Autonomous Task Sequencing and Assignment for In-Space Multi-Robot Assembly

Joshua Moser
Advisor: Erik Komendera
Virginia Polytechnic Institute and State University

Abstract

As endeavors into space exploration and related scientific experimentation expand, the use of autonomous robotic systems to construct and maintain infrastructure will be necessary. To accomplish autonomous assembly in a realistic, stochastic environment, it is necessary to have the ability to reschedule robots and tasks to deal with a changing environment. To this end, the work presented here developed a novel stochastic problem definition that is capable of articulating the different elements present in the autonomous assembly problems through the use of different graph structures and a Markov decision process formulation. While this formulation can be utilized by many different schedule generation methods, a reinforcement learning schedule generation method is explored in this work. It was found that the complexities present in a realistic assembly problem prove to be a challenge for a single policy reinforcement learning formulation which leads to a future work recommendation of pursuing a multi-policy reinforcement learning approach to handle the complexities present in a realistic assembly problem.

Introduction

As the frontier of in-space exploration and research is pushed forward, technological advances will be required to facilitate this effort. One such advancement that shows great potential is the use of autonomous systems for the construction and maintaining of the necessary infrastructures in the in-space context. Such systems, can be deployed to prepare a habitation structure for astronauts or construct and maintain an unmanned research site. Autonomous systems seen in industry currently implement robots to handle repetitive tasks such and pick and place or repetitive welding jobs. While these are elements that can be present in an assembly process, an autonomous assembly project in an in-space environment will often be required to operate for timespans without human input. This will require the autonomy to be capable of making changes in the task assignment in response to changes that cause deviation from the original schedule, many introduced as a result of working in a a real world, stochastic environment. This level of autonomy will require a framework to articulate the the assembly problem, breaking it down into base components that can be modified to respond as required. This articulation must also have the ability to describe the stochastic elements present, providing a way for the decision making policy to account for risk associated with different assignment configurations. Using this formulation, an assembly task assignment map (assembly schedule) can then be developed to build or repair the structures the system is assigned to maintain.

Problem Formulation

To this end, a framework, noted as the stochastic problem formulation (SPD), was developed to articulate the elements present in an autonomous assembly problem, the constraints that are required to ensure an assembly is viable, and a mathematical model to represent the assembly state and the stochastic elements present. This formulation can be

broken down three main sections: *Elements*, *Constraints*, and *State Representation*. To facilitate the explanation of this problem formulation, a two piece, two robot, assembly problem, shown in Figure 1, will be used. The goal of this assembly will be to move both the block and post pieces into position where they will be welding together.

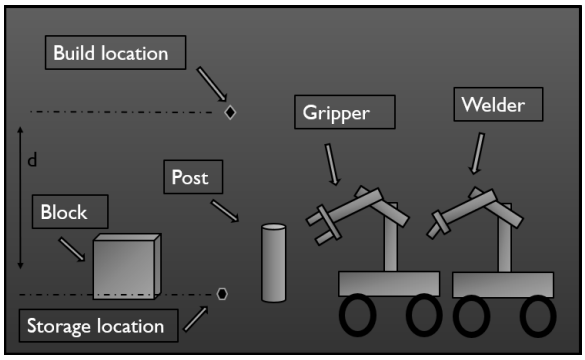

Figure 1: Two robot and two piece assembly project

Elements

The *Elements* portion of the problem formulation describes the "physical" portion of the assembly problem, namely: the important locations in the assembly (*Points*), the structural elements (*Components*) in the assembly, the connection type information (*Joints*), and the autonomous operators (*Robots*). These four categories are divided into two different sets: *Components*, *Joints*, and *Robots* all affect the state of the assembly and are considered stateful elements while *Points* falls into the category of stateless element. For each element, the characteristics are broken up into the two categories: states and properties. As indicated in the name, the states are those that factor into the overall state of the assembly and the properties remain constant throughout the assembly. This distinction will be important when it comes modeling the state of assembly, described in a few sections.

Points: As stated above, it is the only stateless element in the problem formulating. Denoted by the set $W = \{W_1, \ldots, W_i, \ldots W_{|W|}\}$, each point can be thought of as a description of a location in the assembly environment. Since these location designations do not change as the assembly progresses, the Point type is stateless. Each Point usually contains two property features: type and location. Figure 2 contains the Point formulation for this assembly example.

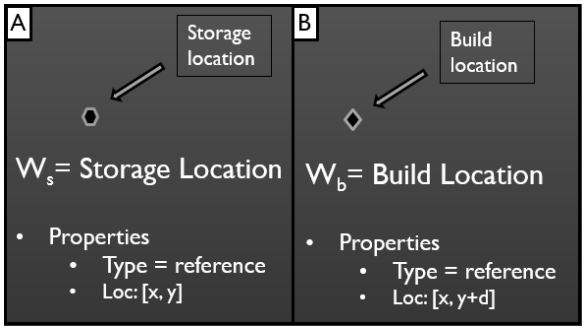

Figure 2: A: Point formulation for the storage location, B: Point formulation for the build location

Components: A component, represented in the set $C = \{C_1, \ldots, C_i, \ldots C_{|C|}\}$, is any physical part that is included in the autonomous assembly problem and is not an autonomous operator. It contains all of the elements that will need to be manipulated for the assembly. Each C_i has a set of possible states and properties. For the component type, there are generally three different state features: in position / not in position, broken / not broken, and current location. The in position / not in position state reflects if the component is correctly installed. The broken / not broken state reflects if that component is going to need to be replaced or if it can remain. Finally, the current location state provides state information as the component transitions around the environment. The property features for each component will contain information such as component type, locations, and weight (if applicable). The location property feature can contain points that are important to the component. Two primary examples are start location and goal location. Figure 3 contains the component formulations at the start of this example assembly project.

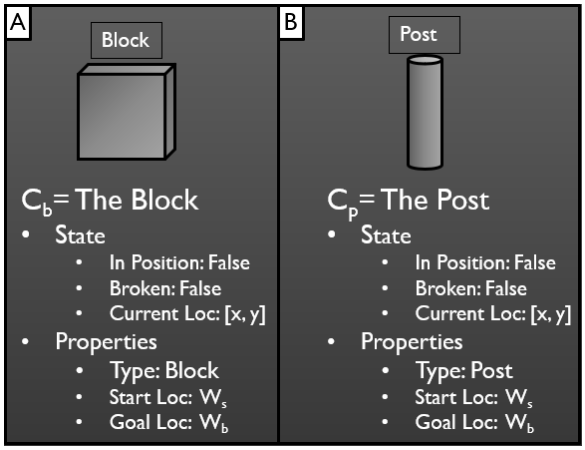

Figure 3: A: Block component formulation, B: Post component formulation

Joints: A Joint, represented in the set $K = \{K_1, \ldots, K_i, \ldots K_{|K|}\}$, can be thought of as an element of the assembly problem that is not in the form of a physical part or a location. This will take the form of welds, bolt joints, etc. Similar to the component type, this element contains states and properties in its definition. For a joint, joined / not joined and completion percent make up the two most common state features. Joint type, locations, and component list are examples of property features. Figure 4 contains the formulation of the weld present in this assembly.

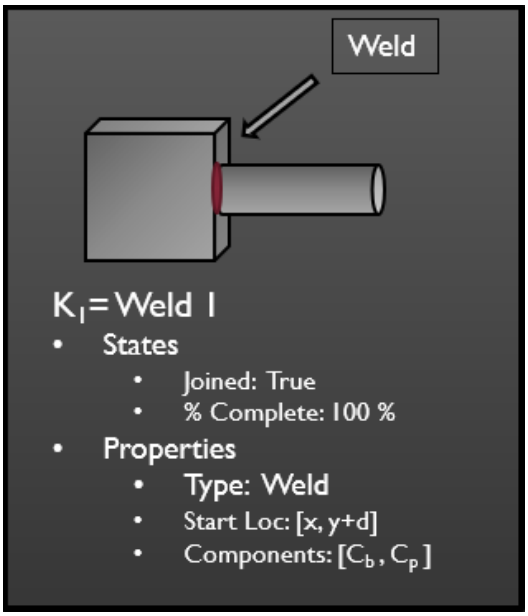

Figure 4: Joint formulation for the weld

Robots: The Robot element type represents the autonomous operators that will be used to complete the assembly project. A given robot, represented in the set $R = \{R_1, \ldots, R_i, \ldots R_{|R|}\}$, can have the following state features: idle / busy, current location, current task, and energy level. Additionally, each robot often has four property features: robot type, locations, workspace, and abilities. The workspace can be thought of as a robot's reach, that is, the area a robot has access to without moving its base. To define the abilities feature, a set of operation types in the project needs to be defined. Based on the types of components and joints in the assembly project and the types of robots in the project, a set of operations will be defined as $O = \{O_1, \ldots, O_i, \ldots O_{|O|}\}$. These operations will be discussed in more depth in a following section. For now, each operation type can be thought of as an action a robot has to preform (ex: $\{hold, weld, locomote\}$). The abilities property for each robot will have a set of entries for each type of operation representing how long it takes a robot to complete that operation (the processing time). If the problem is stochastic, this will take the form of a distribution. This distribution can be accompanied by distributions representing the time cost of a minor failure (one that delays the completion of the operation) and a chance of major failure (requiring the operation to be restarted). For this example, a normal distribution is used. If the schedule generation method being used requires a single value input for processing time, this can also be modeled as the expected value of the processing time distribution. Figure 5 shows the two robots, using an expected value representation for the processing times for each operation.

Constraints

The constraints portion of the formation contains the requirements that describe what a valid assembly sequence must look like. This will include a job shop type formulation to

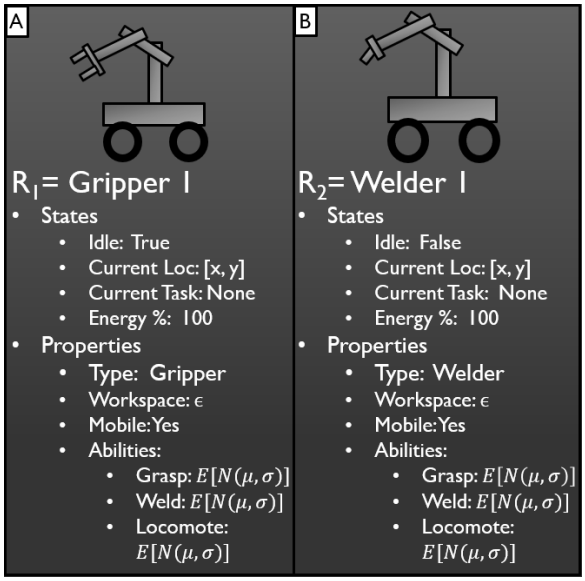

Figure 5: A: Robot formulation for the gripper robot, B: Robot formulation for the welding robot

discretize the steps into different action that must be taken, precedence constraints to ensure tasks are completed in the required order, continuity constraints and machine validity constraints to ensure that the correct machines are used on tasks, and finally, distance constraints to encapsulate the distance robots will have to travel between tasks.[1]

Job Shop Scheduling Problem Formulation: As mentioned above, a job shop scheduling problem formulation (JSSP) is used to describe the discretized tasks required to complete an assembly project. The jobs are represented by the set $J = \{J_1, \ldots, J_i, \ldots J_{|J|}\}$. For this project there are two kinds of jobs: *Move (M)* and *Affix (A)* where *M* represents the task type of moving a component from one point to anther in the environment and *A* represents the task type of connect the block and post together. The complete set of jobs for this assembly project is: $J = \{M_{Block}, M_{Post}, A_{Post}\}$. Each job will have a set of operations (O_j), which represent the individual robot contributions for a given job (i.e. if there are two operations in a job, two different robots have to work together to complete it). It is possible for there to be more than one way to complete a job, leading to multiple sets of operations per job. These different sets are referred to as process plans (P_j). For this project there is one way to complete the move job and one way to complete the affix job. These are given by the process plans: $P_M = \{[Grasp]\}$ & $P_A = \{[Grasp, Weld]\}$ respectively.

Precedence: As stated earlier, the precedence constraints ensure that jobs are only completed in viable temporal sequences. To model the precedence constraints, a Directed Acyclic Graph (DAG), $G_p(V_p, \alpha_p)$, is used where each vertex, V_p, represents a job and each arc, α_p, represents a precedence constraint, thereby encoding the project precedence constraints in the structure of the graph. Figure 6 contains the precedence DAG to get from the starting configuration (S) of this project to the finished configuration (F).

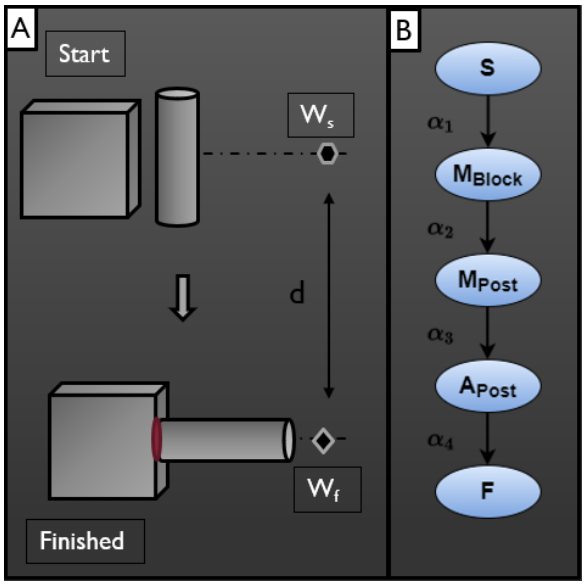

Figure 6: A: Start and finished component configurations, B: Precedence Directed Acyclic Graph

Continuity: In an assembly project, there may be times when a robot needs to continue in the same operation that it had in a previous job. This constraint is represented by the set *H*, which contains arc and operation type parings. The arcs, from the DAG mentioned previously, describe which jobs these constraints

apply to. The operation type represents which operations need to have the same robot across the arcs. For this project, a continuity constraint will be defined to require the robot that moves the post into position is also the robot holding the post during the welding (preventing the need for the part to be set down). This is formulated as: $H = \{(\alpha_3, Grasp)\}$.

Valid Robot Selection: In an assembly problem, there are also instances in which a specific robot must be used for a job even if other robots are technically capable of completing it. To formulate this, a set of pairs (V) is defined that represents what robots can complete what jobs. For this assembly there is no restriction, making all the robots valid choices for all of the jobs: $V = \{(J,R)\}$.

Distance: Finally, as mentioned above, the distance between jobs must also be modeled to allow a scheduler to include the cost of moving between jobs as it selects robots and job sequences. To model this, a fully connected graph is used, $G_d(V_d, E_d)$, where each vertex, V_d, represents each job. In this graph, each edge (E_d) contains the values of the distance, in a project's required units, between the jobs it connects. Figure 7 contains the fully connect graph for this assembly project where the distances are either d or 0 depending on if they are at the same located or across the environment space.

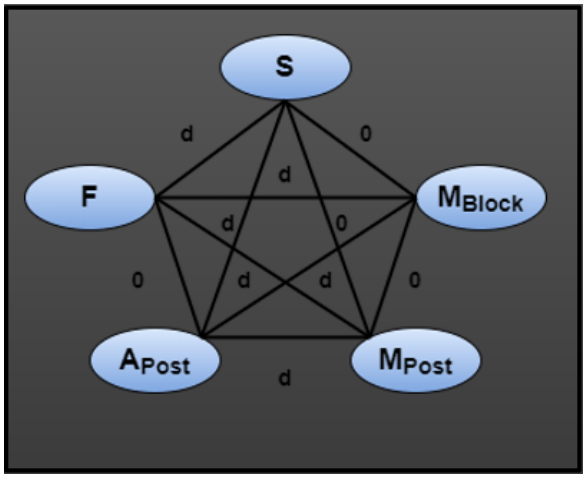

Figure 7: Fully connected distance graph

States

The definitions above provide a way to describe the current state of each element in the assembly project and all the constraints required to complete the assembly. The aggregated states of all the stateful elements form the basis for a description of the overall state of the assembly. For the purpose of task assignment, it is necessary to model how the state of an assembly will change for a particular robot to task assignment. In a realistic autonomous assembly problem, there will often be stochastic elements that factor into the state transition of the assembly. To accommodate this, the SPD models the autonomous assembly as a Markov Decision Process (MDP), represented as the tuple $(S, A, P(s'|s,a), R)$, containing the state set, action set, transition probability, and transition reward respectively. Each of these will be unpacked below.

State Space: The SPD state space, S, is the state set representing all of the possible states a given assembly project can take. Each unique state, $s \in S$, is a different state variation of the stateful elements defined earlier. Some of these are easily thought of as discretized instances, such as what job a robot is currently assigned to. Alternatively, some of the elements fall into a continuous domain and will be discritized as an approximation based on the capability of a given scheduler generator utilizing this formulation.

Action Space: The SPD action space, A, is the action space representing all of the possible actions in the assembly problem. These actions take the form of assigning different robots to different tasks where the set is an accumulation of the different unique assignment possibilities $(a \in A)$.

Transition Probability: The transition probabilities, $P(s'|s,a)$, can be thought of as a measurement of how likely a certain state transition is given an action assignment.

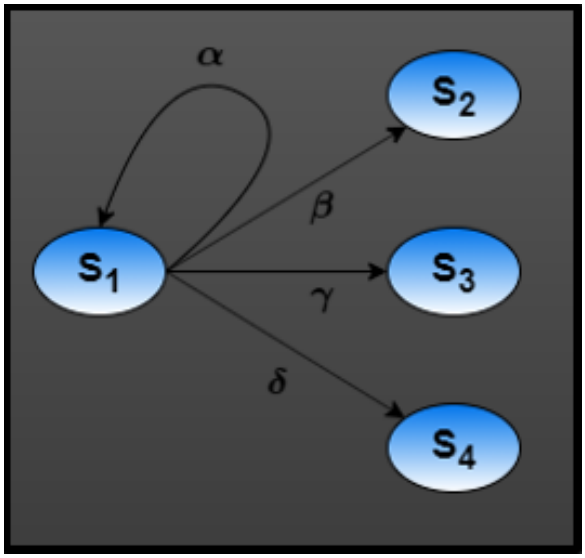

Figure 8: Transition probability example

These transition probabilities are a key feature of this problem formulation as they encapsulate much of the stochastic nature present in the assembly problems. The probability of transition from the current state, s, to the new state, s', is dependent on the probability of success from a chosen action, a. These probabilities can come from the processes times and chances of failure defined in the robot abilities as well as stochastic elements present in the environment.

Figure 8 and Table 1 provide an example of how this probability function is generated through the use of a two job example. The jobs are $J = \{J_1, J_2\}$ where $J_1 = M_{Block}$ and $J_2 = M_{Post}$. Each job has the two possible states, completed, ✓, and not completed, ✗. This leads to four possible assembly states, shown in Table 1. In this example, s_1 represents a state where J_1 is complete and J_2 is not complete $(J_1^{\checkmark}, J_2^{\times})$ i.e. the block is in position but the post is not. By the same logic, the other three states represent the other three possible combinations. The equations in 1 show the probability formulations for each of the possible transition probabilities. Through probability laws these probabilities are equivalent to those shown by the equations in 2.

Table 1: Example MPD with State Transitions

State	Job instances
s_1	$J_1^{\checkmark} J_2^{\times}$
s_2	$J_1^{\checkmark} J_2^{\checkmark}$
s_3	$J_1^{\times} J_2^{\checkmark}$
s_4	$J_1^{\times} J_2^{\times}$

$$\begin{aligned} P_\alpha(\cdot) &= P(s' = s_1 | s = s_1, a = a) = P(J_1^{\checkmark}, J_2^{\times} | a) \\ P_\beta(\cdot) &= P(s' = s_2 | s = s_1, a = a) = P(J_1^{\checkmark}, J_2^{\checkmark} | a) \\ P_\gamma(\cdot) &= P(s' = s_3 | s = s_1, a = a) = P(J_1^{\times}, J_2^{\checkmark} | a) \\ P_\delta(\cdot) &= P(s' = s_4 | s = s_1, a = a) = P(J_1^{\times}, J_2^{\times} | a) \end{aligned} \tag{1}$$

$$\begin{aligned} P_\alpha(\cdot) &= P(J_1^{\checkmark}, J_2^{\times} | a) = P(J_1^{\checkmark} | J_2^{\times}, a) P(J_2^{\times} | a) \\ P_\beta(\cdot) &= P(J_1^{\checkmark}, J_2^{\times} | a) = P(J_1^{\checkmark} | J_2^{\checkmark}, a) P(J_2^{\checkmark} | a) \\ P_\gamma(\cdot) &= P(J_1^{\checkmark}, J_2^{\checkmark} | a) = P(J_1^{\times} | J_2^{\checkmark}, a) P(J_2^{\checkmark} | a) \\ P_\delta(\cdot) &= P(J_1^{\checkmark}, J_2^{\times} | a) = P(J_1^{\checkmark} | J_2^{\times}, a) P(J_2^{\times} | a) \end{aligned} \tag{2}$$

This formulation allows for scenarios such as *failures*, *restarts*, and *backtracking*. It also allows for impossible paths to be blocked by asserting zero probability for these transitions. For this example, the assertion can be made that there is no real state where J_2 is successful when J_1 is not (the post can not be in the correct position if the block is not already there if the correct position of the post includes a reference from the block). Modeling this would require that $P(J_1^{\times} | J_2^{\checkmark}, a) = 0$. Using this method of modeling, more complicated interactions from uncertainties can be accounted for. In this general form, *job insertions* are not an issue since they would simply add a set of nodes into the MDP. Unknown states would also not be a concern since this type of formulation describes **every possible state**. This is an important element when it comes to replanning because it means that the required transitions for replanning are *already embedded* in this MDP description. It should be acknowledged here that while this is possible in the theoretical case, in practice, extremely large state spaces are not practical for many schedule generator methods to handle. Therefore, some schedule generation formulations will use a reduced set of states in

the MDP model to approximate the entire assembly.

Transition Reward: The final element of the MDP formulation is the state transition reward, $R_a(s', s)$. This represents how good it is for the state transition (s', s) to occur with respect to completing the assembly. This might include rewarding specific job completions, the completion of the overall assembly, or focusing on elements like penalizing the entering of broken states (states where something in the assembly is broken). The general formulation does not constrain how this reward is generated, leaving a framework in place to be utilized by the different schedule generation methods. Some examples of the reward might be a time cost, or some penalty value utilized in a schedule generator's solver.

Reinforcement Learning

The SPD formulation is independent of any one scheduling method. It provides the foundation for articulating the elements present in an assembly problem. As such, a wide range of schedule generation approaches can utilize it, pulling as much or as little of the information as they are capable of modeling (for example, a deterministic schedule generation method may not be capable of utilizing all of the stochastic information but it will still pull from the constraint and job shop formulation portions). In this work, a reinforcement learning (RL) formulation which is inherently seeking to find an optimal control policy for an MDP formulation, will be explored as a schedule generation method. In reinforcement learning, the goal is to learn a policy, $\pi(s)$, that will predict the optimal action to take for a given state by evaluating how good the next state is as a result of the action. This "goodness" is measured by predicting the future reward to give a state value when a certain policy is followed. This value function for a given policy can be thought of as:

$$v_\pi(s) = \mathbb{E}_\pi [G_t | S_t = s] \tag{3}$$

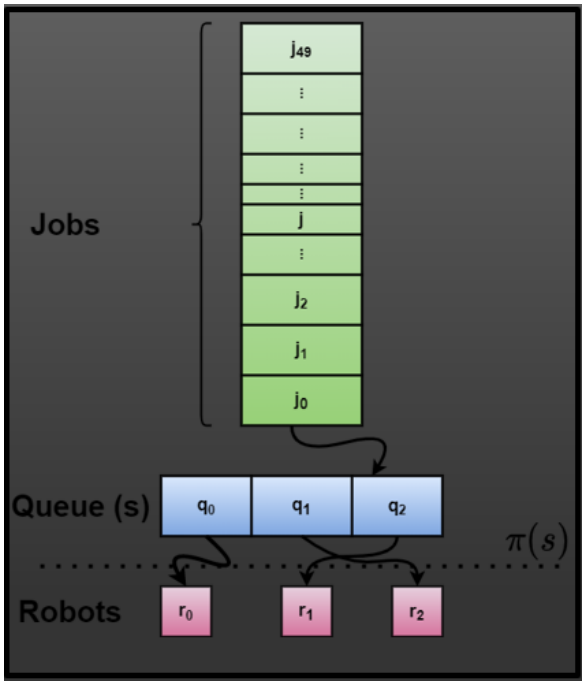

Figure 9: Reinforcement learning project anatomy

where s is the state and the future reward, G_t, is given by:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

where a discount factor, $\gamma \in [0, 1]$ is used to prioritize rewards, R_t, closer to time t.[2]

Formulations

The specific RL methods utilized in the experiments performed for this work were A2C,[3] DDPG,[4] SAC,[5] TD3,[6] and PPO.[7] These are all policy gradient methods, that is, they seek to directly optimize the policy. Using these methods, the experiments below sought to expand the very preliminary work done in this author's previous work[8] by expanding the state space from only evaluating processing times to including additional complicating elements from the SPD such as distance.

The environment consisted of three types of robots: Manipulator, Jigging Robot, and Walker, along with three different types of tasks: Manipulation, Alignment, and Locomotion. The assembly space to be navigated

Table 2: The processing times for each robot type and task type pair given in environment time step units.

Robot Description	Manipulation	Alignment	Locomotion
Manipulator	2	10	5
Jigging Robot	5	2	10
Walker	5	10	2

was thought of as a 20×20 grid. For training, a 50 job queue was populated with a random number of the three types of jobs at random locations in the space. The scheduler (policy) was only able to see 3 jobs at a time in a visible queue and was tasked with assigning each job to one of the three types of robots. It was also allowed to wait on assigning a job to a robot, giving it the ability to wait for a better robot if needed. Figure 9 gives a visual representation of this setup. The processing times for each of the robot and operation type pairs is shown in Table 2. Multiple state space configurations and reward functions were tested to provide the RL with the correct information to learn how to optimally assign in this expanded state space. These variations included:

- When the scheduler queried the policy (at every time step or only when a robot was free)
- If the state space included direct information about how long robots were working
- If the reward function credited an assignment when it took place or after the jobs were completed

The great majority of these formulations did not successfully converge to a learned policy which will be discussed in the next section. This work will discuss the results from a base configuration[8] and an expanded configuration that performed the best. Both of these configurations had a state space vector consisting of what jobs the robots were working on, what jobs were in the queue, how long those jobs had been in the queue, and, when applicable, the distance between those jobs. For both of the configurations, the policy was queried every time step. The base formulation reward function for each time step t was:

$$R_t = -\sum_{i_t}(t - t_{0i}) \tag{5}$$

where i was a given job that completed at t and t_{0i} was the time that job i entered the visible queue. This reward function sought to teach the policy to move jobs through the queue as quickly as possible. The shorter amount of time the job was in the queue and the faster it was completed the better (less negative) the reward would be. The best performing reward function, the expanded formulation, was:

$$R_t = \sum_{j_t}(t_{jw} - t_{jp}) - \sum_{i_t}(t - t_{0i}) \tag{6}$$

where j was a job being assigned at time t and i was a job in the visible queue at time t. For this formulation, the job was taken out of the queue as soon as it was assigned to a robot. The idea of this reward function was to give an immediate positive reward when an assignment was made by subtracting the longest (worst case) a robot could take to complete the job t_{jw} by the actual projected completion for the job t_{jp} based on the robot assigned. This immediate feedback was balanced by if the job it chose had been in the queue long by taking the difference of the current time and how long each job in the queue had been there t_{0i}. After training, the policies were testing on 100 different 50 job projects. Their performance on these test projects were compared against a random assignment policy and a greedy assignment policy. The greedy policy would assign the best robot to the job at a given time step without any consideration of future jobs. The training and testing results will be discussed in the next sections.

Results & Discussion

To evaluate training, there are two graphs that are important. The reward value vs time, which shows the return of the reward value

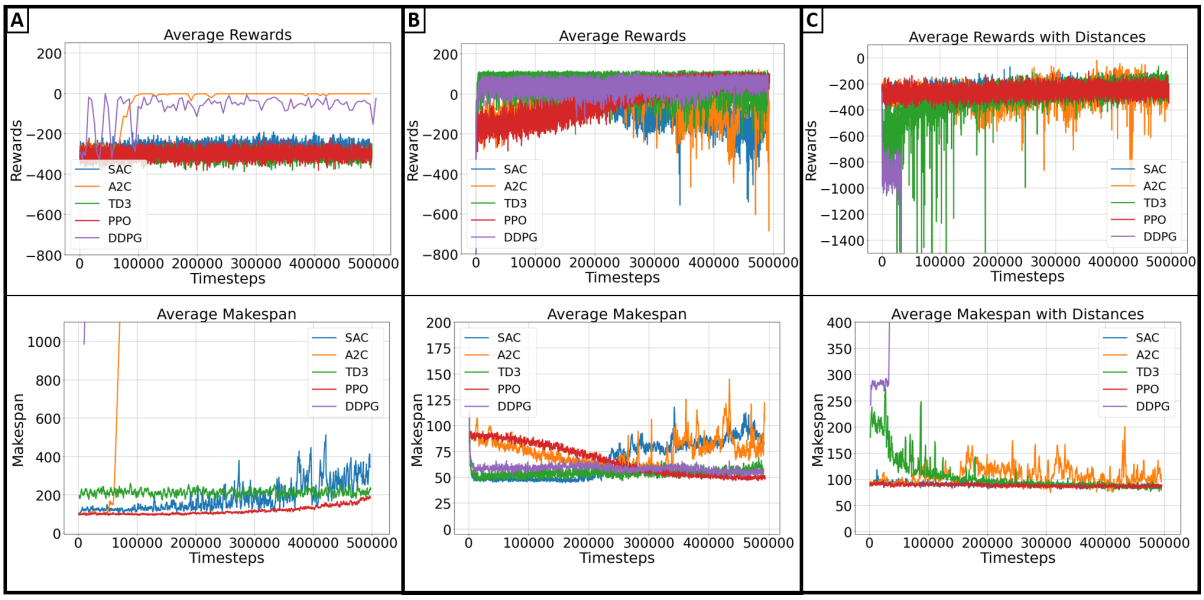

Figure 10: A: Training reward and makespan graphs for base formulation, B: Training reward and makespan graphs for the expanded formulation, C: Training reward and makespan graphs for the expanded formulation with distances

at each time step, and the makespan (how long it takes to complete the training project) vs time. If a model successfully converged to a learned state, the reward values should trend at a slope before leveling out. If the training successfully learned to minimize the makespan and thereby learn to assign the robots in a way that improved the project completion time, the makespan should have a downward trend before converging. Shown in Figure 10, the base formulation was not able to learn even the simple environment containing only processing times considerations. The expanded reward formulation was able to learn for the simple environment but only with the PPO formulation. However, once the distances were added to increase the complexity, even the PPO formulation was no longer able to converge to a useful policy. This is illustrated in Figure 11 where each of the expanded formulation models is compared against a greedy and random assignment policy. As shown, most of the other RL policies performed worse then guessing randomly. For some of these models, it appears from the makespan graphs that they start to learn the correct way to assign but then they to diverge. This points to a need for more work to be done in developing a better reward function. In the expanded form, the two parts in equation 6 balancing local impact vs future impact can become unbalanced depending on the size of the time differences. In the next section, options for mitigating this and ways for handling the increasing complexity of the state information will be discussed.

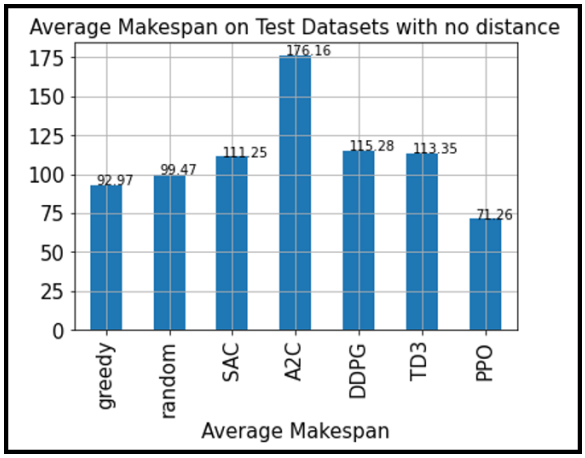

Figure 11: Average makespan from 100 test projects using the expanded formulation

Future Work

Future work with the SPD will include testing its use with other schedule generation methods. One such method of interest is a mixed integer programming formulation which is often used for job shop type problems. In the context of a reinforcement learning based solution generation methods. Future work will continue to develop the reward function. A possible addition to preserve the balance of current and future affects would include the introduction of weighting factors on the two parts of the reward. Another alternative is splitting decision making across multiple policies. It is possible to have the different policies "focus" on different factors contributing to the reward. This, combined with additional models to embed the different complexities in the state space may provide a better feedback loop on how the actions affect the current and future state of the assembly.

Acknowledgements

Special thanks to Paolo Fermin for his assistance with the reinforcement learning experiments.

References

[1] Joshua Moser, Anderson Matthew, Holly Everson, Amy Quartaro, William D. Chapin, Benjamin Beach, Julia Hoffman, Robert Hildebrand, and Erik E. Komendera. Recent Developments in Robust, Accurate Autonomous Assembly Methods for Surface and Orbital Structures. In *ASCEND 2020*, Virtual Event, November 2020. American Institute of Aeronautics and Astronautics.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, June 2016. arXiv: 1602.01783.

[4] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, July 2019. arXiv: 1509.02971.

[5] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, August 2018. arXiv: 1801.01290.

[6] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. *arXiv:1802.09477 [cs, stat]*, October 2018. arXiv: 1802.09477.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, August 2017. arXiv: 1707.06347.

[8] Joshua Moser, John Cooper, James Neilan, Liam Chapin, Samantha Glassner, and Erik Komendera. A Reinforcement Learning Approach for the Autonomous Assembly of In-Space Habitats and Infrastructures in Uncertain Environments. In *70th International Astronautical Congress*, page 11, October 2019.