it's similarities with the rewards in MDPs. Drawing a parallel to some Reinforcement Learning (RL) concepts, the loss function defines the same concept as the optimal state value function $v_*(s)$ is RL, this is is defined as the expected *return* when starting in state s and choosing future actions according to the optimal policy. We can then interpret the problem the Simulated Annealing algorithm is trying to solve as trying to find the optimal state given the state value function.

Going back to the original Tetravex problem. In the presented implementation the state space defined by all of the possible grid configurations. And the loss function is defined such as it decreases by 1 for every matching side of each tile. Each matching pair is counted exactly only once and not twice, or once for each tile.

The initial loss is set to 0 which makes the optimal loss equal to $l^* = -2(n^2 - n)$, where n is the side of the grid being solved.

3. Optimizations

3.1. Simulated Annealing optimization

The algorithm was optimized in two aspects. First, by searching for a good initial Temperature T_0. And second, by making modification to the annealing schedule (process used to decrease the temperature at each iteration).

The search for T_0 parameter was done manually by tuning this one all through the development by testing it against multiple input board of different sizes.

For the annealing schedule, this one required more research to be performed. Initially the temperature was decreased linearly and was simply reset when the algorithm did not manage to find the solution before the temperature reached 0. However, this was much improved upon, currently the algorithm is set to decrease the temperature as such that 0 is reached at a given predefined max time-step t_{max}. With this new method there is no longer need to completely reset the board (by resetting the temperature). Now, when the algorithm does not manage to find a solution in time and reaches t_{max}, t_{max} will simply be increased by a predefined value (which will also increase the temperature) and allow the algorithm to the out of any sort of local minima, all without having to restart from 0.

3.2. General program optimizations

Some additional optimization were performed on the source code side of things. During the implementation the algorithm featured a surprisingly high execution time, even for board of small size. Profiling the code resulted in the `callgrind` results shown in figure 4. In this ones we see that by far most of the execution time was spent on the member function of the Tetravex class `chooseTilesToSwap()`, which itself was part of the smallest simpler function in the main loop in `solve()`. Further inspection showed that there was a unusually high amount time being spent on kernel calls. This was later on found out to be due to the usage of the `std::default_random_engine`

class, which was used to generate the random number used to randomly select movable tiles. Fixing this issue allowed to decrease the execution time by a factor of over 100.

4. Results

4.1. Execution time sampling

The two Bash and Python presented in figures 5 and 6 were used for both tuning the parameters of the Simulated Annealing algorithm as well as for computing the final execution time analysis presented in the following section.

```
g++ -std=c++17 *cc

grid_size=6

echo "exec_time" > $grid_size.csv
for i in $(seq 100); do
  ./a.out in-$grid_size.txt out-$grid_size.txt \
  >> $grid_size.csv
done

python3 stat.py $grid_size.csv
```

Figure 5: time.sh: Bash script for collecting execution times

```
import sys, pandas

print(pandas.read_csv(sys.argv[1]).describe())
```

Figure 6: stat.py: Python for getting execution time statistics

The `time.sh` script was used to generate a CSV file containing the execution times of the `solve()` function. And the `stat.sh` was used to generate some useful statistics from the previously sampled execution times.

4.2. Execution time analysis

The following table show multiple statistics related to the sampled execution times of the presented program on the multiple board sizes ranging from 2x2 to 6x6 in seconds. For these tests only input boards without fixed tiles were given to the program.

Stat.	2x2	3x3	4x4	5x5	6x6
mean	6.4e-5	0.02	0.27	0.97	10.92
std	2.7e-5	0.02	1.34	1.54	8.96
min	1.6e-5	2.1e-3	7.1e-3	0.02	0.28
25%	4.7e-5	4.5e-3	9.2e-3	0.10	4.30
50%	5.9e-5	0.01	0.01	0.19	8.58
75%	7.7e-5	0.03	0.03	1.22	15.01
max	1.4e-5	0.12	10.56	8.17	46.68

Figure 7: Execution times statistics averaged accross 100 samples in seconds

5. Improvements

The following is a list of possible improvements:

1. Consider other annealing algorithms aside from the Simulated Annealing one. Or implement other methods for solving the problem all together, for this Reinforcement Learning could be considered.

2. Do some additional parameter tuning.

3. Implement a multi-threaded version that for each thread shuffles the board and launches a `solve()` instance. This method would decrease the execution time variance.

6. Conclusion

As directed we made use of a Metropolis–Hastings variant, in this case Simulated Annealing, to implement a Tetravex solver. Through the implementation process, extensive research was made on the algorithm and all of its related concepts. Particularly after having implemented a working version, while trying to further optimize the current implementation.