Title Coming Soon

Anonymous Author(s)
Affiliation
Address
`email`

Abstract: The purpose of this document is to provide both the basic paper template and submission guidelines. Abstracts should be a single paragraph, between 4–6 sentences long, ideally. Gross violations will trigger corrections at the camera-ready phase.

Keywords: CoRL, Robotics, Learning, Q-Learning, DQN, Reinforcement Learning

1 Scratchpad

1.1 Outline

- Abstract
- Introduction
  - Define Problem
  - Discuss Challenges
- Preliminaries and Motivation
- Our Method (Find a better name!)
  - Formulation
  - Constraints Definitions
  - Dynamics Decoupling
  - Dynamic Programming Algorithms for Intuition
  - DQN Architecture
- Experimental Results
- Conclusion

Still To do:

- Probability Transition can be used to encode the chance of failure and the cost of entering the failure state can be a parameter that the user can tune to their accepted level of risk! (Mention in conclusion)
- Add notes about Running example as seen in figures 1, 2 and 3
- ~~mention that edge removal is isomorphic to ordering~~
- Finish DQN section
- Add result plots and flush out
- Add future directions to conclusion
  - Structure partitioning
  - Full stack assembly solution (agent assignment and path planning)

- Mention number of subassemblies constraint in constraints section
- stress power of dynamics decoupling
- fix constraint figure
- fix example figures in experimental results

2 Introduction

With the rise of automated manufacturing, there has been much interest in also developing a method for determining the order in which parts should be assembled to create a product. This "Assembly sequencing" task has wide-ranging applicability from planning the order of work performed on a specific home, to optimizing a generic manufacturing process by determining the most efficient order in which parts should be assembled. These assembly sequencing algorithms can even work to avoid errors that can cause significant costs, such as delays, rework, and even scrap, ultimately improving the quality and profitability of the final product.

Furthermore, Robots are becoming increasingly commonplace in manufacturing and production scenarios. As such, there is immense interest in providing assembly planning algorithms for these robotic workers, and current commercial solutions are insufficient outside of very structured environments. add citation

Current solutions primarily help to improve manufacturing efficiency by reducing the time or cost required to assemble a product and reducing the need for rework or adjustments. These algorithms can also help to ensure that the intermediate states of the product are stable, reducing the risk of safety hazards or other issues that could arise from assembly errors.

Furthermore, assembly sequencing algorithms can be applied in a variety of manufacturing contexts, from automotive and aerospace to consumer goods and electronics. As manufacturing processes become increasingly complex and automated, the need for efficient and accurate assembly sequencing algorithms will only continue to grow.

As stated above, assembly sequencing is the process of determining the order in which parts should be assembled to create a product. This problem is made challenging by the many factors that need consideration, from the compatibility of parts, to the availability of tools and resources at certain points in the assembly process, to constraints made on intermediary assemblies.

Traditionally, assembly sequencing problems have been solved using heuristic methods which often produce near-optimal or occasionally, even optimal solutions. These heuristics typically exploit a characteristic of the cost structure specific to their given problem and so are effectively based on rules of thumb or experience, which do not generalize.

Our key insight is a reformulation which views assembly sequencing as a sequential decision-making problem, which simplifies to an optimal control for deterministic settings. Utilizing tools from Dynamic Programming and Deep Reinforcement Learning (RL), we showcase results that surpass the state-of-the-art and provide solutions for large structures which have been previously computationally intractable. Furthermore, this framework is not only capable of handling stochastic settings, but also arbitrary cost structures.

[paragraph which does an overview of what is included in this paper. Like, we discuss our formalism and talk about the simulation and experiments that we run, etc.]

Mention DQNs

3 Preliminaries and Motivation

[Introductory sentence] In the biological field, Assembly Sequencing methods are often utilized for genomics and DNA sequencing [1][2][3]. However, these algorithms primarily utilize considerations specific to the biological setting. Most notably, the ordering of elements in the finished structure is profoundly important, the form of the final assembly is not always known apriori, and there is no obvious cost structure. As such, many of these methods are not easily adapted to work with robotic systems for automated manufacturing and production tasks.

Some work from has been from the point of view of the manufacturing sciences, but these methods often rely on heuristics specific to a given cost structure or fail to be computationally tractable for large scenarios. Methods that employ Monte Carlo Tree Search [4] often fail to find the true optimal strategy, as they primarily operate as a blind search through a solution space. A similar paper which attempted to utilize Q-Learning [5] failed to generalize to more diverse settings than those directly posed in their paper, and additionally was computationally intractable for large structures.

The best result in line with this work is that of [6] which utilizes Integer Linear Programming and Mixed Integer Linear Programming to pose the assembly sequence as an optimal control problem, but is constricted to a very specific form of dynamics and cost structure. However, due to this work's promise of optimal sequences, this paper will be treated as state-of-the-art.

4 Our Method

The focus of this paper will be on performing assembly planning for manufacturing or construction environments, with pre-provided reward structures. Observe that many choices of reward function exist. For example, it could represent the time required to perform a given task, or even the cost associated with performing certain tasks during construction. Note that minimizing a cost function is equivalent to maximizing a reward function if we set $R(s, a) = -C(s, a)$ where R is the reward for the given state s and action a and C is a similarly defined cost function. As such, this framework can handle such generic reward functions, making it much more versatile than previous works.

Observe that when assembling a product, only the final state (i.e. the structure of the final product) is fixed, as the goal is always to produce the fully assembled structure or product. As such, it is often useful to flip this task, and do *Assembly by Disassembly*, where the initial state s_0 of the system is the fully constructed structure, and the actions a are the removal of pieces or connections. Our primary insight is a smart reformulation of this disassembly problem to a sequential decision-making problem. We pose this final assembled structure as the graph $\mathcal{G}$, where different nodes in the graph correspond to different parts in the assembly, and the edges between these nodes correspond to connections required to connect these parts together in the finished product, as seen in Fig. 1.

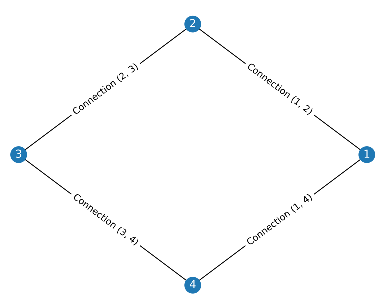


Figure 1: A simple example for a Full Assembly Graph $\mathcal{G}$ with 4 Parts/Nodes and 4 Connections/Edges

We formulate the assembly sequencing problem as a sequential decision-making problem, which can be modeled via the Markov Decision Process (MDP) defined by the tuple $\langle s_i, \mathcal{S}, \mathcal{A}, \mathcal{T}, R\rangle$, where s_i is the deterministic initial state, $\mathcal{S}$ and $\mathcal{A}$ are state and action spaces respectively, $\mathcal{T} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is the probability transition function, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ gives the reward for a given transition. The agent acts with a stochastic policy $\pi : \mathcal{S} \rightarrow P(\mathcal{A})$, generating a sequence of state-action-reward transitions or trajectory. The standard objective is then to find a return-maximizing policy that satisfies Eqn. 1.

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi,\tau}\left[\sum_{t=1}^{\tau} R(s_t, a_t) \middle| \begin{array}{c} s_0 = s_i \\ s_{t+1} \sim \mathcal{T}(s_{t+1}|s_t, a_t) \\ a_t \sim \pi(\cdot|s_t) \end{array} \right] \tag{1}$$

In the robotic construction setting, the deterministic initial state s_i corresponds to the fully assembled structure, the action space $\mathcal{A}$ where $a \in \mathcal{A}$ corresponds to the action of removing a connection from the assembly graph $\mathcal{G}$, and the state space $\mathcal{S}$ comprised of $s \in \mathcal{S}$ which denote semi-connected subassemblies which include only a subset of the edges in $\mathcal{G}$. Observe, that under this definition of state and action the mapping to the next state s_{t+1} from a given state s_t and action a_t is deterministic, meaning that $\mathcal{T}(s_{t+1}|s_t, a_t) = 1$ for the appropriate s_{t+1} and $\mathcal{T}(s_{t+1}|s_t, a_t) = 0$ otherwise. Additionally, as there are a finite number of edges in the full assembly, the disassembly process will produce trajectories of length τ, where τ is the number of edges in the graph $\mathcal{G}$. As such, the return-maximizing policy is $\pi^* = \arg\max_\pi \mathbb{E}_{a_t \sim \pi(\cdot|s_t), s_{t+1} \sim \mathcal{T}(\cdot|s_t, a_t)}[\sum_{t=1}^{\tau} R(s_t, a_t)]$ where $s_0 = s_i$ as previously defined.

Therefore, we can simplify Eqn. 1 to the following,

$$\begin{aligned} \max_{\pi} \quad & \sum_{t=0}^{\tau} R(s_t, a_t) \\ \text{s.t.} \quad & a_t \sim \pi(\cdot|s_t) \\ & \mathcal{T}(s_{t+1}|s_t, a_t) = 1 \\ & s_0 = s_i \end{aligned} \tag{2}$$

Under this specified setting, the state-action of this MDP is illustrated by a directed tree graph $\mathcal{H}$ as seen in Fig. 3a, with the root being the fully assembled structure s_i, the edges corresponding to the removal of certain connections a, and the leaves corresponding to the states where the structure is fully disassembled s_τ.

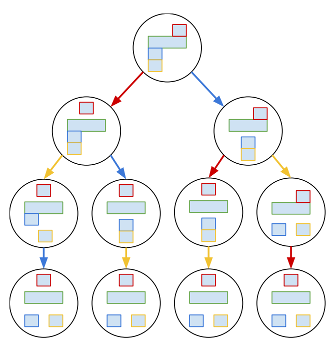
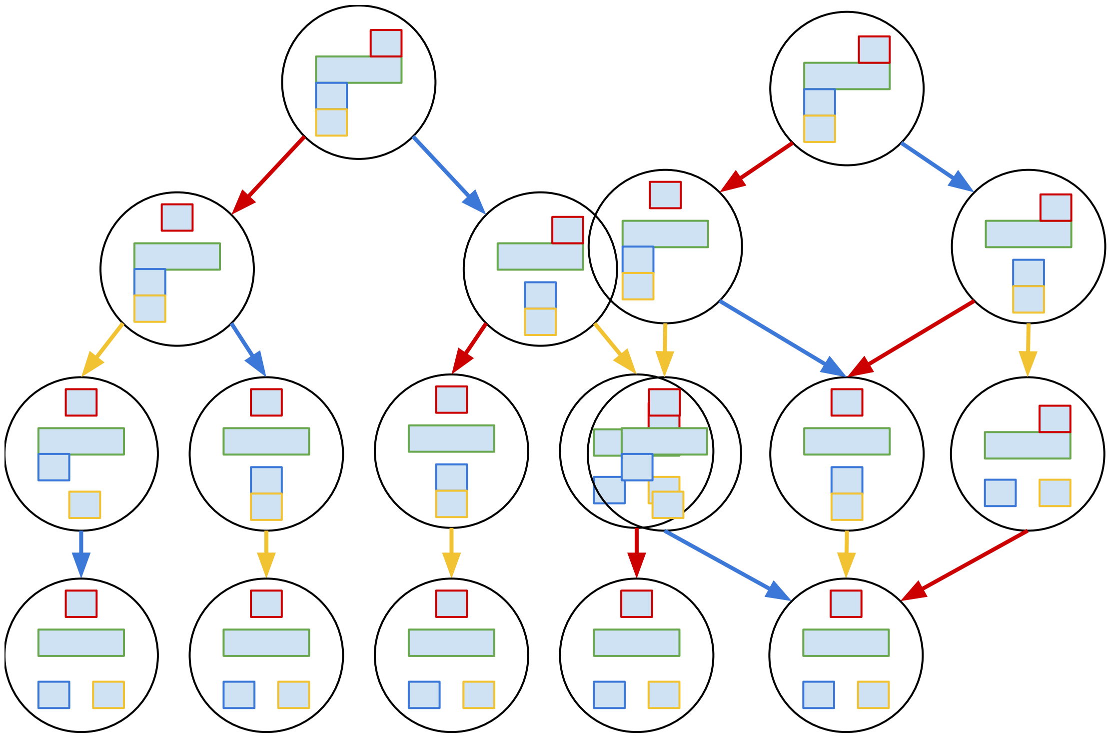


Figure 2: The directed tree graph $\mathcal{H}$ for the disassembly of a 4-Connection Structure

Observe that this definition of State is isomorphic to the ordering of edge removal, and so states can be effectively combined. As such, the expansion rate of the tree $\mathcal{H}$ is greatly reduced and $\mathcal{H}$ instead resembles Fig. 4.

Additionally, observe that for multi-agent scenarios, the action a can be redefined to involve the removal of multiple connections at once. Furthermore, under the assumption that a given agent is capable of transporting larger loads, the transportation of multi-part structures can also be codified as singular actions. For this simplified problem, it is rather clear that a Dynamic Programming solution is sufficient, but given that our state-action space is represented by a tree graph and now the initial state and final states are fixed, more traditional graph exploration techniques are also possible.



(a) The directed tree graph $\mathcal{H}$ for the disassembly of a 4-Connection Structure

(b) The *Consolidated* directed tree graph $\mathcal{H}$ for the disassembly of a 4 Part Structure

Figure 3: Two examples of optimal paths through $\mathcal{H}$ with different Reward functions

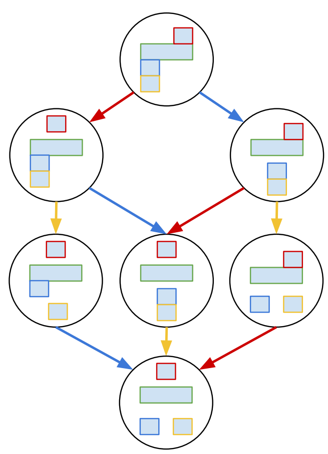


Figure 4: The Consolidated tree graph $\mathcal{H}$ for the disassembly of a 4 Part Structure

4.1 Constraints

Staying consistent with RL norms, the probability transition function $\mathcal{T}(s_{t+1}|s_t, a_t), s \in \mathcal{S}, a \in \mathcal{A}$ can be utilized for simple constraint definitions. For example, if there is some kind of sequential constraint to the assembly (i.e. the center part in a lattice structure must be placed before the parts around it are placed), then this constraint is equivalent to the probability transition between certain state transfers being 0, i.e. impossible.

This kind of constraint is very well-behaved, as when generating the tree graph $\mathcal{H}$, this constraint translates to a particular branch being abandoned. As such, this technique not only ensures satisfaction of these constraints by construction, but also reduces the size of the state-action space, as seen in Fig. 5. Similarly, any constraint on future states s_{t+1} that only utilizes characteristics of the current state of the structure s_t, is well-behaved.

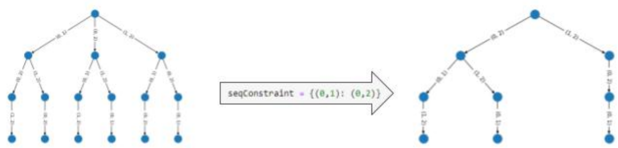


Figure 5: The effect of adding a sequential constraint on the tree graph $\mathcal{H}$

Note that in Fig. 5, the equivalent states are not consolidated for the simplicity of the diagram.

As laid out in the previous section, Dynamic Programming can be easily utilized for solving this problem. Utilizing the Bellman optimality principle, a recursive formula can be constructed, which produces the optimal value function of the Markov decision process (MDP), effectively reinventing dynamic programming. The Bellman optimality principle states that the optimal value function $V^*(s)$ satisfies Eqn. 3.

$$V(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') V(s') \right] \tag{3}$$

where $R(s,a)$ is the expected reward of taking action a in state s, $\mathcal{T}(s,a,s')$ is the transition probability from state s to state s' after taking action a, γ is the discount factor, and $\mathcal{A}$ is the set of available actions in state s.

The Bellman optimality principle states that the optimal value of a state s is equal to the maximum expected return that can be obtained by taking any action a in that state and then following the optimal policy thereafter. This principle forms the basis of the value iteration algorithm 1.

Algorithm 1 Value Iteration Algorithm

Require: MDP $(\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \gamma)$,
Require: $\epsilon > 0$ (Convergence Condition), N (Maximum Number of Iterations)
Ensure: π (Deterministic Policy) s.t. $\pi \approx \pi^*$
Ensure: Optimal Value Function $V(s)$

1: Initialize $V_0(s)$ for all $s \in \mathcal{S}$ randomly (or to some initial value based on a prior)
2: **for** $k = 0, 1, 2, ..., N$ **do**
3:     $\Delta \leftarrow 0$
4:     **for** all $s \in \mathcal{S}$ **do**
5:         $V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a) V_k(s') \right]$
6:         $\Delta \leftarrow \max \Delta, |V_{k+1}(s) - V_k(s)|$
7:     **end for**
8:     **if** $\Delta < \epsilon$ **then**
9:         **break**
10:     **end if**
11: **end for**
12: $V(s) \leftarrow V_k(s)$ for all $s \in \mathcal{S}$
13: **return** π s.t. $\pi(a|s) = \arg\max_a \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a) \left[R(s,a) + \gamma V(s') \right]$

In the Assembly Sequencing problem, there is rarely a use for discounting the effect of future actions, so $\gamma = 1$. Additionally, with a finite state and action space, the convergence criterion ϵ can be set to be very low s.t. $\epsilon \approx 0$, and N can be set based on user preference, but note that too small of a value can cause the policy π to produce non-optimal behavior.

This value iteration method can also be expanded to incorporate not only the value of being in a certain state, but also the quality of taking a certain action in that state, which reproduces Q-Learning, as expressed in Eqn. 4

$$Q^{\pi}\left(s_t, a_t\right) = \mathbb{E}_{\pi}\left[\sum_{k=t}^{T}\left(s_t, a_t\right) \,\middle|\, s = s_t, a = a_t\right] \tag{4}$$

As mentioned earlier, since the state-action space $\mathcal{H}$ is a tree, there will be little re-visitation required for these dynamic programming methods to converge. As such, more traditional graph exploration techniques, such as Breath-First Search (BFS) would be fully exploratory and would also produce an optimal result.

4.2 Q-Learning and DQNs

While this dynamic programming method is capable of producing optimal results, it can only operate on small structures, as the size of the state-action space $\mathcal{H}$ grows very quickly, even with the state consolidation improvement. As such, Heuristic methods will have to be utilized for especially large structures. Utilizing this same MDP framework, a Deep Q Network (DQN) would be quite sufficient for this task. A DQN is a reinforcement learning algorithm that uses a deep neural network to approximate the Q-function of an agent. This deep neural network structure allows the DQN to handle high-dimensional input spaces and in our case, the ability to learn directly from raw state-action-reward data. The output of this DQN would be a vector $q \in \mathbf{R}^n$ with q_i indicating an estimate of the Q-value of the given action i (i.e. removal of edge i). Observe that as this method returns q-values, and so any constraints placed on state transitions can still be employed.

For the assembly sequencing problem, the DQN algorithm, will utilize the same definition of state and action as above, and to translate this result to the input of a neural network, the following indicator function will be utilized:

$$\mathcal{I}(E_i) = \begin{cases} 1 & \text{if Edge } i \text{ is Connected} \\ 0 & \text{if Edge } i \text{ is Disconnected} \end{cases}$$

such that a given state s is indicated via a vector $s \in \mathbf{R}^n$ where n is the number of edges in the completed assembly. The output of this DQN will then be a vector $q \in \mathbf{R}^n$ with q_i indicating an estimate of the Q-value of the given action i (i.e. removal of edge i). Observe that as this method returns q-values, any constraints placed on state transitions can still be employed. As such, the DQN follows the neural network architecture laid out in Fig. 6, and employs the use of an experience replay buffer during training to reduce correlations between consecutive updates of the network. While this method was sufficient for our results, additional modifications can supercharge the DQN algorithm, such as the use of double Q-learning, prioritized experience replay, or dueling network architectures.

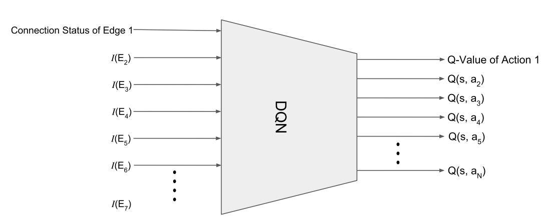


Figure 6: An overview of the DQN Architecture

[Forgot to mention ϵ-greedy!]

$$\pi(a \mid \hat{s}) = \begin{cases} \arg\max_a Q(\hat{s}, a) & \text{with Probability } 1-\epsilon \\ a \sim Unif\left(\mathcal{A}_{\hat{s}}\right) & \text{with Probability } \epsilon \end{cases} \tag{5}$$

where $\mathcal{A}_{\hat{s}}$ is the action space available at the given state $\hat{s}$, which in the robotic manufacturing setting translates to the edges remaining in the subassembly $\hat{s}$.

5 Experimental Results

Utilizing the value iteration method, an optimal path from the start of the tree $\mathcal{H}$ to the end, can be found, which indirectly prescribes a disassembly ordering, which can then be reversed to produce the optimal assembly sequence. This optimal path through H can be seen in Fig. 7 (Note that the states haven't been consolidated in this figure for clarity, but can be found in the attached code).

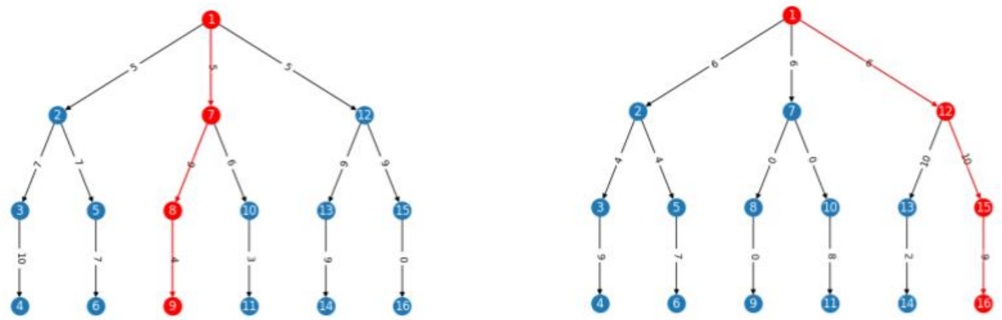
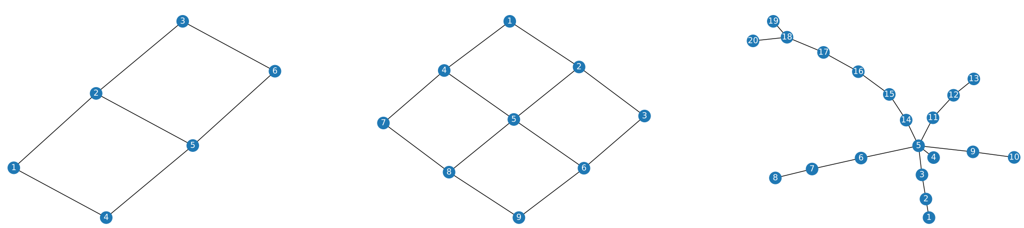


Figure 7: Two examples of optimal paths through $\mathcal{H}$ with different Reward functions

With the method established, the next step was to construct complex structures as seen in Fig. 8, and to show the usefulness of this result, we compare it to the State of the Art ILP solution presented by Culbertson et al. [6] in Table 1.



(a) "2x3" Structure (b) "Lattice" Structure (c) "Hubble" Structure"

Figure 8: The example structures evaluated in Table 1

	Integer Linear Programming [6]	**Dynamic Programming method**	
		Graph Generation	**Value Iteration**
2x3		0.069551 s	0.001235 s
Lattice	Minimum-time: 4s Minimum-travel: 5800s	1.319223 s	0.035178 s
Hubble	Minimum-time: 5000s Minimum-travel: 20000s	107.806627 s	0.648012 s

Table 1: Comparison of Run Times between our Dynamic Programming method with the ILP method used in Culbertson et al. [6]. The ILP paper utilized two cost structures, with one denoting the minimum time taken, while the other utilized a psuedo minimum fuel objective, simplified to minimum distance traveled. Constraints in the ILP were translated to sequential constraints for the Dynamic Programming method.

As seen in the table, our method converges to the solution much faster than the ILP method, while producing the same results (assuming the same cost structure is used).

6 Conclusion

As we were able to reproduce the results seen in Culbertson et al. [6], we can be confident that our method is capable of producing optimal results. This is intuitive, as our reframing of the assembly sequencing problem as a sequential decision-making problem follows from the structure of the problem. At a given point in an assembly procedure, the goal is to identify the next part to attach to the subassembly in order to minimize some cost incurred over the course of the entire assembly procedure, whether that is in terms of time taken or minimizing some kind of fuel expended to perform each action.

While our method is capable of producing optimal results, it can primarily only operate on small structures, as the size of the state-action space $\mathcal{H}$ grows very quickly, even with the state consolidation improvement. As such, Heuristic methods will have to be utilized for especially large structures. Utilizing this same MDP framework, a Deep Q Network (DQN) would be quite sufficient for this task. A DQN is a reinforcement learning algorithm that uses a deep neural network to approximate the Q-function of an agent. This deep neural network structure allows the DQN to handle high-dimensional input spaces and in our case, the ability to learn directly from raw state-action-reward data. The output of this DQN would be a vector $q \in \mathbf{R}^n$ with q_i indicating an estimate of the Q-value of the given action i (i.e. removal of edge i). Observe that as this method returns q-values, and so any constraints placed on state transitions can still be employed. Additionally, a DQN employs the use of an experience replay buffer during training to reduce correlations between consecutive updates of the network. While this method is probably sufficient for this problem, additional modifications can supercharge the DQN algorithm, such as the use of double Q-learning, prioritized experience replay, or dueling network architectures.

References

[1] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, jun 2010. doi:10.1016/j.ygeno.2010.03.001.

[2] J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Research*, 17(11): 1697–1706, oct 2007. doi:10.1101/gr.6435207.

[3] J. Warnke-Sommer and H. Ali. Graph mining for next generation sequencing: leveraging the assembly graph for biological insights. *BMC Genomics*, 17(1), may 2016. doi:10.1186/s12864-016-2678-2.

[4] A. D. Giorgio and Filmon Yacob. Assembly sequences with mcts. 2018. doi:10.13140/RG.2.2.11175.44968.

[5] A. D. Giorgio and Filmon Yacob. Assembly sequences with q-learning. 2018. doi:10.13140/RG.2.2.35502.41286.

[6] P. Culbertson, S. Bandyopadhyay, and M. Schwager. Multi-robot assembly sequencing via discrete optimization. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, nov 2019. doi:10.1109/iros40897.2019.8968246.