Solving the 15-Puzzle & Rubik's Cube

François Berrier

Submitted for the Degree of Master of Science in

Artificial Intelligence

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

September 8, 2022

Abstract

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 Rubik's 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. Finally, in chapters 6 and 7, I will present all my various results on respectively the sliding puzzle and the Rubik's cube before offering some conclusion in chapter 8.
A first appendix details how to use the code base, from constructing puzzles, to learning and solving them using all the different methods I have implemented for this project. A second appendix describes a limitation I bumped into while using the kociemba library (an open source 3x3x3 **RC** solver), and how I circumvented it.

Contents

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, 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. I am also glad to have followed his advice at our last meeting: I stopped shaving for a month, which did free enough time for me to implement DQL (joint policy and value learning), as well as a Monte Carlo Tree Search algorithm, to nicely complement my assortment of sliding-puzzle and Rubik's cube solvers.

Last but not least, my wife Thanh Nhã is my biggest supporter in everything I do, and her continued encouragement in my pursuit of this MSc has made it so much easier to juggle between a demanding banking job, running 50 to 70 miles a week in preparation for the Chicago marathon, and the coursework, lectures and exam revisions.

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 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, 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 Rubik's cube. I am in particular looking to solidify my understanding of DRL and DQL 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 Puzzles 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 puzzles 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.

2 Deep Reinforcement Learning Search

In this chapter, I will succinctly go through the different techniques that underly the algorithms I have implemented and compared in this project and present them in (very simplified) pseudo code. Along the way, I will give some references for the interested reader.

2.1 Graphs Search & Heuristics

It is quite fruitful to think of single-player, deterministic, fully-observable puzzles such as the **SP** and **RC** as unit-cost graphs. We start from an initial *node* (usually some scrambled configuration of the **SP** or **RC**), and via legal moves, we transition to new nodes, until hopefully we manage to get to the goal in the shortest number of moves possible.

Graphs search (or graphs traversal) is a rather large, sophisticated and mature branch of **CS** that studies strategies to find *optimal* paths from initial to goal node in a graph. What is meant by *optimal* can vary, but usually is concerned with one or several of minimizing the run-time of the strategies, their memory complexity/usage or the total cost/length of the solutions they come up with. In the general theory, different transitions can have different costs associated with them. In this project however, we can reasonably consider that each move of a tile in the **SP** or each rotation of a face in the **RC** are taking a constant and identical amount of time and effort to perform, and I will therefore consider all the graphs to be unit-cost (i.e. all moves have cost 1).

Search strategies typically maintain a frontier, which is the collection of unexpanded nodes so far. They keep expanding the frontier, by choosing the next node to expand (expanding a node simply means adding all of its children nodes - i.e. those which are reachable via legal transitions - to the frontier for future evaluation) until a solution is found. The question is how to choose the next node to expand! Some graphs search strategies are said to be *uninformed*, in the sense that they do not exploit any domain-specific knowledge or insight about what the graphs represent to choose the next node to expand. In that category fall for instance Breadth First Search (**BFS**) and Depth First Search (**DFS**), which as their name suggest expand nodes from the frontier based on, respectively, a **FIFO** and **LIFO** policy. **BFS** and **DFS** will only work for modest problems where the number of transitions and states are reasonably small; they can however work in a wide variety of situations as they make no assumptions and require no knowledge. It is quite easy to see that **BFS** is an *optimal* search strategy, in that when it does find a solution, that solution is of smallest possible cost (i.e. optimal). **DFS** is obvisouly not optimal as it could very well find a solution very deep in the graphs while expanding the very first branch, not realising that a solution was one level away from the start on another branch!

pseudo code – BFS & DFS

```
################################################
def blind_search(initial_node, time_out, search_type=Search.BFS):
   """ pseudo code for BFS/DFS """
    initialize_time_out(time_out)
    if initial_node.is_goal():
        return initial_node
    explored = set()
    frontier = [initial_node]
    while True:
        check_time_out() # -> raise TimeoutError if appropriate
        if frontier.empty():
            return None # -> Failure to find a solution
       # FIFO/LIFO for BFS/DFS
        node = pop_front(frontier) if search_type is Search.BFS else
                                              pop_back(frontier)
        explored.add(node)
        for child in node.children() and not in frontier.union(explored):
            if child.is_goal():
                return child
            frontier.add(child)
################################################
```

Informed strategies, on the other hand, exploit domain-specific knowledge. One very popular such strategy is **A*** (see Dechter and Pearl, 1985), which always first examine the node of smallest expected total cost (**f**), itself the sum of the cost-so-far from initial to current node (**g**) plus the expected cost-to-go from current node to solution (**h**). The expected cost-to-go **h** is often called a *heuristic*. The better the heuristic is, the better **A*** usually performs. An important property of heuristics is *admissibility*. In short, admissible heuristics never over-estimate the real cost-to-go, and can easily be shown to render **A*** optimal.

pseudo code – A*

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

def A*(initial_node, time_out, heuristic):
    """ pseudo code for A*
    g = cost from initial_node to a current node
    h = heuristic (expected remaining cost from current node to goal)
    """
    initialize_time_out(time_out)
    node = initial_node
    cost = heuristic(node) # g = 0
    frontier = sorted_multi_container({cost: {node}})
    explored = set()
    while True:
        check_time_out()
        if frontier.empty():
            return None # -> Failure to find a solution
```

```
        (cost, node) = frontier.pop_smallest() # smallest total expected
                                               cost
        if node.is_goal():
            return node
        explored.add(node)
        for child in node.children(): # -> children have g = node.g + 1
            child_cost = child.g + heuristic(child)
            if child in frontier and child_cost < cost:
                frontier.update_cost(child, child_cost)
            elif child not in explored:
                frontier.add(child, child_cost)
#
                                        ##########################################################
```

2.2 Reinforcement Learning

As mentioned in the abstract, **RL** (see Sutton and Barto, 2018 for a brilliant exposition of the basic concepts and theory) has known a bit of false start in the 1950s and 1960s but recently been extremely successfully applied to a variety of problems (from Atari to board games and puzzles, robotics and else). **RL** is concerned with how intelligent agents can learn optimal decisions from observing/experiencing rewards while interacting in their environment. One of the fundamental concept in the field is that of value function $s \rightarrow \mathbf{V}(s)$, which tells us the maximum expected reward - or equivalently and better suited to our puzzle solving task, the minimum cost - the agent can obtain from a given state s (if it takes optimal decisions). In finite state and transitions spaces, the value function can be remarkably computed by a rather mechanical and magic-like procedure called value-iteration (kind of the equivalent of the Bellman equation from optimal control), where each state s's value is iteratively refined by combining the value of s's reachable children states.

pseudo code – RL Value Iteration

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

def value_iteration(states):
    """ pseudo code for RL value iteration """
    V = {state: inf for state in states}
    change = True
    while change:
        change = False
        for state, cost in V:
            V[state] = min(V[child] + t_cost for child, t_cost in state.
                                                children())
            change |= V[state] < cost
    return V
```

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

2.3 Deep Learning

Quite similarly to **RL**, Deep Learning (**DL**) has not initially had the success the **AI** community was hoping it would. Marvin Minsky, who often took the blame for having *killed* funding into the field, even regretted writing his 1969 book (Minsky and Papert, 1969) in which he had proved that three-layer feed forward perceptrons were not quite the universal functions approximators some had hypothesized them to be. Since around 2006 (see Goodfellow, Bengio, and Courville, 2016 for more history of the recent burgeoning of **DL**), the field has known a rebirth, probably due to a combination of factors, from ever more powerful computers and larger storage, the availability of data sets large and small on the internet, the advances in many techniques and heuristics (backward propagation, normalisation, drop-outs, different optimisation schemes, etc ...) and the huge amount of experimentation with different architectures (from very deep feed forward fully connected networks, to convolutional, recurrent and more exoctic networks). It is not an exageration to say that **DL** has rendered obsolete entire fields of research and bodies of knowledge, most notably in **CV**, robotics, computational biology and **NLP**. **DL** has often become the method of choice to learn or approximate highly dimensional functions or random processes. The beauty of it is that the relevant features are autodiscovered during the training of the network, via back-ward propagation and trial-and-error, rather than having to be postulated or handcrafted as is often the case with other **ML** techniques.

2.4 DL and DLR Heuristics

So what is the link between all of these: A*'s heuristics, **DL**, **RL**'s value iteration?

Sometimes we have ways of computing good solutions/heuristics to our puzzles, but cannot computationally do this for all possible states (maybe the state space is simply too large!). One approach in that case is to compute the solution/heuristic for *some manageable number of* states, and let a **DL** model learn how to extrapolate to the rest of the state space. This is a case of *supervised* learning in some sense, since we need a teacher-solver to tell us good solutions or heuristics to extrapolate from. In very simplified pseudo code, the procedure looks like the following:

pseudo code – DL heuristics

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

def deep_learning(puzzle_type,
                  teacher_heuristic,
```

```
                  loss_function,
                  network_architecture):
    """ pseudo code for learning a puzzle via deep learning and a teacher
                                          heuristic """
    target_value_function = {state: teacher_heuristic(state)
                             for state in generate_random_states(
                                                                 puzzle_type
                                                                 )}
    dl_heuristic = train_neural_network(target_value_function,
                                        loss_function,
                                        network_architecture)
    return dl_heuristic
#
                                    ##################################################
```

For this project, I have implemented the above procedure of training a **DL** network from the solutions of other solvers/heuristics in a class called DeepLearner (see section 4.6 for details).

An alternative approach in order to compute the cost-to-go is via **RL** value-iteration as described in the previous section (2.2). The state space is often so large however - and this certainly is the case for pretty much all **SP** and **RC**, except maybe the smallest dimensional **SP**s - that we cannot practically perform the value iteration procedure for all states. This is where **DL** comes again to the rescue to act as function approximator, combining with **RL** into what has become known as **DRL**. The subtelty is then that both the left-hand-side and right hand side in the value-iteration procedure are given by a **DL** network which we are constantly tweaking. For my implementation of **DRL**, I followed the same approach as in Mnih et al., 2013, utilising two loops. During iteration over the first (inner) loop, the right hand-side V of the value-function update equation is computed using a fixed *target-network*. The left hand side V is using another *current-network* which alone is modified by the **DL** learning (backward propagation and optimization), until convergence is deemed reached. When convergence in the inner loop is deemed obtained, we break out of it, and run the outer loop, which updates the *target-network* via a copy of the *current-network*. The outer loop is itself broken out of when deemed appropriate (some kind of convergence criteria). In simplified pseudo-code, this looks like:

pseudo code – DRL heuristics

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

def deep_reinforcement_learning(puzzle_type,
                                loss_function,
                                max_epochs,
```

```
                                target_network_update_criteria,
                                puzzles_generation_criteria,
                                network_architecture):
    """ pseudo code for learning a puzzle via deep reinforcement learning
                                                    """
    net = get_network(puzzle_type, network_architecture)
    target_net = get_network(puzzle_type, network_architecture)
    epoch = 0
    puzzles = list()
    while epoch < max_epochs and other_convergence_criteria(network):
        epoch += 1
        """ Generate a new bunch of puzzles """
        if puzzles_generation_criteria(puzzles):
            puzzles = generate_random_states(puzzle_type)
        V = dict()
        """ Compute their updated cost via value iteration ...
             all moves are assumed to have cost 1 """
        for puzzle in puzzles:
            if puzzle.is_goal():
                V[puzzle] = 0
            else:
                V[puzzle] = 1 + min(target_net(child) for child in puzzle
                                                          .children()))
        """ Train lhs network to approximate rhs target network better
        i.e. perform a forward / backward-propagation update
        """
        network = train_neural_net(V,
                                   loss_function,
                                   network_architecture)
        """ Update the target network if criteria are met
        (e.g. epochs, convergence, no progress, etc...) """
        if target_network_update_criteria(net, target_net):
            target_net = copy(net)
    return net
#
                                        ##################################################
```

For this project, I have implemented the above algorithm in a class called DeepReinforcementLearner which I shall describe in details in section 4.6.

2.5 Deep Q-Learning

Another fundamental concept of **RL** is that of the Q-function $Q(s,a)$, the maximum expected reward the agent can obtain from state s, taking action a and acting optimally afterwards. Notice that the knowledge of V everywhere is equivalent to that of Q, since obviously:

$$V(s) = max_{a \in \mathcal{A}(s)} Q(s,a) \tag{1}$$

where $\mathcal{A}(s)$ are the transitions/actions available from state s. Conversely, we can recover Q from V since:

$$Q(s,a) = R(a) + V(a(s)) \tag{2}$$

where $R(a)$ is the reward of performing action a and $a(s)$ is the state we reach by performing action a from state s. Similarly to the way one can compute V via value-iteration (at least in finite state and action spaces), Q can also be computed by iteration (See Watkins and Dayan, 1992 for details of convergence of that procedure).

Similarly to our discussion about **DRL**, in cases where the state space is large, it can be useful to approximate Q, or a somewhat related quantity (such as e.g. a probability distribution over the actions that can be taken from s) by a neural network, and use a similar procedure to that of the previous section to update the right-hand-side and left-hand-side of the iterations. That becomes **DQL**!

I have for this project followed the exact same procedure described in McAleer et al., 2018b (and also used by alpha-go Silver et al., 2016): the output of the neural network that is learnt by **DQL**, which actually is surprisingly similar to the approach and pseudo code shown for **DRL** earlier. The only 3 differences are that:

- the networks (target and current) now output a vector of size $a + 1$, where a is the number of possible actions/moves. That is, one dimension for each of the possible actions, representing the probability/desirability of that action, and one dimension for the value-function's value.
- the target update by Q-iteration now needs to update both the value-function and the optimal decision (a-dimensional vector of 0s, with a 1 for the optimal decision only)
- the loss function used need to be compatible with the size of the output and target vectors. I have used as loss the average of the mean square error of the value function difference (current network and target) and the cross entropy loss of the actions probability versus the actual optimal move.

Since in my **DRL** pseudo-code above (as well as in my actual code), the loss function is abstracted, and the network architecture (including the size of the output) configurable, the only modification to get **DQL** in lieu of **DRL** are extremely minimal, and essentially only to do with the Q-iteration-update section:

pseudo code – DQL heuristics

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

def deep_q_learning(... same as DRL ...):
    """ pseudo code for learning a puzzle via deep q learning """
    # ... same as DRL ...
    nb_actions = puzzle_type.get_nb_actions()
    while epoch < max_epochs and other_convergence_criteria(network):
        # ... same as DRL ...
        Q_and_V = dict()
```

```
        """ Compute updated cost and actions via V & Q iteration ...
             all moves are assumed to have cost 1 """
        for puzzle in puzzles:
            value = target_network(puzzle)[-1]
            actions = [0] * nb_actions
            best_action_id = 0
            if puzzle.is_goal():
                value = 0
            else:
                for action_id, child in puzzle.children():
                    child_value = target_network(child)[-1]
                    if child_value < value:
                        value = child_value
                        best_action_id = action_id
            # We update both the value function and the best action
            actions[best_action_id] = 1
            Q_and_V[puzzle] = actions + [value]
        # ... same as DRL ...
#
                                        ##################################################
```

For this project, I have implemented the above algorithm in a class called DeepQLearner which I shall describe in details in section 4.6.

2.6 Monte Carlo Tree Search

Since my implementation of **DQL** is no longer just a cost-to-go estimate as with **DRL**, but a joint cost-to-go and distribution over actions (to be concrete, that means a 5-dimensional vector for the **SP** and a 13-dimensional vector for the **RC**). This leaves us with the question of how can we use the output of such a network to search for a solution. One simple answer (which I have tried, see e.g. subsection 5.2.4) is that we could disregard the distribution and only take into account the cost-to-go by simply using A^*. The rationale for doing so is that we might hope that jointly learning the cost-to-go and the actions via a combination of MSE and cross-entropy-loss produces a better network and leads to better heuristics, since we use more information during the learning and iterations.

Another possibility is to use an algorithm which combines both the actions distribution and the value function to inform its search. The Monte Carlo Tree Search (**MCTS**) is such an algorithm. In short, it generates at each step (and possibly in a distributed or multithreaded manner) paths to a *leaf* on the frontier of unexpanded nodes, expands that node and checks if it is a goal state (and keeps going until the goal is found). The paths followed from initial state to leaves are decided via a combination of the probability distribution over actions (following the most promising actions) as well as the value function, according to a trade-off which can be tuned via a hyper-parameter. In order to prevent always following the same paths (which might take us very far from the goal), **MCTS** algorithms usually keep track of actions they have followed already

and underweight their probability so that subsequent paths explore other actions. A pseudo code description of it looks like this:

pseudo code – MCTS

```
######################################################################
def MCTS(initial_node, time_out, q_v_network):
    """ pseudo code for Monte Carlo Tree Search """
    initialize_time_out(time_out)
    tree = Tree(initial_node)
    while True:
        check_time_out()
        leaf = construct_path_to_new_leaf(tree, q_v_network)
        if leaf.is_goal():
            return BFS(tree) # path to leaf can usually be improved by
                                                        BFS
######################################################################
def construct_path_to_new_leaf(tree, q_v_network):
    leaf = tree.initial_node
    while leaf.expanded:
        actions_probas, actions_values = q_v_network(leaf)
        """ choose next node based on joint actions probas & values
        as well as history of actions taken
        """
        leaf = leaf.choose_child(actions_probas,
                                 actions_values)
    # add children of leaf to tree
    leaf.expand()
    # penalize path taken to favour exploration
    leaf.increment_historical_actions_count()
    return leaf
######################################################################
```

I have implemented the exact **MCTS** procedure described in McAleer et al., 2018b and outlined in the above pseudo code in a class called MonteCarloTreeSearch (see section 4.7 for details).

3 Puzzles

3.1 Sliding Puzzle

3.1.1 History - the 15-Puzzle

The first puzzle I will focus on in this thesis is the sliding puzzle (see Wikipedia, 2022b) whose most common variant is the 15-puzzle, seemingly 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 adjascent; the additional move that this allows renders all 16! configurations solvable.

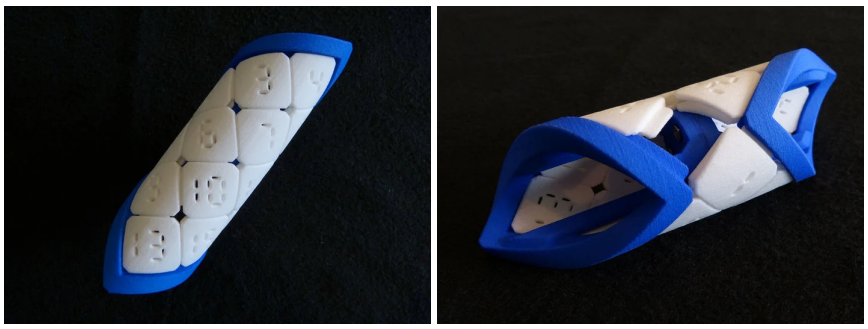


Figure 1: Coiled 15-puzzle

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 adjascent 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 globe 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 2.1, 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 Rubik's Cube

3.2.1 History – from Erno Rubik to speed cubing competitions

The second puzzle I will focus on is the well known Rubik's Cube (**RC**), invented in 1974 by Erno Rubik, originally known under the name of *Magic Cube* (see Wikipedia, 2022a). Since it was commercialized in 1980, it has kown a worldwide success: countless people are playing with the famous original 3x3x3 cube, as well as with many variants of it, more or less difficult and contrived. Competitions nowadays bring together *speedcubers* who have trained for years and memorized various algorithms to solve the Rubik's 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 cube in (only) two and a half hours! (see Cubastic, 2022). The key insight to solving high dimensional **RC**s (short obviously of coming up with specific and sophisticated strategies) is the following: if we manage to group all the inside (i.e. not located on a corner or an edge of the cube) cubies by colour, and then all the non-corner edges on a given tranche of a face by colour, we are essentially reducing the cube to a 3x3x3, since no face rotation will ever undo the inside cubies and non-corner-edge cubies' grouping. For instance, the following picture shows faces of my own 4x4x4 and 3x3x3 cubes which I have arranged to be in identical configurations.

It is quite interesting to note that the difficuly of hand-solving cubes does not grow nearly as much with increasing dimensionality (most of the time difference comes rather from the difficulty of handling and manipulating the cubes of high dimensions at speed).

Figure 2: Reducing a 4x4x4 **RC** to a 3x3x3 – home made picture

It is quite clear from that picture above that if we can bring the 4x4x4 cube to such a configuration on all faces (which is not super hard), and we know how to solve a 3x3x3, we are essentially done. (well, almost, since the additional degrees of freedom coming from the higher dimension of the 4x4x4 mean that certain arrangements of the colors which are possible on the 4x4x4 are not on the 3x3x3. For instance we can never have the center red and white on opposite faces of a 3x3x3, but we easily can on a 4x4x4! Fortunately that is nothing that a couple of easily learnable algorithms cannot fix).

3.2.2 Conventions: faces, moves and configurations

In my code as well as in this write-up, I follow the conventions and notations of the **RC** literature and community as much as possible, I will refer to the faces as F (front), B (back), U (up), D (down), L (left) and R (right). The cubes' tiles are of 6 different colors, and without loss of generality (though the interested reader is referred to appendix 9 for some subtlety around that), 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 Rubik's has, as one would expect, an extremely large state space. I will compute the state space size for the

3x3x3 and 2x2x2 cubes in the next two subsections and discuss conditions under which a cube is solvable.
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. Some people refer to the convention I have chosen as the *quarter turn metric*, and obvioulsy the length of solutions is affected by which convention/metric we use.
Finally, let me make a remark on what I consider in this project to be distinct configurations of the Rubik's cube. Since I am interested in experimenting with, among other methods, **DRL** and **DQL**, and that they are usually presented as methods which learn *without-human-knowledge*, I thought it would be a bit akin to cheating to consider that two cubes equivalent by full space rotation are one and the same and represent them as such in my code. Ideally, we would rather let the algorithms learn (if they are able to) about symmetries. For instance, the two following cubes (which are clearly just a rotation away or equivalently a change-in-observer away from one another) are the same for us humans. I however represent them as different tensors in my code, and this is up to the solvers to make sense of things.

Figure 3: Two equivalent configurations by full-cube-rotation

3.2.3 3x3x3 Rubik's – search space, solvability & God's number

A good strategy to compute the state space size of a the Rubik's of a given dimension is to compute:

- N: the number of possible arrangements of the *cubies* (individual small cubes that constitute the Rubik's) in space, taking into account physical constraints and some invariants.

- D: the number of possible configurations which are equivalent via full cube rotation in space.

The state space can then be considered to be either $\frac{N}{D}$ or N, depending on our representation (as mentioned earlier my code will treat each rotation of the cube in space as a different representation, so I shall not be dividing by D below).

In order to compute D, we just need to realise that each configuration can be arranged in (only) 24 different ways via full cube rotation in space. In particular that means that there are 24 possible goal states (and not 6! as we might naively think at first), so for instance in figure 3 I could only have picked 2 out of 24 different configurations. Why is this?
Any of the six colors can be placed on the F face, and then any of the four adjacent colors can be placed on the U face (say). Once these two colors 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 cube 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 color (e.g. enumerated clock-wise) is invariant. For instance, consider the corner cubie with colors (F, R, U). No scrambling of the cube will ever 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).

Computing N is a bit tricker. For the 3x3x3 Rubik's, each of the 8 corner cubies could be placed at one of the physical corner locations and oriented three different ways (rather than six, since as discussed earlier, the clock-wise orientation of a given corner cubie can never be changed). We have therefore $8!.3^8$ corner arrangements. Similarly, the 12 edge cubies can be placed at 12 physical edge locations and oriented 2 different ways, so we have another multiplicative factor of $12!.2^{12}$. There are however some well know parities (see Martin Schoenert, 2022 for details) which are invariant under legal moves of the Rubik's cube and which reduce the number of arrangements that are physically attainable via face rotations. The *corner orientation parity* (**COP**) for instance sums over the corner cubies the number of 120^o rotations it would take to bring a given cubie to its standard orientation (i.e. compared to the top cube in figure 3). It is easy to observe that the **COP** is invariant (% 3) since any face rotation will displace four corner cubies, two of which see their individual **COP** increase by 2, and the other two by 1 (so a total increase of 6). Similarly, there are two additional parities called *edge orientation parity* (**EOP**) and *total permutation parity* (**TPP**) which are both invariant (% 2) under face rotation. Overall, these three parities mean that only 1/12 of the $8!.3^8.12!.2^{12}$ arrangements are physically attainable, leaving us with a grand total of

43, 252, 003, 274, 489, 856, 000 different cubes (still a rather staggeringly large number!).

Finally, let me mention that there is a long history of research into finding tighter and tighter bounds on the 3x3x3 God's number (usually, proofs of upper bounds were obtained and refined by decomposing the Rubik's moves group into subgroups, then finding upper bounds of distance between two elements of a given subgroup, and finally summing these upper bounds over the different subgroups). See e.g. Alexander Chuang, 2022 for an exposition of this type of approach. In 2007, an upper bound of 34 in the quarter turn metric was published (Silviu Radu, 2007) but the question got properly settled in Tomas Rokicki and Morley Davidson, 2022 by Google's researchers which used clever symmetry arguments to reduce the search space to the maximum, followed by Google's vast computing power capabilities to brute force solve all of the configurations they were left with. They concluded that the 3x3x3 Rubik's God number is in fact a mere 26.

3.2.4 2x2x2 Rubik's – search space, solvability

For the 2x2x2 cube, there are eight possible ways of choosing the physical position of the eight corner cubies (and 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 we know all can indeed be obtained). Each corner cubie can be oriented in 3 different ways, so we get a total of $8!.3^8$ configurations. However, due to the **COP** parity discussed in the previous section, we conclude that only one third of these are attainable (namely those for which the **COP** is equal to 0 % 3). Overall, this gives $8! * 3^7 = 88{,}179{,}840$ possible configurations (probably a larger number than most people would expect for a mere 2x2x2 cube).
The insight and knowledge of this section has been useful for me to implement *perfect scrambling* for the 2x2x2 **RC**. Indeed, the way I generate perfectly shuffled cubes is by randomly placing the 8 corners, then randomly choosing the orientation of the first 7, and finally fixing the 8^{th} corner's orientation to make sure COP is equal to 0 % 3.
Some quick googling suggests the 2x2x2 Rubik's God number to be 11 in the quarter turn metric, but I could not find any clear and definitive paper on this.

3.3 Solving puzzles without human knowledge

In the upcoming implementation (4) and results (5, 6) sections, I will be discussing various solvers. I would put them broadly in four different categories, by decreasing level of human knowledge that has been assumed and handcrafted in them. Notice that this human-knowledge-required scale is obviously somewhat arbitrary, but good to keep in mind in later (and in particular results) sections.

Domain-specific solvers At 4 on the human-knowledge-required scale, I would put my NaiveSolver (handcrafted to solve the **SP**) as well as the hkociemba (2x2x2) and kociemba (3x3x3) open source implementations which I have integrated in my Kociem-

baSolver. They make heavy use of specific knowledge about the two puzzles, not only about their mechanics, but also about how to actually (provably) solve them.

Domain-specific heuristics At 3 on the human-knowledge-required scale, I put A*-with-handcrafted-heuristics (e.g. Manhattan heuristic for the **SP**) algorithms. Constructing the heuristics usually require some knowledge about the puzzle's mechanics, but not necessarily about how to solve them efficiently (though of course knowing the latter might help come up with more efficient heuristics).

Supervised learning At 2 on the human-knowledge-required scale, I put my DeepLearner solver (also using A* but whose heuristic is learnt via **DL**). This solver does not require any knowledge of the structure or mechanics of the puzzle, nor of how to solve them, but just need a teacher algorithm to learn/extrapolate from.

Blind search& unsupervised learning Finally at 1 on the human-knowledge-required scale, I put the blind solvers such as **BFS** and **DFS**. Also in that group I put A* with heuristics learnt using my DeepReinforcementLearner and DeepQLearner, and my MonteCarloSearchTree which all know (almost) nothing about the puzzles and do totally work from generic interfaces alone (states, transitions, etc ...) and are operating in an unsupervised way.

One important comment I would make to end this section is that it is easy to inject, without realising or acknowledging it, specific knowledge into the solvers, or in the tensor representation of the puzzles that the solvers take as input. Pretty much all the papers about **RC** that I have surveyed chose a more compact representation for the cube than I have. They remove the center cubies of the 3x3x3 cube since, they argue, those never move via face-turns. Some of them go much further and make use of various symmetry arguments to reduce the search space and reduce the **RC** to having one goal state (instead of 24). One of the main papers I have taken inspiration from for this project, McAleer et al., 2018a, reduces the representation of the Rubik's considerably more, by making use of arguments about the physical structure of the edge and corner cubies: knowing the color of one of their face allows you do deduct the other faces, and therefore, they only need to track one face. I of course understand that they do that for the sake of reducing the size of the tensors representing the cube, and in order to remove redundant information. This is still *cheating* a bit in my opinion, especially if you are claiming to solve these puzzles *without-human-knowledge*, as we can only perform these dimensionality reduction tricks if we have external world knowledge about the physical structure of the puzzles. In this project, I have chosen as pure an approach as I could, and not only did not reduce the dimensionality of the tensor representation of the puzzles, but also, specifically for the Rubik's cube, I did not make use of invariance by spacial rotation of the full cube.

4 Implementation

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..).

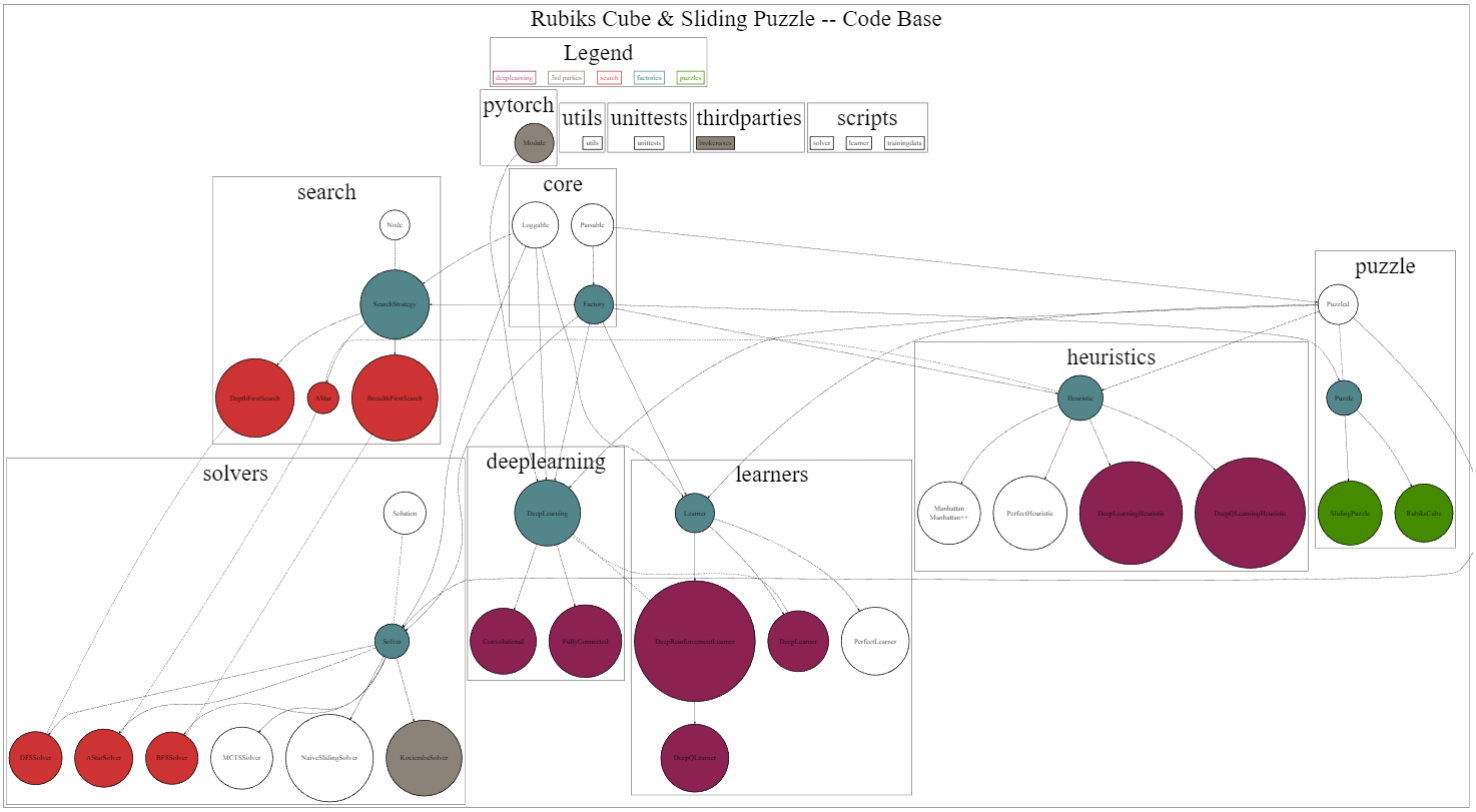


Figure 4: Code base

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

4.1 rubiks.core

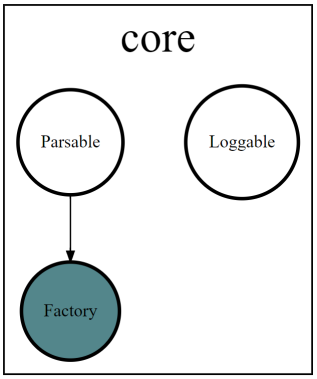


Figure 5: rubiks.core

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

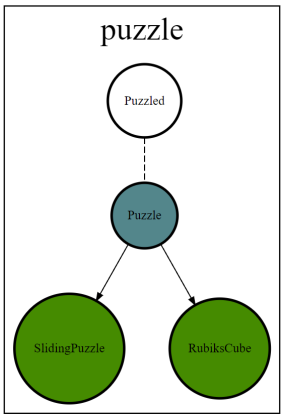


Figure 6: rubiks.puzzle

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

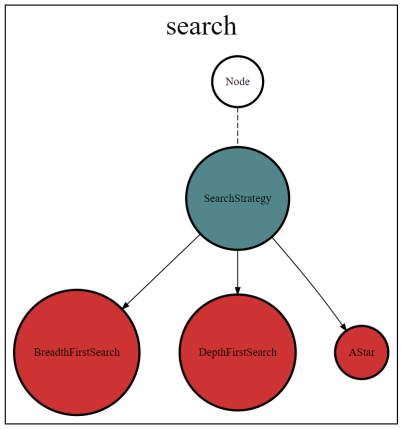


Figure 7: rubiks.search

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

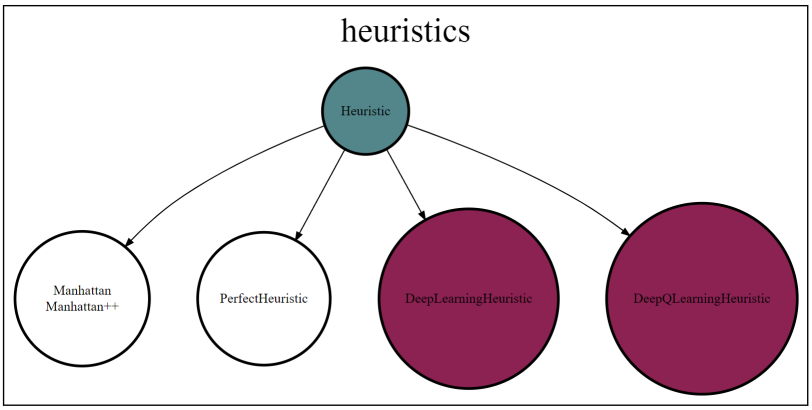


Figure 8: rubiks.heuristics

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 displaces 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 8.3.4 as well as a thorougher performance comparison on the 2x5 **SP** in 5.1.2). 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 2x3 puzzle, 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 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 the left-most or the 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 nxm, the appropriate linear constraint penalties

are computed and populated in a database.

Notice that for an nxm **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 each 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 $nxm \leq 5x5$ 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 **DL** (DeepLearner) **DRL** (DeepReinforcementLearner) or **DQL** (DeepQLearner). See 4.6 for a discussion of these three learners.
- **DeepQLearningHeuristic**: this uses a network which has been trained using **DQL** by the DeepQLearner (see below 4.6). The DeepQLearningHeuristic follows the same interface as the other heuristics, and therefore can be used by A* for instance. It also has an additional method *optimal_actions* that returns the learnt probability distribution over actions for a given puzzle, for use in search algorithms (e.g. **MCTS**) which make use of this to inform their search.

4.5 rubiks.deeplearning

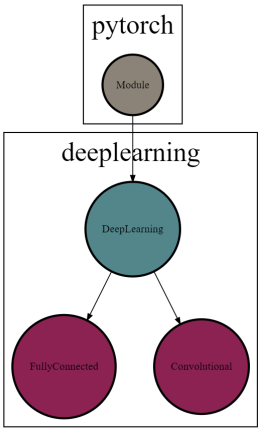


Figure 9: rubiks.deeplearning

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). There is also importantly a *joint_policy* parameter to indicate whether we want the output to be 1 dimensional (e.g. for value function learning) or $a + 1$-dimensional, where a is the size of the actions space (for joint policy-value learning).
- **Convolutional**: similar wrapper to FullyConnected, but with the ability to add some parallel convolutional layers to complement fully connected layers.

4.6 rubiks.learners

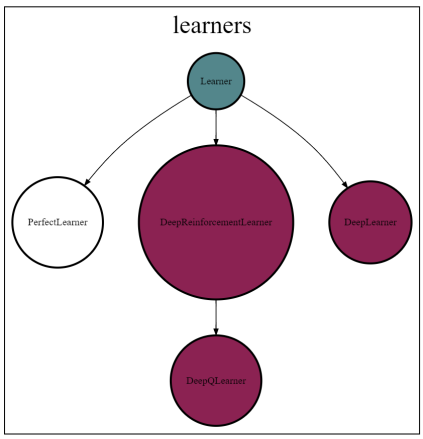


Figure 10: rubiks.learners

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. The four 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 8.2.1 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** DeepLearner instanciates a DeepLarning (neural network), and trains it on training data by minimizing the mean square error between the cost-

to-go from the target solver versus the neural network. The main parameters driving this learner are:

  - *training_data_from_data_base* which controls whether the training samples are generated on the fly via another solver (defaults to $A^*[Manhattan++]$ for **SP** and $A^*[Kociemba]$ for **RC**) or read from a data base (similarly constructed from other solvers, but has the advantage of being done offline)
  - *nb_epochs* and *threshold* which control exit criteria for the network training.
  - *nb_sequences*, *nb_shuffles_min*, *nb_shuffles_max*, *training_data_freq* and *training_data_every_epoch* which control how often we regenerate new training data, how many sequences of puzzles it contais, how scrambled the puzzles should be.
  - *learning_rate*, *scheduler*, *optimizer* and assorted parameters which control the optimisation at each backward propagation.

  Once it has completed (or on a regular basis), DeepLearner saves the trained network, which can then be used with DeepLearningHeuristic to guide A^*.

- **DeepReinforcementLearner**: It instantiates a DeepLearning (network), and trains it using **DRL** essentially following the pseudo-code from 2.4. Unlike DeepLearner, the targets are here generated via value iteration rather than via another solver (teacher). It has very similar parameters to the DeepLearner, in particular some parameters to control exit criteria (same as DeepLearner's plus a few that help it exit early when the value function's range has not been increasing anymore for a while), some parameters to control and configure the optimiser, some to control the training puzzles (how many, how often, how scrambled).

- **DeepQLearner**: The implementation of the DeepQLearner is actually only a few lines of code, for it inherits pretty much all of its behaviour from the DeepReinforcementLearner. The only overwritten functions are:

  - *get_loss_function* which constructs the loss function to be used by the network's training. Instead of returning a simple torch.nn.MSELoss, it returns a function which computes the average of torch.nn.MSELoss on the value-function and torch.nn.CrossEntropyLoss on the actions.
  - *__construct_target__* which performs the the Q-V-iteration to update the targets, as described in pseudo code form in 2.5. This function updates not only the value function of a node given its children nodes, but also updates the optimal action vector (0 for non optimal actions, 1 for the optimal action leading to the child node of highest value function).

4.7 rubiks.solvers

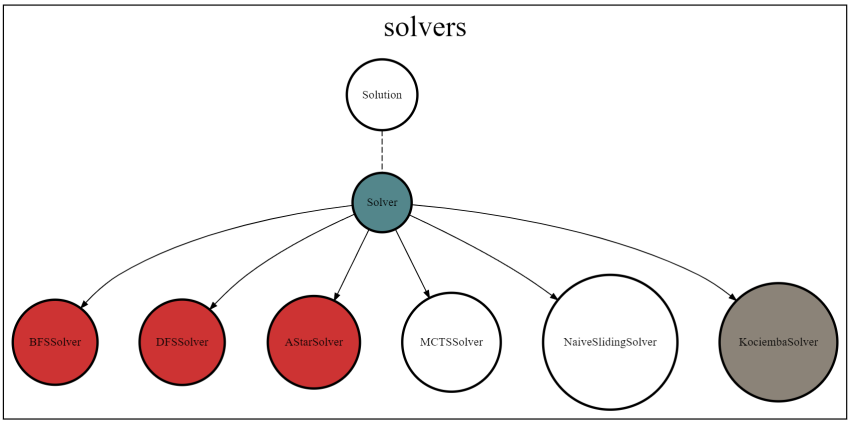


Figure 11: rubiks.solvers

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 (chosable from percentage of puzzles solved before time out, percentage of optimal puzzle (if optimality can be ascertained), mean-median-maximum run time, mean-median-maximum expanded nodes (for solvers where it makes sense), mean-median-max cost, and an optimality score which compares the total sum of cost over all puzzles versus a benchmark solver). Ideally the benchmark solver to compute the optimality score should be one which is provably optimal, but in the case of the **RC** that was not possible so I used A* with KociembaHeuristic as my benchmark. For the **SP** I used the optimal A* with Manhattan++ heuristic.

- **DFSSolver** This solver is literally a few lines of code, extremely close to the pseudo code detailed in 2.1
- **BFSSolver** Ditto, see 2.1
- **AStarSolver** Ditto, see 2.1
- **NaiveSlidingSolver** The NaiveSolver is conceptually simple, but was far from trivial to implement and debug. Its functioning is detailed in the appendix section 8. Basically, it does what a beginner player would do, certainly what I do when I play the sliding puzzle: solving the top row first, then the left column, hence reducing the puzzle to a smaller dimensional one, and keep iterating until done.
- **KociembaSolver** This solver is a wrapper around two libraries. The first one, hkociemba (see Herbert Kociemba, 2022) is a 2x2x2 implementation written by

Kociemba himself, the second one is a Python pip-installable library which solves the 3x3x3 Rubik's. My wrapper simply translates my cube representation in that accepted by these 2 solvers, and their result string into moves usable by my code. I also do some additional massaging for the 3x3x3 case, due to a shortcoming/bug in the 3x3x3 Python library, as I have detailed in appendix 9.

- **MonteCarloSearchTreeSolver** I have implemented the algorithm described in the paper from McAleer & Agostilenni et al McAleer et al., 2018b. My implementation is single-threaded however (in part due to my using Python for this project, which would have made using multiprocessing not efficient at all for their algorithm). A consequence of that is that I dropped the ν parameter from their implementation, since it only introduces a penalty to discourage parallel threads from generating the same paths from initial node to leaves (in single threaded mode, this parameter is inconsequential since the generation of a path decrements it, only to reincrement it once the leaf is generated).
  This solver basically generates (pseudo-random) paths from initial node to a new leaf at each iteration, until we obtain a goal leaf. I use the word pseudo-random since the paths are actually completely deterministic. They indeed result from a trade-off between the probability of actions (penalised over time by discounting actions which have been taken already) and the value function, but there is no random draws performed anywhere in there.
  As is described in McAleer & Agostilenni's paper, I have implemented a final pruning via BFS of the tree resulting from the above *Monte-Carlo* process consisting in generating random paths until a goal state is reached. Indeed, it is clear that the paths generated have very little chance of being optimal since there is no correction mechanism to avoid long and unecessary loops. I have however made this tree trimming by BFS optional via a *trim_tree* parameter as I wanted to study its effect on solutions' optimality and run time (see results section 5.2.3).
  Similarly to their implementation, the trade off between optimal actions and value function is driven by a hyper-parameter c, as I wanted to study its effect as well, and confirm my intuitions as to how it would affect solutions' quality and run time (again see results section 5.2.3).

4.8 data

Finally it is worth noting that the code can/will save on disk a lot of data, including:

- Manhattan++ generates databases of linear constraint penalties (see 4.4)
- TrainingData: (see e.g. 8.2.2) generates databases of puzzles and their associated costs for later training of a DeepLearner.
- Learners: models generated by the DeepLearner, DeepReinforcementLearner, DeepQLearner as well as some convergence data useful to understand how the networks learnt. Similarly PerfectLearner saves its results for later reuse (in e.g. PerfectHeuristic) or display.

- Solver: Sequences of scrambled puzzles which we use to make the performance tests (comparing different solvers) fair.
- Solver.performance_test: Results of the performance tests (all the metrics discussed earlier in 4.7, for each solver and each difficulty level)
- etc . . .

The data is saved according to a file system tree of the form:

$$root/data\_type/puzzle\_type/dimension/file\_name.pkl$$

where the root is driven by the "RUBIKSDATA" environment variable or, if not set, will go somewhere in your HOME.

5 Results - Sliding Puzzle

In this chapter, I show and discuss results on **SP** of various dimensions, going from *small* dimensions (which I define as those I can reasonably *fully* solve given my computing resources, the finite time I have to complete this project, and the choice of programming language), then I move to studying in much detail the case of the 3x3, and finally move on to slightly higher dimensions 3x4 and 4x4. Table 1 below shows which solvers I have run on each of the dimensions attempted. The reader is referred to the implementation chapter 4 for a detailed explanation of each of these solvers, and to the appendix section 8 for code examples on how to run them.

solver \ **Sliding Puzzle**	**2x2**	**2x3**	**2x4**	**2x5**	**3x3**	**3x4**	**4x4**
BFS					x		
A*[Manhattan]				x	x	x	
A*[Manhattan++]				x	x	x	x
A*[Perfect]	x	x	x	x	x		
Naive					x	x	x
A*[DL[FullyConnected]]					x	x	x
A*[DL[Convolutional]]					x		
A*[DRL]					x	x	x
A*[DQL]					x		
MCTS[DQL]					x		

Table 1: Solvers used vs **SP** dimension

I am neither making claims of depth (each solver could surely be tuned and optimized/adapted, possibly differently for each dimension) nor breadth (I have not run all of the solvers versus each dimension). Running these experiments takes a lot of computing time (even though I have very decent computing resources), so I had to be somewhat selective. I wanted however to run all of the solvers I have implemented one one particular dimension (the intermediary 3x3 seemed appropriate to do so) and try to answer as many of the following interesting questions as possible:

- How hard is too hard for **BFS** to complete in *reasonable* time?
- How much, if any, improvement do we get with Manhattan++ over Manhattan?
- Is there much loss between **DL** and **DRL**, i.e. is losing the teacher used by **DL** a deal breaker?
- Is **DQL**, all things being equal, able to learn a better cost-to-go heuristic than **DRL**?
- How does **MCTS** perform? How important is it to tune the hyper-parameter c? How much improvement, if any, does the post trim of the **MCTS** tree via **BFS** bring?

- Are any of the **D*L** techniques able to perform on par with A* with Manhattan++ or perfect heuristic?

All of these questions are answered (obviously not in generality, but in the context of the experiments I have been running) in this chapter.

5.1 Low dimension

5.1.1 God numbers and hardest puzzles

As mentioned in chapter 3, the state space cardinality for the SP grows very quickly with n and m. The only dimensions which have less than 239.5 millions states are shown in table 2. Note I am also only considering $n \leq m$ since dimension mxn can always be solved by symmetry from the solutions mxn:

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

Table 2: # puzzles for *small* dimensions

In this section, I will discuss *full* results for these 5 dimensions. 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 8.2.1. I obtained the following God numbers (table 3) for these puzzles:

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

* provisional result

Table 3: God numbers for *small* dimensions

Notice that among the above dimensions, 2x5 is the largest and hardest one. My current displayed result in table 3 is provisional, as I have only solved a bit over 1.1 million puzzles out of the 1.8 million possible, so it is well possible that I haven't yet bumped into the hardest configuration. I ran a combined 600 hours, much of which run over 16 cores, and even over a full week-end on a c5.18xlarge instance (72 cores) on Amazon EC2.

The perfectLearner also keeps track of the hardest puzzle it has encountered (i.e. a configuration whose optimal solution has a cost equal to the God number). I obtained the following hardest puzzles for each of the 5 *small* dimensions:

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

Most difficult 3x3 (31 moves):

8	6	7
2	5	4
3		1

5.1.2 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 2x5 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:

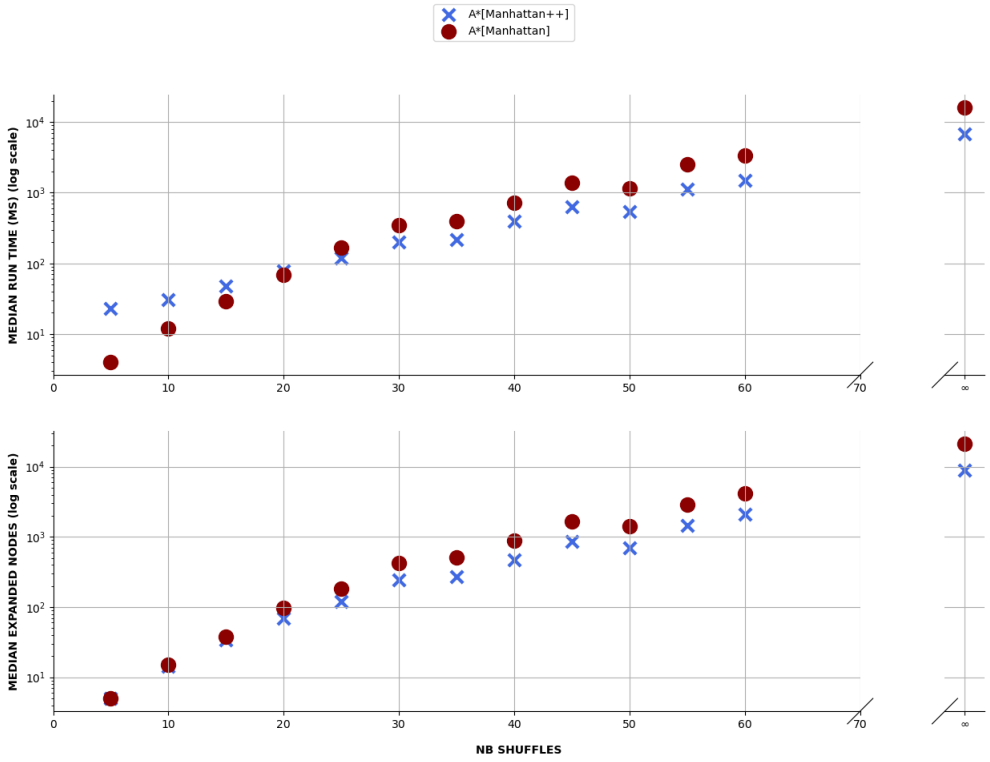


Figure 12: Manhattan vs Manhattan++ for the 2x5 **SP**

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). Table 4 below summarizes the results for the 250 *perfectly* shuffled instances:

	avg cost	max cost	avg nodes	max nodes	avg run time (ms)	max run time (ms)
Manhattan	34.6	49	46,483	575,050	36,088	428,887
Manhattan++	34.6	49	18,780	213,557	14,665	165,830
Improvement	n/a	n/a	x2.5	x2.7	x2.5	x2.6

Table 4: Manhattan++ outperformance on perfectly scrambled 2x5 **SP**

5.2 Intermediary case - 3x3

In this section, which focusses on the 3x3 **SP**, I detail some results from the PerfectLearner (5.2.1), from the DeepReinforcementLearner (5.2.2), before moving to a discussion of the **DQL MCTS** solver and the effect of its trade-off hyper-parameter c and of its optional **BFS** tree-trimming on the quality of the solutions (5.2.3). I then run a comprehensive comparison of my 10 different solvers on the 3x3 **SP** in 5.2.4. Finally, I end the section by running a few select solvers on one of the two hardest 3x3 puzzle in 5.2.5

5.2.1 Perfect learner

As discussed in the previous section section 5.1, the 3x3 **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 **DL**, **DRL** and **DQL**. To start with, I ran the PerfectLearner and the results are shown below in figure 13. It is interesting to note that there are only two hardest configurations (of cost 31) and 221 configurations of cost 30.

PUZZLE TYPE	SLIDING PUZZLE
DIMENSION	(3, 3)
# POSSIBLE PUZZLES	181,440
# SOLVED PUZZLES	181,440
MAX COST	31
HARDEST PUZZLE	8 6 7 2 5 4 3 1

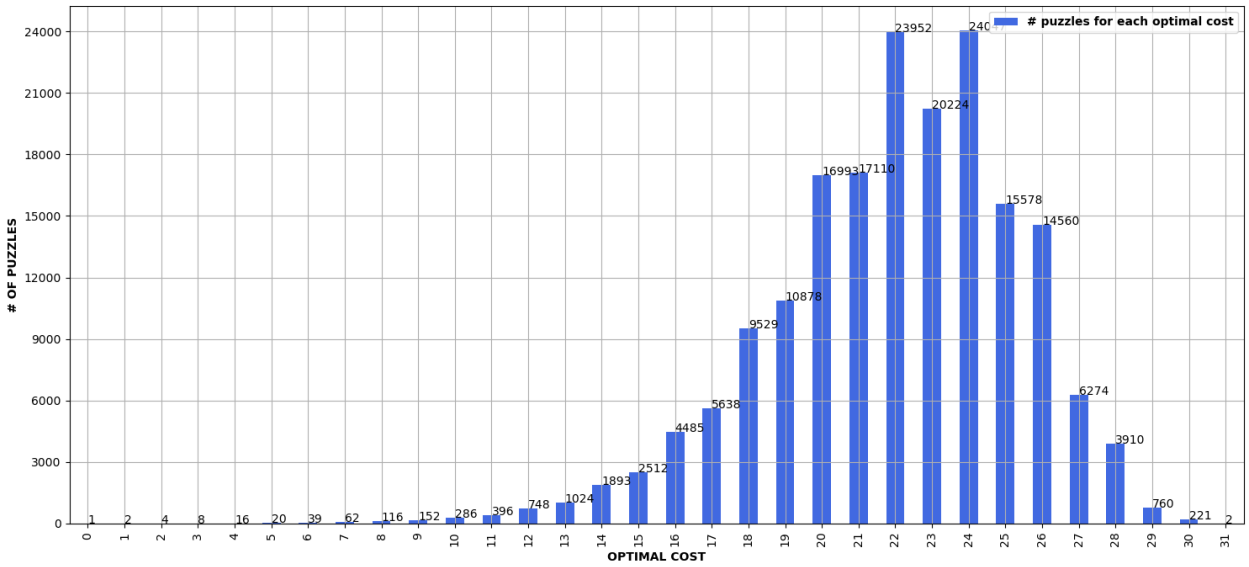


Figure 13: Perfect Learning of the 3x3 **SP**

5.2.2 Deep reinforcement learner

Next I have run the DeepReinforcementLearner with the following main parameters:

- Update the target network every 1,000 epochs (*update_target_network_frequency=1000*) or when the MSE falls below 10 basis points of the maximum target cost-to-go (*update_target_network_threshold=1e-3*)
- Run for a maximum of 11,000 epochs (*nb_epochs=11000*) or when the maximum target so far has not been increasing by more than 1% in the last 5,500 epochs (*max_target_uptick=1e-2* and *max_target_not_increasing_epochs_pct=0.5*)
- At every update of the target network (*training_data_every_epoch=False*), generate 100 sequences of puzzles (*nb_sequences=100*), each comprised of a 50-scramble puzzle back to goal state, with all intermediary puzzles along the path (*nb_shuffles=50*).
- Use a fully connected network (*network_type=fully_connected_net*) with three hidden layers of size 600, 300, 100 (*layers_description=(600,300,100)*) and use

torch.optim.RMSProp as optimiser (*optimiser=rms_prop*) with an exponential scheduler (*scheduler=exponential_scheduler*) with gamma 0.999 (*gamma_scheduler=0.9999*).

Below I show some of the DeepReinforcementLearner's quantities tracked over the epochs (see 14). I have shown only the following quantities:

- learning rate, which decreases due to the scheduler, but gets reset at each network update.
- max target. One interesting dynamic during learning is that due to the way targets are constructed using value-iteration update, and starting from an all 0 network, the max target only increases by about 1 at best at every network update. That is, the first network only learns how to differentiate goal from non-goal, then the second one learns how to differentiate goals, cost 1 and cost 2+ in some sense. This is why I have introduced the first set of parameters discussed just above. The first one makes sure we do not waste time initially training the left-hand-side network when training is easy (as consist of just goal versus not goal). The second ones make sure that we stop training altogether when the max target does not grow anymore. This is because we do not know a-priori what the God number of a given puzzle/dimension is, but let's assume for the sake of the argument that it is 15. Then that means that after 15 updates of the target network we might not really get so much benefit by keeping training and value-iterating.
- the MSE loss
- the MSE loss divided by the max target. The exit criterion *update_target_network_threshold* is based on that quantity, since a loss of e.g. 0.01 means little if the we do not compare it to the possible range that the cost-to-go can take.
- The percentage of all the possible puzzles of that dimension *seen* by the DeepReinforcementLearner. This is an interesting quantity, as it tells us later on when testing out of sample and comparing with other solvers, how well the solvers are able to generalise. In this case (3x3) we can see that the DeepReinforcementLearner quickly has seen a very large percentage of all puzzles. That being said, it is good to keep in mind that *seen* does not mean really much here, since the learning is unsupervised. This is quite in contrast to the DeepLearner, for which puzzles seen come along with the cost-to-go computed by a *teacher*, usually an optimal or at least efficient solver.

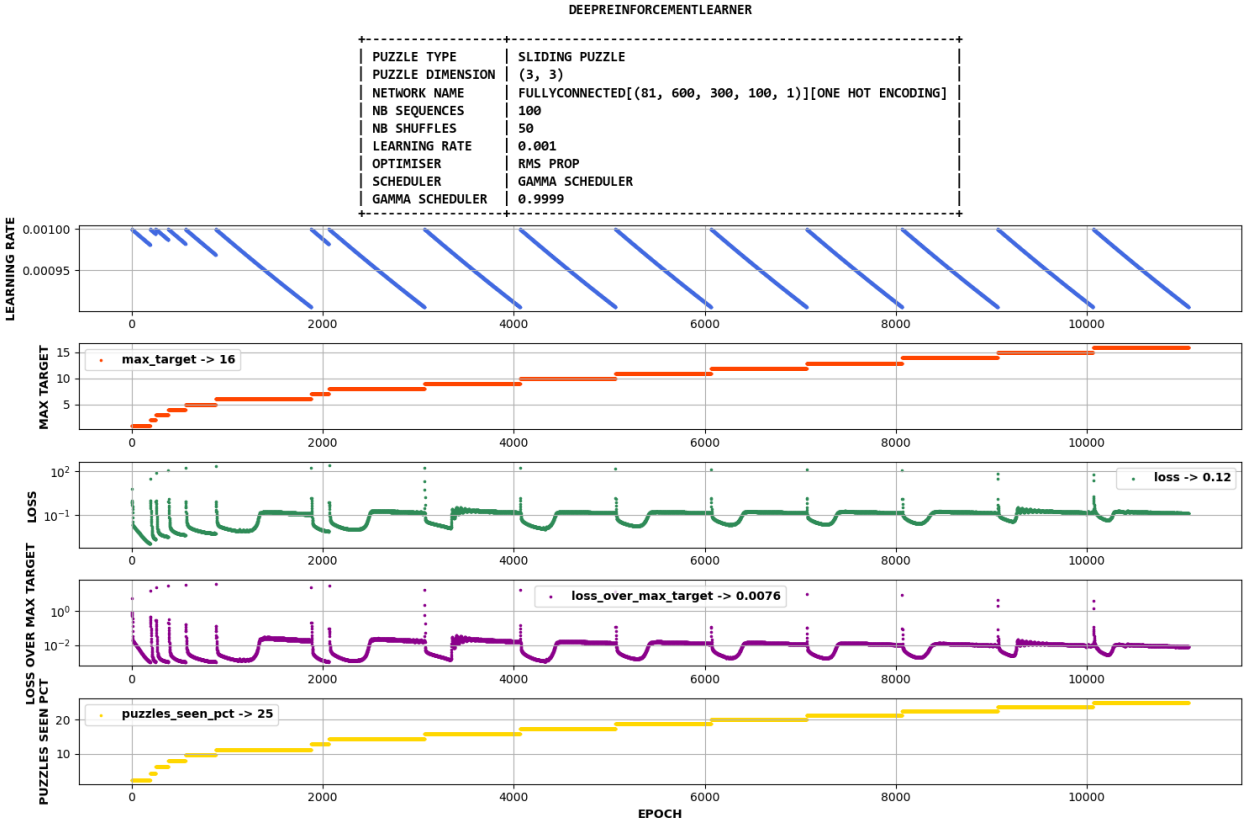


Figure 14: **DRL** of the 3x3 **SP**

Notice that the DeepLearner and DeepQLearner have very similar sets of parameters and track similar metrics during their learning for later display, but I shall not detail it here since there is not much extra insight to be had compared to the DeepReinforcementLearning example we just went through.

5.2.3 MCTS DQL

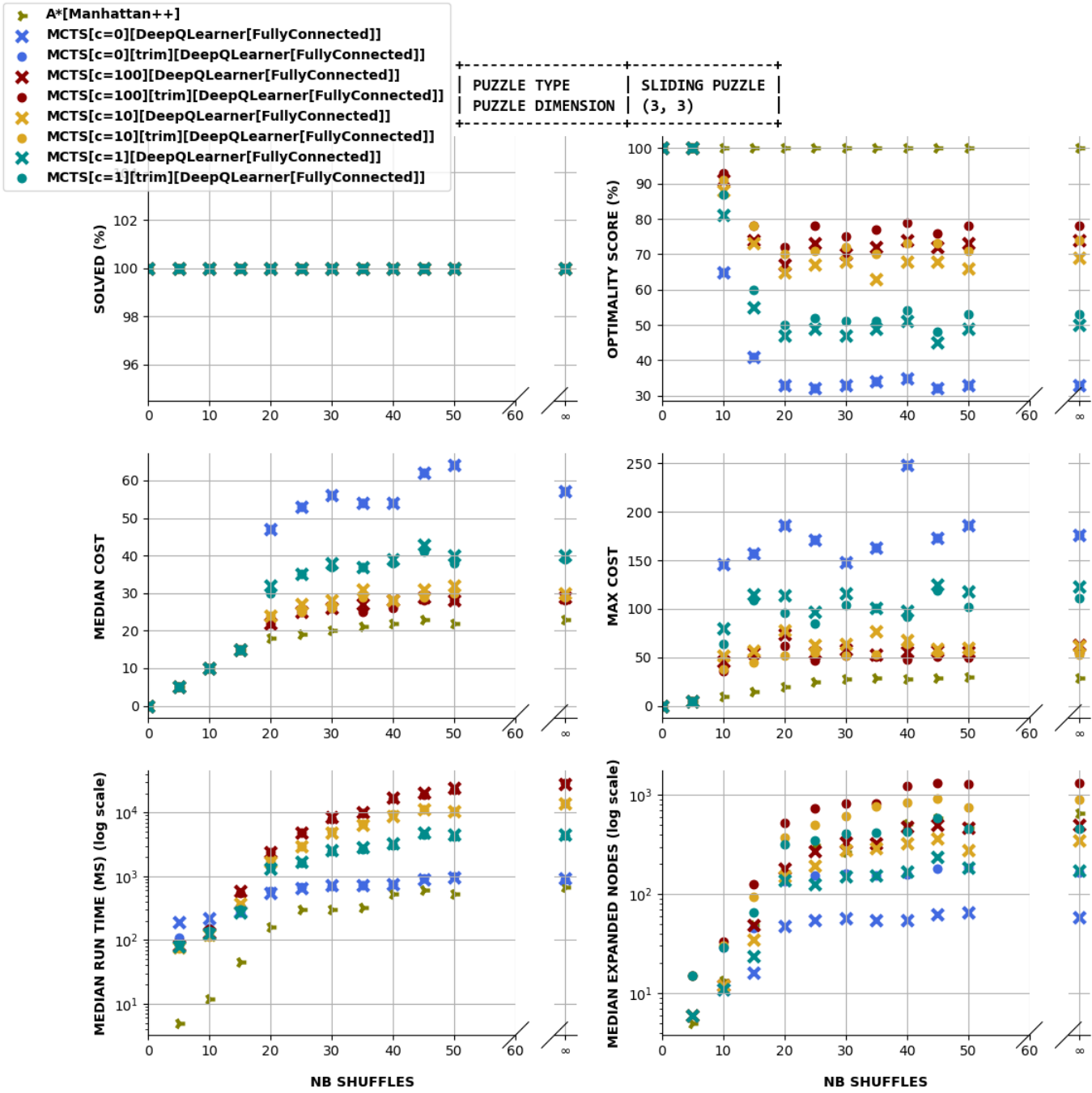


Figure 15: **MCTS** tuning 3x3 **SP**

5.2.4 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 16
DL: 100 seq 15 to 31 shuffles 10k epochs connected 600 300 100
Manhattan vs ++
DQL vs DRL
DxL vs Perfect
Put table for nb_shuffle=inf and highlight best in diff categories (cost, optimality score, run time, nodes)
say how much of the puzzles the solvers have seen

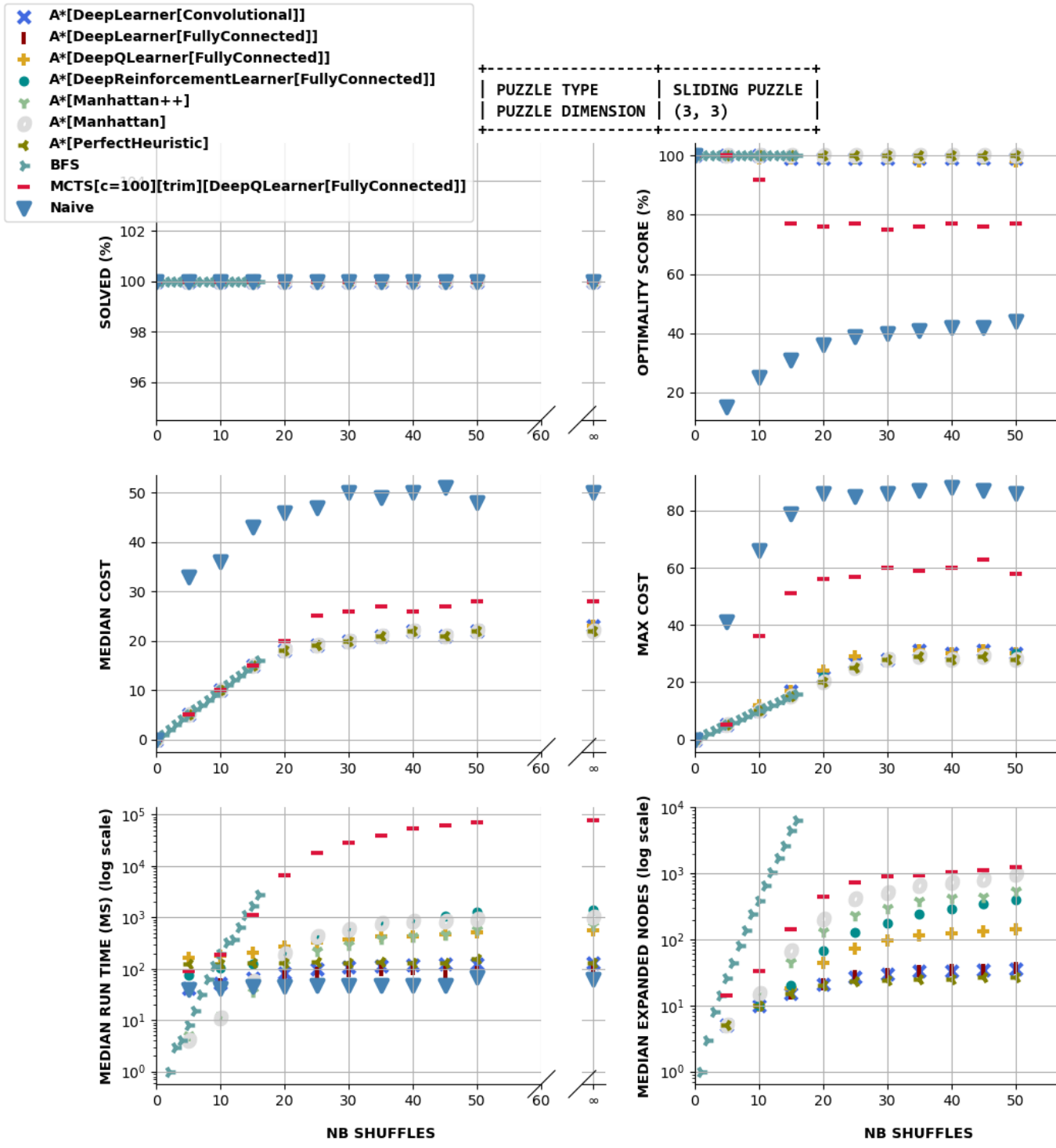


Figure 16: Solvers' performance comparison 3x3 **SP**

5.2.5 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[Manhattan++]	31	34,224	7,080
AStarSolver[PerfectHeuristic]	31	1,585	202
AStarSolver[DRLHeuristic]	31	101	58
MCTSSolver[DQLHeuristic][c=0]	101	103	456
MCTSSolver[DQLHeuristic][c=69]	35	2,244	8,873
BFS	-	-	time out
NaiveSlidingSolver	61	n/a	18

On this specific configuration, **BFS** was unable to complete before the time-out of sixty seconds. In an implementation without duplicate pruning, **BFS** would time out no matter what the bound is set. Indeed, it would need to explore in the order of 3^{31} - roughly 617 trillions - nodes to reach the goal! Even with my implementation which does pruning, it would need to pretty much traverse the entire 3x3 **SP** tree.
Rather interestingly, the **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 5.2.4, it is not the case on average.
MCTS with $c = 0$ performs rather poorly, finding the longest solution (even than my Naive solver). It does behave a bit like **DFS** when c is very small, expect the direction of travel is a bit more informed. I increased c, and for values over 69, it always gave me a solution of cost 35, which is not bad at all.
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

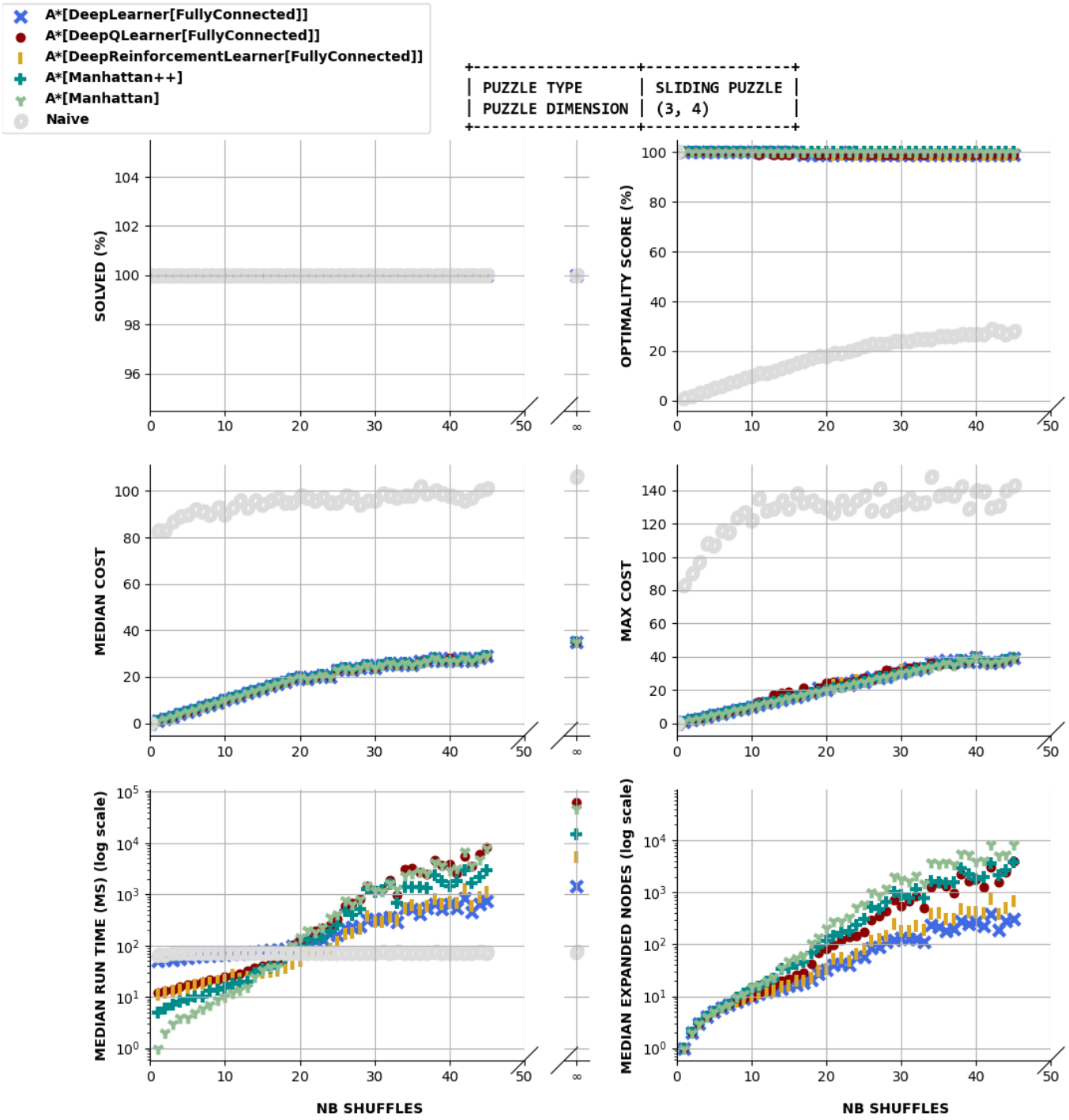


Figure 17: Solvers' performance comparison 3x4 **SP**

5.4 4x4

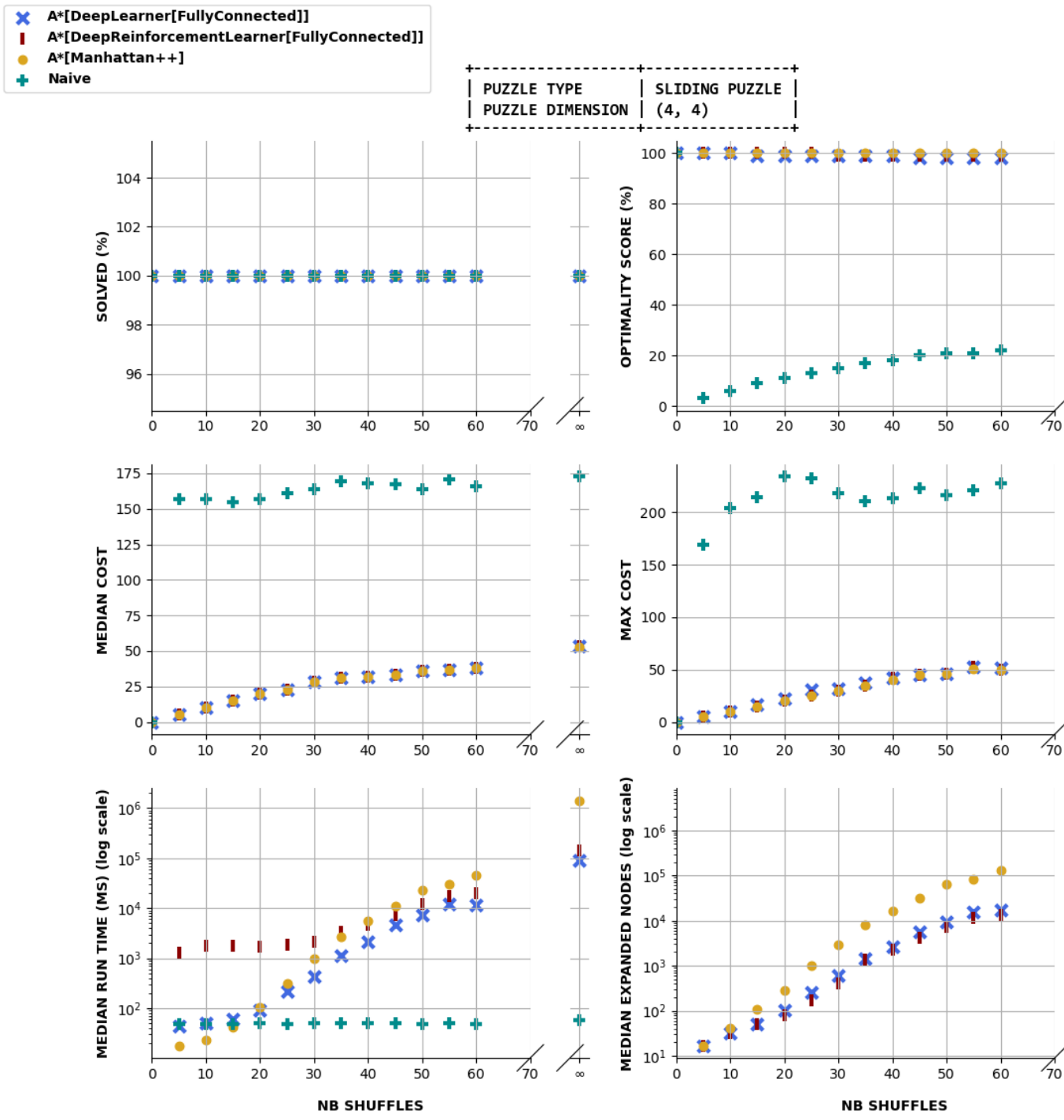


Figure 18: Solvers' performance comparison 4x4 **SP**

6 Results - Rubik's Cube

solver	Rubik's	2x2x2	3x3x3
BFS		x	
Kociemba		x	
A*[Kociemba]		x	
A*[DL[A*[Kociemba]]]		x	
A*[DRL]		x	
A*[DQL]		x	
MCTS[DQL][c=0]		x	
MCTS[DQL][c=69]		x	

6.1 2x2x2

1000 seq 30 shuffles training each target network update seen 1.1% of all puzzles 199 of the 200 perfectly shuffled got solved well under 1h (median 83 seconds), only one outlier took about 67 minutes.

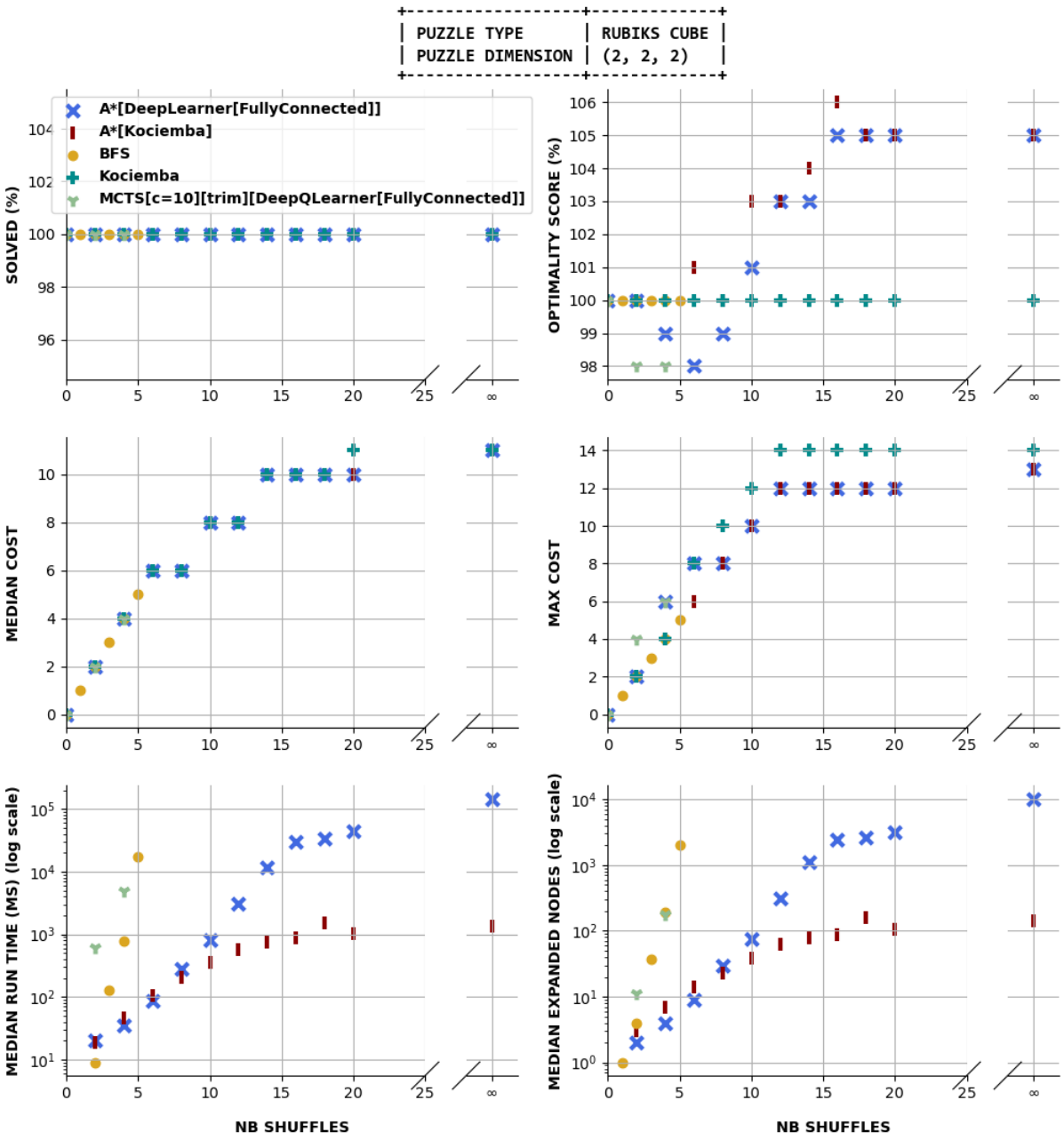


Figure 19: Solvers' performance comparison 2x2x2 **RC**

6.2 3x3x3

For now just showing Kociemba as a baseline. Will first try out DL trained on data generated from Kociemba, then will try out DRL. If unsuccessful or too slow, might try

similar to paper using DQL/MC search.
Notice Kociemba 3, unlike the 2x2x2 implementation that I found on github, does not do automatic color mapping. So if you present it with a cube which is not with the *standard* centers (facing red and white up), it will actually not solve the cube. I have therefore added a bit of logic in Kociemba solver to look for the equivalent cube among the 24 equivalent cubes to the one being solved, with standard colors. We then solve this one using Kociemba and add the whole cube rotations to the solution (which in my code are deemed to have 0 cost anyway). This way, not only can I use Kociemba 3 to solve any (solvable) 3x3x3 cube irrespective of rotations, but I can also use that in A* or to train a DL network (also for A*)

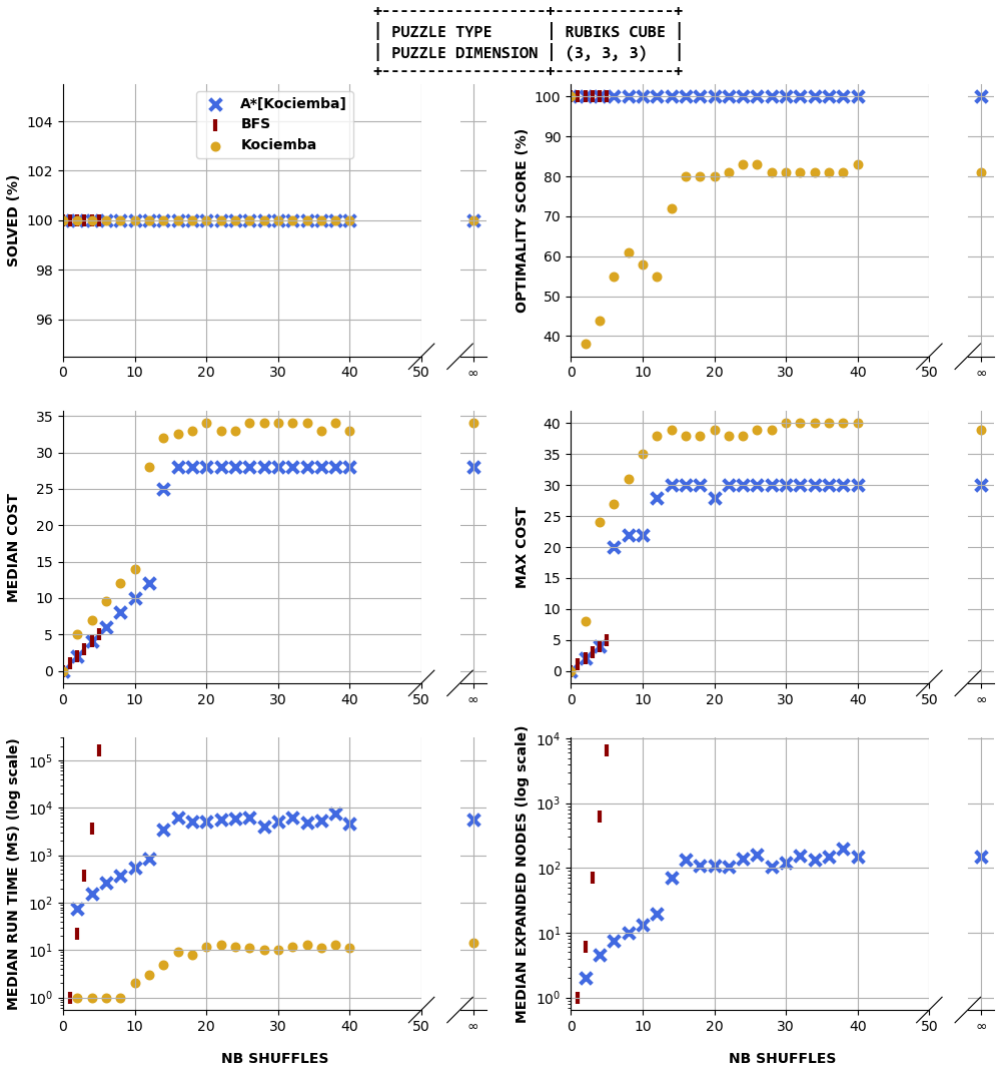


Figure 20: Solvers' performance comparison 3x3x3 **RC**

7 Conclusion & Professional Issues

7.1 Learning

Learnt about puzzles, and also even took the time to learn a couple of sequences of moves that are sufficient to solve **RC**, 2x2x2 in labout one minute, 3x3x3 in about 2 minutes.

7.2 Implementation

Implemented a bunch of solvers. Ideally should do in C++ for speed and multithreaded (esp for MCTS) but time was limited so I prioritized breadth over depth.

7.3 Deep Reinforcement/Q Learning

Quite impressive, without too much effort, how well they perform. However, I feel to really get more out of them (higher dimensions in particular) would need careful tuning of the convergence criteria.

7.4 Professional issues

See Hugh Everrett, 1957 See Bostrom, 2019 and Bostrom, 2014

References

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.

Bostrom, Nick (2019). "The Vulnerable World Hypothesis". In: *Global Policy* 10.4, pp. 455–476. DOI: https://doi.org/10.1111/1758-5899.12718. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1758-5899.12718. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1758-5899.12718.

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

Bostrom, Nick (2014). *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press.

Goodfellow, Ian J., Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. Cambridge, MA, USA: MIT Press.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley professional computing series. Addison-Wesley. ISBN: 9780321334879. URL: https://books.google.co.uk/books?id=eQq9AQAAQBAJ.

Minsky, Marvin Lee and Seymour Papert, eds. (1969). *Perceptrons: an introduction to computational geometry*. Partly reprinted in **shavlik90**. Cambridge, MA: MIT Press.
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.
Herbert Kociemba (2022). *2x2 Kociemba Python*. github. Accessed: 2022-06-22.
Hugh Everrett (1957). *The Many-Worlds Interpretation of Quantum Mechanics*. Many-Worlds. Accessed: 2022-09-02.
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.
Tomas Rokicki and Morley Davidson (2022). *God's Number is 26 in the Quarter-Turn Metric*. GodNumber26. Accessed: 2022-08-22.
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.

8 Appendix - Examples

In this appendix, I will go through many examples, illustrating how to use the code base to create, learn and solve various puzzles. For each section, a snippet 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 Python IDE.

8.1 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, simply specifying *puzzle_type* and the expected parameters to construct a puzzle. The factories will just pick up the relevant parameters indicating the puzzle type and dimension and everything should work seamlessly.

For instance, let us create a 5x6 **SP**, shuffle it a thousand times, and print it:

python code – sliding puzzle construction

```
########################################################################
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):

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

Figure 21: scrambled 5x6 **SP** example

.. similarly to construct a perfectly scrambled 2x2x2 **RC**:

python code – Rubik's 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))
########################################################################
```

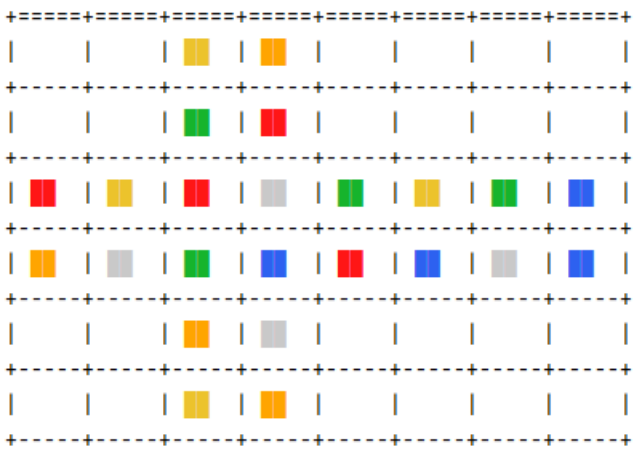


Figure 22: rubiks cube construction example

8.2 Learners

8.2.1 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 2x3 **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
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 solves the 2x3 **SP** and then displays the results, showing the distribution of optimal costs, as well as the most difficult puzzle:

```
+--------------------+--------------------+
| PUZZLE TYPE        | SLIDING PUZZLE     |
| DIMENSION          | (2, 3)             |
| # POSSIBLE PUZZLES | 360                |
| # SOLVED PUZZLES   | 360                |
| MAX COST           | 21                 |
| HARDEST PUZZLE     | +=====+=====+=====+ |
|                    | |   4 |   5 |     | |
|                    | +-----+-----+-----+ |
|                    | |   1 |   2 |   3 | |
|                    | +-----+-----+-----+ |
+--------------------+--------------------+
```

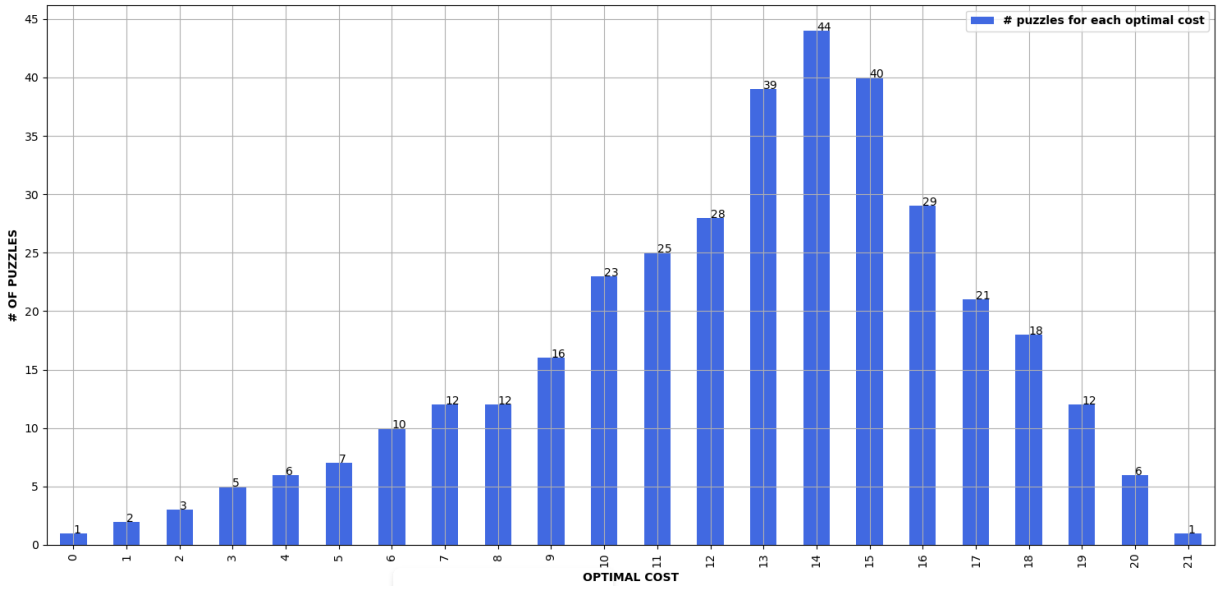


Figure 23: perfect learner example

8.2.2 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 5x2 **SP** via A* with Manhattan++.

python code – deep learner

```
########################################################################
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.

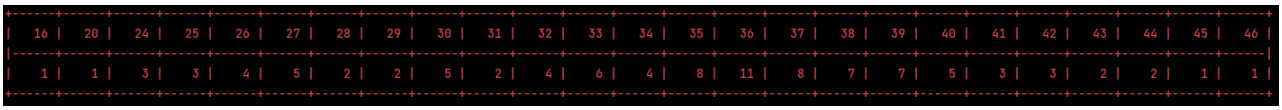


Figure 24: deep learner training example

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.

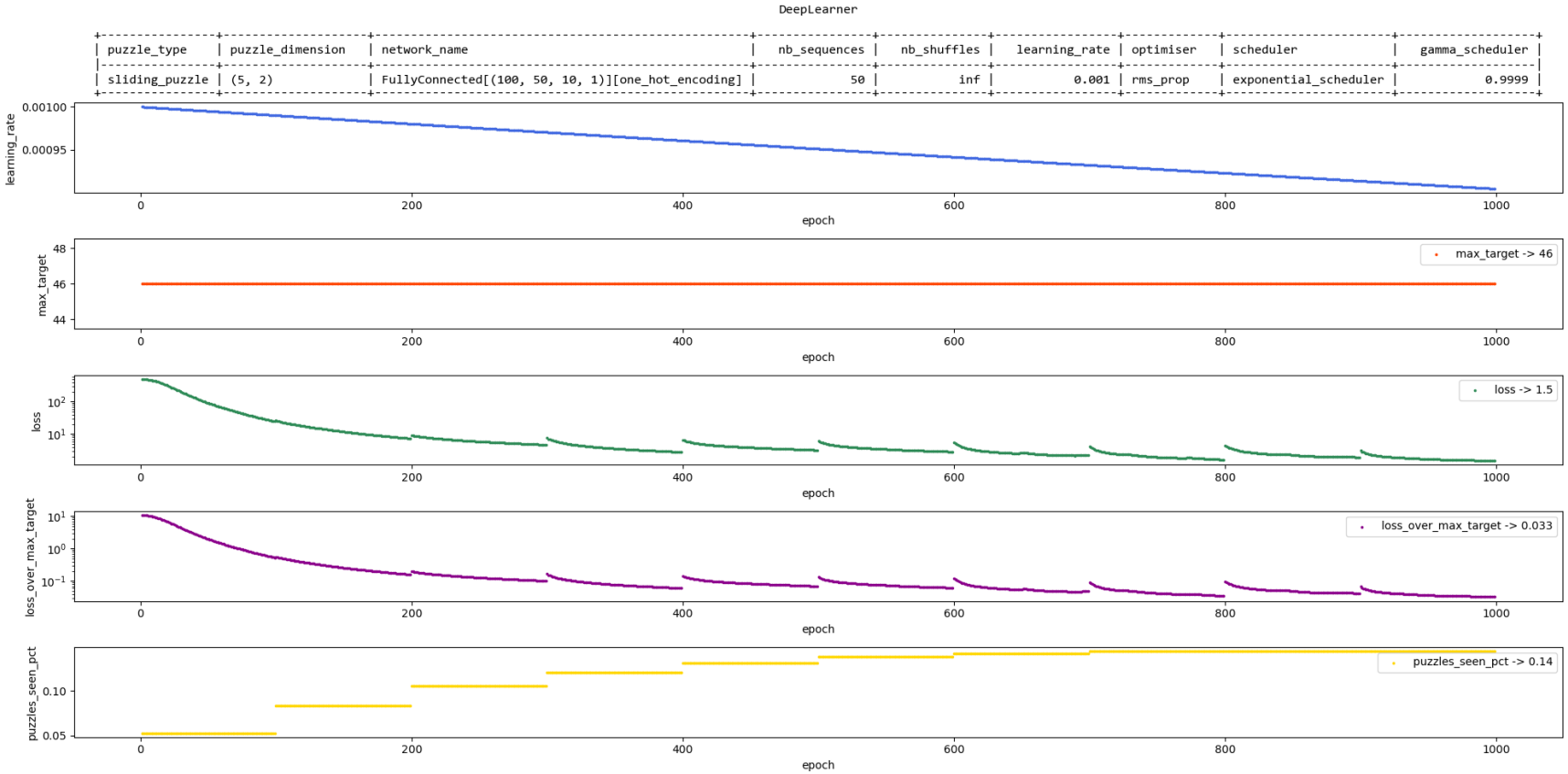


Figure 25: deep learner learning data example

8.2.3 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

```
########################################################################
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 5.1). It is therefore likely that the resulting network would not produce very optimal solutions for puzzles whose cost is in the region [25, 36].

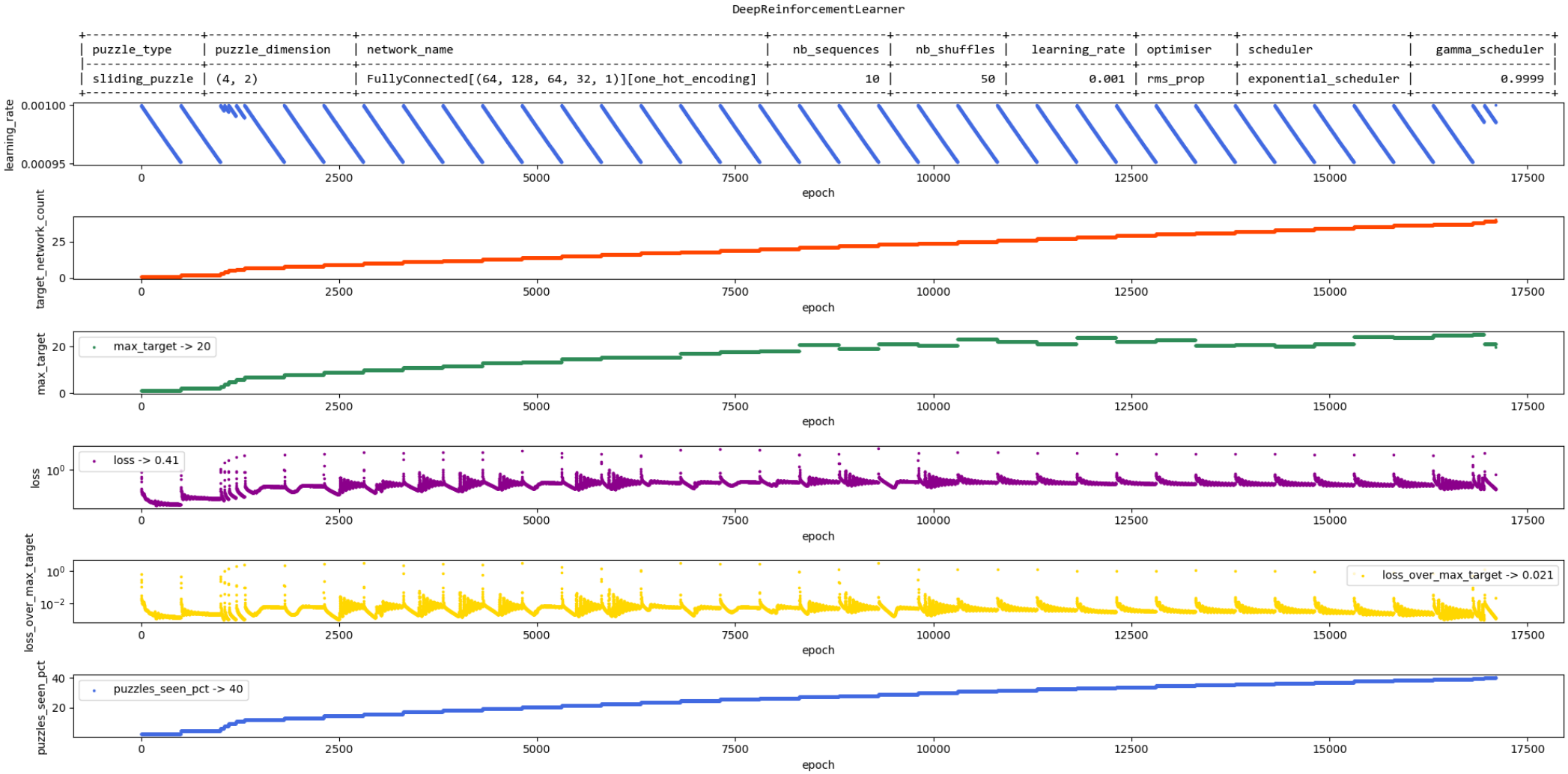


Figure 26: deep reinforcement learner learning data example

8.3 Solvers

8.3.1 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

```
########################################################################
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.

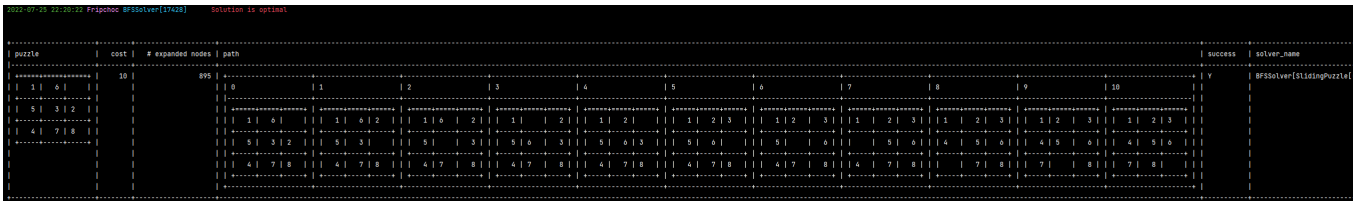


Figure 27: breadth first search solver example

DFS

python code – depth first search solver

```
########################################################################
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).

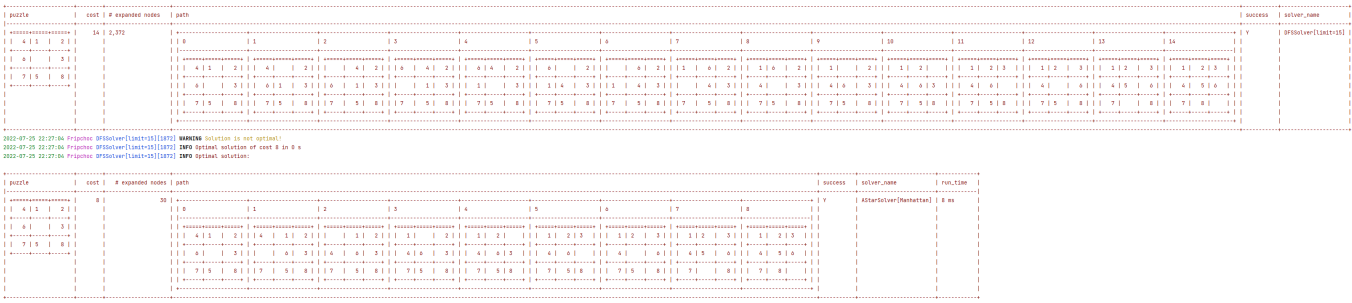


Figure 28: depth first search solver example

8.3.2 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 undoing 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

```
################################################
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())
################################################
```

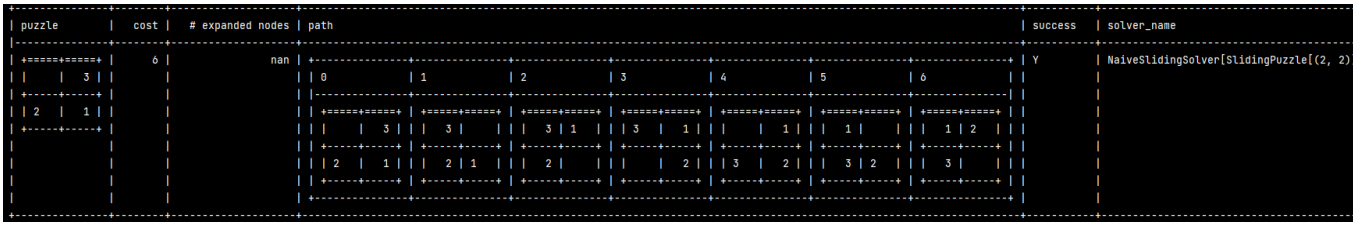


Figure 29: naive solver example

8.3.3 Kociemba

python code – Kociemba solver

```
#####################################################################
from rubiks.puzzle.puzzle import Puzzle
from rubiks.solvers.solver import Solver
#####################################################################
puzzle_type=Puzzle.rubiks_cube
n=2
cube=Puzzle.factory(**globals()).apply_random_moves(2)
solver_type=Solver.kociemba
solver = Solver.factory(**globals())
print(solver.solve(cube))
#####################################################################
```

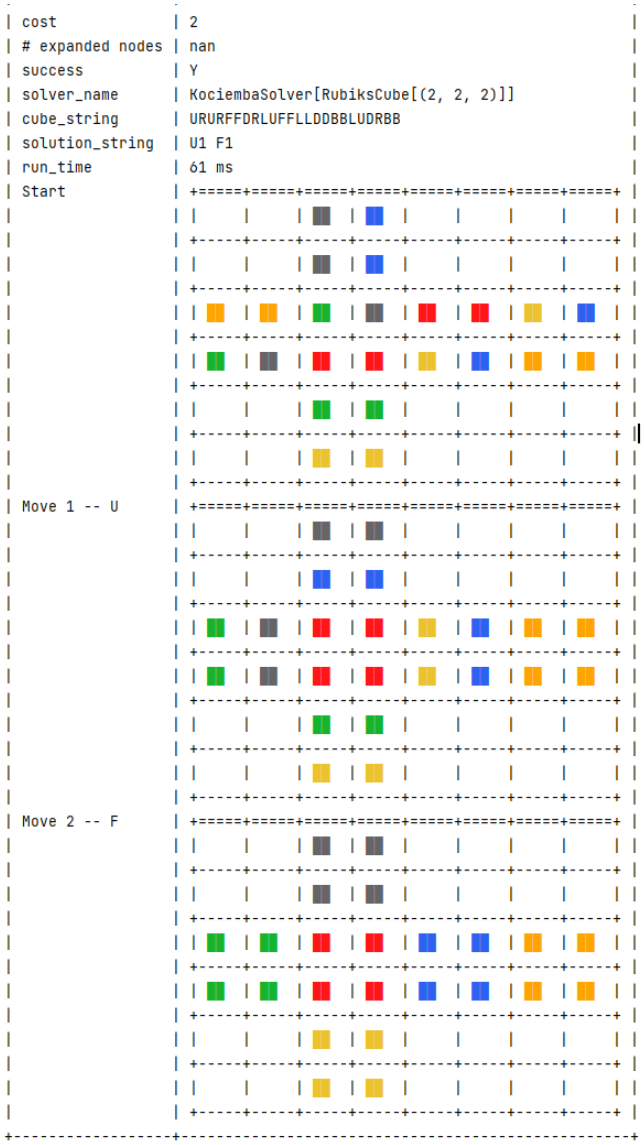


Figure 30: Kociemba solver example

8.3.4 A*

Manhattan heuristic

python code – depth first search solver

```
#######################################################################
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 31: 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 32: 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 8.2.1).

python code – depth first search solver

```
#####################################################################
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 33: a* perfect solver example

Deep Learning Heuristic

Deep Reinforcement Learning Heuristic

Deep Q Learning Heuristic

9 Appendix - Kociemba Bug

Let me discuss in this appendix in more details the issue I mentioned in section 6.2 regarding the *poor* interface of the kociemba (3x3x3) library.

I personally like to follow the great Scott Meyers' advice (see item 18 of Meyers, 2005), which is to not merely make interfaces easy to use, but also difficult to misuse. In that instance, I would argue they have made it very easy to shoot oneself in the foot with their interface which assumes the cubes passed to the solver are in *standard* form (my terminology), not only without making that explicit anywhere, but worse than that: sometimes non standard initial configuration raise an exception, sometimes not and returning a solution which is not really one! So what is the problem exactly?

To start with, let us remember from chapter 3 that each **RC** configuration has 24 equivalent configurations under invariance by full rotation of the cube in space. Out of 24 equivalent configurations, and for an observer facing the cube, there is only one whose center front (F) cubie is red and center up (U) cubie is white.

In hkociemba (the third parties 2x2x2 implementation which I used under the hood of my KociembaSolver), the problem of these equivalent puzzles is irrelevant (since full cube rotation can always be achieved by 2 rotations of 2 opposite faces in opposite directions, e.g. FB' is equivalent to full cube rotation around F (CF in my jargon)). That being said, the hkociemba is smarter than just using the equivalence between full cube rotation and two faces' rotations. It does automatic color scheme recognition when passed a *cube string*, and gets a smart solution given that coloring scheme. For instance, if we pass it a solved configuration, but not in standard form, it knows there is no need to do anything. In the general case of a scrambled cube, it will also endeavour to find solutions to the closest among the 24 equivalent solved configurations. Let's look at the following concerete example:

Kociemba bug – 2x2x2 RC – example of good interface design

```
#####################################################################
from rubiks.puzzle.puzzle import Puzzle
from rubiks.solvers.solver import Solver
from rubiks.solvers.kociembasolver import KociembaSolver
from_kociemba = KociembaSolver.from_kociemba
to_kociemba = KociembaSolver.to_kociemba
#####################################################################
puzzle_type = Puzzle.rubiks_cube
n=2
init_from_random_goal=False
cube = Puzzle.factory(**globals()).get_equivalent()[-1].
                                    apply_random_moves(nb_moves=1)
solver_type = Solver.kociemba
solver = Solver.factory(**globals())
print(solver.solve(cube))
```

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

which, subject to randomness, gave me the following scrambled cube and solution:

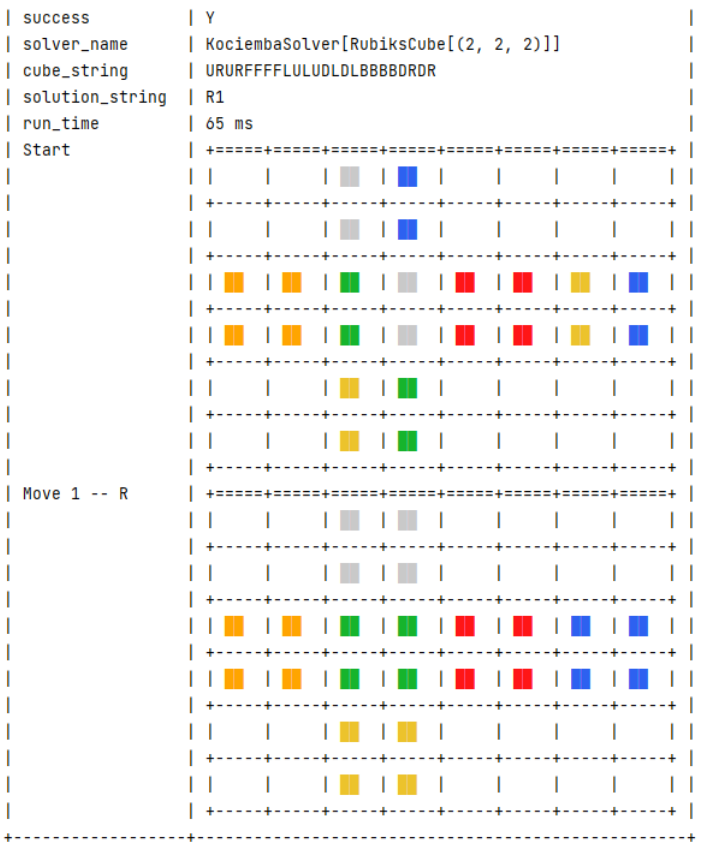


Figure 34: 2x2x2 **RC** – hkociemba finds path to closest of the 24 equivalent goals

As we can see, hkociemba happily returned a 1-move solution R to green F, white U, instead of the costlier solution it would have taken to get to *standard* form red F white U (clearly at a cost of 3 with solution RUD').

In the 3x3x3 case, things are not quite that simple. There is no way to reproduce full cube rotation from faces rotations alone, since the center cubies will never move via the latter, but do via the former. If we run the equivalent of the above problem with the (3x3x3) kociemba library, we silently get to a false solution as is apparent from printing the resulting cube.

Kociemba bug – 3x3x3 RC – example of bad interface design

```
########################################################################
from kociemba import solve
```

```
########################################################################
from rubiks.puzzle.puzzle import Puzzle
from rubiks.solvers.kociembasolver import KociembaSolver
from_kociemba = KociembaSolver.from_kociemba
to_kociemba = KociembaSolver.to_kociemba
########################################################################
puzzle_type = Puzzle.rubiks_cube
n=3
init_from_random_goal=False
cube = Puzzle.factory(**globals()).get_equivalent()[-1].
                                    apply_random_moves(nb_moves=1)
print(cube)
cube_string = to_kociemba(cube)
solution = solve(cube_string)
print('Kociemba Solution:', solution)
print(cube.apply_moves(from_kociemba(solution)))
########################################################################
```

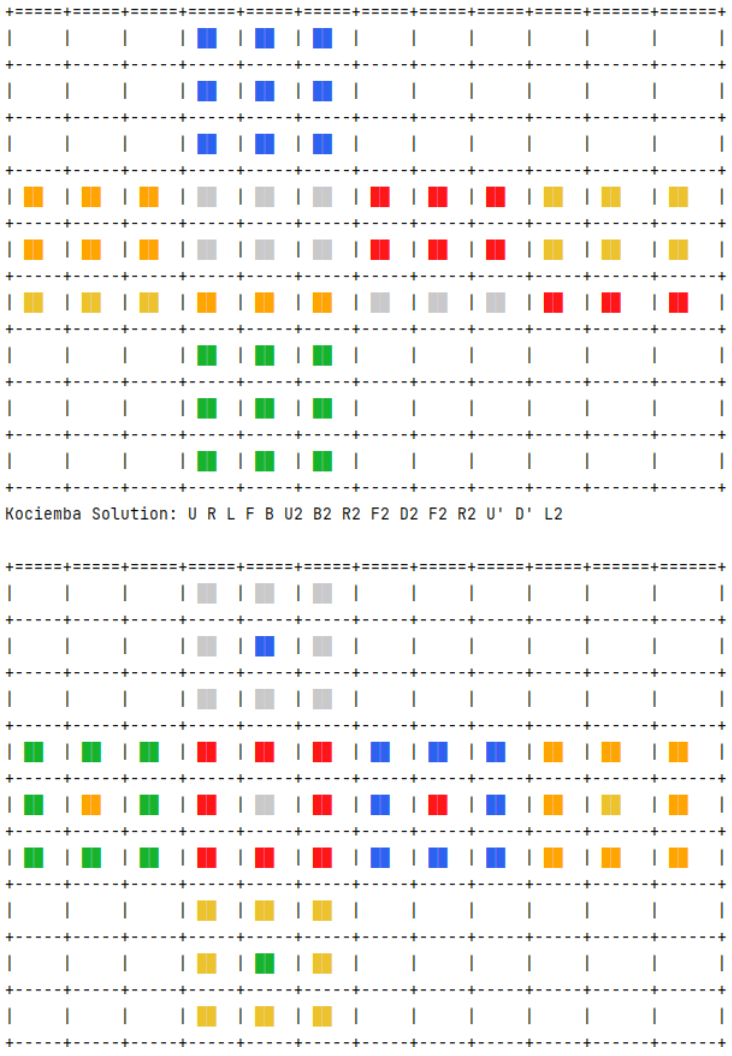
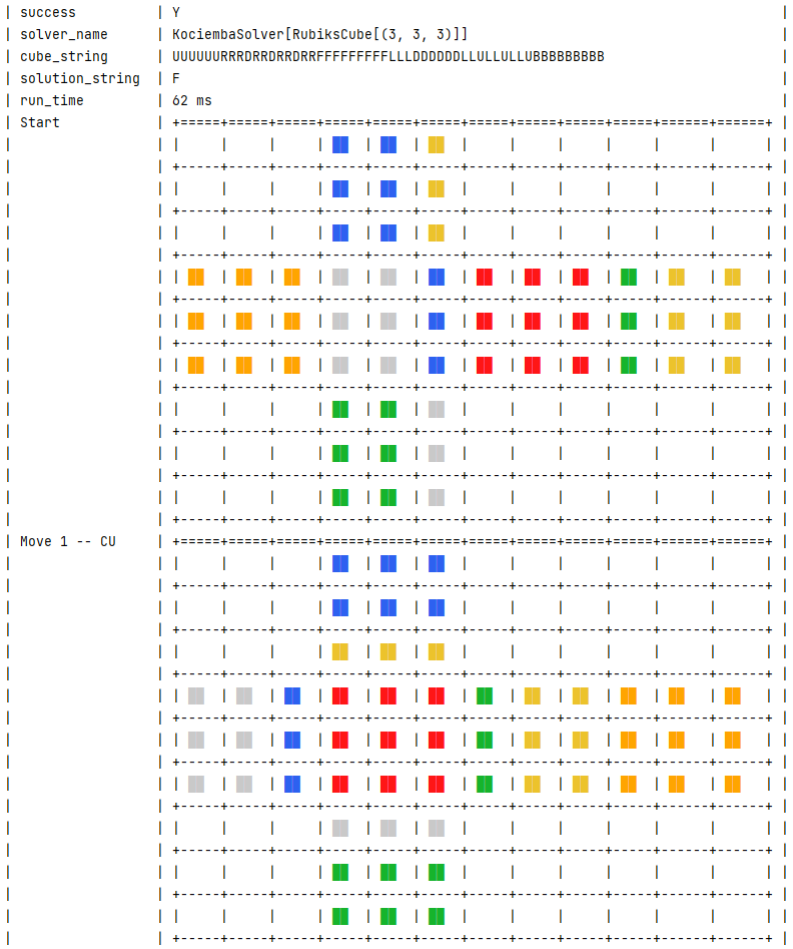


Figure 35: 3x3x3 **RC** – kociemba does not check center cubies and returns wrong solution

In order to counteract that, and make the comparisons between Kociemba 3x3x3 and other solvers and methods fairer, I have had to implement a layer on top of kociemba. My Kociemba solver basically will check if a cube is in *standard* form, and if not, it will find the sequence of full cube rotations that bring us to an equivalent configuration in *standard* form, pass that to equivalent *standard* cube to kociemba, and stich the cube rotation to the kociemba solution (treating cube rotation as 0 cost everywhere in the code). That way, we seamlessly get to an answer that makes sense from kociemba, without the user having to worry about whether or not the cube they are passing as input is in *standard* form or not! Running the same example as above via my KociembaSolver, we get the expected result, composed of 3 cube rotations to bring it to *standard* form, followed by one face rotation to solve it:

Kociemba bug – 3x3x3 RC – massage to fix the broken interface

```
#######################################################################
from rubiks.puzzle.puzzle import Puzzle
from rubiks.solvers.solver import Solver
from rubiks.solvers.kociembasolver import KociembaSolver
from_kociemba = KociembaSolver.from_kociemba
to_kociemba = KociembaSolver.to_kociemba
#######################################################################
puzzle_type = Puzzle.rubiks_cube
n=3
init_from_random_goal=False
cube = Puzzle.factory(**globals()).get_equivalent()[-1].
                                    apply_random_moves(nb_moves=1)
solver_type = Solver.kociemba
solver = Solver.factory(**globals())
print(solver.solve(cube))
#######################################################################
```



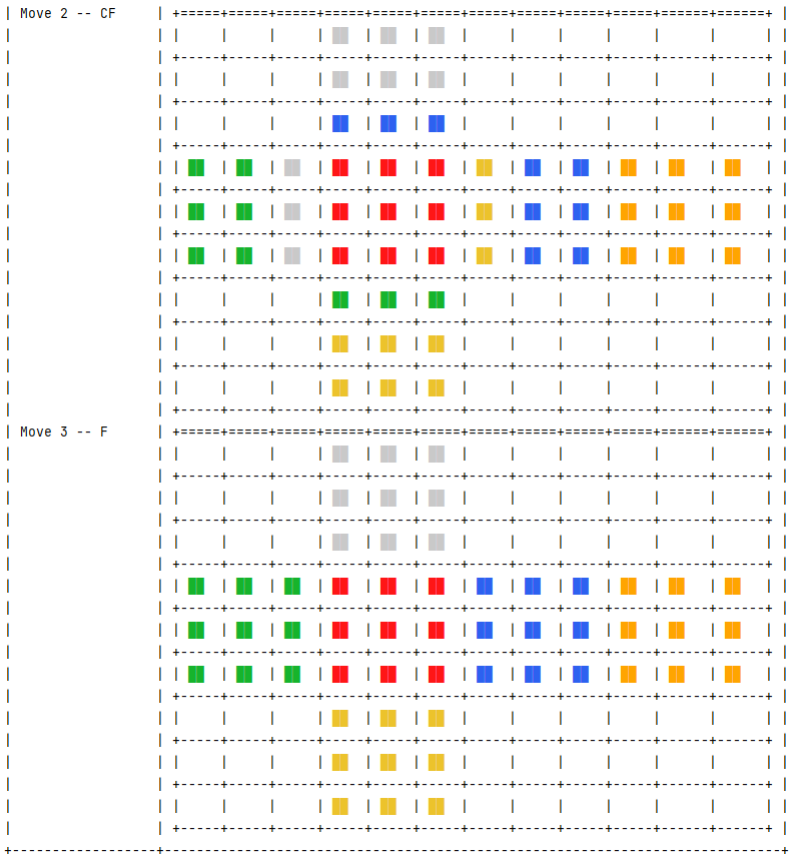


Figure 36: 3x3x3 **RC** – Fixed kociemba interface