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

June 26, 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

blablabl...

Acknowledgements

I am grateful to my employer Bank of America for sponsoring my part time 2-year MSc in Artificial Intelligence at Royal Holloway. I entered in the study of AI with a heavy dose of skepticism and am delighted to have learnt so much and revised my judgment of this field. In particular I would like to thank my managers Mitrajit Dutta and Stephen Thompson for supporting me in pursuing this MSc.

I have been fortunate to have had the company of my colleague Saurabh Kumar, as he decided to follow me in this MSc journey. We have had countless opportunities to debate about all the new cool stuff we learnt during the MSc, and that has definitely made the whole process much more enjoyable.

I would also like to thank Professor Chris Watkins, inventor of QL :), for agreeing to supervise my project. It was an honour to attend his very lively lectures on Deep Learning during the first year of the MSc, as well as to absorb some of his wisdom and enthusiasm at our meetings during my work on this project.

Last but not least, my wife Thanh Nhã is my biggest supporter in everything I do, and her continued encouragement and enthusiasm in my pursuit of this MSc has made it so much easier to juggle between a demanding banking job, running 50 to 60 miles a week in preparation of multiple marathons and half marathons, and the coursework, lectures and exam revisions.

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 Sliding Puzzle and the Rubik's Cube. Both these games 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 adjascent; 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 adjascent to the empty compartment into it. Notice from a programming and proof-design perspectiv, it is often easier to equivalently think of the empty compartment being moved (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:

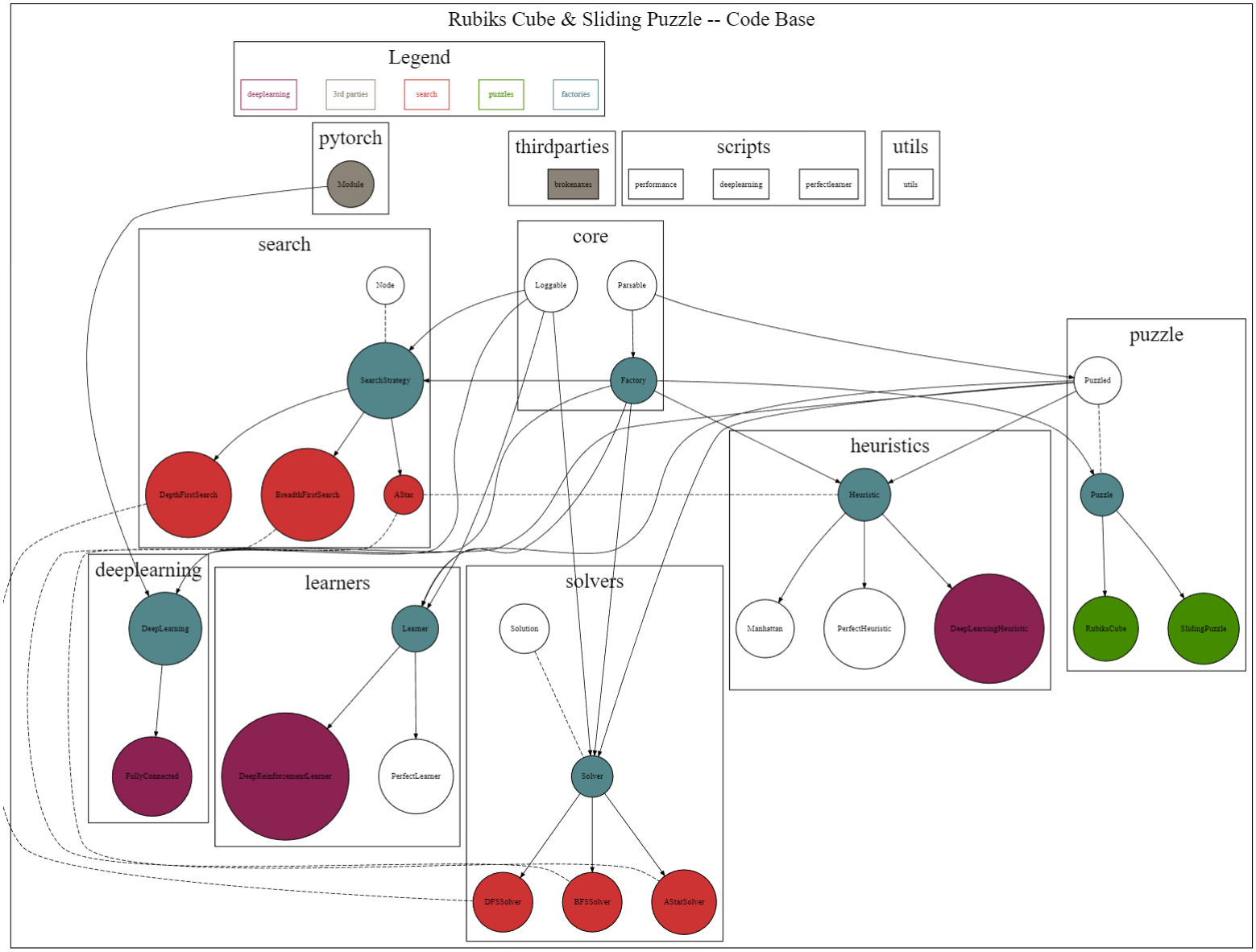

FIGURE 4.1: Code base

Let me describe what each submodule does:

4.1.1 rubiks.core

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

- 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.

4.1.3 rubiks.heuristics

This module contains base class Heuristic, also a Factory (the supported widges are Manhattan, specific to the sliding puzzle, discussed in more details in 5.2,)

4.1.4 rubiks.search

4.1.5 rubiks.deeplearning

4.1.6 rubiks.learners

4.1.7 rubiks.solvers

4.2 Running An Example

blabla

4.3 DL Training

blabla

4.4 DRL Training

blabla

4.5 Solvers Comparison

blabla]

Chapter 5

Results - Sliding Puzzle

5.1 Low dimension

blablabla

5.1.1 Perfect Heuristic

blablabla

5.1.2 Results

blablabla

5.2 Intermediary case - 3x3

As seen above, we have actually been able to solve the 3 by 3 case 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 to make trying deep reinforcement learning.

5.3 3x4

blabla

5.4 4x4

blabla

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.

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.

Misc

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

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.