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

August 7, 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 (**DQL**) (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 before detailing a few examples in chapter 5. Finally, in chapters 6 and 7, I will present all my various results on respectively the sliding puzzle and the Rubiks' cube.

Acknowledgements

List of Abbreviations

AIPnT	Artificial Intelligence Principles and Techniques
BFS	Breadth First Search
CS	Computer Science
CV	Computer Vision
DFS	Depth First Search
DL	Deep Learning
DQL	Deep Q-Learning
DRL	Deep Reinforcment Learning
ML	Machine Learning
NLP	Natural Language Processing
RC	Rubiks' Cube
RL	Reinforcement Learning
RHUL	Royal Holloway, University of London
SP	Sliding Puzzle

Contents

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

In this section, I will succintly go through the different techniques I have used and implemented for this project as well as give some references.

2.1 Reinforcement Learning

see Sutton and Barto, 2018

2.2 Deep Learning

TBD

2.3 Graph Search & Heuristics

See Dechter and Pearl, 1985

2.4 Deep Reinforcement Learning Heuristics

TBD

2.5 Deep Q-Learning

see Watkins and Dayan, 1992 and McAleer et al., 2018b

2.6 Monte Carlo Tree Search

see McAleer et al., 2018b and Silver et al., 2016

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, 2022b).The 15-puzzle seems to have been invented in the late 19th century by Noyes Chapman (see WolframMathWorld, 2022), who applied in 1880 for a patent on what was then called "Block Solitaire Puzzle". 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
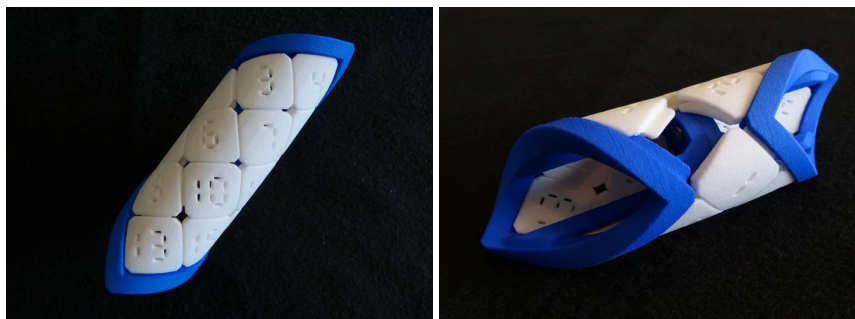
It is rather easy to see why this is the case, let us discuss why: given a configuration c of the tiles, let us define the permutation p(c) of this configuration according to the following schema: we enumerate the tiles row by row (top to bottom), left to right for odd rows and right to left for the 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 below 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, we conclude by invoking symmetry arguments that 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 are 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.

As hinted in the previous section when discussing the coiled 15-puzzle, we can note that the parity argument used there implies that in general, out of the $(n * m)!$ possible permutations of all tiles, only half are attainable and solvable. This gives us a neat way to generate *perfectly shuffled* puzzles, and we make use of this in the code. When comparing various algorithms in later sections, I will compare them on a same set of shuffled puzzles, where the shuffling is either done via a fixed number of random moves from goal position, or via what I call *perfect shuffle*, which means I give equal probability to each attainable configuration of the tiles. A simple way to achieve this is therefore to start with one randomly selected among the $(n * m)!$ permutations, and then to verify that the parity of that permutation is the same as that of the goal (for the given choice of n and m), and simply swap the first two (non-empty) tiles if it is not.

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

3.2.1 History

The second puzzle I will focus on is the well known Rubiks' Cube (**RC**), invented in 1974 by Erno Rubik, originally known under the name of *Magic Cube* (see Wikipedia, 2022a). Since it was commericalised in 1980, it has kown a worldwide success: countless people are playing with the famous original 3x3x3 cube, as well as with countless variations, more or less difficult of it. Competitions nowadays bring together *speedcubers* who have trained for years and memorized various algorithms to solve the Rubiks in astonishly effective and fast times (literally in seconds for the best ones). Interestingly, some of the principles used by speedcubers generalise to different dimensions. As an example, it is quite impressive to watch *Cubastic* solve a 15x15x15 RC in two and a half hours (Cubastic, 2022).

3.2.2 Search Space & Solvability

In my code as well as in this write-up, as is very customary in the RC literature, I will refer to the faces as F (front), B (back), U (up), D (down), L (left) and R (right). The RC's tiles are of 6 different colors, and without loss of generality, we can consider that these colors are also called F, B, U, D, L and R, and consider the goal state as one where, up to rotations of the whole cube, the color's name match the faces (color F on face F, etc). The RC has, as one would expect, an extremely large state space. If we ignore redundant cubes via rotations, there are 24 goal states, since any of the colors can be placed on the F face, and then any of the 4 adjacent colors can be placed on the U face (say). Once these 2 are chosen, the remaining faces are determined. This is not immediately obvious, but not difficult to convince oneself that this is the case with the following two observations about the structure of corner *cubies*: first observation is that they have 3 determined colors, which are fixed once and for all. In particular, the fact that there is no corner with colors F and B, means that once the RC is in solved state, we have to have colors F and B on opposite faces, and similarly for colors U and D as well as colors L and R. Hence, having fixed the color on face F, the color on face B is fixed too. The second observation is that when facing a corner cubie, the order of its three colors, enumerated clock-wise) is invariant. For instance, consider the corner cubie with colors (F, R, U). There is no scrambling of the cube that will produce clock-wise order of e.g. (F, U, R). This is obvious once you consider that there is no move of the Rubik's cube you could not perform while holding a given corner fixed in space (e.g. by pinching that corner cubie with your fingers and never letting go of it).
Finally a further word of notation. It is common in RC jargon to refer to moves by the name of the face that is rotated clock-wise (when facing it), and by adding a prime ′ for counter-clock-wise rotations. Sometimes people use a number after the face to indicate repeated similar moves, though I will count in my code that as 2 moves, since obviously e.g. U2 = U U = U′ U′. As another example: F B′ F2 means rotating the front face clock-wise, the back face counter-clock-wise, followed by the front face twice.

2 x 2 x 2

Up to rotations of the full RC, there are $\frac{8!*3^7}{24}$ = 3,674,160 possible combinations. Indeed, there are 8! possible ways of choosing the position of the eight corner cubies (since there are known algorithms, such as R′ U R′ D2 R U′ R′ D2 R2, to swap exactly two and only two adjacent corners), each of which can be oriented in 3 different ways (not 6, since as discussed earlier, the clock-wise order of a given corner cubie can never be changed). That gives us 3^8 permutations of the corners orientation. However,

the *corner orientation parity* (**COP**) (see Martin Schoenert, 2022 for details) is invariant (% 3) by any legal move of the RC. Since rotating by 120° a given corner cubie increases the COP by 1, we conclude that only one third of all these 3^8 permutations are attainable (namely those for which the COP is equal to 0 % 3). Finally, the denominator of 24 is due to the equivalence by whole cube rotation as discussed in the previous section.

The insight and knowledge of this section will be useful for me to implement *perfect scrambling* for the 2x2x2 RC. Indeed, the way I generate perfectly shuffled RCs is by randomly placing the 8 corners, then randomly choosing the orientation of the first 7, and finally fixing the 8th corner's orientationto make COP equal to 0 % 3.

3 x 3 x 3

3.2.3   Optimal Cost & God's Number

Alexander Chuang, 2022
Silviu Radu, 2007

Chapter 4

Code

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 except the parameters passed in the command line. Here is a visual overview of the code base with the main dependencies between the main submodules and classes. Solid arrows indicate inheritance (e.g. AStar inherits from SearchStrategy), while dotted lines indicate usage (e.g. AStar uses Heuristic, DeepReinforcementLearner uses DeepLearning, etc..).
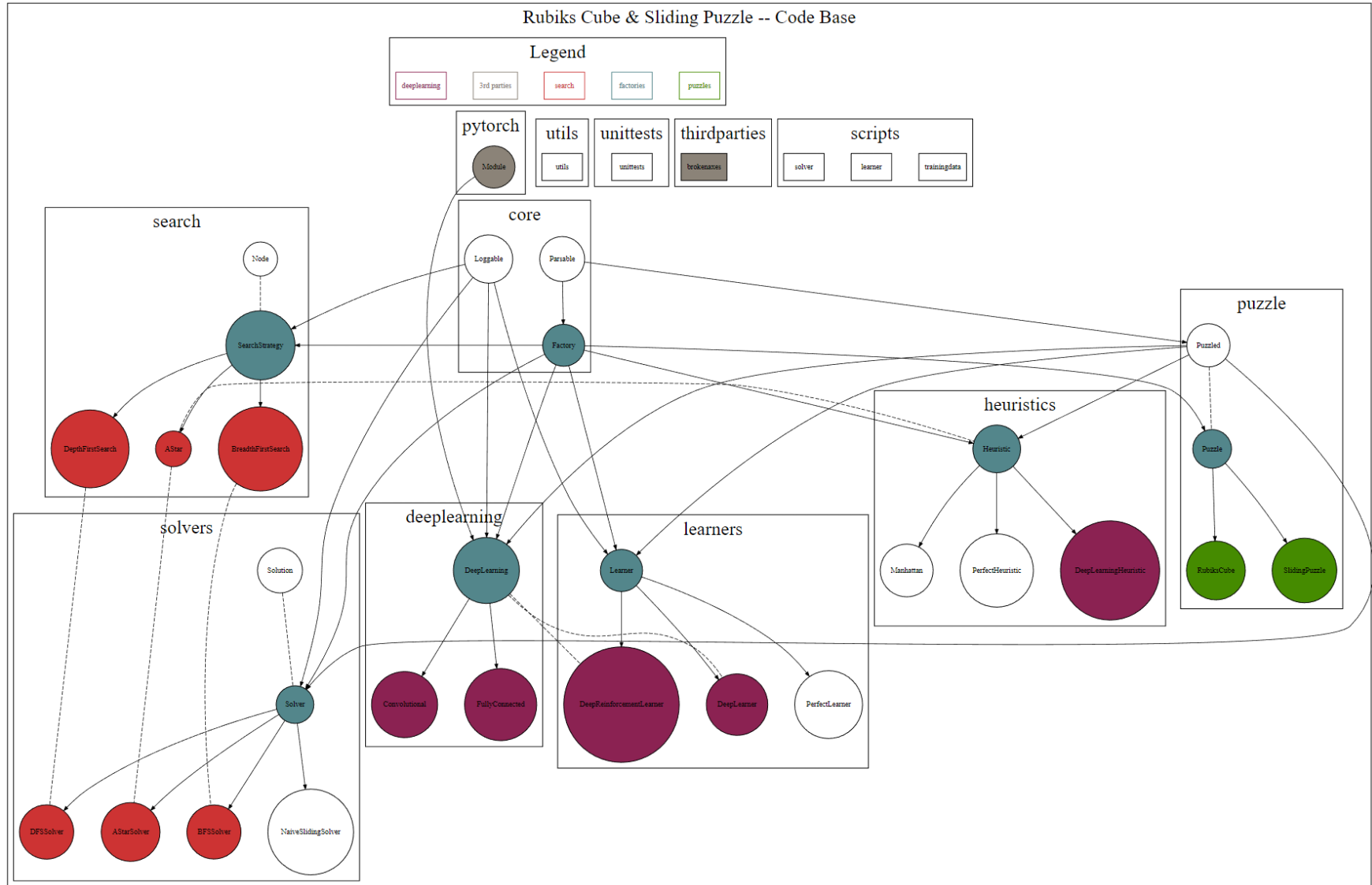
FIGURE 4.1: Code base

Let me now describe what each submodule does in more details:

4.1 rubiks.core
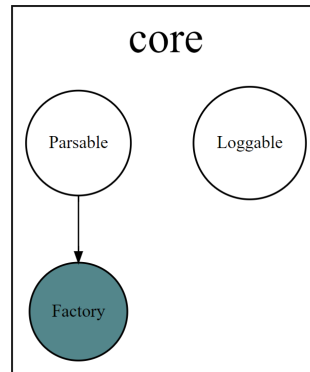
FIGURE 4.2: rubiks.core

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)

4.2 rubiks.puzzle
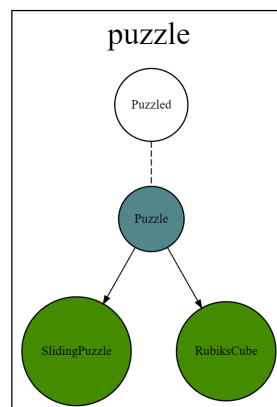
FIGURE 4.3: rubiks.puzzle

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, this module contains a **Puzzled** base class which most classes 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.

4.3   rubiks.search
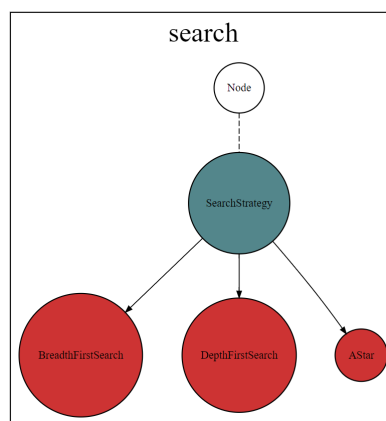
FIGURE 4.4: rubiks.search

This modules contains graph search strategies. I have actually reused the code I implemented for one of the AIPnT assignments here. 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.

4.4 rubiks.heuristics
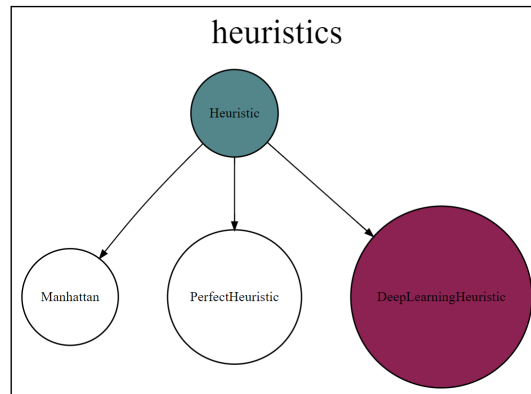
FIGURE 4.5: rubiks.heuristics

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**: This heuristic is specific to the SP. It simply adds for each tile (other than the empty tile) the L_1 distance between their current position and their target position. We can quickly see that this heuristic is admissible. Indeeed, each move in the SP moves one (and one only) tile by one position. In other puzzles, it is generally the case that a move will affect the position of many tiles (e.g. RC). Therefore, if tiles could somehow freely move on top of one-another (that is, we remove the constraint that there can be at most one tile per compartment, the number of moves necessary to solve the SP would exactly be the Manhattan distance.

  The reason why it is interesting to have a known admissible heuristic is that we can obvioulsy compare other heuristics (DL, DRL, DQL, etc) in terms of optimality.

  I have also implemented an improvement to the Manhattan distance, which I shall call Manhattan++ in here (and in code logs and graphs) and can be activated by simply passing *plus=True* to the Manhattan Heuristic (see example 5.3.4 as well as a thorougher performance comparison on the (n=2, m=5) SP in 6.1.2). It is based on the concept of linear constraints that I read about in lecture notes from Washington University (Richard Korf and Larry Taylor, 2022). The idea is simply that when tiles are already placed on their target row or column, but not in the expected order, we will need to get the tiles out of the way of one another to get to the goal state, and this is not accounted for by the Manhattan distance. For instance, in the following (n=2, m=3) SP, the Manhattan distance does not account for the fact that, at best, tile 3 needs to get out of the way for tiles 1 and 2 to move, and then needs to get back in its row, adding a cost of 2 to the Manhattan distance.

3	1	2
4	5	0

Two important things to notice are that linear constraints across rows and columns (which I might generically refer to as *lines* in the following) can to be added without breaking admissibility (hence

giving more pessimistic, or accurate, cost estimates than simple Manhattan). This is because if a tile is involved in two linear constraints, it will need to get out of the way both horizontally and vertically. The second thing is that when several pairs in a line are not in order, we cannot simply add 2 for each distinct out-of-order pair, the right penalty to add is more subtle than that and needs to be computed recursively. For instance, let us now consider the following configuration:

3	2	1
4	5	0

The correct penalty to add is not 6 (3 times 2 since all pairs (1, 2), (1, 3) and (2, 3) are out of order, but only 4. Indeed if, say, tile 3 got somehow out of the way at the back of the SP (imagine just another dimension there where we can move tiles) and tile 2 got out of the way by moving down to let tile 1 pass across, we could be done by simply adding 4 to the Manhattan distance (2 to move tile 3 out and back, and 2 to move tile 2 out and back). The correct way to compute the penalty cost for linear constraints is therefore to do it recursively, taking the minimum additional cost of moving either left-most or right-most tile of the line under consideration out of the way (that additional cost to move these left-most or right-most tile is 2 if not at their expected order in the line, 0 otherwise) plus the penalty of reordering the rest of the line.

Finally, as suggested by the reference lecture notes (which give very vague details about the above subtleties), I have precomputed all the penalties for all possible rows, columns and all possible tiles ordering they could have and saved the corresponding penalties in a database. For memory efficiency, I also only saved penalties which are non-zero. The very first time any call to Manhattan++ is made, for a given dimension (n, m), the appropriate data base is computed and populated.

Notice that for an (n, m) SP, there are n rows, each of which can have $\frac{(n*m)!}{(n*m-m)!}$ different ordering of tiles and m columns which can have n columns, each can have $\frac{(n*m)!}{(n*m-n)!}$ different ordering of tiles. This means the pre-computations and data-base sizes for the Manhattan++ heuristic are actually manageable, as it grows much slower than the number of possible puzzles. The maximum number of penalties to compute for $(n, m) \leq (5, 5)$ are:

n	m	2	3	4	5
2		48	330	3,584	60,930
3			3,024	40,920	1,094,730
4				349,440	8,023,320
5					63,756,000

Taking also into account that we only store non-zero penalties, we actually get the following (quite smaller) number of penalties in our data bases:

n	m	2	3	4	5
2		2	46	1,238	32,888
3			278	7,122	328,894
4				40,546	1,456,680
5					8,215,382

- **PerfectHeuristic**: this reads from a data base the optimal costs, pre-computed by the PerfectLearner (see below 4.6)

- **DeepLearningHeuristic**: this uses a network which has been trained using DRL by the DeepReinforcementLearner (see below 4.6)

4.5   rubiks.deeplearning
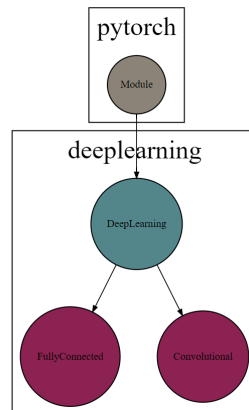
FIGURE 4.6: rubiks.deeplearning

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 hidden layers and size. There are some params as well to add drop out, and to indicate whether or not the inputs are one hot encoding (in which case the first layer is automatically adjusted in size, using information from the puzzle dimension).

- **Convolutional**: similar wrapper to FullyConnected, but with the ability to add some parallel convolutional layers to complement fully connected layers.

4.6   rubiks.learners
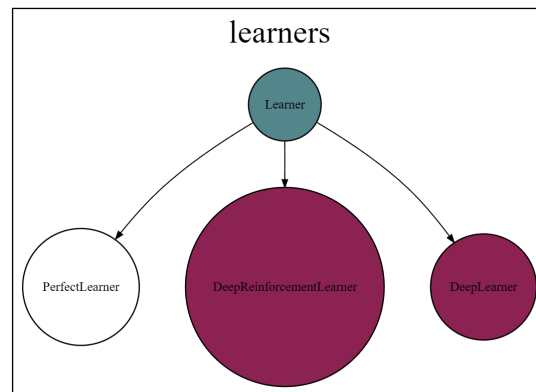
FIGURE 4.7: rubiks.learners

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 5.2.1 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 4.4. The DeepReinforcementLearner runs for a number of epochs (or less if, based on other parameters discussed below, it is deemed to have converged).

4.7 rubiks.solvers
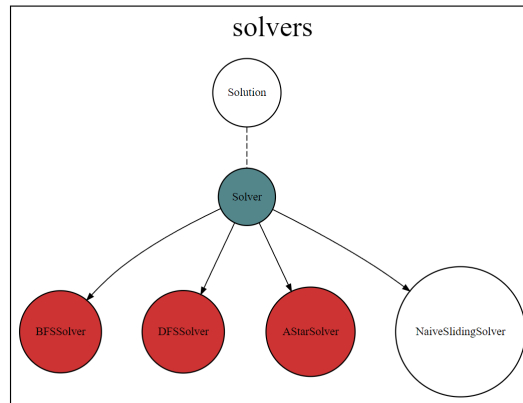
FIGURE 4.8: rubiks.solvers

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** TBD

- **BFSSolver** TBD

- **AStarSolver** TBD

- **NaiveSlidingSolver** TBD

- **MonteCarloSearchTreeSolver** TBI I want to implement this in August when working on the Rubiks' ... following the Rubik's paper fromMcAleer & Agostilenni et al McAleer et al., 2018b

4.8 rubiks.scripts
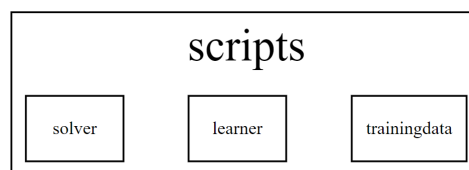
FIGURE 4.9: rubiks.scripts

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 :)

Chapter 5

Examples

In this chapter, I will go through some examples, illustrating how to use the code base to learn and solve various puzzles. For each section, simple examples of code will be indicated in **python code** paragraphs, and can easily be run from command line or copied into a script and run from your favourite IDE.

5.1 Puzzles

Let me start by showing how to construct puzzles, using the Puzzle factory. Notice that in order to run a learner or solver of any kind (assuming of course that they are meant to work on the puzzle type in question), we can just use the exact same code, but just specify *puzzle_type* and the expected parameters to construct a puzzle.

For instance, let us create a (n=5, m=6) SP, shuffle it a number of times, and print it:

python code – sliding puzzle construction

```python
################################################################
from rubiks.puzzle.puzzle import Puzzle
################################################################
puzzle_type=Puzzle.sliding_puzzle
n=5
m=6
nb_moves=1000
print(Puzzle.factory(**globals()).apply_random_moves(nb_moves))
################################################################
```

The output from the above code snippet will look like (subject to randomness):

```
+=====+=====+=====+=====+=====+=====+
|   6 | 4   |  24 |  22 |   5 |  17 |
+-----+-----+-----+-----+-----+-----+
|  21 | 28  |  13 |  25 |   9 |  10 |
+-----+-----+-----+-----+-----+-----+
|  11 | 23  |  16 |  15 |   8 |   2 |
+-----+-----+-----+-----+-----+-----+
|  20 | 27  |   7 |  14 |  19 |  18 |
+-----+-----+-----+-----+-----+-----+
|   3 | ██  |  26 |   1 |  12 |  29 |
+-----+-----+-----+-----+-----+-----+
```

FIGURE 5.1: sliding puzzle construction example

.. similarly to construct a (n=2) RC and shuffle it perfectly:

python code – rubiks cube construction

```
################################################################
from math import inf
from rubiks.puzzle.puzzle import Puzzle
################################################################
puzzle_type=Puzzle.rubiks_cube
n=2
""" Here we use perfect shuffle by spcifying infinite number of shuffles """
nb_moves=inf
print(Puzzle.factory(**globals()).apply_random_moves(nb_moves))
################################################################
```
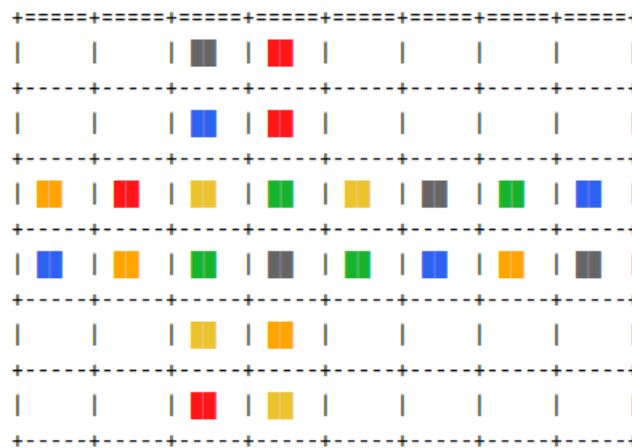

FIGURE 5.2: rubiks cube construction example

5.2   Learners

5.2.1   Perfect Learner

The perfect learner has been discussed in details in 4.6. We simply show here how to run it to learn the value function for the (n=2, m=3) SP, that is, to solve all 360 possible configurations via an optimal solver (A* with Manhattan heuristic).

python code – perfect learner

```
################################################################
from rubiks.heuristics.heuristic import Heuristic
from rubiks.puzzle.puzzle import Puzzle
from rubiks.learners.learner import Learner
from rubiks.utils.utils import get_model_file_name
################################################################
action_type=Learner.do_learn
n=2
m=3
puzzle_type=Puzzle.sliding_puzzle
learner_type=Learner.perfect_learner
heuristic_type=Heuristic.manhattan
nb_cpus=4
```

```python
learning_file_name=get_model_file_name(puzzle_type=puzzle_type,
                                        dimension=(n, m),
                                        model_name=Learner.perfect)
if __name__ == '__main__':
    # we fully solve the 2 x 3 SP ... should take ~5s
    Learner.factory(**globals()).action()
################################################################
action_type=Learner.do_plot
if __name__ == '__main__':
    # we display the results
    Learner.factory(**globals()).action()
################################################################
```

The above snippet of code will solve the 2 by 3 SP and then display the results, showing the distribution of optimal costs, as well as the most difficult puzzle.

puzzle_type	dimension	# possible puzzles	# solved puzzles	max cost	hardest puzzle
sliding_puzzle	(2, 3)	360	360	21	4 5 / 1 2 3
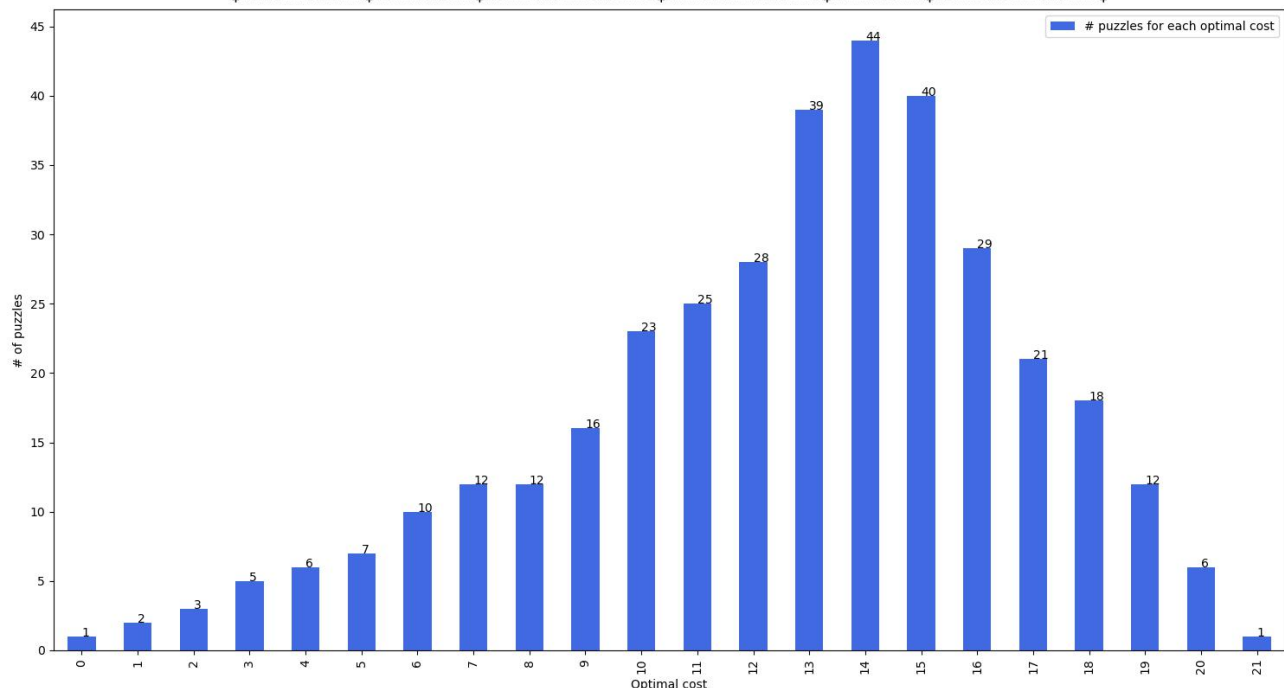
FIGURE 5.3: perfect learner example

5.2.2 Deep Learner

The DeepLearner, also discussed in details in 4.6, needs some training data. It could in principle of course be trained on any data, not necessarily optimal data (i.e. generated by an optimal solver). Here however, I make use of the TrainingData class to generate 100 sequences of fully solved (n=5, m=2) SP via A* with Manhattan++.

python code – deep learner

```
################################################################
```

```python
from math import inf
######################################################################
from rubiks.deeplearning.deeplearning import DeepLearning
from rubiks.learners.learner import Learner
from rubiks.learners.deeplearner import DeepLearner
from rubiks.puzzle.puzzle import Puzzle
from rubiks.puzzle.trainingdata import TrainingData
######################################################################
if '__main__' == __name__:
    puzzle_type=Puzzle.sliding_puzzle
    n=5
    m=2
    """ Generate training data - 100 sequences of fully
    solved perfectly shuffled puzzles.
    """
    nb_cpus=4
    time_out=600
    nb_shuffles=inf
    nb_sequences=100
    TrainingData(**globals()).generate(**globals())
    """ DL learner """
    action_type=Learner.do_learn
    learner_type=Learner.deep_learner
    nb_epochs=999
    learning_rate=1e-3
    optimiser=DeepLearner.rms_prop
    scheduler=DeepLearner.exponential_scheduler
    gamma_scheduler=0.9999
    save_at_each_epoch=False
    threshold=0.01
    training_data_freq=100
    high_target=nb_shuffles + 1
    training_data_from_data_base=True
    nb_shuffles_min=20
    nb_shuffles_max=50
    nb_sequences=50
    """ ... and its network config """
    network_type=DeepLearning.fully_connected_net
    layers_description=(100, 50, 10)
    one_hot_encoding=True
    """ Kick-off the Deep Learner """
    learning_file_name=Learner.factory(**globals()).get_model_name()
    Learner.factory(**globals()).action()
    """ Plot its learning """
    action_type=Learner.do_plot
    Learner.factory(**globals()).action()
######################################################################
```

As can be seen in the code snippet, this example will generate 100 perfectly shuffled (n=5, m=2) SPs and solve them. Once done, a summary of the training data is printed, indicating, for each optimal cost, how many sequences have been generated.

16	20	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
1	1	3	3	4	5	2	2	5	2	4	6	4	8	11	8	7	7	5	3	3	2	2	1	1

FIGURE 5.4: deep learner training example

Then the Deep Learner will get trained on this data for 999 epochs. In the above example, I have chosen to fetch, every 100 epochs, 50 random sequences of puzzles from the training data. Each sequence is composed of a random puzzle of cost between 20 and 50, fetched from the training data, along with its optimal path to the solution. The default optimiser (rms_prop) is used, together with an exponential scheduler starting with a learning rate of 0.001 and a gamma of 0.9999. We can see that the (MSE) loss on the value function decreases rapidly, and jumps back up every time we change the training data (since it has not yet seen some of it). By the end of the training, the in-sample MSE loss has dropped to 1.5 and the Deep Learner has seen 0.14% of the possible (n=5, m=2) puzzles.
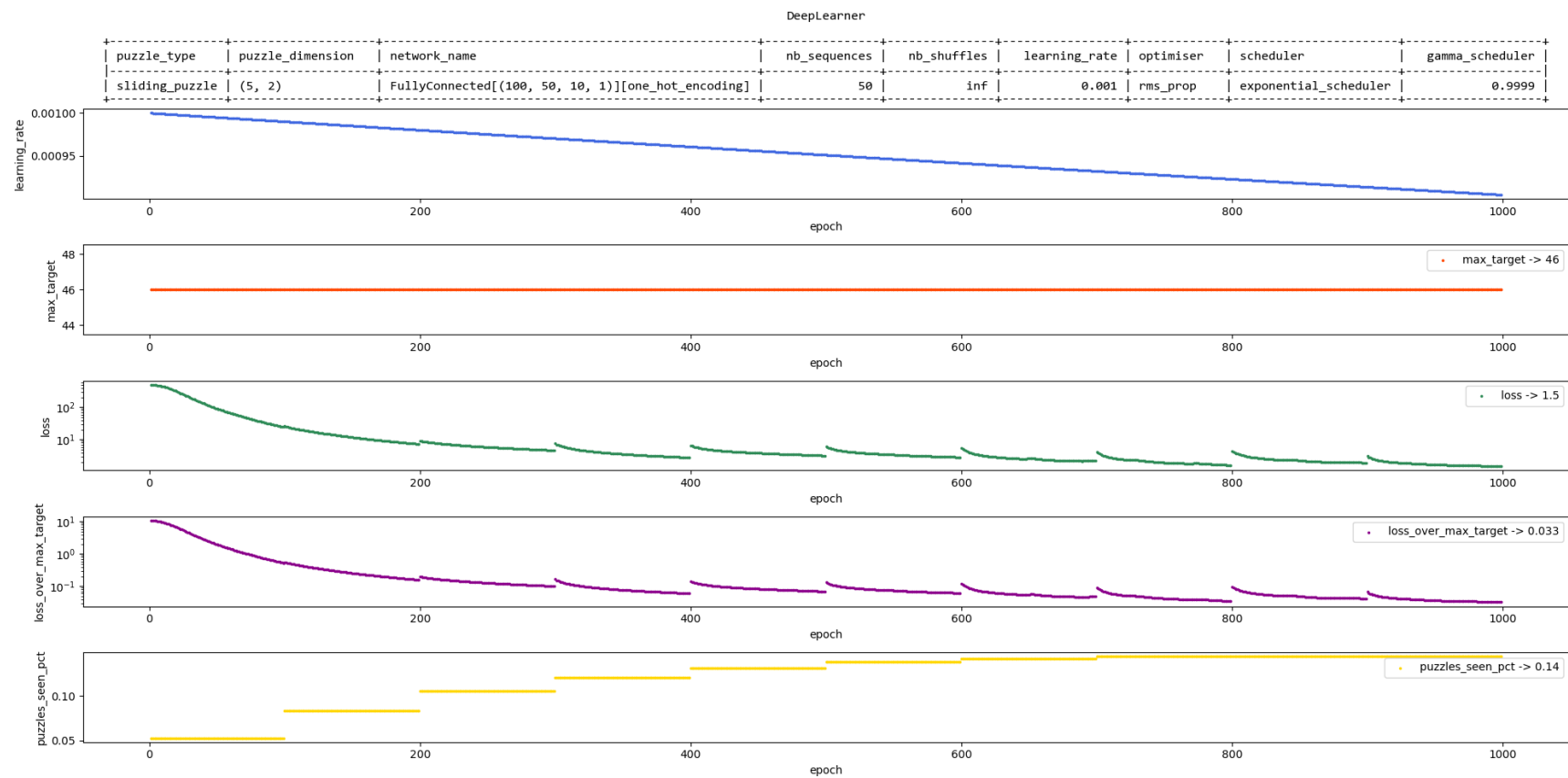
FIGURE 5.5: deep learner learning data example

5.2.3 Deep Reinforcement Learner

Unlike the Deep Learner, the Deep Reinforcement Learner learns unsupervised (in the sense that there is no need to pre-solve puzzles to tell if what the actual (target) costs are), since it generates its own target using a combination of a target network and the simple min rule described in 4.6. Let us see here how to run it on the (n=4, m=2) SP for instance. The following code snippet will run a Deep Reinforcement Learner for a maximum of 25,000 epochs, generating randomly 10 sequences of puzzles shuffled 50 times from goal state every time it updates the target network. That update will happen either after 500 epochs or when the (MSE) loss on the value function gets under one thousands of the max target value. The learner will stop if it reaches 25,000 epochs or if the target network updates 40 times. The network trained is a fully connected network with 3 hidden layers and the puzzles are one-hot encoded. We use the same optimiser and scheduler as in the previous subsection.

python code – deep reinforcement learner

```python
########################################################################
from rubiks.deeplearning.deeplearning import DeepLearning
from rubiks.learners.learner import Learner
from rubiks.learners.deepreinforcementlearner import DeepReinforcementLearner
from rubiks.puzzle.puzzle import Puzzle
########################################################################
if '__main__' == __name__:
    puzzle_type=Puzzle.sliding_puzzle
    n=4
    m=2
    """ Generate training data - 100 sequences of fully
    solved perfectly shuffled puzzles.
    """
    nb_cpus=4
    """ DRL learner """
    action_type=Learner.do_learn
    learner_type=Learner.deep_reinforcement_learner
    nb_epochs=25000
    nb_shuffles=50
    nb_sequences=10
    training_data_every_epoch=False
    cap_target_at_network_count=True
    update_target_network_frequency=500
    update_target_network_threshold=1e-3
    max_nb_target_network_update=40
    max_target_not_increasing_epochs_pct=0.5
    max_target_uptick=0.01
    learning_rate=1e-3
    scheduler=DeepReinforcementLearner.exponential_scheduler
    gamma_scheduler=0.9999
    """ ... and its network config """
    network_type=DeepLearning.fully_connected_net
    layers_description=(128, 64, 32)
    one_hot_encoding=True
    """ Kick-off the Deep Reinforcement Learner ... """
    learning_file_name=Learner.factory(**globals()).get_model_name()
    Learner.factory(**globals()).action()
    """ ... and plot its learning data """
    action_type=Learner.do_plot
    Learner.factory(**globals()).action()
########################################################################
```

As can be seen on the next page, which I obtained from running the above code snippet (keep in mind that every run is going to be slightly different due to the random puzzles being generated), the training stopped after about 17,100 epochs as the 40 target network updates had been reached. By that point, the DRL learner had seen 40% of the possible puzzles, and the very last MSE loss (after update, so out-of-sample) was around 0.4, corresponding to 2% of the max target cost. It is interesting to notice that since I have shuffled the sequences only 50 times, and since the (n=4, m=2) SP is quite constrained in terms of possible moves, the max target ever produced by the network was only around 25, whereas we know the God number for this dimension is 36 (see later section 6.1). It is therefore likely that the resulting network would not produce very optimal solutions for puzzles whose cost is in the region [25, 36].
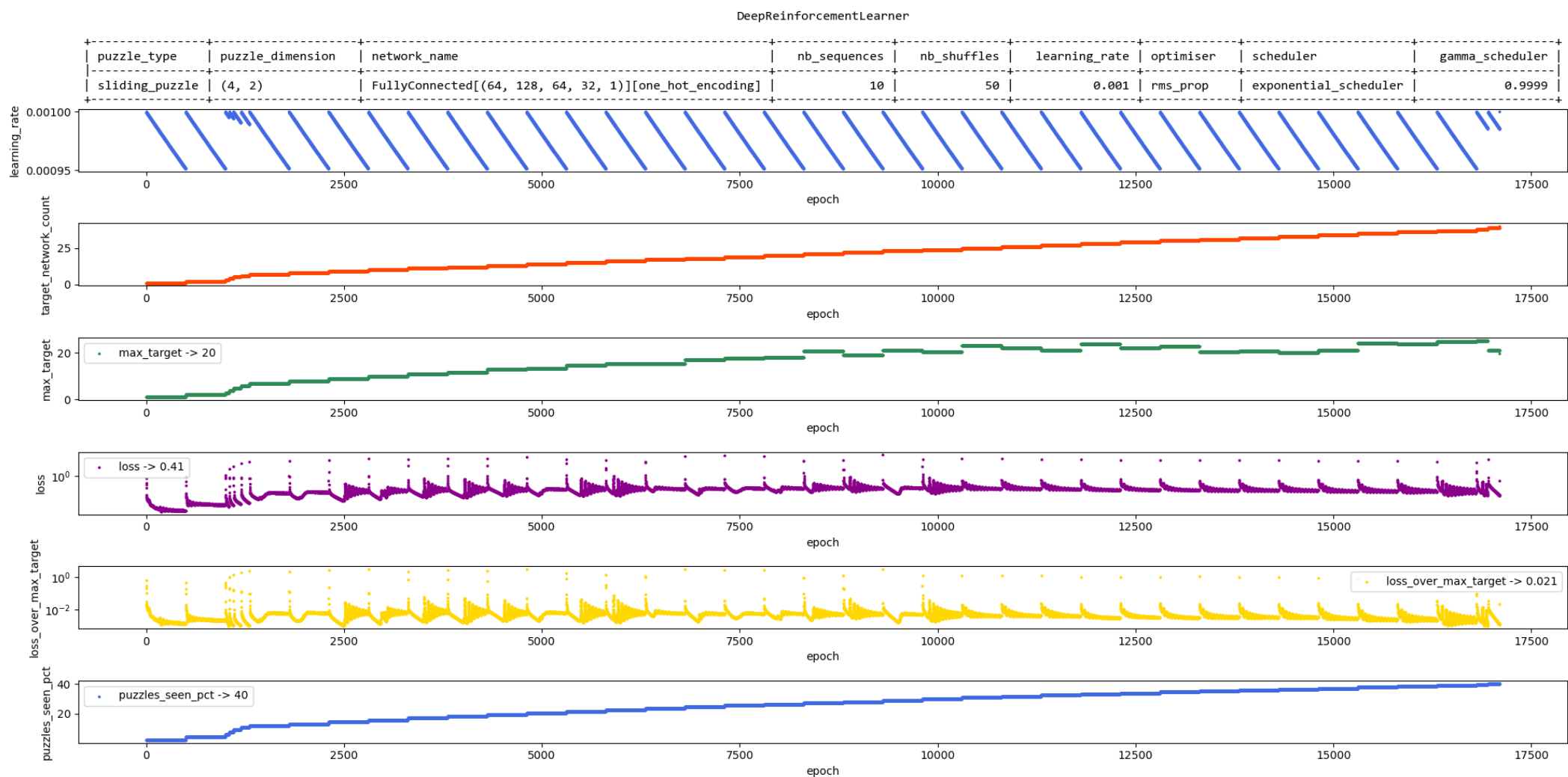
FIGURE 5.6: deep reinforcement learner learning data example

5.3   Solvers

5.3.1   Blind search

First let me start with some of the blind search algorithms, which I only have been able to use on small dimension SP. They quickly become too memory hungry to be practical on anything but the smallest puzzles.

BFS

The following example shows how to use Breadth First Search to solve a (n=3, m=3) SP which we do not *shuffle* too much.

python code – breadth first search solver

```python
###################################################################
from rubiks.solvers.solver import Solver
from rubiks.puzzle.puzzle import Puzzle
###################################################################
if '__main__' == __name__:
    puzzle_type = Puzzle.sliding_puzzle
    n=3
    m=3
    nb_shuffles=10
    solver_type=Solver.bfs
    check_optimal=True
    action_type=Solver.do_solve
    print(Solver.factory(**globals()).action())
###################################################################
```

As expected (since BFS is optimal), the solution found, printed by the snippet of code above, is optimal. The check_optimal flag in the code snippet indicates that the solver should let us know if solution is optimal. Since BFS advertises itself (via the Solver base class API) as an optimal solver, the solution is deemed optimal.

FIGURE 5.7: breadth first search solver example

DFS

python code – depth first search solver

```python
###################################################################
from rubiks.solvers.solver import Solver
from rubiks.puzzle.puzzle import Puzzle
###################################################################
if '__main__' == __name__:
    puzzle_type = Puzzle.sliding_puzzle
    n=3
    m=3
```

```
    nb_shuffles =8
    limit =15
    time_out =60
    solver_type =Solver.dfs
    check_optimal =True
    log_solution =True
    action_type =Solver.do_solve
    Solver.factory(** globals()).action()
##################################################################
```

Here we can see, in contrast with the previous example, that the solution obtained by DFS is not optimal.
The Solver indicates it and shows an optimal solution. Clearly since we shuffled 8 times from goal
configuration, the optimal path cannot have a cost higher than 8 (it could be lower though of course).

FIGURE 5.8: depth first search solver example

5.3.2 Naive Sliding Puzzle Solver

As a comparison point for otherSP solvers, 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, therefore the only
  possible sequence of move in a (n=2, m=2) puzzle is to either keep moving clock-wise or counter-
  clock-wise.

- 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:
naive algorithm - top-row solver

1. we sort the tiles (which since we are potentially dealing with a sub-problem, are not 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-left corner

3. we place $t_1, ..., t_{m-2}$ to their respective positions (in that order, and making sure not to undo any
   previous steps 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 end up with t_3 under the empty tile. We have handcrafted a sequence of moves to solve this particular position. Other than this one particular case, the above naive algorithm is guaranteed to succeed (and is obviously quite fast in terms of run time, though not elegant).

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 (n=5,m=5) sub-puzzle in blue:

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

Let us now detail how the **naive algorithm** will solve the top row if that 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 $t_1, ..., t_5 = 8, 9, 10, 11, 12$ 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):

29	27	32	18	
23	21	33	35	24
15	17	22	14	16
30	20	26	34	28

which the naive solver can keep solving iteratively by taking care of the left-most column, etc...

Below is a simple code snippet to run the naive solver on a randomly generated (n=2, m=2) SP:

python code – naive solver

```python
##############################################
from math import inf
from rubiks.puzzle.puzzle import Puzzle
from rubiks.solvers.solver import Solver
##############################################
action_type=Solver.do_solve
n=2
puzzle_type=Puzzle.sliding_puzzle
solver_type=Solver.naive
nb_shuffles=inf
##############################################
print(Solver.factory(**globals()).action())
##############################################
```


FIGURE 5.9: naive solver example

5.3.3   Kociemba

TBI & TBD

5.3.4   A*

Manhattan heuristic

python code – depth first search solver

```python
######################################################################
from rubiks.heuristics.heuristic import Heuristic
from rubiks.puzzle.puzzle import Puzzle
from rubiks.solvers.solver import Solver
######################################################################
if '__main__' == __name__:
    puzzle_type = Puzzle.sliding_puzzle
    tiles=[[3, 8, 6], [4, 1, 5], [0, 7, 2]]
    solver_type=Solver.astar
    heuristic_type=Heuristic.manhattan
    plus=False
    action_type=Solver.do_solve
    print(Solver.factory(**globals()).action().to_str_light())
######################################################################
```

We run this example twice, one with simple Manhattan and one with Manhattan++. As can be seen from the output below, the Manhattan++ improves on the number of expanded nodes (as expected, since the heuristic is less optimistic while retaining its optimality property). See a more detailed analysis in the SP results section **??**

```
+----------------------+--------+----------------+----------+----------------------+----------+
| puzzle               |  cost  | # expanded nodes | success | solver_name          | run_time |
|----------------------+--------+----------------+----------+----------------------+----------|
| +=====+=====+=====+ |   18   |            472 | Y        | AStarSolver[Manhattan] | 95 ms  |
| | 3   |  8  |  6  | |        |                |          |                      |          |
| +-----+-----+-----+ |        |                |          |                      |          |
| | 4   |  1  |  5  | |        |                |          |                      |          |
| +-----+-----+-----+ |        |                |          |                      |          |
| |     |  7  |  2  | |        |                |          |                      |          |
| +-----+-----+-----+ |        |                |          |                      |          |
+----------------------+--------+----------------+----------+----------------------+----------+
```

FIGURE 5.10: a* manhattan solver example

```
+----------------------+--------+----------------+----------+------------------------+----------+
| puzzle               |  cost  | # expanded nodes | success | solver_name            | run_time |
|----------------------+--------+----------------+----------+------------------------+----------|
| +=====+=====+=====+ |   18   |            340 | Y        | AStarSolver[Manhattan++] | 76 ms  |
| | 3   |  8  |  6  | |        |                |          |                        |          |
| +-----+-----+-----+ |        |                |          |                        |          |
| | 4   |  1  |  5  | |        |                |          |                        |          |
| +-----+-----+-----+ |        |                |          |                        |          |
| |     |  7  |  2  | |        |                |          |                        |          |
| +-----+-----+-----+ |        |                |          |                        |          |
+----------------------+--------+----------------+----------+------------------------+----------+
```

FIGURE 5.11: a* manhattan++ solver example

Perfect Heuristic

To run A* with a perfect heuristic, we just need to specify the heuristic type as such, and set the parameter *model_file_name* to point to a pre-recorded database populated by the PerfectLearner (see earlier section 5.2.1).

python code – depth first search solver

```python
################################################################
from rubiks.heuristics.heuristic import Heuristic
from rubiks.puzzle.puzzle import Puzzle
from rubiks.solvers.solver import Solver
from rubiks.utils.utils import get_model_file_name
################################################################
if '__main__' == __name__:
    puzzle_type = Puzzle.sliding_puzzle
    n=3
    nb_shuffles=40
    solver_type=Solver.astar
    heuristic_type=Heuristic.perfect
    model_file_name = get_model_file_name(puzzle_type=puzzle_type,
                                          dimension=(n, n),
                                          model_name=Heuristic.perfect)
    action_type=Solver.do_solve
    print(Solver.factory(**globals()).action().to_str_light())
################################################################
```

Running this code snippet will output something like (the shuffle is random obviously):

```
+---------------------+--------+-------------------+----------+------------------------------+-----------+
| puzzle              |  cost  |  # expanded nodes |  success |  solver_name                 |  run_time |
|---------------------+--------+-------------------+----------+------------------------------+-----------|
| +=====+=====+=====+ |    24  |                84 | Y        | AStarSolver[PerfectHeuristic] | 51 ms    |
| | 2   |   8 |   7 | |        |                   |          |                              |           |
| +-----+-----+-----+ |        |                   |          |                              |           |
| | 6   |   4 |   5 | |        |                   |          |                              |           |
| +-----+-----+-----+ |        |                   |          |                              |           |
| |     |   1 |   3 | |        |                   |          |                              |           |
| +-----+-----+-----+ |        |                   |          |                              |           |
+---------------------+--------+-------------------+----------+------------------------------+-----------+
```

FIGURE 5.12: a* perfect solver example

Deep Learning Heuristic

Deep Reinforcement Learning Heuristic

Chapter 6

Results - Sliding Puzzle

6.1 Low dimension

6.1.1 God numbers and hardest puzzles

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 5.2.1 I obtained the following God numbers for these puzzles:

n m	2	3	4	5
2		6	21	36	55*
3			31	

* provisional result

Notice that among the above dimensions, (n=2, m=5) is the largest and hardest one. I had to run its PerfectLearner over a couple of days, on a c5.18xlarge instance (72 cores) on Amazon EC2.

The perfectLearner also keeps track of (one of) the hardest puzzles it has encountered (i.e. requiring a number of steps equal to their respective God number to solves):

Most difficult 2x2 (6 moves):

	3
2	1

Most difficult 2x3 (21 moves):

4	5	
1	2	3

Most difficult 2x4 (36 moves):

	7	2	1
4	3	6	5

Most difficult 2x5 (55* moves):

	9	3	7	1
5	4	8	2	6

* provisional result

Most difficult 3x3 (31 moves):

8	6	7
2	5	4
3		1

6.1.2   Manhattan heuristic

In this section, I verify empirically that, as expected, the overhead of adding penalty in Manhattan++ for the linear constraint (which have all been precomputed and stored in a database) is more than compensated for by the reduction in nodes expansion. I have run my solver script for (n=2, m=5) in performance test mode, for both Manhattan and Manhattan++, with 250 randomly shuffled puzzles with *nb_shuffles* from 0 to 60 by increment of 5, as well as with *nb_shuffles* = inf. The resulting run time and nodes expansions are as follows:
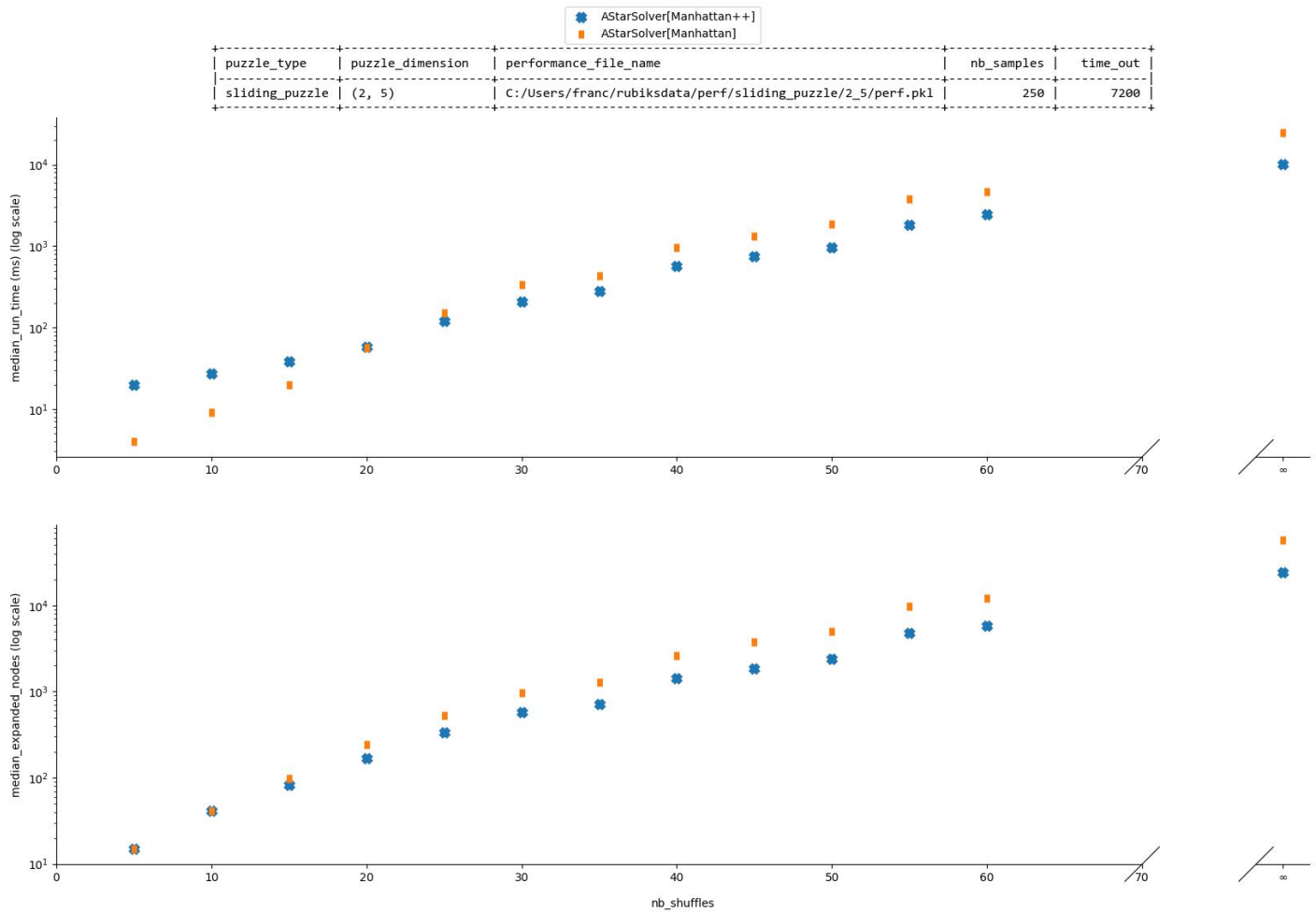
FIGURE 6.1: Manhattan vs Manhattan++

As can be seen, for low difficulty (up to *nb_shuffles = 20*), the node expansions are about the same in both cases, and the overhead of adding the linear constraints penalty increases the run time. However, for any non trivial case, Manhattan++ outperforms considerably (by a factor of about 2.5). For the 250 *perfectly* shuffled instances, I got the following:

	avg cost	max cost	avg nodes	max nodes	avg run time (ms)	max run time (ms)
Manhattan	34.7	50	133,332	2,110,887	49,561	606,838
Manhattan++	34.7	50	53,637	962,324	19,723	239,468
Improvement	n/a	n/a	x2.5	x2.2	x2.5	x2.5

6.2 Intermediary case - 3x3

6.2.1 Perfect learner

As discussed in the previous section section 6.1, 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 6.2. 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 6.2: Perfect Learning 3x3 SP

6.2.2 Deep reinforcement learner

The DeepReinforcementLearner's learning is shown in figure 6.3:

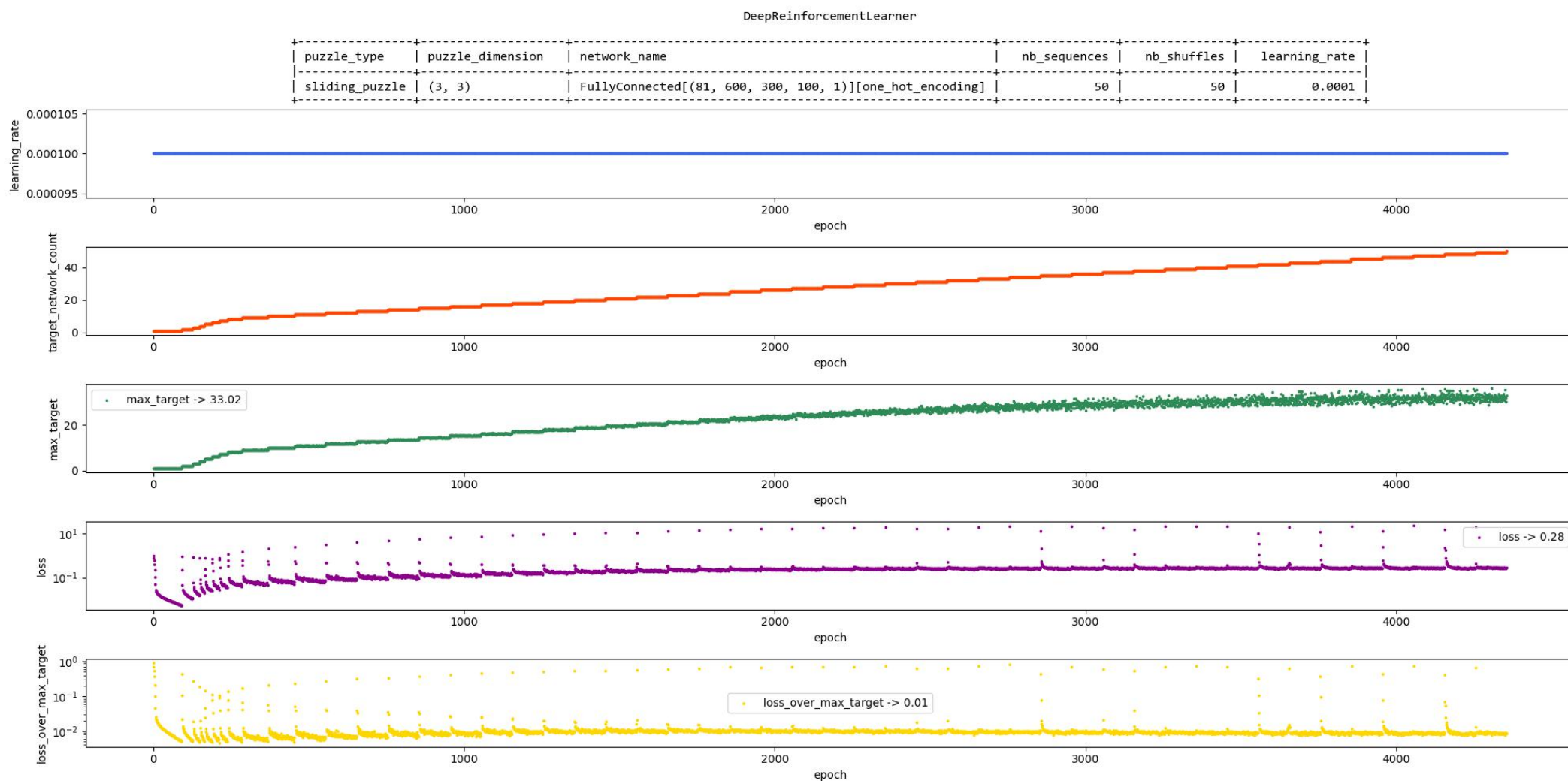FIGURE 6.3: Deep reinforcement learner 3x3 SP

6.2.3 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 6.4

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

Legend:
- AStarSolver[Manhattan++]
- AStarSolver[Manhattan]
- AStarSolver[PerfectHeuristic]
- BFSSolver[SlidingPuzzle[(3, 3)]]
- NaiveSlidingSolver[SlidingPuzzle[(3, 3)]]
- deep_learner_[puzzles_seen=0.67%]
- deep_learner_[puzzles_seen=7.92%]
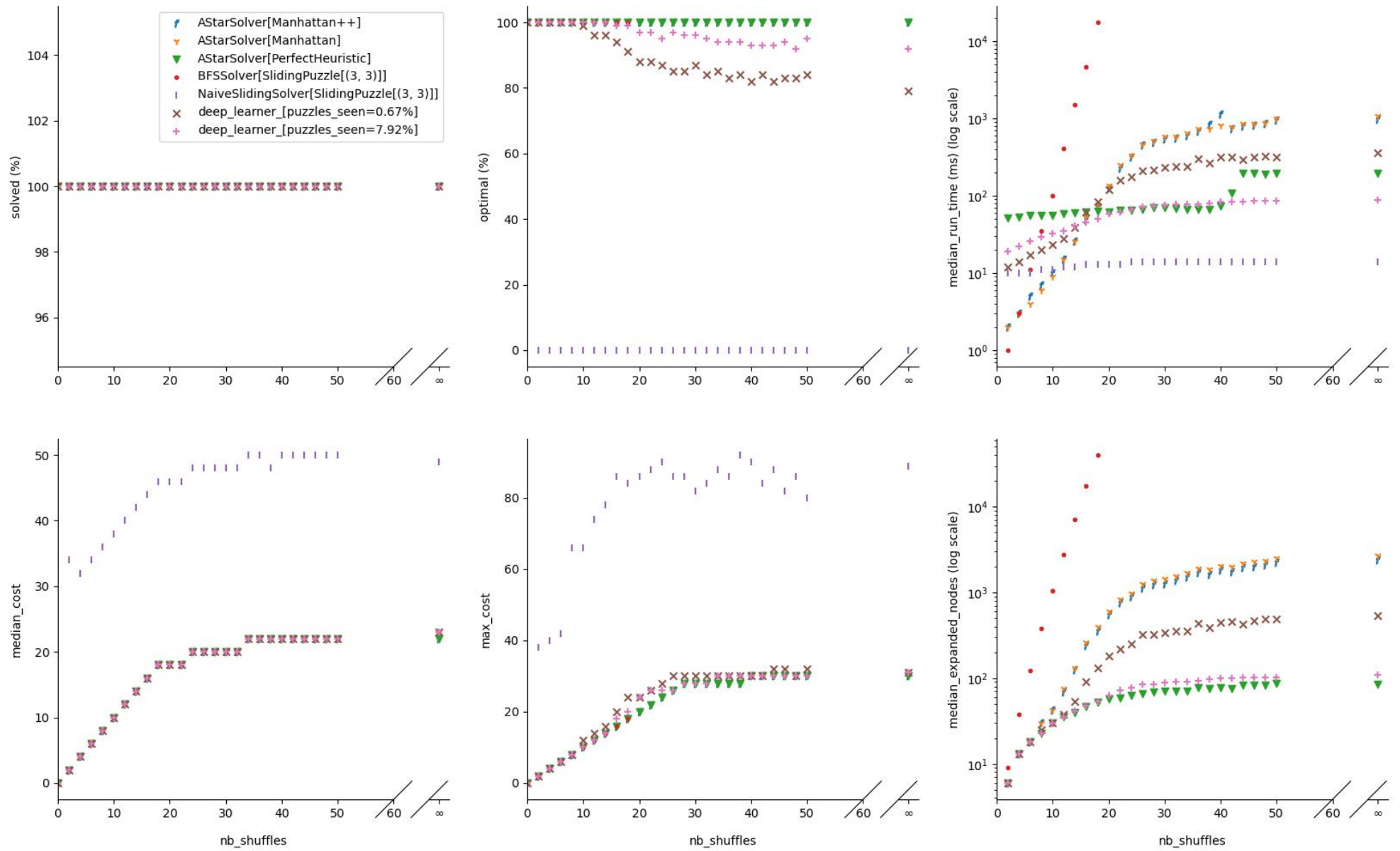
FIGURE 6.4: Solvers' performance comparison 3x3 SP

6.2.4   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 6.1) 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 6.2.3, 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.

6.3   3x4

TBD

6.4   4x4

TBD

6.5   5x5

TBD

Chapter 7

Results - Rubiks' Cube

7.1   2x2x2

blablabla

7.2   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

Alexander Chuang (2022). *Analyzing The Rubik's Cube Group Of Various Sizes And Solution Methods*. RubiksChicago. Accessed: 2022-07-31.

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

Cubastic (2022). *Cubastic 15 Rubiks*. youtube. Accessed: 2022-06-22.

Martin Schoenert (2022). *Orbit Classification*. OrbitClassification. Accessed: 2022-08-06.

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.

Richard Korf and Larry Taylor (2022). *Sliding Puzzle Washington Uni*. washington. Accessed: 2022-07-31.

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

Silviu Radu (2007). *Ner Upper Bounds on Rubik's Cube*. RubiksRadu. Accessed: 2022-08-01.

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

Wikipedia (2022a). *Rubiks Cube*. wikipedia. Accessed: 2022-08-06.

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

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