Results of the Airlift Challenge: A Multi-Agent AI Planning Competition

Adis Delanovic[a], Carmen Chiu[a], John Kolen[b], Abeynaya Gnanasekaran[c], Amit Surana[c], Kunal Srivastava[c], Hongyu (Alice) Zhu[c], Yiqing Lin[c], Norman Bukingolts[d], Devin Willis[d], Nickolas Arustamyan[d], Adam Sardouk[d], Matthew Huynh[d], Dali Grimaux-De camps[d], Kaleb Smith[e], David Bragg[d], Andre Beckus*[a]

[a] Air Force Research Laboratory, Information Directorate, Rome, NY 13441, USA
[b] Independent Researcher
[c] Raytheon Technologies Research Center
[d] University of Florida, Gainesville, FL, 32611, USA
[e] NVIDIA AI Tech Center

ABSTRACT

Planning the delivery of cargo as part of an airlift operation is a notoriously complex problem. Transportation routes can suddenly become inaccessible due to poor weather or other unexpected occurrences. Factors such as airplane speed, carrying capacity, and airport maximum-on-ground must also be considered to ensure on-time and efficient delivery. Unforeseen cargo needs may require quick re-planning to meet tight deadlines.

To address the problem, we held the Airlift Challenge, an online multi-agent planning competition which concluded in early 2023. Competitors in the Airlift Challenge created innovative algorithms to execute a simulated airlift operation. The algorithms were scored against a set of increasingly complex evaluation scenarios while contending with unexpected events and disruptions.

In this paper, we describe the competition and simulation environment, summarize results, and include write-ups of the top approaches provided by the winning teams.

Keywords: Airlift Planning, Multi-agent Systems, Competitions, Machine Learning, Artificial Intelligence, Reinforcement Learning, Optimization, OpenAI Gym

1. INTRODUCTION

The airlift cargo problem consists of moving large amounts of cargo between airports in the most efficient way possible. The cargo must be delivered in a timely manner, but airlift planners must contend with limits on fuel and airport capacity, as well as a changing environment where certain routes may be delayed or unavailable. The optimization goal is to deliver all cargo on time while finding the lowest-cost way of doing so. However, cargo delivery faces an ever-evolving environment that includes possible delays, various bottlenecks, and different disruptions. As a result, one of the biggest challenges is recalculating routes quickly to avoid any delay in deliveries.

Airlift planning has roots in such problems as the dynamic vehicle routing problem with pickup and deliveries (VRPPD) [1] and the pickup and delivery problem with time windows (PDPTW) [2]. Often, these problems are solved using mixed integer linear programming, but methods such as reinforcement learning are showing promising results recently.

The real-world environment is complex, and each factor added to a problem increases the difficulty and time required to find a proper solution. Simplifying the environment allows faster iterations and removes unnecessary detail. Building upon our previous work of “flattening” the Command & Control problem space [3], we have created an open-source simulation environment that aids researchers in creating air cargo planning algorithms.

The paper is organized as follows. Section 2 covers a general overview of the simulator. Section 3 covers more about the recent competition that was held from January 23rd to March 1st, along with results. Section 4, 5 and 6 include algorithm descriptions written by the two winning teams, as well as the honorable mention team. Section 7 covers the conclusion and acknowledgements from the airlift challenge development team. Some of the material in this paper was derived from the documentation hosted on the competition website [4].

2. APPROACH

2.1 Simplification Methodology

We focused on adding in the most important features that create common problems with air cargo delivery while adding in a simplified interface via PettingZoo [5] and OpenAI Gym [6]. A simplified and familiar interface helps abstract air cargo terminology away for machine learning researchers. This allows for a wider range of individuals to explore and create solutions. By including only the most important features in the simulator we allow researchers to focus on creating a solution that would not be skewed by less important factors. In addition, a lightweight simulator allows for easier changes in the environment. In conjunction with the development of our simulator we are in the process hosting a series of international competitions.

We were inspired by competitions such as the Flatlands competition [7], which provided a simulator for train routing planning. There is a rich academic knowledgebase outside of the DoD and its contractors, and international competitions relying on open-source code may be a way to tap into that.

2.2 Airlift Environment

The airlift environment is a discrete time-step simulator. All scenarios are random in regards to dynamic events occurring, placement of airports, routes, cargo and agents. It is built using a graph-based network. Each node is a capacity-constrained airport with a processing time. The edges between those nodes are routes associated with cost and flight time. These edges can become randomly unavailable making it impossible for an airplane to traverse for a random duration. If an airport is at capacity, the airplane will be allowed to land; however, they will be placed into a queue for processing and will have to wait for an extended duration. Cargo is distributed uniformly amongst the airports that are in the designated pick-up area and must be delivered to airports located within the designated drop-off area. Each piece of cargo has two delivery deadlines: a soft deadline by which the cargo is desired, and a hard deadline after which the delivery is considered missed. There is a heavy penalty for missing the hard deadline.

The simulation also allows for modeling discrete airplane types. Disparate airplane models have separate route networks. The network for a specific model may in-fact be disconnected, and not allow the airplane to reach all airports. These airplane models are also limited by range, speed and carrying capacity. Time for processing is also needed after landing in order to refuel and to load or unload cargo. All of these must be taken into consideration when creating a plan to deliver cargo from one airport to another. An important feature of our environment is simulation of finite working capacity at each airport. Routing too much traffic through a specific node could lead to the creation of a bottleneck.

In Figure 1, a small example scenario is shown. Airports (small squares) are shown with connecting routes (white lines). Cargo is staged at three airports in the pickup area (green rectangle). Each is designated for delivery at a specific airport in the area of need (yellow circle). The agent algorithm guides four airplanes through the network to pick up and deliver the cargo. Routes undergo random disruptions, requiring airplanes to either wait for the disruption to clear, or follow a different route [4].

Figure 1 Visualization of the Airlift Environment

2.3 Interface

The Airlift environment is created using Python. Developer tools such as PettingZoo and OpenAI Gym lay the framework for the simulator. Each airplane takes an action picked from a small selection of possible interactions. The agents can load cargo, unload cargo, move from one node to another using an available edge or take no action. At each time step the agent receives an observation from the environment. This includes the status of the airplane, route availability, current location, cargo locations and indication of new cargo that was generated. The agent can then decide on their next action based on this observation.

- **Load Cargo:** Load a list of cargo specified by id.
- **Unload Cargo:** Unload a list of cargo specified by id.
- **Travel:** Move to another node
- **No Action:** Take no action

The agent can receive multiple actions in one step and the agent will complete the list of orders. Late or missed deliveries, as well as agents moving between airports, are penalized by the environments reward system. Agents are encouraged to only take the most effective routes in order to diminish the effect of this. Figure 2 shows the minimum code requirement for a solution to work. The policy function defines the algorithm for the agents' actions. In this case we are using the built in random agent that is selecting a legal action from a set of valid actions. The policy is then run in a loop until either the max steps allowed is reached or all agents are done.

```
from airlift.envs import AirliftEnv, AirliftWorldGenerator, ActionHelper

# Agent algorithm goes here
def policy(obs):
    actions = ActionHelper.sample_valid_actions(obs)
    return actions

env = AirliftEnv(AirliftWorldGenerator())
obs = env.reset()
while True:
    actions = policy(obs)
    obs, rewards, dones, infos = env.step(actions)
    env.render()
    if all(dones.values()):
        break
```

Figure 2 Minimal Agent Code

2.5 Dynamic Events

During the execution of a scenario several dynamic events can occur. All of these events are randomly generated using the Poisson process. One of these events is that routes can become unavailable for a set duration. The duration that the route is unavailable for is generated uniformly at random within a given range. This range is a parameter set prior to the creation of the scenario and can vary between different scenarios. Agents are able to observe the end time of when the route will become available again.

Another event is the generation of new cargo orders. New cargo can become available at any point throughout a scenario. These cargo orders are placed in the pick-up zone and are given deadlines based on the time-step they were created at. Agents are able to observe if a new cargo order was generated.

2.6 Generators

The airlift environment uses several generators to create scenarios that are each distinct from one another. These are the map, airport, route, cargo, and dynamic event generators. By using different seeds for the environment, we can create a specific scenario that is unique to that seed. For the competition we created a set of hidden scenarios with increasing difficulty. We verified that the difficulty of each scenario scales up with the number of airplanes, airports, cargo, and the rate at which dynamic events are generated. This determination was made by scoring the baseline shortest path algorithm against the random agent algorithm over a large set of scenarios.

- **Maps:** The map is randomly generated using Perlin noise and contains land masses surrounded by bodies of water. The airports are constrained to land only and are placed uniformly. Options are also available for placing airports on a grid for debugging purposes.
- **Airports:** Airports are nodes where the agents can be processed. However, the processing capacity is limited to a certain number of planes and all other planes must wait. This creates a potential source of bottlenecks. Airports can be part of pickup or drop-off zones for cargo.
- **Routes:** The routes are represented by the edges of the graph. The edges are generated according to each airplane models' maximum range. To encourage sparsity, a subset of the possible routes are chosen to be generated. If the resulting graph contains multiple connected components, edges will be added until the graph is fully connect.
- **Cargo:** There is an initial set of cargo orders generated on simulation start. Additional dynamic cargo is generated throughout the simulation level. Each cargo order has a source, destination and soft and hard deadlines for each piece of cargo.

2.7 Evaluation

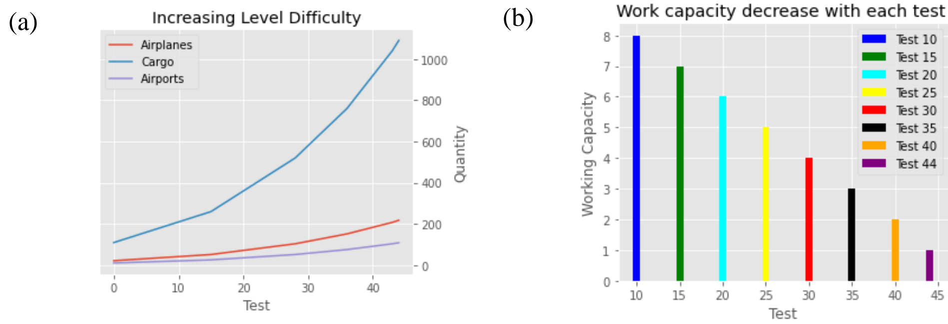
Algorithms are evaluated against a series of Tests which become progressively more difficult with respect to "static" parameters.

Figure 3(a) shows how the values for the number of agents, max cargo per episode and number of airports was tuned for all the levels in the hidden set of scenarios. For the competition phase, the first scenario had 22 agents, 110 cargo, 11 airports, with no dynamic events occurring and the final scenario had 218 agents, 1123 cargo, 109 airports and a higher probability of dynamic events occurring. In Figure 3(b) we also observe how much the working capacity decreases over Tests. At Test 10 each airport has a capacity to process 8 airplanes, but by Test 45 this goes down drastically to only 1.

In Figure 3(c) we observe the increase of how likely any route may become unavailable as well as the range (in steps) that it can be unavailable for in Figure 3(d). The more environments that a competitor's algorithm can go through, the higher the rate of route disruption, and the longer the route may be offline for. In Figure 3(e) we observe how many dynamic cargo are generated at specific test and levels. Dynamic cargo generation rate is slowly increased throughout both tests as well as levels.

Although scenarios are randomly generated, they are initialized with the same seed for every evaluation. Therefore, every participant is presented with the same realization.

Static Parameters



Dynamic Parameters

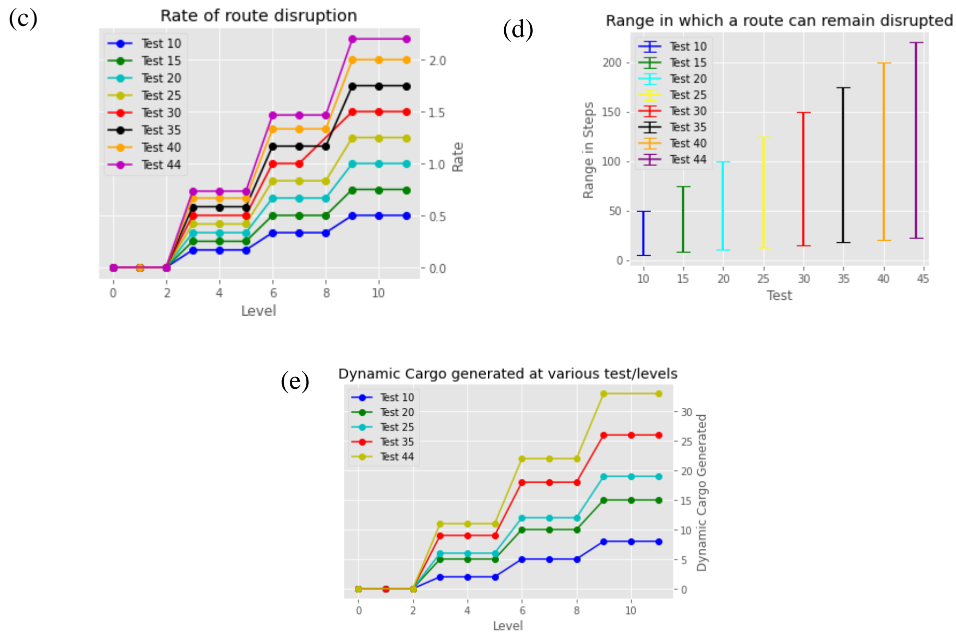


Figure 3 Scaling of difficulty in evaluation scenarios

Within each Test, the algorithms are presented with a series of Levels with increasing frequency and duration of dynamic disruptions. Evaluations ends when all tests are completed or either of the following stopping criterion are met:

- The percentage of missed deliveries exceeds 30% (averaged over the levels within a given Test).
- An overall evaluation time of three hours is reached.

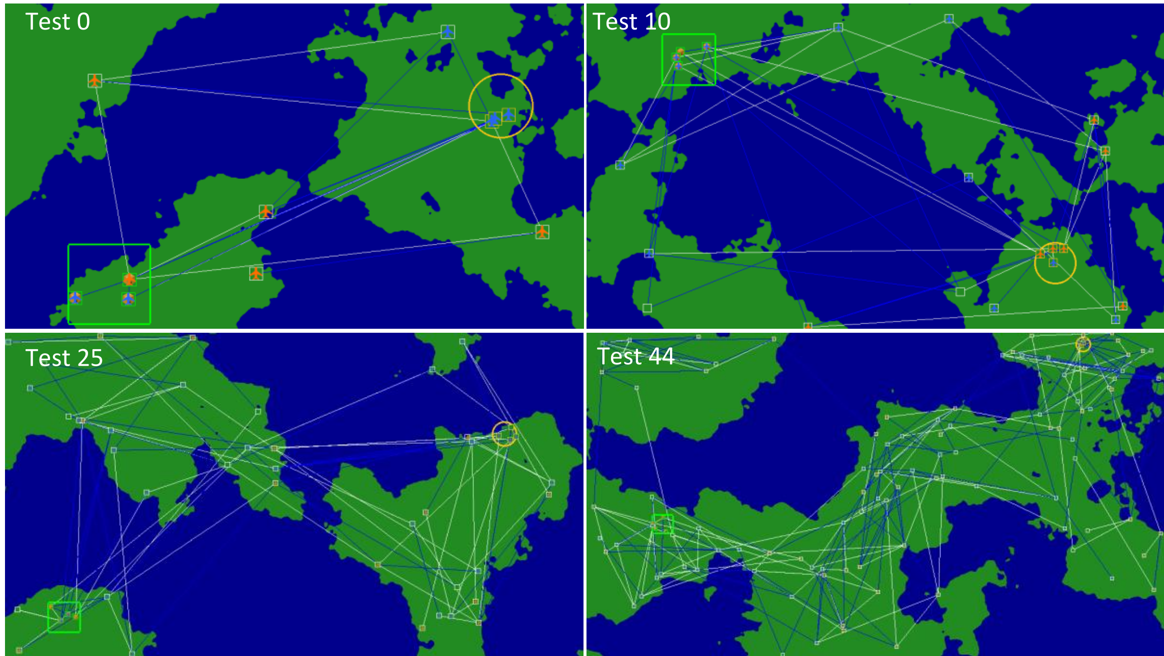


Figure 4 Scenario Examples

In Figure 4 we can see examples of scenarios from various tests that were used in the evaluation. We observe the starting scenario in the top-left labeled as "Test 0" and subsequently we also see what the starting scenario looked like for the final test in the bottom right labeled "Test 44". In all cases, we show level 0 of the corresponding tests. The steady increase in scale provides competitors a diverse set of scenarios to test their solutions on.

2.8 Scoring

An important factor in our simulation is the delivery deadline. We have a "hard deadline" after which a cargo is considered completely missed. There is also a "soft deadline" in which there is no lateness penalty applied if it is delivered by this time. Figure 5 shows the delivery window for any cargo to be considered on-time, late or missed.

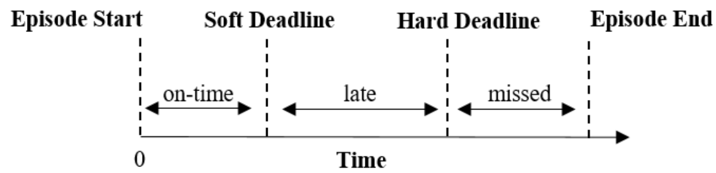


Figure 5 Delivery time windows

Each episode is assigned a score based on missed deliveries, late deliveries, and total flight cost[1]. This score is based on three raw metrics:

[1] This scoring information was made available to competitors on our website and is part of our documentation.

$$missed = \sum_{c \in \mathbf{C}} \mathbf{I}(actualdelivery_c > deadline_c) \ (1)$$

$$lateness_c = \max\{0, actualdelivery_c - \ targetdelivery_c\} * \mathbf{I}(actualdelivery_c \ \leq \ deadline_c) \ (2)$$

$$flightcost_p = \sum_{r \in \mathbf{R}} \left(routecost_{rp} * numflights_{rp}\right) \ (3)$$

for which $\mathbf{I}(\cdot)$ is the indicator function, $actualdelivery_c$ is the actual time (in discrete time steps) for when the cargo reached its delivery location (or ∞ if the delivery was not completed), $routecost_{rp}$ is the cost of plane p to fly route r, and $numflights_{rp}$, is the number of flights by plane p over route r. For cargo c, the hard deadline is represented by $deadline_c$, and the soft deadline is represented by $targetdelivery_c$.

The number of missed deliveries is captured by (1). For deliveries that are not missed, (2) indicates the amount of time by which these deliveries miss the soft delivery deadline. The cost incurred by airplane p flying routes during the course of the episode is identified by (3).

In order to encourage more uniformity across scenarios, we derive two scaled metrics. First, we scale the lateness so that its value will range between 0 and 1:

$$scaledlateness = \sum_{c \in \mathbf{C}} \frac{lateness_c}{deadline_c - \ targetdelivery_c} \ (4)$$

Second, we scale the flight costs of each airplane against the diameter of the route map graph, the number of cargo generated, the total cargo, and the capacity of the airplane. Specifically, the total scaled flight cost over all planes is defined as

$$scaledflightcost = \sum_{p \in \mathbf{P}} \frac{flightcost_p * \ weightcapacity_p}{diameter_p * \ totalcargo} \ (5).$$

When assigning an episode score, we use a large weight α to give missed deliveries the largest penalty, and then give a smaller penalty to lateness using a smaller weight β. Flight cost is penalized by γ. Then, the objective is to minimize the episode score

$$EpisodeScore = \ \alpha * missed + \ \beta * scaledlateness + \ \gamma * scaledflightcost \ (6)$$

To determine an overall score for the submission, we find it useful to normalize against baseline scores and produce a score that increases with improved performance. For each scenario, we calculate episode scores RandomScore and BaselineScore for the Random Agent and "shortest path" baseline algorithm, respectively. Then given a episode score for a solution, we calculate a normalized score as

$$NormalizedScore = \frac{RandomScore - EpisodeScore}{RandomScore - BaselineScore} \ (7)$$

A solution will receive a normalized score of 0 if it only performs as well as a random agent and will receive a score of 1 if it performs as well as a simple "shortest path" baseline algorithm. Scores greater than 1 indicate that the algorithm is exceeding the performance of the baselines. Note that it is possible to obtain negative normalized scores if the agent performs worse than the random agent.

Finally, an overall score is calculated by summing the normalized score over all tests/levels.

$$OverallScore = \sum_{i=1}^{\# of\ completed\ tests} \sum_{j=1}^{\# levels} NormalizedScore(i,j) \quad (8)$$

3. COMPETITION

The competition began on Jan 23rd, 2023, and ended on March 1st, 2023. The competition consisted of a warmup phase and a competition phase. The warmup phase allowed competitors to familiarize themselves with the environment and starter kit while asking questions. Competitors were free to utilize a large range of methods such as machine learning, heuristics, optimization or anything they could come up with to solve environments that become more difficult the further they progressed.

In total the first iteration of the competition had 27 registered users with three active teams.

The competition was hosted on CodaLab and all information concerning it can be found on the website at: https://airliftchallenge.com/

3.1 Results

The winner of the competition was John Kolen who submitted under username "jkolen". In second place was "Raytheon Technologies AlphaLift" from the Raytheon Technologies Research Center. We will often refer to the latter as "AlphaLift". Their team members included Abeynaya Gnanasekaran, Amit Surana, Kunal Srivastava, Hongyu (Alice) Zhu, and Yiqing Lin. Both winners accomplished very long runs by keeping a low percentage of missed deliveries well below the threshold of 30% for the duration of evaluation. In both cases, evaluation was stopped automatically after reaching the maximum allowed time of three hours.

An honorable mention goes to team "Gator Lift" from the University of Florida, which was comprised of Nickolas Arustamyan, Norman Bukingolts, Dali Grimaux-De camps, Matthew Huynh, Adam Sardouk, and Devin Willis. Their advisors were David Bragg; Florida Applied Research in Engineering (FLARE) and Dr. Kaleb Smith; Senior Data Scientist, NVIDIA AI Tech Center. The teams' approach was made to be run on a local evaluator due to its use of the NVIDIA AI-Powered optimization tool "cuOpt". Due to this approach the team was unable to run their solutions successfully when using our remote evaluator services.

Following the results, we include the write ups from the three teams jkolen, Raytheon Technologies AlphaLift, and University of Florida GatorLift.

Table 1 shows the top two rankings, number of tests and scenarios complete, as well as their top scores.

Table 1 Top two finishers and number of environments complete

Rank	Competitor Name	Score	Tests Complete	Scenarios Complete
1	John Kolen (jkolen)	980.688	35	429
2	Raytheon Technologies AlphaLift	927.587	34	422

In Figure 6 we observe the progress made by the winners during the course of the competition. Dots indicate individual submission scores (up to three submissions were allowed per day), while the maximum daily score is shown as a line. jkolen achieves the highest score of the competition and plateaus on Feb. 12th. The next day team AlphaLift makes their first submission and steadily improves the algorithm, obtaining their best score on Feb. 27th. Both teams consistently exceed the baseline score except for a few low outliers.

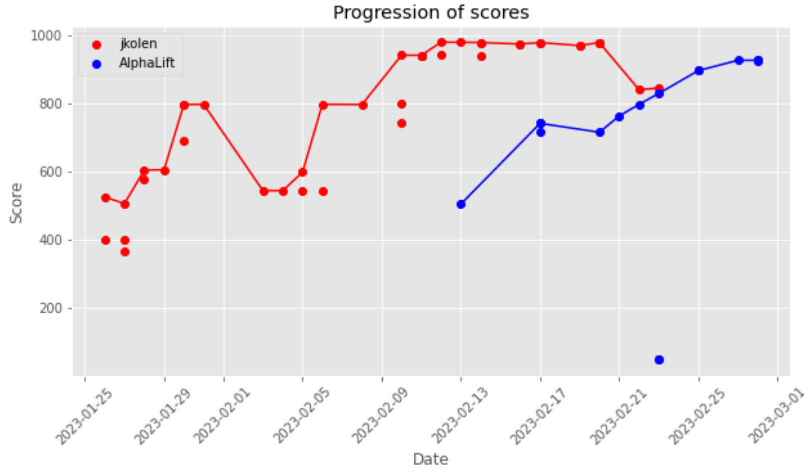


Figure 6 Progression of Scores during competition

Figure 7(a) shows the score accumulation as the evaluation progresses through each test, i.e., for Test k we show $\sum_{i=1}^{k} \sum_{j=1}^{\#\,levels} NormalizedScore(i,j)$. The algorithms of jkolen and AlphaLift have similar scores until the end of evaluation. To highlight the difference at the end, we show jkolen's lead (i.e., the difference between the accumulated scores of jkolen and AlphaLift) in Figure 7(b). This lead is < 1 until Test 29. The sharpest increase is in the last test owing to the fact that jkolen's algorithm runs slightly faster and finishes 35 tests, whereas AlphaLift only completes 34 (they complete some levels in Test 35, but receive a score of zero for that test due to incompletion). For comparison, we show the Baseline score, which obtains a score of approximately 1 for each scenario.

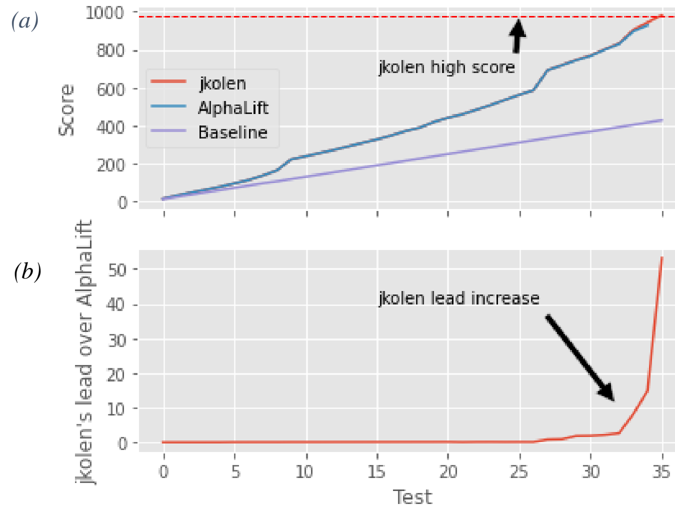


Figure 7 Score breakdown

In Figure 8 Metric breakdown we show metrics which impact the score. Figure 8(a) shows fraction of missed deliveries averaged over the levels in each test. As stated earlier, both winners fall well under the stopping threshold on missed deliveries, with jkolen having especially small misses throughout evaluation. In contrast the baseline exceeds this threshold at Test 19 and the random agent always exceeds it. Figure 8(b) shows the average missed deliveries for different dynamic parameterizations. The leftmost datapoint shows the average value for levels 0-2 over all tests, i.e., we show $\frac{1}{3T}\sum_{i=1}^{T} \sum_{j=0}^{2} m(i,j)$, where $m(i,j)$ is the fraction of missed deliveries for test i and level j, and T is the number of completed tests. Recall that these levels have no dynamic events. Likewise, we show the average value for levels 3-5,

levels 6-8, and levels 9-11, each of which has an increasing rate of dynamic events. As can be seen, when the number of dynamic events increases in levels 6-11, AlphaLift has a large increase in missed deliveries as compared to jkolen.

In Figure 8(c) and (d) we show similar plots with average lateness per cargo. As with missed deliveries, we see a large increase in the lateness for AlphaLift in later tests. We also see AlphaLift has more lateness across all dynamic parameterizations (including the static case).

Figure 8(e) and (f) show the average flight cost per airplane. The cost of a flight is determined by the Euclidean length of the edge it must traverse. We can observe that jkolen's solution was very efficient in keeping the cost down, indicating it was able to fly less distance while at the same time make more on-time deliveries. As static complexity increases, there is a consistent increase in the gap in costs incurred by jkolen and AlphaLift. We also see a consistent gap across all dynamic parameterizations.

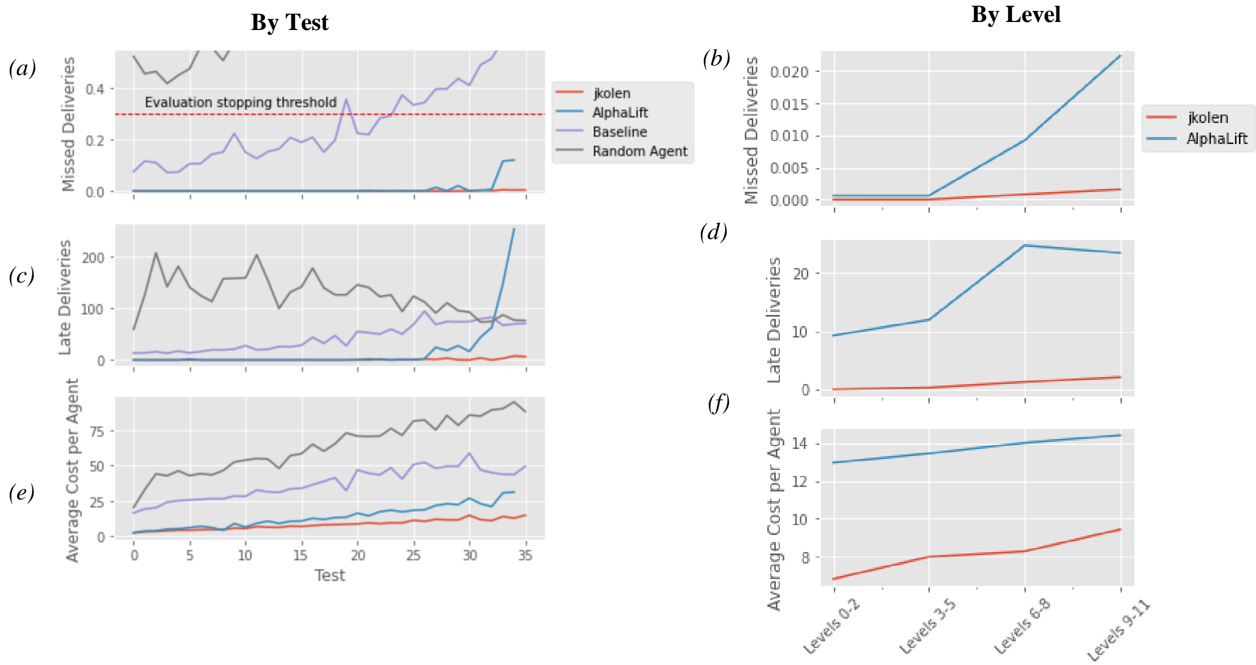


Figure 8 Metric breakdown

In Figure 9(a), we show the average steps required to complete each episode within a test. As expected, we see a general increase in number of steps with more difficult tests. In later tests, AlphaLift requires more tests to complete, which is consistent with the increase in lateness and missed deliveries shown earlier. We note that all algorithms, including the random agent and baseline, complete these episodes before hitting the maximum of 5,000 steps. In Figure 9(b), we show the steps by dynamic parameterization using the same approach as for Figure 8. As expected, there is a clear correlation between frequency of dynamic events and number of steps, with both jkolen and AlphaLift exhibiting similar performance in this regard.

Finally, Figure 9(c) and (d) show the number of steps, on average, that airplanes spend waiting to process at an airport. We can see both winners have an increase in wait time as the tests progress. However, jkolen has much longer wait times

across all tests, and shows a large increase as the number of dynamic events increases.

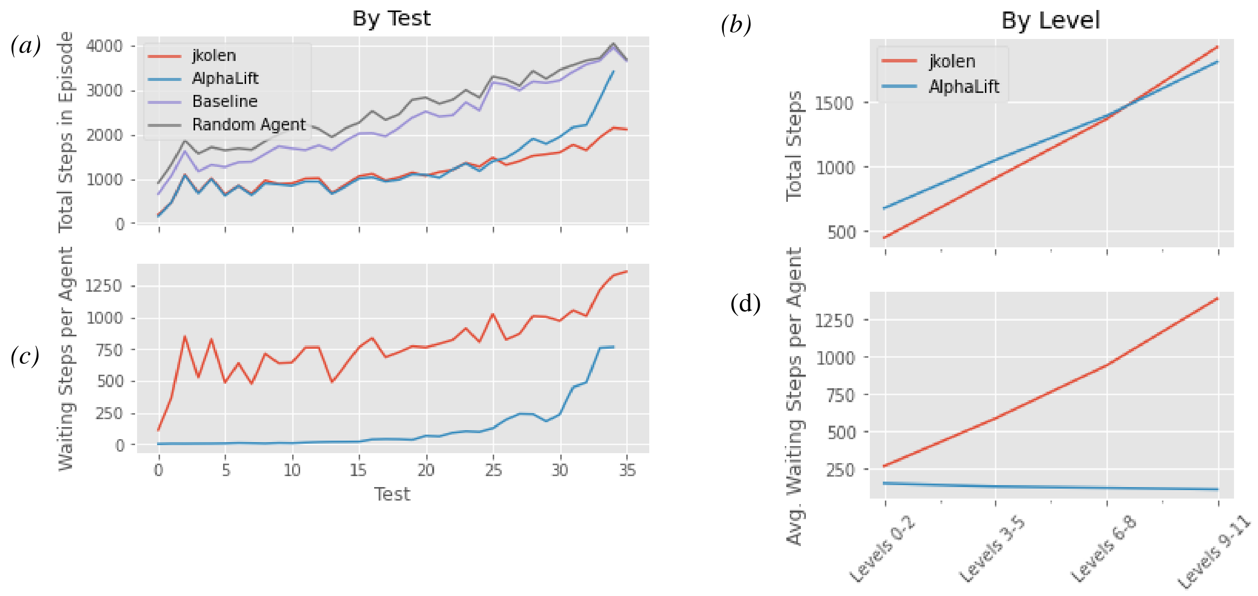


Figure 9 Total episode steps vs. waiting steps

4. WINNING SUBMISSION APPROACH: JOHN KOLEN

4.1 Method

This section describes the structure and methodology behind the winning submission. First, the underlying multiphase architecture to solve the airlift problem is presented. The discussion then turns to addressing the competitive context and how it impacted design choices.

The airlift agent controller used a multilayered approach to produce action sequences. First, cargo items were assigned goal stacks dependent upon current route constraints. Second, a flight planner would generate one or more flight plans for an aircraft that would move cargo toward their goals and other appropriate behaviors. Third, a scheduler would select a flight plan for each aircraft in order to reduce airport congestion. Finally, cargo loading and unloading actions were opportunistically determined given the cargo present at the airport, the current load, and the aircraft's flight plan. These levels of processing separated dependencies allowed for reasonable responses to changing route conditions as well as opportunities for optimizations that still produced satisfactory action sequences.

Each item of cargo appears with an assigned destination. Unfortunately, the route from cargo source to destination requires the coordination of multiple aircraft types. The first phase solves this problem aspect by calculating a stack of goals. The top most goal can be satisfied by a single aircraft. Once that goal is achieved, the next goal on the stack may be reached by another aircraft type. The bottom-most goal is the destination of the cargo item. The intermediary goals are computed from a meta-map. In this map, a layer of nodes represent each airport. Each of the route maps are then connected to this layer with load and unload edges to the corresponding airport nodes. These edges have the airport processing time as their time attribute. Thus, it is possible to request all shortest paths between the current cargo location and its destination. While time would be a useful metric in optimal circumstances, routes were subject to malfunctions. A sum of 1000 times malfunction time and travel time (*malxtime*) were ultimately used in the shortest path calculations. Transitions in aircraft type from these shortest paths are recorded a partial goals in the goal stack. The paths computed during this phase were discarded and only the goal stacks were retained.

The second phase addressed the generation of flight plans that could route aircraft to satisfy cargo goals. The collection of aircraft were passed through a series of behavioral filters that would test for appropriate preconditions and then generate a corresponding flight plan (sequence of airports and actions). Once a behavior had been assigned for the current timestep, it was removed from the collection. The behaviors were

- Load local cargo
  - If the aircraft is at an airport and can load cargo, then load it.
- Deliver cargo
  - If the aircraft has cargo then head toward airports on top of the cargo goal stacks. The flight plan is a shortest path from the current location through the possible cargo goals. The path metric, *malxcost*, is 1000 times the malfunction time plus the fuel cost for the route edge.
- Pickup assigned cargo
  - Head toward the airport where the assigned cargo is located (see pickup unassigned cargo below). The flight plan is a shortest cost path from the current location through the possible cargo goals and the target airport.
- Pickup unassigned cargo
  - Find a nearby reachable unassigned cargo that has the shortest pickup path (see path description above). Assign the aircraft to that cargo and generate a flight plan.
- Pickup unreachable and unassigned cargo
  - Same as pickup unassigned cargo, but release the reachability constraint.
- Unload cargo
  - If the aircraft is at an airport and is carrying cargo with the airport as a goal, unload it. The rest of the flight plan is determined by the remaining cargo as well as potential new cargo.
- Stage for secondary cargo goals and possible new cargo
  - Head for an airport that is either a drop off point for an enroute cargo or one of the possible origin sites were new cargo could appear.
- Refuel
  - Last resort is to refuel if there is nothing else to do.

Once each aircraft has a set of possible flight plans, a single flight plan must be selected for each aircraft. The next phase attempts to eliminate airport congestion across flight plans. A candidate flight plan is selected for each aircraft. Estimated times for airport occupancy are determined. If any airport exceeds its working capacity, one of the aircraft is selected to replace its flight plan. The candidate aircraft are identified and alternate flight plans selected. This process is repeated for a fixed number of attempts or until no constraint violations have been found.

Finally, an action must be identified for each aircraft from its flight plan. Depending upon the aircraft's current simulator state and the next event from the flight plan, the aircraft will load and/or unload, takeoff, or continue on its way. Aircraft loading is a greedy operation where candidate cargo is either assigned to the aircraft or unassigned and has a goal airport along the current flight plan.

The above description outlines the core behavior of the airlift agent controller. While it performed well in terms of avoiding missed cargo, its time requirements restricted the scores it could achieve. To this end, a score rate metric (points per second) was defined and the controller's performance against test scenarios was measured with it. This view allowed for parameter tuning that would lead to increasing the controller's score under the time constraints of the competition. Over thirty parameters were available to tune its performance. These included how often the cargo goals were updated, maximum number and length of flight plan alternatives, sequencing of aircraft for flight plan generation. From these optimization efforts some features were simply adopted. For example, smart time was one of the shortest path metrics that was used to generate cargo goals and flight plans. Smart time incorporated malfunction time delays during the shortest path calculations. Consider a partial path cost of 20 time units and a new edge with malfunction time of 25 and travel time

of 3. Only 5 time units remain of the malfunction time, so the new partial path would have a cost of 28 time units. This metric was very effective in solving the problem, but required too much time to deliver its benefits. The *malxtime* metric described above allowed for higher score rates across all tested route map sizes and was permanently adopted in the controller.

Another approach that had significant competitive impact was to reduce the number of active aircraft–sending them to the hangar. As cargo units reach their destination, the number of aircraft required to deliver the remaining cargo is reduced. The fleet size was controlled by two parameters, minimum size and ratio. Minimum fleet size ranged between 7 and 10 across scenario sizes, while the cargo ratio (fleet size less than ratio times cargo size) ranged from 0.01 to 0.25. Aircraft were not permanently sent to the hangers. If a new cargo appeared in the simulation and there were no active units in its vicinity, the nearest inactive aircraft would be activated. The staging behavior described above deposited aircraft at points where they might be useful in the future. The impact of this change was significant. The time spent processing aircraft flight plans depended upon the decreasing number of cargo units and not the fixed number of aircraft initially in the fleet.

4.2 Conclusion

This submission encapsulated a multipronged approach to winning the airlift competition. First, a multiphase action controller was developed. This controller separated cargo goal identification, aircraft behavioral subgoals, minimizing subgoal interactions, and local opportunistic decisions for delivery and pickup of cargo. Given an action controller capable of addressing the target task, efforts were shifted to its operation in a competitive context. Through parameterization of a wide range of functional controls, it was possible to search for parameters that maximized the system's scoring rate. Many of the parameter selections traded accuracy for time, a rational decision given that time was the scarce resource during the competition.

5. TEAM APPROACH: RAYTHEON TECHNOLOGIES ALPHALIFT

5.1 Method

Following [2], our overall approach is built on decomposing the optimization problem into two subproblems: (1) cargo assignment problem, and (2) single vehicle pickup delivery problem (SVPDP). We formulate the cargo assignment problem as a minimum weight bipartite matching (also known as linear sum assignment) problem and solve it using the Hungarian algorithm [8]. The agents are routed to the cargo pickup and delivery points following the shortest path heuristic. We refine the initial solution computed in this fashion to handle airport congestion, dynamic cargo generation, and edge malfunctioning. Our implementation involved two stages: offline planning and online replanning. The offline stage generates an initial nominal plan before the execution, while the online stage modifies this plan during execution to account for the dynamic cargos and malfunctions.

5.1.1 Offline Planning

The offline planning stage involves an initial planning step and subsequent plan refinement to account for airport congestion. In the initial planning step, we use two simple but effective heuristics for solving the assignment problem and SVPDP, respectively. We further considered two approaches for the assignment problem.

5.1.1.1 Initial planning with agent-cargo assignment

In this approach, the cargo assignment problem is formulated as a minimum weight bipartite matching problem, where the worker is the agent/aircraft and the job is a single cargo pickup/delivery. The edge weights are proportional to the cost of delivering a cargo by an agent which includes the flight cost and delivery delays. If an agent cannot reach the delivery location, an incomplete delivery penalty is added to the edge cost based on how far the closest airport, which the agent can reach, is to the delivery location. We use the well-known Hungarian method to solve the matching problem in polynomial time. Once the assignment is done, an agent is routed to the cargo's pickup and delivery locations by following the shortest path from its current location.

The matching approach discussed above assigns at most a single cargo to each agent. Since, there are more cargos than the #agents, to assign the remaining cargos we employ an iterative approach by formulating successive bipartite matching problems. The edge weights in each such iteration are computed as the "best" cost of inserting the new cargo in the agent's
current route. The agent's existing pickup and delivery order of cargos is kept fixed, and all possible insertions of the new cargo's pickup and delivery order are examined. The pickup/ delivery order with the least cost is selected. These iterations are repeated till all cargos have been assigned or cannot be handled currently due to agent capacity constraints. The cargos not assigned during this initial planning stage are considered as "dynamic cargos" and reconsidered for assignment during execution phase as discussed next in Section 5.1.2.

5.1.1.2 Initial planning with agent-airport assignment

To take advantage of the fact that cargo pickup and delivery nodes are often grouped into zones, we considered an alternative assignment problem where the worker is the agent/aircraft as before, but the job is defined as servicing the airport location. An agent tries to pick up as many cargos as possible in order of increasing deadlines till agent capacity is met and delivers them by following the shortest path heuristic. In this case, the edge weights are proportional to the average cost of delivering the cargos picked up from a particular airport. Since number of cargo pickup locations is typically less than #agents, this process is repeated till all agents are assigned to service an airport. Finally, any remaining cargo is assigned using the iterative approach discussed above.

5.1.1.3 Initial plan refinement for congestion control

In the assignment approaches discussed previously, the edge weights are determined using shortest path heuristic applied to each agent independently and does not consider the coupling effects with other agents. As the airport working capacity is limited, agents can experience a processing delay at congested airports which can impact the delivery times. To reduce such congestion effects, we identify the agents that fly through congested airports and reroute them as follows. The agents' routes are revisited sequentially, and a histogram is populated for each airport based on their arrival time at that airport. Airports are tagged as congested during a specified time interval if the number of agents visiting in that time interval exceeds its working capacity. The remaining agents that fly through the airport during the congested time interval are rerouted through the next shortest path that doesn't include a congested airport. The new route is chosen only if its cost is less than the original cost (including the extra processing time).

5.1.2 Online replanning

We employed a reactive approach to handle dynamic cargo pick/delivery requests and link malfunction events that can happen during execution. At each time step, we check if any new cargo request has been generated and/or if any link has been disabled and replan as discussed below. The agents follow the updated plan till new requests/malfunction events are detected, and the process is repeated.

5.1.2.1 Handling dynamic cargo

To assign new dynamically generated cargos and/or any unassigned cargos from the initial planning stage or previous time steps, we repurpose the iterative approach used in agent-cargo assignment formulation. To compute the edge weights for an agent, we now consider the cargos remaining to be processed at the current time step as the existing cargo list and determine best order to insert the new cargo as discussed above in Section 5.1.1. The iterations are applied till all available cargo requests have been assigned or cannot be currently handled due to agent capacity constraints. Any unassigned cargos are treated as dynamic cargos to be considered for assignment in subsequent time steps.

5.1.2.2 Handling malfunctions

To handle edge malfunction, we considered two approaches. In the first approach, the agents wait for the failed edge to recover and continue following the original plan that was generated offline. In the second approach we first determine if it's necessary to replan due to the malfunction(s) using several criteria and then replan if deemed necessary. The criteria

are based on whether: 1) the edge-malfunction appears on any planned routes of agents; 2) duration for which a malfunctioning edge will be un-available exceeds a prescribed threshold; and 3) the edge unavailability appears within a prescribed number of immediate few flight segments from agent current airport location. Once, the decision to replan is made, we increment the cost of travelling on the malfunctioning edges by a prescribed value and re-compute shortest paths for the affected agents while preserving the prior computed pickup/delivery order. The prescribed value/thresholds were selected by trial and error.

5.2 Results

Our final solution achieved a score of 927 and was able to complete 78% of episodes in the allocated time on the online CodaLab tests. In our first test using the offline solution approach based on agent-cargo assignment, we completed 54% of episodes in the allocated time with a score of 505. After code profiling and more efficient implementation, the same solution approach completed 68% episodes with a score of 742. Switching to the agent-airport assignment approach, we were able to further increase the number of completed episodes to 72% with a score of 798. Finally, with further profiling and improvements to the code and with incorporation of plan refinement with congestion control, we were able to obtain our highest score of 927 and completed 78% of the episodes. Our first approach for handling malfunctions was used in all the test results described here as our second approach turned out to be too expensive. Moreover, our approaches to handle congestion and malfunction provided more improvement in the given offline test scenarios compared to the online CodaLab tests. Further evaluation is needed to characterize this difference in behavior and improve the performance.

6. TEAM APPROACH: GATOR LIFT

6.1 Concept

The Airlift Challenge documentation suggests "machine learning, optimization, path planning heuristics, or any other technique" [4] to be used by its participants to design their airlift operation planning agents. Enter NVIDIA cuOpt: a tool that employs "GPU-accelerated logistics solvers relying on heuristics, metaheuristics, and optimization to calculate complex vehicle-routing-problem (VRP) variants with a wide range of constraints" [9]. NVIDIA provides a Python developer API to interface with cuOpt.

Exploring this API sparked a deep interest in the team to employ cuOpt in its agent's decision-making logic because of significant structural similarities. Conceptually, the tool explicitly focuses on solving problems such as the Airlift Challenge. Practically, the same data structure - a directed, weighted graph - is used by the tool and the competition API to represent a network of traveling vehicles.

Instead of focusing development time on creating a novel algorithm to solve the problem, the team found it an interesting notion and challenge to adapt an existing, industry-approved tool crafted by skilled data scientists at NVIDIA which leverages trained artificial intelligence.

Moreover, the team was fortunate to have access to the University of Florida's research supercomputer, HiPerGator [10] and intended to leverage its incredible computing power and memory to run cuOpt at near-maximum attainable efficiency.

As such, the task at hand became translating the components of the Airlift Challenge into a structure parsable by cuOpt, empowering the tool's computation with a supercomputer's processing power and memory capacity, and then returning the decision made by its algorithms into an executable action for the agent to receive[2].

6.2 Approach

To jumpstart development, team members made use of NVIDIA's available resources to learn cuOpt. Each team member completed a free course called "Optimized Vehicle Routing" [11] that demonstrated cuOpt being used with Python. After

[2] The code is available at: https://github.com/UF-Airlift-Challenge-2023/gator-lift/tree/dynamic-solver

learning how cuOpt intakes information to generate an optimized route, the task at hand became identifying which components of cuOpt matched the competition's model and how they would generate a readable result. Below are disambiguations of cuOpt's Python API classes and functions [12] as they may be adapted to the Airlift Challenge.

6.3 Data Model

Routing

Data Model

Objective

DataModel

…

The *Objective* class serves to identify the target parameter for optimization that will affect how the model is constructed and processed by the solver. It accepts one of seven possible options: number of vehicles, total cost of path traversal, total time, cumulative time an object is in transit, earliest departure time, variance in route size, and variance in route traversal times. The team chose to optimize for total cost because the Airlift model provided pre-calculated weights for its edges representing cost.

The *DataModel* class contains all the vehicle (airplane), package order (cargo), and constraint information. Critically, the resulting object contains many customizable variables representing characteristics like *slack* (waiting time), maximum *lateness* per route, *penalties* for unfulfillment, order *priorities*, time *windows*, delivery *pairs*, *destinations*, *break locations*, *fleet size*, *vehicle types*, and more. Modifying these characteristics within the cuOpt data model is crucial to adapting it to a given problem.

Fundamentally, the data model must feature a representation of the space being optimized over. This includes accessible nodes a plane could stop at as well as the cost of travel between them. cuOpt accepts two possible representations: either a cost matrix where the $(i,j)^{th}$ element of the matrix denotes the cost to travel from airport i to airport j; or a waypoint graph where each node represents an airport, an edge between nodes a connection and hence possible flight path, and the weight associated with an edge the cost to travel between airports. The team used the waypoint graph as this allowed easier adaptation to changing states such as airports closing and let the solver know more clearly which airports had possible flight paths and which didn't.

6.4 Solver Settings

Routing

…

Solver Settings

InitialStrategy

SolutionStrategy

Scope

SolverSettings

…

The *InitialStrategy* and *SolutionStrategy* classes determine the searching algorithms cuOpt uses to solve the path optimization problem. The initial strategy options are iterative insertion (cheapest node at cheapest position) and the Clarke-Wright savings algorithm [13]. The solution strategy options are hill climbing [14], tabu search [15], tabu guided local search [16], or none.

The *Scope* class specifies the solution's result and features two options: feasible (a solution with no penalties), or soft time window (a solution that minimizes cost and lateness, allowing penalties).

The *SolverSettings* class is a modifiable interface encapsulating the above options as related to a particular data model, allowing the developer to get and set associated variables. For example, the number of climbers, number of iterations, error logging mode, time limit, and more may be changed. Selection of a strategy and modification of various settings

encapsulates the decision-making logic for the Airlift Challenge; the team tested various combinations to varying degrees of success during development.

6.5 Output

Routing
    …
    Solve
    Assignment

The *Solve()* function yields the solution to a vehicle routing problem represented by a data model and customized via the solver settings. The result is stored in an *Assignment* object that contains the number of vehicles in the solution, the final total cost, the final value of the optimized objective, the route as a data frame, solver status, any error message(s), and any impossible orders. The solution cuOpt yields is comprehensive, containing an optimized set of routes for every given vehicle. However, the output still needed to be translated into terms parsable by the Airlift Challenge API. This was accomplished by parsing the output and identifying the agents and their target locations for a given iteration to generate a policy.

6.6 Results & Challenges

Code referencing cuOpt was initially written to run locally, aided by a remotely-accessible instance of HiPerGator that had the necessary packages and dependencies for it installed. To meet cuOpt's high computational demands, the team's first major hurdle was simply getting the program loaded and accessible for testing. The team quickly began drafting an adaptation of cuOpt's data model and made extensive use of its modifiable nature. Getting cuOpt to provide a solution required substantial data formatting to convert from the Airlift Challenge's configuration to one that the solver could understand and use. However, upon running code which used cuOpt's output to inform the policies function within Airlift's *Solution* class, a major bottleneck was quickly discovered: memory capacity.

Crafting a data model for cuOpt's solver requires iteration through variably-sized elements from the entire environment - every given node, airplane, and route. The time complexity of this process is far from trivial and must be factored into any evaluation of the solution's efficiency.

Furthermore, cuOpt's solution output contains all of the routes needed to be traversed by every given vehicle over a particular set of connected nodes. In other words, the entire scenario is processed and solved upon execution of cuOpt's *Solve()* function.

The Airlift Challenge testing suite involves iterating through every stage of a solution for a given scenario. Crucially, there is a random chance for change to dynamically occur (ex. *dynamic_cargo_generation_rate*) with every iteration, to which the agent's decision-making algorithm must adapt.

To address this dynamic change, the code was designed to run cuOpt's solution algorithm every iteration which caused buffer overflow; cuOpt was allocating more memory resources than were allotted to the team with HiPerGator. Similarly, any submission which referenced cuOpt attempted via the docker container failed. As such, no solution submitted yielded significant results using cuOpt.

Thankfully, these issues are entirely fixable. It was clear that cuOpt's solver should have only been used once at the start and every time a dynamic change to the scenario was detected, rather than at every iteration. Specifically, once detection of a relevant change to the state was implemented, then the solver could be re-run only when needed - thus allowing for a greater performance. Additionally, more optimal extraction of a given scenario's environment data as well as further configuration of the solver settings would have likely yielded a faster solution.

6.7 Conclusions

The advantage of using a ready-made algorithm for an optimization use case is that the decision-making logic is abstracted from the implementer. This allows for the implementer to customize their own specific constraint and objectives for any domain-based problem they are attempting to optimize. However, the disadvantage is that even in the case of very similar problem structures, the ready-made solution must be modified and tested very thoroughly. And in order for a tool's implementation to yield valid and optimal results for a target project, the tool itself must be very well understood. There is no "one size fits all" - rather, "a tailor can make your garment fit better".

That said, companies or agencies in search of an efficient vehicle routing optimizer will find technologies like NVIDIA's cuOpt capable of catering to most of their needs. Many industries have a broad range of optimization problems, and a plug-in-based architecture enabled by such tools can allow developers to integrate algorithms based on their use-cases. To achieve more accurate and efficient solutions, developers can modify the tool with domain-specific knowledge, custom algorithms, and heuristics tailored to their unique requirements. Businesses can harness the power of GPU-accelerated optimization for many applications: vehicle routing, supply chain management, workforce scheduling, and more. Moreover, integrating machine learning, artificial intelligence, and data analytics enables using valuable insights from various data sources to inform the optimization process.

For example, a company with strict environmental regulations might want to create a custom plug-in algorithm that attributes both the travel distance and CO_2 emissions when considering their optimization. The company can then create a plugin that utilizes their subject-matter expertise in the company's operation and the local environmental regulations to create an effective and optimal approach that adjusts to their specific use cases.

For cuOpt specifically, NVIDIA has ready-made Jupyter Notebooks containing application scenarios for various routing problems [17]. In fact, cuOpt has demonstrated impressive results, enabling packages to be delivered 120X faster with the same accuracy and achieving a world-record error gap of 2.98% [9] on the Gehring and Homberger benchmark. It is evident that this comprehensive approach to optimization allows companies to enhance decision-making, reduce operational costs, and improve overall performance.

7. ENDING THE FIRST AIRLIFT CHALLENGE

This was the first competition that we had the opportunity to put out to the public. We were excited to see the diligence of the participants and the high quality submissions they made. There are many lessons learned that we can carry forward into future iterations of the competition, with the goal of improving the model, software, diagnostic tools, and metrics collection. For example, in future competitions we will consider adding other events, like malfunctioning airplanes. This would require immediate re-routing of the airplane and take them offline for a specified duration.

One particular feature we plan to improve is the evaluation scenario set. There are many parameters that can be tuned when scaling the complexity of the scenarios, and care must be taken to strike a balance between accessibility and difficulty. On one side of the coin, we want a diverse set of scenarios, including some easier scenarios that are within reach of simple algorithms. We do not want to discourage participation. On the other side of the coin, we wish to include challenging scenarios which push algorithms to their limits in later tests.

We also hope to encourage more machine learning submissions in the future. Participants identified some difficulties with such approaches which we hope to address in the future:

- Incompatibility between the simulator interface and the RLlib reinforcement learning library [17].
- Each scenario has an entirely different placement of airports and routes, requiring machine learning algorithms to be of a very general nature. Limiting variation to the air network among scenarios could lower the barrier to entry for machine learning algorithms, and better mimic real-world conditions.

7.1 Links

Airlift Starter Kit - https://github.com/airlift-challenge/airlift-starter-kit/releases/tag/v1.0.1

Airlift Challenge Environment - https://github.com/airlift-challenge/airlift/releases/tag/v1.0.2

Airlift Documentation – https://airliftchallenge.com

University of Florida, “Gator Lift” - https://github.com/UF-Airlift-Challenge-2023/gator-lift/tree/dynamic-solver

Acknowledgements

We extend a huge thank you to John Kolen, the Raytheon Technologies AlphaLift team, students from the University of Florida as well as their advisors not only for participating in this challenge, but also for providing detailed and constructive feedback. Without your valuable participation this challenge would not have been considered a success, and for that we are very grateful.

We thank Jill Platts for setting up infrastructure (including email account, website, web domain name, and github accounts), sending participant communications, as well as many helpful discussions throughout the conception, design, and execution of the challenge.

We thank SPIE for giving us a venue to host our first airlift challenge competition.

Finally, the structure and technical details of our competition were heavily influences by the Flatland challenge train routing competition. We are grateful to the companies and people that created this great open source environment.

The views expressed are those of the authors and do not reflect the official guidance or position of the United States Government, the Department of Defense, the United States Air Force or the United States Space Force

8. REFERENCES

[1] G. Berbeglia, G. Laport and J.-F. Cordeau, "Dynamic pickup and delivery problems," *European Journal of Operational Research,* vol. 202, no. 1, pp. 8-15, 2010.

[2] D. Bertsimas, A. Chang, V. V. Mišić and N. Mundru, "The Airlift Planning Problem," *Transportation Science,* pp. 773-795, 2019.

[3] A. Beckus, A. Delanovic, J. Platts, A. Loy and C. Chiu, "A methodology for flattening the command and control problem space," in *Proc. SPIE 12113, Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications IV*, Orlando, 2022.

[4] A. Beckus, C. Chiu, A. Delanovic and J. Platts, "Airlift Challenge," 19 October 2022. [Online]. Available: https://airliftchallenge.com/SPIE2023. [Accessed 16 March 2023].

[5] J. K. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sullivan, L. Santos, R. Perez, C. Horsch, C. Dieffendahl, N. L. Williams, Y. Lokesh and P. Ravi, "PettingZoo: Gym for Multi-Agent Reinforcement Learning," *Advances in Neural Information Processing Systems.,* vol. 34, 6 Dec 2021.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "OpenAI Gym," *CoRR,* 2016.

[7] AICrowd, "Flatland Challenge: Multi Agent Reinforcement Learning on Trains," [Online]. Available: https://www.aicrowd.com/challenges/flatland-challenge. [Accessed 4 February 2022].

[8] H. W. Kuhn, "Variants of the Hungarian method for assignment problems," *Naval Research Logistics Quarterly,* vol. 3, pp. 253-258, 1956.

[9] Nvidia, "NVIDIA CuOpt: Accelerated AI Logistics and Route Optimization," [Online]. Available: https://developer.nvidia.com/cuopt-logistics-optimization. [Accessed 20 March 2023].

[10] University of Florida, "HiPerGator," [Online]. Available: https://www.rc.ufl.edu/about/hipergator/. [Accessed 20 March 2023].

[11] Nvidia, "Optimized Vehicle Routing," [Online]. Available: https://courses.nvidia.com/courses/course-v1:DLI+T-FX-05+V1/. [Accessed 20 March 2023].

[12] Nvidia, "Nvidia CuOpt Python API Reference," 2022. [Online]. Available: https://docs.nvidia.com/cuopt/py_api.html. [Accessed 20 March 2023].

[13] Neo: Networking and Emerging Optimization, "Savings Algorithm," 7 January 2013. [Online]. Available: https://neo.lcc.uma.es/vrp/solution-methods/heuristics/savings-algorithms/).. [Accessed 20 March 2023].

[14] O. Mbaabu, "Understanding Hill Climbing Algorithm in Artificial Intelligence," 16 December 2020. [Online]. Available: https://www.section.io/engineering-education/understanding-hill-climbing-in-ai/. [Accessed 20 March 2023].

[15] F. Liang, "Optimization Techniques - Tabu Search," 26 July 2020. [Online]. Available: https://towardsdatascience.com/optimization-techniques-tabu-search-36f197ef8e25. [Accessed 20 March 2023].

[16] C. Voudouris, E. P. K. Tsang and A. Alsheddy, "Guided Local Search," in *Handbook of Metaheuristics*, Springer, 2010, pp. 321-361.

[17] "Nvidia: CuOpt Resources," 19 September 2022. [Online]. Available: https://github.com/NVIDIA/cuOpt-Resources/tree/branch-22.08/notebooks/routing/python. [Accessed 20 March 2023].

[18] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez and I. S. Michael Jordan, "RLlib: Abstractions for Distributed Reinforcement Learning," in *Proceedings of the 35th International Conference on Machine Learning, PMLR*, 2018.