Modeling, Identification and Control, Vol. 42, No. 4, 2021, pp. 197–204, ISSN 1890–1328

Model-Free All-Source-All-Destination Learning as a Model for Biological Reactive Control

M. Knudsen[1] S. Hendseth[1] G. Tufte[2] A. Sandvig[3]

[1]*Department of Engineering Cybernetics, Norwegian University of Science and Technology, N-7491 Trondheim, Norway. E-mail:* {Martinius.Knudsen,Sverre.Hendseth}@ntnu.no

[2]*Department of Computer Science, Norwegian University of Science and Technology, N-7491 Trondheim, Norway. E-mail:* {Gunnar.Tufte}@ntnu.no

[3]*Department of Neuromedicine and Movement Science, Norwegian University of Science and Technology, N-7491 Trondheim, Norway. E-mail:* {Axel.Sandvig}@ntnu.no

Abstract

We present here a model-free method for learning actions that lead to an all-source-all-destination shortest path solution. We motivate our approach in the context of biological learning for reactive control. Our method involves an agent exploring an unknown world with the objective of learning how to get from any starting state to any goal state in shortest time without having to run a path planning algorithm for each new goal selection. Using concepts of Lyapunov functions and Bellman's principle of optimality, our agent learns universal state-goal distances and best actions that solve this problem.

Keywords: Control, Learning, Shortest path

1 Introduction

We are all naturally born explorers. If a child is not sleeping or feeding, they are most likely moving. In seemingly sporadic ways the baby moves arms and legs without much goal or intent. Movement provides valuable sensory data streams which is essential in developing the child's sensory motor skills and their knowledge of their bodies state-space. The brain is in fact constantly bombarded with some 11 million bits per second of sensory feedback Britannica (2020), to provide the brain with data to keep track of the body's state as well as information from the surrounding environment. All this data enables one of the brain's core functions; to process this input stream and apply motor actions in accordance to the individuals needs and objectives. Exactly how the brain achieves this feat of learning control policies is still an active research topic. Better understanding in this area may contribute greatly towards novel methods in AI. Such knowledge could also facilitate rehabilitation of patients with stroke, spinal cord and traumatic brain injuries as well as restore neurological functions to a level which is currently not possible.

Control can largely be divided into *prediction* and *reaction*. Prediction involves forward simulation, which estimates future outcomes given an understanding of the system dynamics and the effect of actions, which together comprise a *model* of the system. Reactive control requires no forward prediction, a property which allows reaction to be computationally more efficient than prediction. In addition, reaction can be accomplished even without a model, as it only requires a mapping from state to action. In control engineering, a highly successful predictive method is that of Model-Predictive Control (MPC) Camacho and Alba (2013). MPC has the ability to anticipate future events and thereby take control actions according to the desired outcome Camacho and Alba (2013). While design of conventional controllers typically requires a system/-plant model in order to design a state-action mapping, intelligent control methods such as reinforcement learning (RL) learn mappings through exploration and experimentation of the environment Sutton and Barto (2018). It is likely that the brain utilizes both model-based predictive control and model-free reactive control. Either for different types of tasks or for different stages of the same task. An example of the latter is learning to juggle: at first the beginner uses intrinsic knowledge of physics to predict where the ball will land and how hard to toss the ball into the air. At this stage many mistakes are made and the mental load is high. As the individual becomes more skilled, mental load decreases, even though precision increases. This effect can be observed when comparing EEG signals of beginners and experts Schiavone et al. (2015). It would seem the practitioner moves from more predictive based control to reaction. As such, many experts often cite their reactive behavior as *intuition* Izquierdo-Torres and Di Paolo (2005). This is interesting from a biological point of view as the brain would continuously strive to solve its computational problems at hand as cost efficient as possible using the minimum energy to solve the problem/task.

Inspired by the brains ability to learn through exploration Roussou (2004), we ask how we can implement more efficient learning by utilizing the feedback we get about every state-action transition from the environment. In this paper, instead of defining a single goal upfront, we treat every state we observe as a potential goal. This way, should our goal be redefined, we already have the knowledge to quickly and efficiently find our way to this newly defined goal. This requires no additional time-consuming and computationally expensive training, and also addresses the problem of catastrophic forgetting McCloskey and Cohen (1989). Our aim is to explore the environment once and by this learn how to take actions as to get to any potential goal in the environment. Using principles from Lyapunov theory and dynamic programming Bertsekas (2016), we present an algorithm for solving the all-source-all-destination (ASAD) problem in a model-free way in an unknown world. With this, we hypothesize that the brains reactive control may result from learning spatio-temporal maps of the distances between states in state-space. These maps are continuously updated as new state-space trajectories are explored and faster paths are found.

The paper is laid out as follows: Section 2 provides the background of the ASAD problem and the core principles of the method. In Section 3 we describe the environment world in which we will apply our method, as well as the method itself applied through an agent. In section 4 we present the results, and discuss these in section 5. Finally we conclude the work in section 6.

2 Background

2.1 All-pairs shortest path

The ASAD problem is essentially an all-pairs shortest path problem Deo and Pang (1984). The main difference between our method and the established methods for solving this problem is the assumption that there exists a model or encoding of the system that the algorithm can be applied to. Most algorithms will be given a known graph from the get go, while our method does not. Being model-free we also do not attempt to generate a model of the graph nor do we store information about the graph structure. In addition, most of the established algorithms, with some exceptions, primarily aim to find the shortest paths between states, and not the actual actions that need to be taken at each state in order to move towards the goal. For our context, which is to enable an agent to learn to operate in an unknown world, learning and remembering what the optimal actions are is essential.

One of the more established algorithms for solving the all-pairs problem is the Floyd-Warshall (FW) algorithm Floyd (1962). Like the Bellman-Ford algorithm Bellman (1958); Ford Jr (1956) or the Dijkstra's algorithm Dijkstra et al. (1959), it computes the shortest path in a graph. However, Bellman-Ford and Dijkstra are both single-source, shortest-path algorithms, while FW computes the shortest distances between every pair of vertices V_e in the input graph. FW accomplishes this by incrementally improving an estimate of the shortest path between two vertices, by evaluating if there exist better routes through any of the other vertices in the graph. The algorithm is able to find the optimal solution in $O(|V_e|^3)$ time.

While the original FW algorithm only finds the shortest paths between vertices in the graph, it can be extended with the possibility for path reconstruction by also saving the actions during its run. Our Python code for the extended FW algorithm can be seen in Listing 1. We will be using this algorithm for comparison with our own method in the results section 4. While the FW method requires the complete graph up front, we will show how this is not necessary in our method.

```python
def floydWarshall(graph):
  dist = np.array(graph)
  action = np.ones((Ve, Ve), dtype=int)*np.NaN
  for i in range(Ve):
    for j in range(Ve):
      if graph[i][j] != INF:
        action[i][j] = j
  for k in range(Ve): # source
```

```
    for i in range(Ve): # destination
      for j in range(Ve): # intermediate
        if dist[i][j] > dist[i][k] + dist[k][j]:
          dist[i][j] = dist[i][k] + dist[k][j]
          action[i][j] = action[i][k]
  return dist, action
```

Listing 1: Our Python code for the Floyd-Warshall algorithm with path reconstruction.

2.2 Lyapunov functions

Our method is inspired by the idea of Lyapunov functions Freeman and Primbs (1996), which states that if you have a function V that returns a positive scalar for all non-goal states x, i.e. $V(x_t) \geq 0$. and you can have this function continuously decrease while the system moves towards some goal state, i.e. $\dot{V}(x_g) \leq 0$, then you essentially have a Lyapunov function. In control theory, Lyapunov functions are often constructed using a conservation law, but any function satisfying the definition is applicable Freeman and Primbs (1996). If a state is reachable from a set of initial states, then there *must* exist a Lyapunov function for these initial states. Put more formally:

If, from a state A a goal B is reachable then, given some cost metric, there must exist at least one optimal path between A and B. If this optimal path is followed, the associated cost along this path should continuously decrease while remaining positive for all states as the cost is assumed positive.

This cost could be a Euclidean distance, a path length, the systems energy or even the *time-to-destination*. We here use this latter metric by imagining that the Lyapunov function is the number of steps it takes to traverse the optimal path between two states. This time-to-destination is always a positive value, except at the goal state where it is 0. We wish to select actions as to continuously decrease this time-to-destination, which will then satisfy the conditions of a Lyapunov function. Unlike most traditional Lyapunov-based control algorithms where one finds a control law that makes the system stable for a single given goal (ASSD) Freeman and Primbs (1996), our method finds the set of optimal actions that gets us to all other reachable states in the system.

2.3 Principle of optimal subpaths

Another closely related concept is that of Bellman's principle of optimality, which states that for any optimal path between two states, all subpaths of this path are also optimal Bellman (1952). For us, this property means that we only need to know the optimal action to take for the current state we are at, and not the whole future action trajectory as required in predictive control. This is because taking the optimal action moves us to a state along the optimal subpath, where we again can select the optimal action for this new state. Thus, we only need to store a single optimal action at each state. The path reconstruction method for the FW algorithm utilizes a similar strategy.

3 Method

Using the principles of Lyapunov functions and optimal subpaths, we find that each state only needs two pieces of information for each state-goal pair: (i) the best action to take in the current state and (ii) a value that encodes the distance between the current state and the goal. We use the state transition feedback at each timestep in the environment efficiently to update these state-goal actions and values.

3.1 The world

Many environments are discrete, such as board games, grid worlds, choices, or routing (vehicle routing problem Toth and Vigo (2002)). In reinforcement learning, discrete worlds have played an important role and arena for testing out new algorithms, especially for tabular RL methods. Almost all the powerful deep RL methods we hear about today have a tabular beginning. Similarly. the worlds in which we demonstrate our method will be discrete and finite in both state- and action-space.

3.1.1 Graph world

We shall first demonstrate our method on a hypothetical world which can be represented by a graph like the one in Figure 1. Here, each node S represents a state of the system, and each edge A represents an action. At each state all actions are available for selection. An action performed in a given state at the current timestep will result in a transition to another state or back to itself. Doing nothing is also regarded an action as the system evolves with time regardless of any input. Each transition is of step length one. This means that an action is applied at every step, and that the minimal possible path between 2 sequential states is one timestep/transition.

3.1.2 2-link manipulator

The second world we test our method on is a discrete 2-link manipulator as depicted in Figure 2. This system is comprised of two joints and accompanying motors to rotate them. The action space is thus made up of 4 possible actions; motor 1 clockwise and counter-clockwise

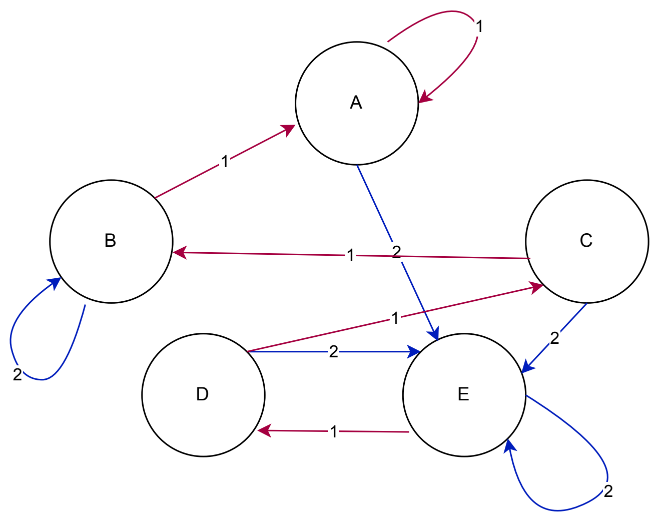


Figure 1: Example graph showing the state-action transitions. Here, the nodes are the 5 states A-E and the edges are the actions. There are 2 possible actions, action 1 (red) and action 2 (blue). The agent does not have knowledge about the transitions prior to exploration.

rotations, and similarly motor 2 rotations. The state-space is the angles of the joints, θ_1 and θ_2, and is discretized; a single rotation leads to a unique state. Low-level motor controllers can ensure the realistic implementation of such a system. Within this system, not all states are allowed as some states violate the operation boundaries either virtually (restricted manipulator configurations) or physically (e.g. obstacles in the work-space Spong et al. (2005)). In this system the goal is to move the manipulator through state-space without violating state constraints. We also want to be able to define new goals at any initial state without having to initiate path planning each time (path planning would also fall into the category of predictive control). The 2-link manipulator task is meant as an illustrative model to how animals may learn to move their limbs in a processing efficient manner, yet with great precision; a feat still quite difficult to achieve in robotics.

3.2 The agent

The agent's task is to learn how to optimally get to other states within its world. Upon initialization, the agent knows nothing about the structure of its environment, it only knows what actions it can take. The individual states can be learned or known in advance. The agent goes through two main phases: (i) an exploration phase and (ii) an exploitation phase. In the exploration phase, the agent learns the all-to-all optimal actions and the path distances, not the actual paths themselves. The agent does not learn or store the structure of the graph. The agent's objective is: *Solve the ASAD problem such that: given any state S_x, and any goal state S_g, apply the appropriate action in S_x that will take it to S_g in a minimal amount of steps/time.*

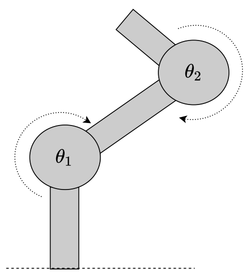


Figure 2: 2-link manipulator

Table 1: Main table for storing state-goal values and best actions. The table is empty prior to learning.

State	A	B	C	D	E
A, shortest time A, best action					
B, shortest time B, best action					
C, shortest time C, best action					
D, shortest time D, best action					
E, shortest time E, best action					

3.2.1 Storing the best actions and shortest times

For storing the best actions and shortest times, we create a table with size given by the number of states and the action-space. This table could be dynamically generated through exploration if one does not have prior information about the number of states beforehand. No information about the transitions between states is yet encoded in the table as this is to be learned. The table is an $N_s x N_s$ table where N_s is the number of system states. Each row is a state which records two values for each of the other states in the system (columns); (i) the shortest time that has currently been found from a state in row R to get to some state in column C, and (ii) the best action taken by a state in R that resulted in this shortest time to a state in C. Such a table with 5 states A-E is shown in Table 1.

Table 2: Memory table.

State	Time since state	Last action
A		
B		
C		
D		
E		

3.2.2 Memory: keeping track of previous states and actions

In addition to the above table that eventually stores the actual solution to the problem, we also require a memory table. This tables keeps track of (i) the number of timesteps since a state was last visited, and (ii) the last action performed in a state. This is illustrated in Table 2.

It is necessary for the agent to keep track of these two metrics as they are essential to comparing trajectory lengths with Table 1, and in the case of finding a shorter path, recalling the last action that was taken as to update Table 1.

3.3 Exploration and exploitation phases

3.3.1 Exploration: filling in the table

The agent first explores its environment in which it performs novel actions in each state in order to explore alternative paths between states. Updating the table is done through the following steps:

1. Say we are transitioning from state S_A to S_B. Upon arriving at S_B, we iterate through all previously visited states S_x and compare the *shortest path time* values stored for each (S_x, S_B) pair with each S_x's *time since visited* value. If the latter value is lower than the former, we update (S_x, S_B) by replacing its *shortest path time* with *time since visited*. At the same time, we also update the *shortest path action* with the *last action* performed in state S_x from memory. So, at a given time step we are not just updating a single cell, we are updating all previously visited cells, which is what makes the algorithm so efficient.

2. Select an action in S_B, and reset its associated *time since visited* counter and update the *last action* with the action currently being performed.

3. Transition to the next state and repeat step 1 and 2 for this new state.

Example exploration illustrated using graph 1: State S_C has stored that the fastest way to S_E is in 3 steps (path $S_C - S_B - S_A - S_E$), and that the action that was taken in S_C for this to occur was *action 1*. Later, again in state S_C, *action 2* is performed (due to random exploration) which gets us directly to state S_E. Now S_C updates the storage table with the new minimum steps to get to S_E (1 step) and the action taken in S_C that accomplished this (*action 2*). This update is done for all states. Upon revisiting a state, a different action from the previous action is taken to ensure exploration.

The following is our Python code for the exploration phase:

```python
def explore():
  s = npr.randint(nStates)
  for _ in range(steps):
      memory.time_since[s] = 0
      for so in range(nStates):
          if memory.time_since[so] < table.shortest_time[so, s]:
              table.shortest_time[so, s] = memory.time_since[so]
              table.action[so, s] = memory.last_action[so]
      for so in range(nStates):
          memory.time_since[so] += 1
      a = action(memory.last_action[s])
      memory.last_action[s] = a
      s = tt[s,a]
```

Listing 2: Python code for the exploration algorithm. tt is the transition table

The exploration phase can easily incorporate simulated annealing Van Laarhoven and Aarts (1987) which is a procedure to decrease the probability of choosing a random action over the current best action as time evolves. This is a commonly used method in RL shown to make the exploration phase more efficient by exploiting learned paths already in the exploration phase.

3.3.2 Exploitation: selecting optimal actions

When the system has been sufficiently trained we utilize the resulting storage Table 3. In this phase, we no longer perform random actions for exploration purposes, but instead always choose the *best action* for the given state-goal pair. This is our exploitation code:

```python
def exploit(s, goal):
  statePath = []
  statePath.append(s)
  if table.shortest_time[s,goal] < INF:
    while s != goal:
      a = int(table.action[s,goal])
      s = tt[s,a]
      statePath.append(s)
  return statePath
```

Listing 3: Python code for the exploitation algorithm

4 Results

4.1 Graph world

Using our algorithm to explore the graph world in Figure 1, we obtain the results found in Table 3. Our agent was not given the graph, nor information about the number of states. We compare our method with the FW algorithm, which we do need to provide the whole

Table 3: The shortest paths found running in the graph world depicted in Figure 1. Our method matches the result from the Floyd-Warshall algorithm, however without knowledge of the graph.

	Our algorithm					Floyd-Warshall				
States	A	B	C	D	E	A	B	C	D	E
A	1	4	3	2	1	1	4	3	2	1
B	1	1	4	3	2	1	1	4	3	2
C	2	1	3	2	1	2	1	3	2	1
D	3	2	1	2	1	3	2	1	2	1
E	4	3	2	1	1	4	3	2	1	1

Table 4: The resulting optimal actions found after exploring the graph world depicted in Figure 1.

States	A	B	C	D	E
A	1	2	2	2	2
B	1	2	1	1	1
C	1	1	2	2	2
D	1	1	1	2	2
E	1	1	1	1	2

graph. FW is known to find the optimal all-pairs distances in minimum time. We see that our method and FW find identical solutions. The FW algorithm solved the graph in 0.22 ms on a single CPU and our algorithm consistantly found a solution after 25 exploring steps averaging 0.51 ms. However, the timing comparison is not quite fair, as solving a known graph vs unknown graph are problems of different complexity.

The resulting *best actions* for graph 1 is shown in Table 4. Given that we found the optimal paths above, these are the optimal actions the agent must take when a goal state is defined.

4.2 2-link manipulator

For the 2-link manipulator, our method efficiently found the optimal paths and actions between goals. First we find the solutions to random 10x10 state-space worlds as shown in Figure 3. In this environment, there were 10x10=100 unique states, The optimality of the solution can be easily verified by inspection.

As a more realistic example, we show in Figure 4 (a) a 2-link manipulator in a workspace subject to obstacles. The manipulator has arm lengths 6 and 5, and is centered in a 24x24 workspace. Any point beyond this is outside of the manipulator's reach. We generate a discretized state-space representation of this workspace in (b) by checking every possible manipulator configuration for collision (both θ_1 and θ_2 have the range $[0, 2\pi]$). We run our exploration algorithm and show in (b) a solution path between two randomly chosen states. Again, the optimality of this path can be easily verified. The solution graph was consistantly found after 60.000 steps of our *explore()* method, which averaged to 9.4 CPU seconds. As expected, the larger the problem the more training steps are required. Any path between two states can now be easily navigated using the *exploit()* method.

5 Discussion

As agents, we learn how to obtain goals in our world, and we are quite efficient at doing so. Through exploration we hypothesize that the brain generates an internal metric that encodes the *closeness* of states given the available action space. This is motivated by the brain's high level of neural network recurrency which enables temporal association Zipser (1991). By means of a simple state-action mapping, efficient reactive control can be achieved. To overcome narrow AI DeepAI.org (2021) we require methods to learn more generally. We need to utilize all the information about the environment, not just update our knowledge when receiving rewards for a specific task. When the agent here explores its environment, there is no defined goal, except to explore as much as possible. Such exploration where the only goal is to find novel states, has been very successful in RL Huang and Weng (2002); Conti et al. (2018); Krebs et al. (2009), and is also biologically motivated through play Roussou (2004).

Compared to other established all-pairs shortest path methods, we believe the method presented here highlights biological motivation for the ASAD problem, and how animals may use such an algorithm in reactive control. Existing shortest path methods traditionally require information about the complete world graph prior to execution, and are therefore less relevant in the agent-based context where the world is unknown. One might be tempted to simply explore and generate a model of the world and then run a traditional shortest path method on this model. Such an approach would be far less efficient as (i) we would have to store information about the world graph, and (ii) we would have to split the process into two steps in an offline manner. In this case, it is difficult to know when the environment has been sufficiently explored such that we can run the path planning algorithm. Also, if we find new states, we would need to run the whole process over. With our method we can continuously update the best paths and actions, and by balancing exploration and exploitation we can continue to have some level of exploration indefinitely. This is especially useful for very

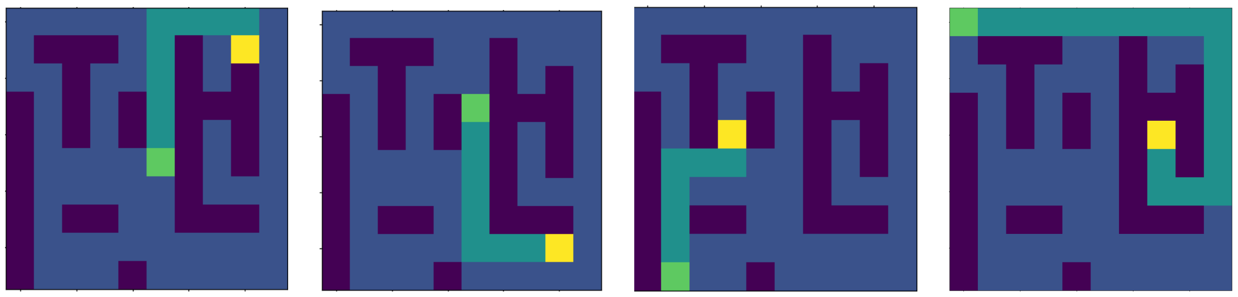
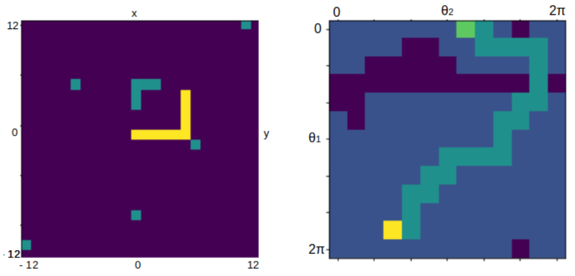


Figure 3: Solutions to the 2-link manipulator as shown in Figure 2. Regular states are blue, prohibited states are dark blue, the starting state is green, the goal yellow and the path turquoise. θ_1 and θ_2 are represented by row and column respectively.



(a) Workspace (a) State-space

Figure 4: Solutions to the 2-link manipulator as shown in Figure 2. (a) the manipulator (yellow) in its workspace. Obstacles are turquoise. (b) the systems state-space where the y-axis is θ_1 and the x-axis θ_2. Dark blue states represent collision with an obstacle and light blue states are allowable states. The turquoise path shows the solution when moving from the green state to the yellow state.

large and even dynamically changing worlds, as the real world is, where we require updating our knowledge continuously. Additionally, we can take advantage of simulated annealing in order to learn faster.

As with all tabular methods this method has some limitations; (i) the *curse of dimensionality* and (ii) confinement to discrete systems. However, future investigation using artificial neural networks will be conducted as to approximate the values and actions in the same way that they have made deep RL algorithms so efficient for large and continuous environments.

6 Conclusion

We have presented here an algorithm that efficiently utilizes state transition feedback from unknown environments in order to learn how to solve the ASAD problem. This was accomplished using ideas from shortest path solvers, Lyapunov functions and Bellman's principle of optimality to generate a state-goal value and best action table. We have addressed how RL methods today often lead to narrow AI, and how our method can be used for more general learning. Furthermore, we have motivated our approach as relevant for biological reactive control. Our method enables agents to efficiently learn optimal solutions.

References

Bellman, R. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 1952. 38(8):716. doi:doi:10.1073/pnas.38.8.716.

Bellman, R. On a routing problem. *Quarterly of applied mathematics*, 1958. 16(1):87–90. doi:doi:10.1090/qam/102435.

Bertsekas, D. P. *Dynamic Programming and Optimal Control*. Number 3 in Athena Scientific Optimization and Computation Series. Athena Scientific, Belmont, Mass, fourth ed edition, 2016.

Britannica, E. Information theory - Physiology. https://www.britannica.com/science/information-theory, 2020.

Camacho, E. F. and Alba, C. B. *Model Predictive Control*. Springer Science & Business Media, 2013.

Conti, E., Madhavan, V., Such, F. P., Lehman, J., Stanley, K., and Clune, J. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Advances in Neural Information Processing Systems*. pages 5027–5038, 2018.

DeepAI.org. What is narrow ai? https://deepai.org/machine-learning-glossary-and-terms/narrow-ai, 2021.

Deo, N. and Pang, C.-Y. Shortest-path algorithms: Taxonomy and annotation. *Networks. An International Journal*, 1984. 14(2):275–323. doi:doi:10.1002/net.3230140208.

Dijkstra, E. W. et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1959. 1(1):269–271. doi:doi:10.1007/bf01386390.

Floyd, R. W. Algorithm 97: Shortest path. *Communications of the ACM*, 1962. 5(6):345. doi:doi:10.1145/367766.368168.

Ford Jr, L. R. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

Freeman, R. A. and Primbs, J. A. Control Lyapunov functions: New ideas from an old source. In *Proceedings of 35th IEEE Conference on Decision and Control*, volume 4. IEEE, pages 3926–3931, 1996. doi:doi:10.1109/cdc.1996.577294.

Huang, X. and Weng, J. Novelty and reinforcement learning in the value system of developmental robots. 2002.

Izquierdo-Torres, E. and Di Paolo, E. Is an Embodied System Ever Purely Reactive? In M. S. Capcarrère, A. A. Freitas, P. J. Bentley, C. G. Johnson, and J. Timmis, editors, *Advances in Artificial Life*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pages 252–261, 2005. doi:doi:10.1007/11553090_26.

Krebs, R. M., Schott, B. H., Schütze, H., and Düzel, E. The novelty exploration bonus and its attentional modulation. *Neuropsychologia*, 2009. 47(11):2272–2281. doi:doi:10.1016/j.neuropsychologia.2009.01.015.

McCloskey, M. and Cohen, N. J. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. In G. H. Bower, editor, *Psychology of Learning and Motivation*, volume 24, pages 109–165. Academic Press, 1989. doi:doi:10.1016/S0079-7421(08)60536-8.

Roussou, M. Learning by doing and learning through play: An exploration of interactivity in virtual environments for children. *Computers in Entertainment*, 2004. 2(1):10. doi:doi:10.1145/973801.973818.

Schiavone, G., Großekathöfer, U., à Campo, S., and Mihajlović, V. Towards real-time visualization of a juggler's brain. *Brain-Computer Interfaces*, 2015. 2(2-3):90–102. doi:doi:10.1080/2326263X.2015.1101656.

Spong, M., Hutchinson, S., and Vidyasagar, M. *Robot Modeling and Control*. Wiley, 2005.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

Toth, P. and Vigo, D. *The Vehicle Routing Problem*. SIAM, 2002.

Van Laarhoven, P. J. and Aarts, E. H. Simulated annealing. In *Simulated Annealing: Theory and Applications*, pages 7–15. Springer, 1987. doi:doi:10.1007/978-94-015-7744-1_2.

Zipser, D. Recurrent network model of the neural mechanism of short-term active memory. *Neural Computation*, 1991. 3(2):179–193. doi:doi:10.1162/neco.1991.3.2.179.