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 7, 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 **??** 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 **??**.

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 **??**.

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 **??**) 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 **??** for details).

References

Journal Papers

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.

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

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

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

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

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

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