ROYAL HOLLOWAY, UNIVERSITY OF LONDON

MSc THESIS

Solving the Sliding Puzzle & Rubiks' Cube by Deep Reinforcement Learning

---

Author:
Francois BERRIER

Supervisor:
Pr. Chris WATKINS

A thesis submitted in fulfillment of the requirements for the degree of MSc in Artificial Intelligence

July 17, 2022

“The best advice I’ve ever received is ’No one else knows what they’re doing either’”

Ricky Gervais

ROYAL HOLLOWAY, UNIVERSITY OF LONDON

Abstract

Computer Science Department

MSc in Artificial Intelligence

Solving
the
Sliding Puzzle
&
Rubiks' Cube
by
Deep Reinforcement Learning

by Francois BERRIER

Motivation
Reinforcement Learning (**RL**), exposed with brilliant clarity in the Sutton book (Sutton and Barto, 2018), has until recently known less success than we might have hoped for. Its framework is very appealing and intuitive. In particular, the mathematical beauty of Value iteration and Q iteration (Watkins, 1989) for discrete state and action spaces, blindly iterating from *any* initial value, is quite profound. Sadly, it had until recently proven hard to achieve practical success with these methods.
Inspired however by the seminal success of Deep Mind's team in using Deep Reinforcement Learning (**DRL**) to play Atari games (Mnih et al., 2013) and to master the game of Go (Silver et al., 2016), researchers have in recent years made a lot of progress towards designing algorithms capable of learning and solving, *without human knowledge*, the Rubiks Cube (**RC**) - as well as similar single player puzzles - by using Deep Q-Learning (**DQR**) (McAleer et al., 2018a) or search and (**DRL**) value iteration (McAleer et al., 2018b).
In this project, I will attempt to implement a variety of solvers combining A^* search and heuristics, some of which will be handcrafted, others which I will train on randomly generated sequences of puzzles using (**DL**) or (**DRL**), to solve the 15-puzzle (and variations of different dimensions), as well as the Rubik's cube.

Organisation of this thesis
In the first chapter, I will quickly describe what I hope to get out of this project, both in terms of personal learning, as well as in terms of tangible results (solving some puzzles!). In chapter 2, I will do a quick recap of the different methods that I will use in the project. Chapter 3 will be dedicated to discussing the mathematics of the sliding puzzle (**SP**) and the (**RC**). I might throw a few random (but hopefully interesting) observations in there and give some references for the keen reader. I will then give an overview in chapter 4 of the code base I have developed - and put in the open on my github page (Berrier, 2022) - to complete this project. Finally, in chapters 5 and 6, I will present all my various results on respecively the sliding puzzle and the Rubiks' cube.

Acknowledgements

Contents

List of Abbreviations

AIPnT	**A**rtificial **I**ntelligence **P**rinciples and **T**echniques
CS	**C**omputer **S**cience
CV	**C**omputer **V**ision
DL	**D**eep **L**earning
DQL	**D**eep **Q**-**L**earning
DRL	**D**eep **R**einforcment **L**earning
ML	**M**achine **L**earning
NLP	**N**atural **L**anguage **P**rocessing
RC	**R**ubiks' **C**ube
RL	**R**einforcment **L**earning
RHUL	**R**oyal **H**olloway, **U**niversity of **L**ondon
SP	**S**liding **P**uzzle

Chapter 1

Objectives

1.1 Learning Objectives

Back when I studied Financial Mathematics, almost 2 decades ago, it was all about probability theory, stochastic calculus and asset (in particular derivatives) pricing. These skills were of course very sought after in the field of options trading, but were also often enough to get a job in algorithmic or systematic trading. By the middle of the 2010s, with the constant advances in computing power and storage, the better availability of off-the-shelves libraries and data sets, I witnessed a first revolution: the field of machine learning became more and more prominent and pretty much overshadowed other (more traditional maths) skills. More recently, a second revolution has taken not only the world of finance, but that of pretty much every science and industry, by storm: we are now in the artificial intelligence age. In 2019-2020, I decided it was time to see by myself what this was all about, and if the hype was justified. What better way to do that than embark on a proper MSc in Artificial Intelligence?
Of all the modules I have studied over the last two years of the Royal Holloway MSc in AI, I have been the most impressed by DL and NLP (itself arguably largely an application of DL) and particularly interested in AIPnT, especially our excursion in the field of graphs search (a very traditional CS topic, but which somehow I had not yet had a chance to study in much details). Even though I still believe there is a tremendous amount of malinvestment everywhere, due in good part to the inability of the average investor to distinguish between serious and scammy AI applications and startups (the same obviously goes for blockchain applications, which might warrant another MSc?), I have totally changed my mind around the potential of DL, DRL and NLP and think they are incredibly promising. I have been astonished to see by myself, through several of the courseworks we have done during the MSc, how incredibly efficient sophisticated ML, DL and DLR algorithms can be, when applied well on the right problems. Sometimes they just vastly outperform more naive and traditional approaches to the point of rendering older approaches entirely obsolete (e.g CV, NLP, game solvers, etc...).
For the project component of the MSc, I thought it would be interesting (and fun) for me to try and apply some of the DL, DRL and search techniques (from AIPnT) to a couple of single-player games, such as the sliding puzzle (of which some variations are well known under different names, e.g. the 8-puzzle and 15-puzzle) and of course the Rubiks' cube. I am in particular looking to solidify my understanding of DRL by implementing and experimenting with concrete (though arguably of limited practical use) problems.

1.2 Project's Objectives

I am hoping with this project to implement and compare a few different methods to solve the SP and the RC. Both these puzzles have tremendously large state spaces (see section Games for details) and only one goal state. I am therefore likely to only succeed with reasonably small dimensional puzzles, especially since I have chosen for simplicity to implement things in Python. Depending on the progress I will be able to make in the imparted time, I am hoping to try a mix of simple searches (depth first search, breadth first search, A* with simple admissible heuristics), then more advanced ones such as A* informed by heuristics learnt via DL and DRL, as well as try different architectures and network sizes and designs for the DL and DRL heuristics. Time permitting I would like to give a go at DQL, and maybe also compare things with some open-source domain-specific implementations (for instance a Kociemba Rubik's algorithm implementation, see e.g. Tsoy, 2019).
Along the way, I am also hoping to learn a bit about these two games that I have chosen to work on, and maybe make a couple of remarks on them that the reader of this thesis might find interesting.

Chapter 2

Deep Reinforcement Learning Search

2.1 Reinforcement Learning

blabla

2.2 Deep Learning

blabla

2.3 Graph Search & Heuristics

See Dechter and Pearl, 1985

2.4 Deep Reinforcement Learning Heuristics

blabla

2.5 Deep Q-Learning

see Watkins and Dayan, 1992

Chapter 3

Puzzles

3.1 Sliding Puzzle

3.1.1 History - The 15 Puzzle

The first puzzle I will focus on is the sliding puzzle (see Wikipedia, 2022).The 15-puzzle seems to have been invented in the late 19th century by Noyes Chapman (see WolframMathWorld, 2022), who applied for a patent in 1980. In 1879, a couple of interesting notes (Johnson and Story, 1879), published in the American Journal of Mathematics proved that exactly half of the 16! possible ways of setting up the 16 tiles on the board lead to solvable puzzles. A more modern proof can be found in Archer, 1999.

Since then, several variations of the 15-puzzle have become popular, such as the 24-puzzle. A rather contrived but interesting one is the coiled 15-puzzle (Segerman, 2022), where the bottom-right and the top-left compartments are adjacent; the additional move that this allows renders all 16! configurations solvable.

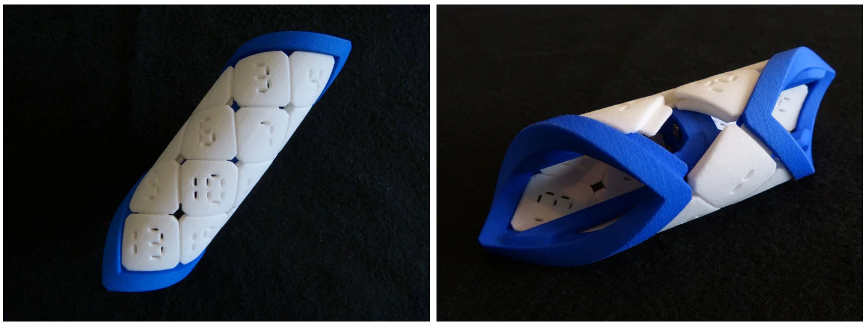

FIGURE 3.1: Coiled 15-puzzle

It is rather easy to see why this is the case, let us discuss why: given a configuration c of the tiles, let us define the permutation p(c) of this configuration according to the following schema: we enumerate the tiles row by row (top to bottom), left to right for odd rows and right to left for the even rows, ignoring the empty compartment. For instance, the following 15-puzzle c:

1	6	2	3
5	10	7	4
9	15	14	11
13	12		8

we have p(c) = (1, 6, 2, 3, 4, 7, 10, 5, 9, 15, 14, 11, 8, 12, 13).
It is easy to see that the parity of p(c) cannot change by a legal move of the puzzle. Indeed, p(c) is clearly invariant by lateral move of a tile, so its parity is invariant too. A vertical move of a tile will displace a number in p(c) by an even number of positions right or left. For instance, moving tile 14 into the empty compartment above it results in a new configuration c_2:

1	6	2	3
5	10	7	4
9	15		11
13	12	14	8

with $p(c_2)$ = (1, 6, 2, 3, 4, 7, 10, 5, 9, 15, 11, 8, 14, 12, 13), which is equivalent to moving 14 by 2 positions on the right. This obviously cannot change the parity since exactly 2 pairs of numbers are now in a different order, that is (14, 11) and (14, 8) now appear in the respective opposite orders as (11, 14) and (8, 14).

This is the crux of the proof of the well known necessary condition (even parity of p(c)) for a configuration c to be solvable (see part I of Johnson and Story, 1879).

In the case of the coiled puzzle, we can clearly solve all configurations of even parity, since all the legal moves of the normal puzzle are allowed. In addition, we can for instance transition between the following 2 configurations, which clearly have respectively even and odd parities:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

	2	3	4
5	6	7	8
9	10	11	12
13	14	15	1

Since it is possible to reach an odd parity configuration, and by symmetry arguments, we conclude we can solve all 16! configurations.

3.1.2 Search Space & Solvability

In this thesis, as well as in the code base (Berrier, 2022) I have written to do this project, we will consider the general case of a board with n columns and m rows, where $(n, m) \in \mathbb{N}^{+2}$, forming $n * m$ compartments. $n * m - 1$ tiles, numbered 1 to $n * m - 1$ are placed in all the compartments but one (which is left empty), and we can slide a tile directly adjacent to the empty compartment into it. Notice from a programming and mathematical analysis perspective, it is often easier to equivalently think of the empty compartment being moved into (or swapped with) an adjacent tile. Starting from a given shuffling of the tiles on the board, our goal will be to execute moves until the tiles in ascending order: left to right, top to bottom (in the usual western reading order), the empty tile being at the very bottom right.

Note that the case where either n or m is 1 is uninteresting since we can only solve the puzzle if the tiles are in order to start with. For instance, in the (n, 1) case, we can only solve n of the $\frac{n!}{2}$ possible configurations. We will therefore only consider the case where both n and m are strictly greater than 1.

3.1.3 Optimal Cost & God's Number

Let us fix n and m, integers strictly greater than 1 and call $\mathcal{C}_{(n,m)}$ the set of all $\frac{(n*m)!}{2}$ solvable configurations of the n by m sliding-puzzle. For any $c \in \mathcal{C}_{(n,m)}$ we define the optimal cost $\mathcal{O}(c)$ to be the minimum number of moves among all solutions for c. Finally we define $\mathcal{G}(n, m)$, God's number for the n by m puzzle as $\mathcal{G}(n, m) = \max_{c \in \mathcal{C}_{(n,m)}} \mathcal{O}(c)$. Note that since $\frac{(n*m)!}{2}$ grows rather quickly with n and m, it is impossible to compute $\mathcal{G}$ except in rather trivial cases.

A favourite past time among computer scientists around the glove is therefore to search for more refined lower and upper bounds for $\mathcal{G}(n, m)$, for ever increasing values of n and m. For moderate n and m, we can actually solve optimally all possible configurations of the puzzle and compute exactly $\mathcal{G}(n, m)$ (using for instance A^* and an admissible heuristic (recall 2.3, and we shall see modest examples of that in the results section later). For larger values of n and m (say 5 by 5), we do not know what the God number is. Usually, looking for a lower bound is done by *guessing* hard configurations and computing their optimal path via an optimal search. Looking for upper bounds is done via smart decomposition of the puzzle into disjoint nested regions and for which we can compute an upper bound easily (either by combinatorial analysis or via exhaustive search). See for instance Karlemo and Ostergaard, 2000 for an upper bound of 210 on $\mathcal{G}(5, 5)$.

A very poor lower bound can be always obtained by the following reasoning: each move can at best explore three new configurations (4 possible moves at best if the empty tile is not on a border of the board (less if it is): left, right, up, down but one of which is just going back to an already visited configuration). Therefore, after p moves, we would span at best $\mathcal{S}(p) = \frac{3^{p+1}-1}{2}$ configurations. A lower bound can thus be obtained for $\mathcal{G}(n, m)$ by computing the smallest integer p for which $\mathcal{S}(p) \geq \frac{(n*m)!}{2}$

3.2 Rubiks' Cube

blabla

Chapter 4

Code

4.1 Code organisation

The code I have developed for this project is all publicly available on my github page (Berrier, 2022). It can easily be installed using the setup file provided, which makes it easy to then use Python's customary import command to play with the code. The code is organised in several sub modules and makes use of factories in plenty of places so that I can easily try out different puzzles, dimensions, search techniques, heuristics, network architecture, etc... without having to change anything but configuration or parameter in the command line. Here is a visual overview of the code base with the main dependencies between the main submodules and classes:

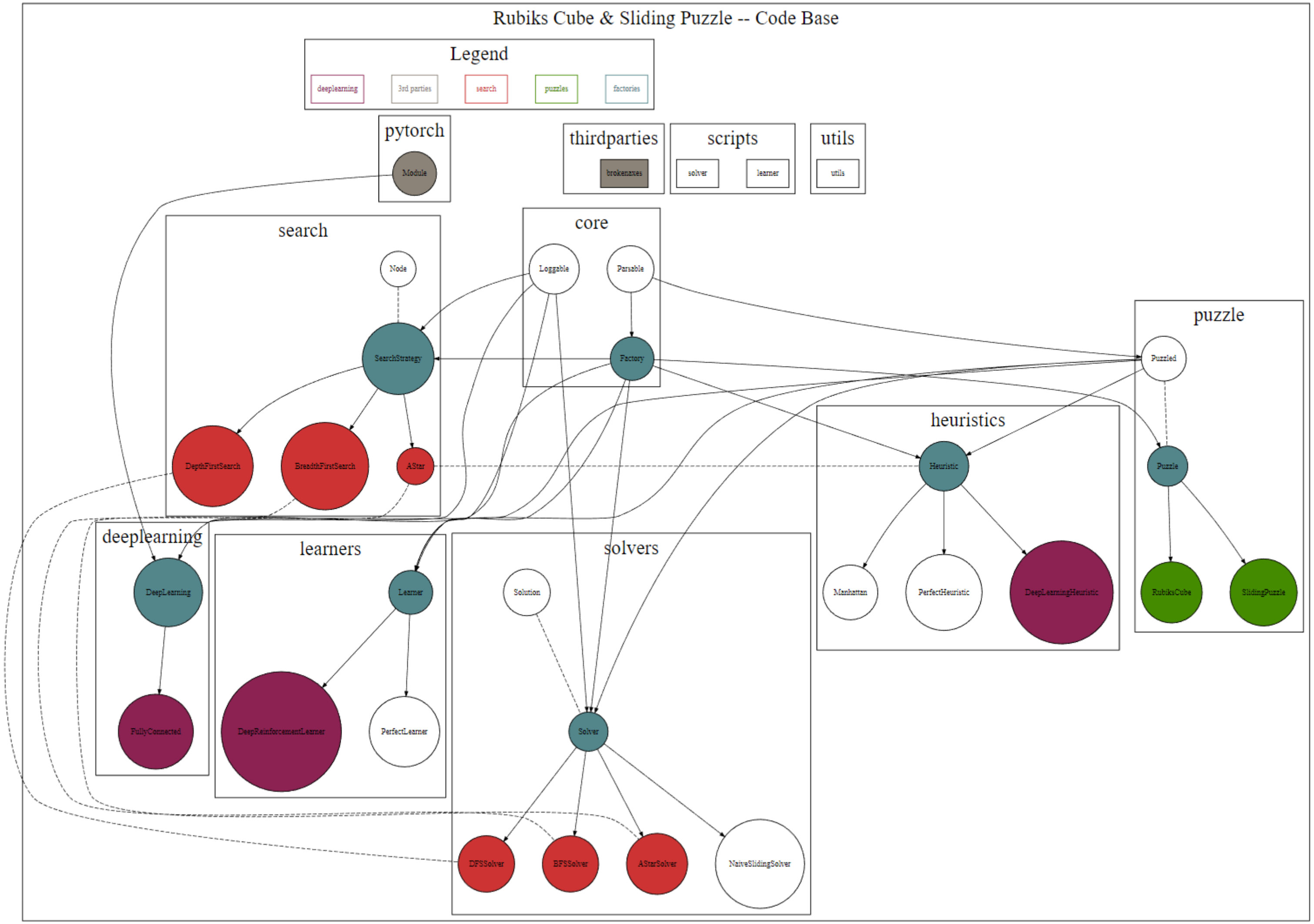

FIGURE 4.1: Code base

Let me describe what each submodule does:

4.1.1 rubiks.core

This submodule contains base classes that make the code base easier to use, debug, and extend. It contains the following:

- Loggable: a wrapper around Python's logger which automatically picks up classes' names at init and format things (dict, series and dataframes in particular) in a nicer way.
- Parsable: a wrapper around ArgumentParser, which allows to construct objects in the project from command line, to define dependencies between object's configurations and to help a bit with typing of configs. The end result is that you can pretty much pass **kw_args everywhere and it just works.
- Factory: a typical factory pattern. Concrete factories can just define what widget they produce and the factory will help construct them from **kw_args (or command line, since Factory inherits from Parsable)

4.1.2 rubiks.puzzle

This submodule contains:

- Puzzle: a Factory of puzzles. It defines states and actions in the abstract, and provides useful functions to apply moves, shuffle, generate training sets, tell if a state is the goal, etc. Puzzle can manufacture the two following types of puzzles:
- SlidingPuzzle. Implements the states and moves of the sliding puzzle.
- RubiksCube. Implements the states and moves of the Rubik's cube. In addition, in contains a Puzzled base class which most below inherit from. That allow e.g. heuristics, search algorithms, solvers and learners to know what puzzle and dimension they operate on, without having to reimplement these basic facts in each of them.

4.1.3 rubiks.search

This modules contains graph search strategies. I have actually reused the code I implemented for one of the AIPnT assignments. It contains the following classes:

- Node: which contains the state of a graph, as well as link to the previous (parent) state, action that leads from the latter to the former and the cost of the path so far.
- SearchStrategy, a Factory class which can instantiate the following three types of search strategies to find a path to a goal:
- BreadthFirstSearch, which is obviously an optimal strategy, but not particularly efficient.
- DepthFirstSearch, which is not an optimal strategy, and also generally not particularly efficient.
- AStar, which is optimal, and as efficient as the heuristic it makes use of is.

4.1.4 rubiks.heuristics

This module contains base class Heuristic, also a Factory. Heuristic can instantiate the following heuristics, which we can use in the AStar strategy from the previous section:

- Manhattan: at current time of writing, this is specific to the SP and will be discussed in more details in **??**,
- PerfectHeuristic: this reads from a data base the optimal costs, pre-computed by the PerfectLearner (see below 4.1.6)
- DeepLearningHeuristic: this uses a network which has been trained using DRL by the DeepReinforcementLearner (see below 4.1.6)

4.1.5 rubiks.deeplearning

This module is a wrapper around Pytorch. It contains:

- DeepLearning: a Puzzled Loggable Factory that can instantiate some configurable deep networks, and provide the necessary glue with the rest of the code base so that puzzles be seemlessly passed to the networks and trained on.
- FullyConnected: wrapper around a Pytorch fully connected network, with configurable layers and size.
- Convolutional: tdb

4.1.6 rubiks.learners

This module implements learners, which learn something from a puzzle, store what they learnt, and can display interesting things about what they learnt.

- Learner is a Puzzled Loggable Factory. It provides some common code to learners (to save or purge what they learnt), kick off learning and plot results. Concrete derived implementation define what and how they learn, and what interesting they can display about this learning process. Currently the two implemented learners are:
- PerfectLearner: It instantiates an optimal solver (A^* with a configurable heuristic - but will only accept heuristic that advertise themselves as optimal. The learning consists in generating all the possible configuration of the considered puzzle, solve them with the optimal solver, and save the optimal cost of it as well as those of the whole solution path. The code allows for parallelization, stop and restart so that we can run on several different occasions and keep completing a database of solutions if necessary or desired. Once the PerfectLearner has completed its job, it can display some interesting information, such as the puzzle's God's number, the distribution of number of puzzles versus optimal cost, the hardest configuration it came across, and how long it took it to come up with the full knowledge of that puzzle. I will show in section 4.2.1 how to run an example. Notice that for puzzles of too high dimension, where my computing resources will not allow to solve exhaustively all the configurations of a given dimension, this class can still be used to populate a data base of optimal costs, which can then be used by DeepLearner. If it is to be used this way, the PerfectLearner can be configured to use perfectly random configurations to learn from, rather than going through the configurations one by one in a well defined order.
- DeepLearner tbd

- DeepReinforcementLearner: It instantiates a DeepLearning (network), and trains it using DRL. It then saves the trained network, which can then be used in the DeepLearningHeuristic we have seen earlier in section 4.1.4. The

4.1.7 rubiks.solvers

This module implements solvers, which solve puzzles. The base class Solver is a Factory of solvers, and in addition to being able to instantiating the following types of solvers, can run different solvers through a similal sequences of random puzzles (for various increasing degrees of difficulty (shuffling), and/or perfectly shuffled ones) and display a comparison of how they perform in a number of metrics.

- DFSSolver
- BFSSolver
- AStarSolver
- NaiveSlidingSolver

4.1.8 rubiks.scripts

Finally it is worth noting that the code will save on disk a lot of data (e.g. the learners will save what they have learnt, e.g. a Pytorch network or a data base of optimal costs, the performance comparison will run solvers versus very many configurations of puzzles and save the results for later being able to display) etc... The base of the tree to save all this data can be chosen by setting up the "RUBIKSDATA" environment variable. If not, it will go somewhere in you HOME :)

4.2 Learners

Here I show how to run some examples... TBC

4.2.1 Perfect Learner

4.2.2 Deep Learner

blabla

4.2.3 Deep Reinforcement Learner

blabla

4.3 Solvers

4.3.1 Blind search

BFS

DFS

4.3.2 Naive Sliding Puzzle Solver

As a comparison point, I have implemented a naive sliding puzzle solver, which does what most beginner players would intuitively do when solving the sliding puzzle by hand: solve the top row, then the left column, and keep iterating until done. Notice that once either the top row or left column is solved, there is no longer any need to modify it, we have simply reduced the problem to a sub-problem of reduced dimension. For the interested reader, the details of the algorithm are as follows:

- if n and m are both equal to 2, we just keep moving the empty tile clock-wise until the puzzle is solved. Notice that this is bound to work, since moving clock-wise or counter-clock-wise are the two ony possible moves, and one of them is just un-doing the other one.
- if n $\geq$ m, we solve the top row
- otherwise we solve the left column

Solving the top row of a n by m puzzle (left column is similar, mutatis mutandis, so I will not detail it) is accomplished as follows:

1. we sort the tiles (which since we are potentially dealing with a sub-problem, are no longer necessarily 1 to $m * n - 1$), and select the m smaller ones $t_1, ..., t_{m-1}, t_m$.
2. we place t_m in the bottom-right corner
3. we place $t_1, ..., t_{m-2}$ to their respective positions (in that order, and making sure not to undo anything as we do so)
4. we place t_{m-1} in the top-right corner
5. we then move t_m just under t_{m-1}
6. we move the empty tile to the left of t_{m-1}
7. finally we move the empty tile right and then down to put t_{m-1} and t_m in place.

In order to move the tiles, we have written a few simple routines which can move the empty tile from its current position next to (above, below, left or right) any tile, and then can move that tile to another position, all the while avoiding to go through previously moved tiles (hence the particular order in which we move the different tiles above). The only case where the above algorithm can get stuck is when both n and m are equal to 3 and that by step 6 we might end-up with t_3 under the empty tile. We have handcrafted a sequence of moves to solve this particular position.

As a concrete example, let us assume we started with the following (n=6, m=6) puzzle:

14	27	6	2	5	18
21	29	13	23	35	30
26	3	7	9	24	19
22	12	11	17	16	33
32	10	20	25	34	28
8	4	15	31		1

After one call to solve the top row and the left column, we are left with solving the following 5 by 5 sub-puzzle:

9	17	27	18	35
23	11	15	24	21
20	8	29	33	10
22	30	14	32	16
	12	26	34	28

step 1 above will decide to solve the top row by placing $t_1, \ldots, t_5 = 8, 9, 10, 11, 12$ in that order as the top row. Steps 2 to 7 will yield in order:

9	17	27	18	35
23	11	15	24	21
20	8	29	33	10
22	30	14	32	16
	12	26	34	28

9	17	27	18	35
23	11	15	24	21
20	8	29	33	10
22	30	14	32	16
12		26	34	28

8	9	10		18
17	15	27	24	35
11	23	29	21	33
20	22	14	32	16
12	30	26	34	28

8	9	10		11
23	29	21	18	24
17	15	27	33	35
20	22	14	32	16
12	30	26	34	28

8	9	10	18	11
29	27	32		12
23	21	33	35	24
15	17	22	14	16
30	20	26	34	28

8	9	10	11	12
29	27	32	18	
23	21	33	35	24
15	17	22	14	16
30	20	26	34	28

and we are left with solving the bottom sub-puzzle (n=4,m=5), etc...

4.3.3 Kociemba

BTD

4.3.4 A*

manhattan heuristic

- **perfect heuristic**
- **deep learning heuristic**
- **deep reinforcement learning heuristic**

Chapter 5

Results - Sliding Puzzle

5.1 Low dimension

As mentioned in chapter 3, the state space cardinality for the SP grows very quickly with n and m. Here are the only dimensions which have less than 239.5 millions states. Note I am also only considering n $\leq$ m since (p, q) can always be solved if we know how to solve (q, p):

n m	2	3	4	5
2	12	360	20,160	1,814,400
3		181,440		

In this section, I will discuss **full** results for these 5 puzzles. In order to fully solve them, one can simply use rubiks.scripts.learner, setting up the PerfectLearner with A* and manhattan heuristic, or instantiate directly a PerfectLearner as have seen in section 4.2.1 I obtained the following God numbers for these puzzles:

n m	2	3	4	5
2	6	21	36	51*
3		31		

* provisional result

and the most difficult puzzles (requiring a number of steps equal to their respective God number to solves):

<u>Most difficult 2x2 (6 moves):</u>

	3
2	1

<u>Most difficult 2x3 (21 moves):</u>

4	5	
1	2	3

Most difficult 2x4 (36 moves):

	7	2	1
4	3	6	5

Most difficult 2x5 (51* moves):

	3	4	2	6
5	9	8	7	1

* provisional result

Most difficult 3x3 (31 moves):

8	6	7
2	5	4
3		1

5.2 Intermediary case - 3x3

5.2.1 Perfect learner

As discussed in the previous section section 5.1, the 3 by 3 SP is one of the cases I have been able to solve perfectly, since it only has 181,440 possible configurations. Its God number is only 31, which definitely makes it manageable. However, this is already an intermediary size, large enough that it is worth trying and comparing a few different methods, including deep reinforcement learning. To start with, I ran the PerfectLearner with n=m=3, and the results are shown below in figure 5.1. It is interesting to note that there are only two hardest configurations (cost 31) and 221 configurations of cost 30.

```
+----------------+-------------+-----------------------+--------------------+-----------+---------------------+
| puzzle_type    | dimension   | # possible puzzles    | # solved puzzles   |  max cost | hardest puzzle      |
|----------------+-------------+-----------------------+--------------------+-----------+---------------------|
| sliding_puzzle | (3, 3)      | 181,440               | 181,440            |        31 | +=====+=====+=====+ |
|                |             |                       |                    |           | |   8 | 6   |   7 | |
|                |             |                       |                    |           | +-----+-----+-----+ |
|                |             |                       |                    |           | |   2 | 5   |   4 | |
|                |             |                       |                    |           | +-----+-----+-----+ |
|                |             |                       |                    |           | |   3 |     |   1 | |
|                |             |                       |                    |           | +-----+-----+-----+ |
+----------------+-------------+-----------------------+--------------------+-----------+---------------------+
```

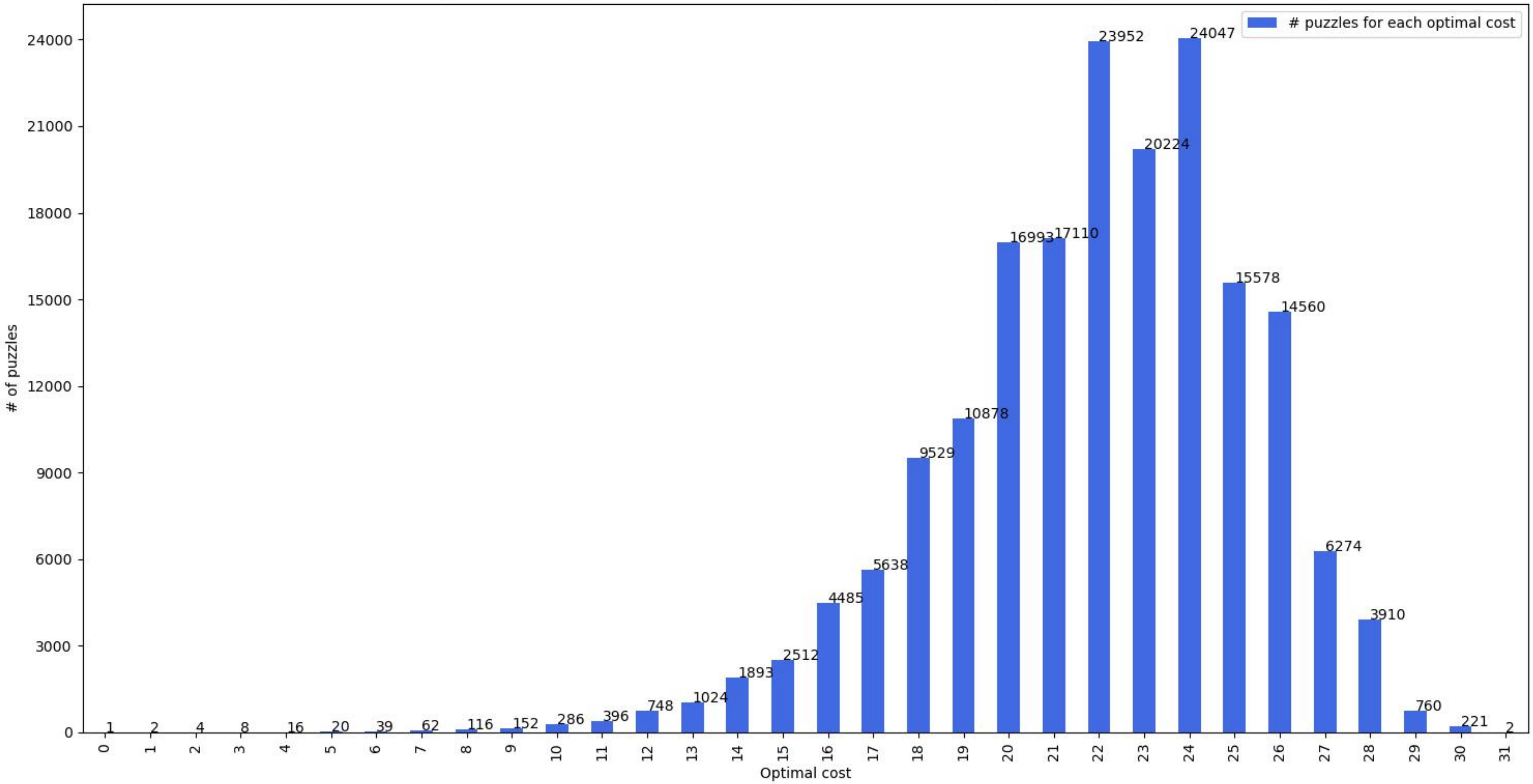

FIGURE 5.1: Perfect Learning 3x3 SP

5.2.2 Deep reinforcement learner

The DeepReinforcementLearner's learning is shown in figure 5.2:

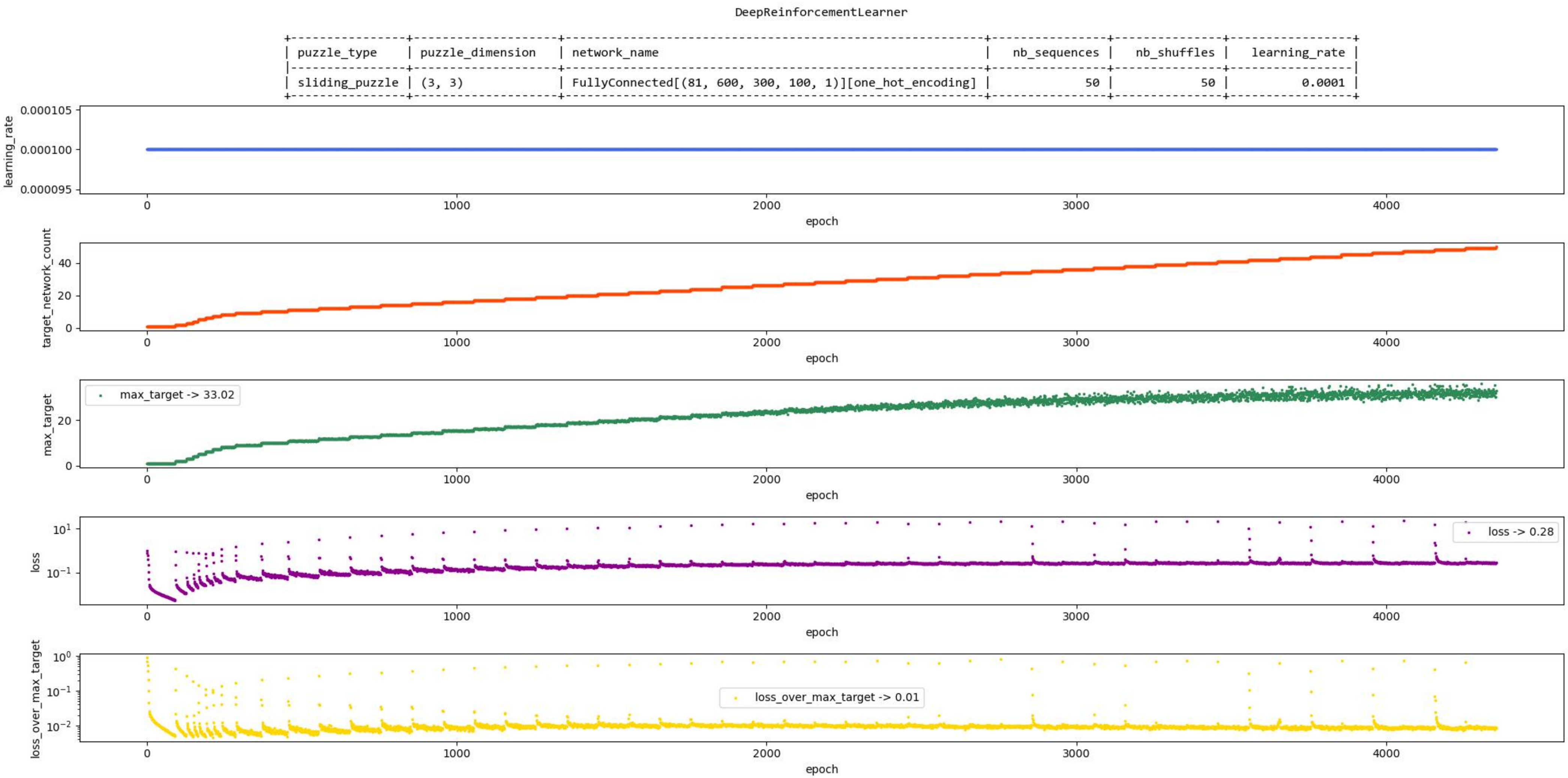

FIGURE 5.2: Deep reinforcement learner 3x3 SP

5.2.3 Solvers' comparison

Let me discuss a comparison of several algorithms on 1000 random puzzles generated for a number of random shuffling (with best-effort-no-backtracking) from 0 to 50 in step of 2, as well as for perfect shuffling (denoted by ∞) on the comparison graphs. The results are shown in figure 5.3

puzzle_type	puzzle_dimension	performance_file_name	nb_samples	time_out
sliding_puzzle	(3, 3)	C:/Users/franc/rubiksdata/perf/sliding_puzzle/3_3/perf.pkl	1000	1200

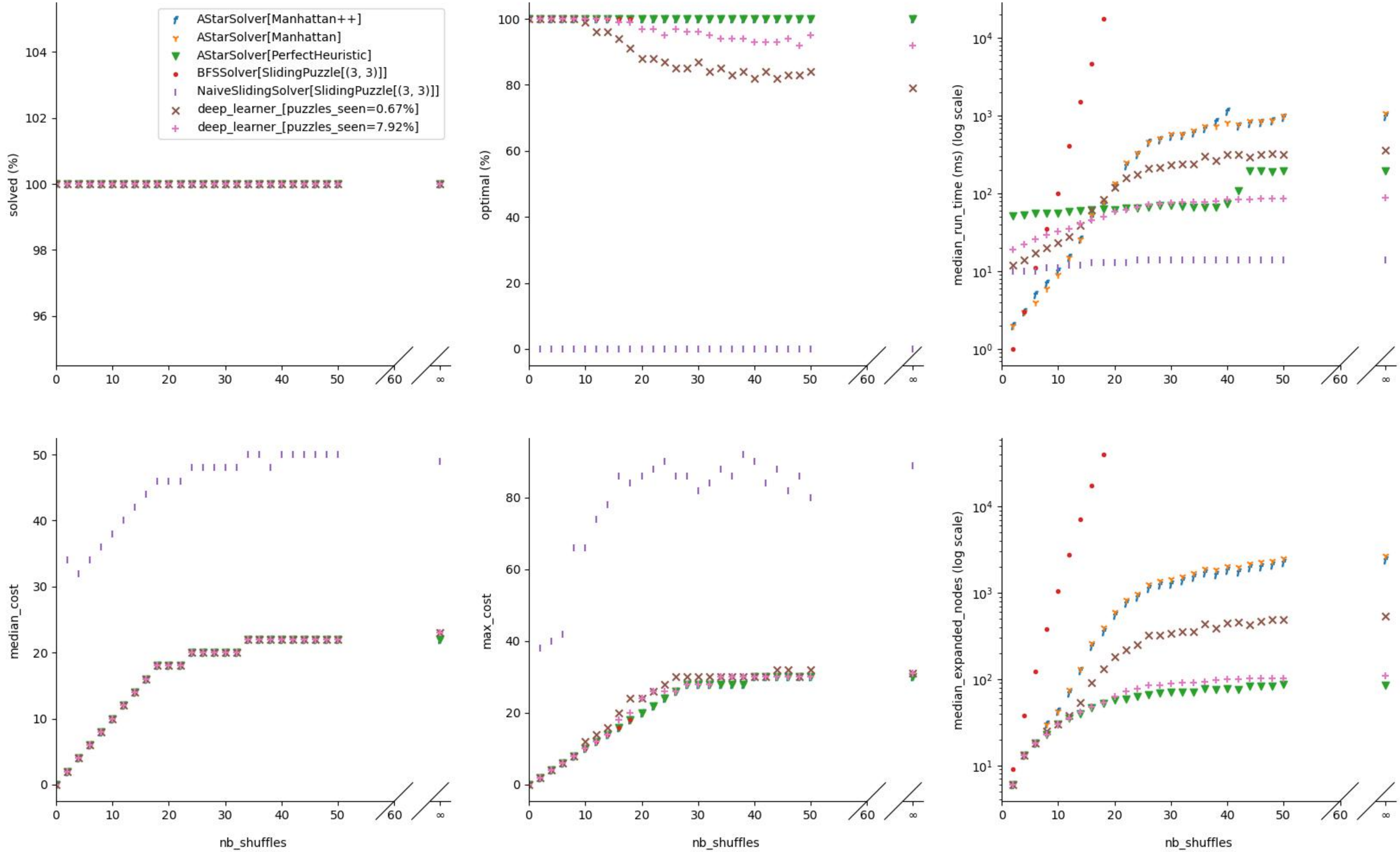

FIGURE 5.3: Solvers' performance comparison 3x3 SP

5.2.4 Solving the hardest 3x3 problem

To finish with the 3x3 SP, let me try to throw one of the two hardest 3x3 configurations (see subsection 5.1) at the different solvers to see how they fare. The results are shown here

solver	cost	# expanded nodes	run time (ms)
AStarSolver[Manhattan]	31	58,859	11,327
AStarSolver[PerfectHeuristic]	31	1,585	202
AStarSolver[DeepLearningHeuristic]	31	101	58
BFS	-	-	time out
NaiveSlidingSolver	61	n/a	18

On this specific configuration, there was obviously no chance that the BFS would complete, hence it timed out. It would have no matter what time out I set. Indeed, since it has no heuristic to guide its search, it would need to explore in the order of 3^{31} - roughly 617 trillions - nodes to reach the goal!
Rather interestingly, my DRL heuristic performs much better than the manhattan heuristic (not super suprising), but also outperforms the perfect heuristic quite significantly both in terms of run time and of nodes expansion. Obviously there is no guarantee that the perfect heuristic will not be outperformed on some random configuration, and it does on this occasion. However, as we have seen in the previous subection 5.2.3, it is not the case on average.
Finally, the naive solver outperforms every other solver in terms of run time, but finds a rather poor solution of 61 moves.

5.3 3x4

TBD

5.4 4x4

TBD

5.5 5x5

TBD

Chapter 6

Results - Rubiks' Cube

6.1 2x2x2

blablabla

6.2 3x3x3

blabla

Bibliography

Journal Papers

Archer, Aaron (1999). "A modern treatment of the 15 puzzle". In: *American Mathematical Monthly* 106, pp. 793–799. URL: https://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/15-puzzle.pdf.

Dechter, Rina and Judea Pearl (1985). "Generalized Best-First Search Strategies and the Optimality of A*". In: *J. ACM* 32.3, pp. 505–536. DOI: 10.1145/3828.3830. URL: https://doi.org/10.1145/3828.3830.

Johnson, Wm. Woolsey and William E. Story (1879). "Notes on the "15" Puzzle". In: *American Journal of Mathematics* 2.4, pp. 397–404. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2369492 (visited on 06/22/2022).

Karlemo, Filip and Patric R.J. Ostergaard (2000). "On sliding block puzzles". In: *Journal of Combinatorial Mathematics and Combinatorial Computing* 34.1, pp. 97–107. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.7558&rep=rep1&type=pdf.

McAleer, Stephen et al. (2018a). "Solving the Rubik's Cube Without Human Knowledge". In: *CoRR* abs/1805.07470. arXiv: 1805.07470. URL: http://arxiv.org/abs/1805.07470.

Mnih, Volodymyr et al. (2013). "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

Silver, David et al. (Jan. 2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529, pp. 484–489. DOI: 10.1038/nature16961.

Watkins, Christopher (Jan. 1989). "Learning From Delayed Rewards". In: URL: https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Watkins, Christopher J. C. H. and Peter Dayan (May 1992). "Q-learning". In: *Machine Learning* 8.3, pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

Books

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. The MIT Press. URL: http://incompleteideas.net/book/the-book-2nd.html.

Misc

Berrier (2022). *FB Code Base*. github. Accessed: 2022-06-22.

McAleer, Stephen et al. (2018b). *Solving the Rubik's Cube Without Human Knowledge*. DOI: 10.48550/ARXIV.1805.07470. URL: https://arxiv.org/abs/1805.07470.

Segerman (2022). *Coiled 15 Puzzle*. youtube. Accessed: 2022-06-22.

Tsoy (2019). *Python Kociemba Solver*. kociemba. Accessed: 2022-06-22.

Wikipedia (2022). *Sliding Puzzle*. wikipedia. Accessed: 2022-06-22.

WolframMathWorld (2022). *15 Puzzle*. wolfram. Accessed: 2022-06-22.