# Decentralized Resource Allocation for Synchronized Tasks through Adaptive Large Neighborhood Search (ALNS)

by

Christian D. Montgomery

B.S. Computer Engineering and Computer Science

United States Naval Academy, 2018

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MAY 2020

Signature of Author: ______________________________________________

Department of Aeronautics and Astronautics
May 19, 2020

Approved by: ______________________________________________

Mark Abramson
Principal Member of the Technical Staff
The Charles Stark Draper Laboratory, Inc
Technical Supervisor

Certified by: ______________________________________________

Hamsa Balakrishnan
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: ______________________________________________

Sertac Karaman
Professor of Aeronautics and Astronautics
Chairman, Committee for Graduate Students

# Decentralized Resource Allocation for Synchronized Tasks through Adaptive Large Neighborhood Search (ALNS)

by

Christian D. Montgomery

Submitted to the Department of Aeronautics and Astronautics
on May 19, 2020 in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

## Abstract

This thesis explores a method for multiple suppliers to coordinate resource scheduling of task requests from multiple consumers using decentralized planning. A time window is associated with each task and some tasks require simultaneous servicing from multiple resources of specified classes to fulfil a request. The suppliers create schedules for their resources that maximize the value of all tasks fulfilled, while minimizing travel cost, and respecting all time window constraints. This thesis presents *Infeasibility Cooling Adaptive Allocation for Resource United Scheduling* (ICAARUS), a novel Adaptive Large Neighborhood Search (ALNS) algorithm that is capable of synchronizing tasks across a variable number of resources. A supplier's individual schedule and cost function is kept private from consumers. An e-commerce style of multi-round bidding is introduced to notify suppliers of resource request parameters and to allow consumers to synchronize resources from independent suppliers. A Mixed-Integer Linear Program (MILP) is used by the consumer to select the least costly bids that can be combined to fulfill a task's requirements.

## TABLE OF CONTENTS

# Chapter 1

## Introduction

Recent advancements in research and technology for robotics has created resources with the ability to operate in a variety of domains. Current operations are unable to take full advantage of disparate capabilities across all domains. Planning of modern day operations are becoming increasingly complex. Challenges ranging from stove piped mission commanders who are only aware of the few resources assigned to them, to rigid schedules that are too complicated to re-plan.

The purpose of this thesis is to develop an algorithm that coordinates resources across multiple domains through decentralized bidding for multiple consumers. This algorithm addresses two main challenges to planning multi-domain operations as envisioned in the *Army in Multi-Domain Operations 2028* [3]. First, to properly manage and fully realize the capabilities of so many decentralized resources, a bidding system is needed. This bidding structure should direct suppliers to service customers that most value resources and require the least cost expenditure. Second, an algorithm that solves the Supplier Scheduling Problem (SSP) must do so in an operationally feasible runtime. Computing solutions is complicated because of the synchronization requirements of a single task across multiple resources.

### 1.1 Thesis Overview

This thesis presents a method to coordinate and create schedules for multi-domain operations. The development of this method and the associated technical challenges are described in the following six chapters. An overview of the chapters follows:

**Chapter 2 – Operational Concept**. In this chapter, the concept of e-commerce is introduced as a solution to decentralized resource allocation. The time and spatial operational constraints in the Supplier Scheduling Problem are assumed fixed in this work.

**Chapter 3 – Model Formulation and Development**. In this chapter, a mathematical model is created for the scope of the SSP. It is shown that this problem can be modeled as a network problem while maintaining the constraints expressed in Chapter 2. Similar problems in past literature are reviewed to lend insight into the development of an algorithm for the SSP. In particular, the *Traveling Salesman Problem* and extended variants are explored due to their similarity to the SSP. A Mixed-Integer Linear Program (MILP) is introduced for providing optimal solutions for the SSP. An analysis of this method is presented in Chapter 5.

**Chapter 4 – Formulation of Algorithm**. This chapter formulates the *Infeasibility Cooling Adaptive Allocation for Resource United Scheduling* (ICAARUS) Algorithm to solve the SSP. This chapter begins with the formulation of the *Composite Operations Planning Algorithm* (COPA) to solve the UAV Planner Problem. However, the Composite generation algorithm falls short of ensuring resource synchronization across a task. To accomplish this, *Adaptive Large Neighborhood Search* (ALNS) is studied in the Vehicle Routing Problem, which has synchronization across a single supplier's resources as a constraint. ICAARUS explores a large region of the state space by allowing infeasible schedules to be created and culled through Simulated Annealing.

**Chapter 5 – Tests and Analysis**. This chapter covers the testing and analysis of the MILP and ICAARUS. It is shown that while the MILP provides an exact solution, ICAARUS is able to find a feasible schedule in much less time. It is shown that the MILP is unable to scale and solve beyond 20 task requests in a reasonable time manner. Meanwhile ICAARUS is able to produce a schedule for large cases including 40 task requests, in under 30 minutes.

**Chapter 6 – Conclusions**. This chapter provides a summary of the work and resulting contributions presented in this thesis. Proposals for modifications to our methods are presented, including the incorporation of robust planning into the resource allocation process.

## 1.2 Contributions

This research makes the following contributions:

1. An e-commerce bidding structure to coordinate multiple consumers with multiple suppliers asynchronously in a decentralized manner.
2. A MILP model for suppliers to solve the SSP with synchronization.
3. A MILP model for consumers to select cheapest resource bids.

4. The development and implementation of ICAARUS, an algorithm to schedule multiple resources for tasks requiring time and spatial synchronization.
5. Testing and analysis of the supplier MILP and ICAARUS.
6. Recommendations for modifications to ICAARUS.

## 1.3 Motivation

In the summer of 2018 at Rim of the Pacific (RIMPAC), the world's largest international maritime exercise, the *USS Racine* underwent a sinking exercise (SINKEX) with a range of military units working together to sink this one ship [1]. Originally the ship was targeted by a P-3 Orion aircraft, but when its ability to communicate targeting information was jammed in the simulation, both a Gray Eagle Unmanned Aerial Vehicle (UAV) and Army AH64E Apache helicopter were able to respond and support targeting through new data-link backups. This re-established communication provided for overwhelming firepower from multiple domains with Naval Strike Missiles and HIMARS artillery launched by the Army, and AGM-84 Harpoon missiles launched by a Navy P-8 aircraft. To finish the exercise, the submarine *USS Olympia* also launched a MK-48 torpedo to sink the *USS Racine*. While this SINKEX was carefully orchestrated to coordinate Army and Navy capabilities from the sea, air, and land, it showed the flexibility and capability of multi-domain operations.

This exercise reflects the US Navy's *A Design for Maintaining Maritime Superiority* call for Distributed Maritime Operations (DMO), described as aiming to "deepen naval integration with other services to realize [strategy] in multi-domain, distributed operations" [2]. In this thesis, domain refers to the five warfighting areas of sea, land, air, cyber, and space. The US Army calls for Multi-Domain Operations (MDO) to converge all domain capabilities across time and space to inundate adversaries. This convergence is envisioned in *Army in Multi-Domain Operations 2028*: (1) Create synergy across domains for overlapping redundancy, and (2) give commanders multiple forms of attack through options that are unforeseen by the enemy [3]. Historically the delivery of effects onto an adversary are referred to as "Kill-Chains" with each effect having siloed planning and execution path, this new level of complexity creates the idea of "Kill-Webs, complex representation of effect chains with multiple possible paths" [4]. However, this new capability presents a problem of scalability with "a factorial increase in possible interrelationships that will test the limits of current analytic approaches" [4]. LCDR Will Spears also remarks that while MDO is the natural evolution of Joint Warfare, communication barriers of information classification levels and technical language discrepancies between disciplines will require a "level of agility that is beyond [current capabilities]" [5]. With this communication though, MG VeraLinn Jamieson describes a vision for the Air Force's Intelligence, Surveillance, and Reconnaissance (ISR) as "Fusion Warfare," which "integrates and synchronizes information from multiple sources and domains" to fly, fight, and win in any battlespace [6].

# Chapter 2

## Operational Concept

This chapter clarifies what is e-commerce and how it is used in the context of this thesis. Advancements in Information Technology in the 21st century have led to supplying services and commodities through telecommunication networks. Rarely does a student nowadays need to physically go to a bookstore, searching for the best deal on textbooks. Instead a student can virtually specify a book's title and condition, and then select the cheapest option from a range of suppliers. This is electronic commerce, or e-commerce. The abundance and speed of access, that defines modern day corporations like Amazon or eBay, are characteristics necessary for MDO and thus e-commerce is an appealing tool to future military planners.

Nanehkaran defines e-commerce through three main components: communication systems, data management systems and security [7]. This thesis assumes hardware capabilities for any communication and data management systems are possible, leaving that work to future researchers. This thesis, as mentioned in Chapter 1, focuses on presenting ICAARUS, an algorithm for scheduling resources from suppliers to consumers. This chapter will define the communications between consumers and suppliers, and what data is shared versus what is kept private in the interest of user security.

The market proposed in this thesis makes several assumptions that depart from classical marketplace features. Firstly, consumers and suppliers do not exchange money for services. Consumers express their level of desirability for a task through assigning value. Consumers are assumed to be honest actors, meaning they do not assign high value to low priority tasks for the purpose of cheating resources from suppliers. Suppliers are also assumed to always be willing to service a task request if feasible in their schedule. However, some suppliers are busier than others and express this through costs associated with offered resources.

### 2.1 Consumer-Supplier Relationship

This thesis will focus on how suppliers can more effectively allocate limited resources to consumers. This interaction begins by consumers, who have missions they are planning, creating tasks which require resources. For the purpose of this thesis only consumers generate tasks, and only suppliers have resources, i.e. no supplier is trying to plan a mission and no consumer has resources. These tasks are defined by several parameters which are common questions a commander may need answered when planning a mission:

*Value*: Assigned number, [1,100], that quantifies the importance of the task.

*Position*: Two-dimensional location at which the task could be performed.

*Minimum Duration*: The time it takes to completely service the task.

*Time Window*: The start and end time in which the task must be serviced. If a resource arrives before the start of the window, it cannot begin servicing until the early edge of the time window. In addition, if the *minimum duration* is not fulfilled before the end time, it is too late and the task is not counted as completed.

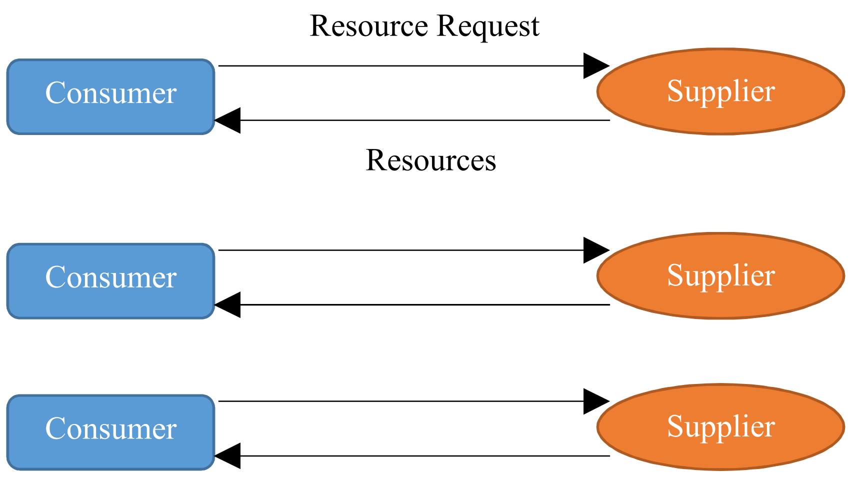
*Resources:* A task may require one or multiple resources. These resources are defined by types, so only resource of one type can fulfill that type requested.

For a task to be serviced, all resources must be in the same position for an overlapping minimum duration within that task's time period.

### 2.1.1 One Consumer to One Supplier

The simplest case of this scenario is one consumer assigned to one supplier. This stovepipe relationship is still prevalent today for the ability to 1) simplify scheduling and 2) keep planning details confidential. A consumer can only get resources from one source, so a task's feasibility is straightforward to find through asking if the one supplier has the requested resources or is pre-occupied with servicing another task. As resources are only servicing one customer, the supplier's schedule is known to only that consumer.

**Figure 1: One Consumer to One Supllier**

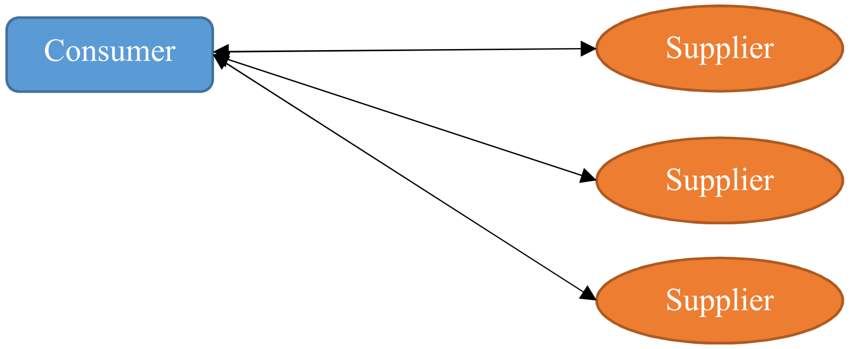


The immediate drawbacks of task planning with this relationship is a consumer's complete reliance on one supplier. Any task requiring a resource that the supplier does not have is instantly infeasible with no other options. Also this relationship can be observed by an adversary and become susceptible to attack. An adversary can predict operational actions by noting historical consumer-supplier relationships, even if dedicated supply chains are not published information. An adversary could be aware of a consumer's immediate actions by observing the movements of its supplier's resources, or even remove this one supplier to cripple the consumer completely.

### 2.1.2 One Consumer to Multiple Suppliers

As the internet opens up connections across the world, so does e-commerce aim to open up connections to suppliers. Through e-commerce comes the ability for a single consumer to expand its network of resource access from one supplier to multiple suppliers.

**Figure 2: One Consumer to Multiple Suppliers**



This multitude of suppliers is a powerful tool for improving mission planning. Through more suppliers comes greater availability of resources, and thus increased probability of a consumer finding the resources they desire when they are needed. Another operational benefit of this network is unpredictability. Through an abundance of options for planning operations, consumers will naturally begin to vary their supplier choices. This will obscure intended actions to adversaries and remove supplier vulnerabilities through redundancy.

With more nodes in a network comes more potential leaks of information in the system. Some operational scenarios could have mission commanders concerned about adversaries eavesdropping into communications and compromising security. That is why in this thesis, suppliers do not share schedules with each other or consumers. A supplier only responds to the consumer through a "bid" with the following information:

*Cost*: This is a measure of how much travel time the supplier must expend for its resources to arrive at the task. Travel time is the expense that is analogous to fuel for resources that are expensive to exercise.

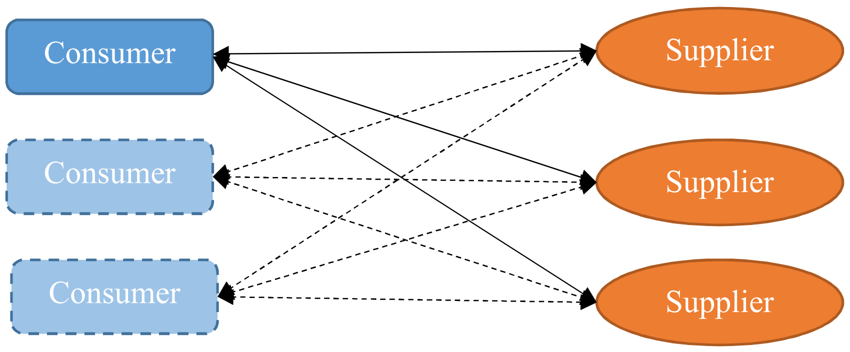
*Time Window*: The arrival and departure time of resources to the task's location.

*Resources*: Which resources, type and quantity, are being allocated to the task.

### 2.1.3 Multiple Consumers to Multiple Suppliers

In theory, multiple suppliers collectively servicing only one consumer would be ideal, but in reality, this system requires multiple consumers. In a finite world, gaining the flexibility of numerous suppliers in a network also requires multiple consumers pooling their supply chains to create this network. For e-commerce to be utilized in a practical setting, consumers will need to compete against each other for supplier resources.

**Figure 3: Multiple Consumer to Multiple Suppliers**

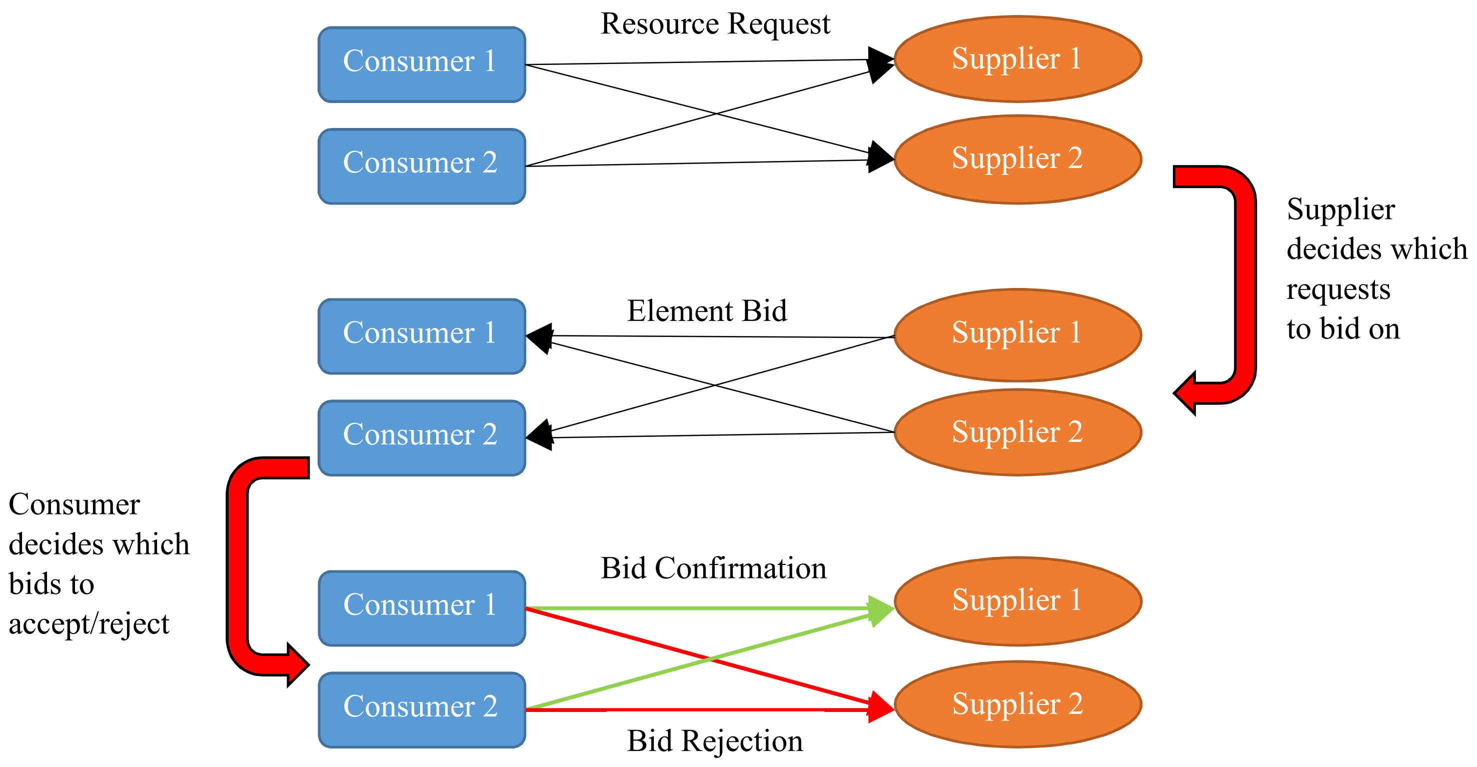


Consumers do not collaborate with other consumers for achieving a global maximum task fulfillment rate. A consumer is concerned with only completing its own tasks. These self-interested actors will continue to request resources from all suppliers until the task is fulfilled or bidding is stopped. This means suppliers in this network will receive a larger volume of resource requests, compared to suppliers operating in a one consumer to one supplier relationship. For the desired option creation feature of e-commerce, comes inherent complexity for the system to manage.

## 2.2 E-Commerce Bidding Structure

In this thesis, a consumer is responsible for stating what resource types are needed and time range that these resources are needed for a task. The challenge of calculating a path for how those resources will get to the task is removed from the consumer’s concern. Consumers interact with suppliers for resources through multiple bidding rounds. The procedure for each round is a three part handshake: 1) Resource requests are sent from the consumer with the task’s critical information to the supplier; 2) element bids are sent from the supplier to the consumer expressing resource availability; and 3) bid confirmations are sent from the consumer to the supplier to accept or reject the element bids. Figure 4 illustrates the flow of information in these resource bidding rounds:

**Figure 4: Information Flow in One Bidding Round**



After resources are requested, the consumer will receive element bids from suppliers with three distinct outcomes for each task:

1. Incomplete Fulfillment

    For at least one resource the task requests, no bids were received.

2. Complete Fulfillment

    From one or multiple suppliers all resources a task requested are fulfilled and have an overlapping service time of at least the minimum required duration.

2. Unsynchronized Fulfillment

    From multiple suppliers a task receives bids on all resources requested, but the received bids have misaligned service times.

These three distinct outcomes require a protocol for handling, with the desired effect that this process would get all tasks to Complete Fulfillment before the end of bidding. MG Jamieson envisions “fusion warfare” of the future as continuously repeating OODA loops for mission planners of ISR operations. The OODA loop is a classic decision-making process distinguished by its four phases of Observe, Orient, Decide, Act. These four phases are the basis of the Consumer Decision Process for this Marketplace:

Observe – Which element bids are received.

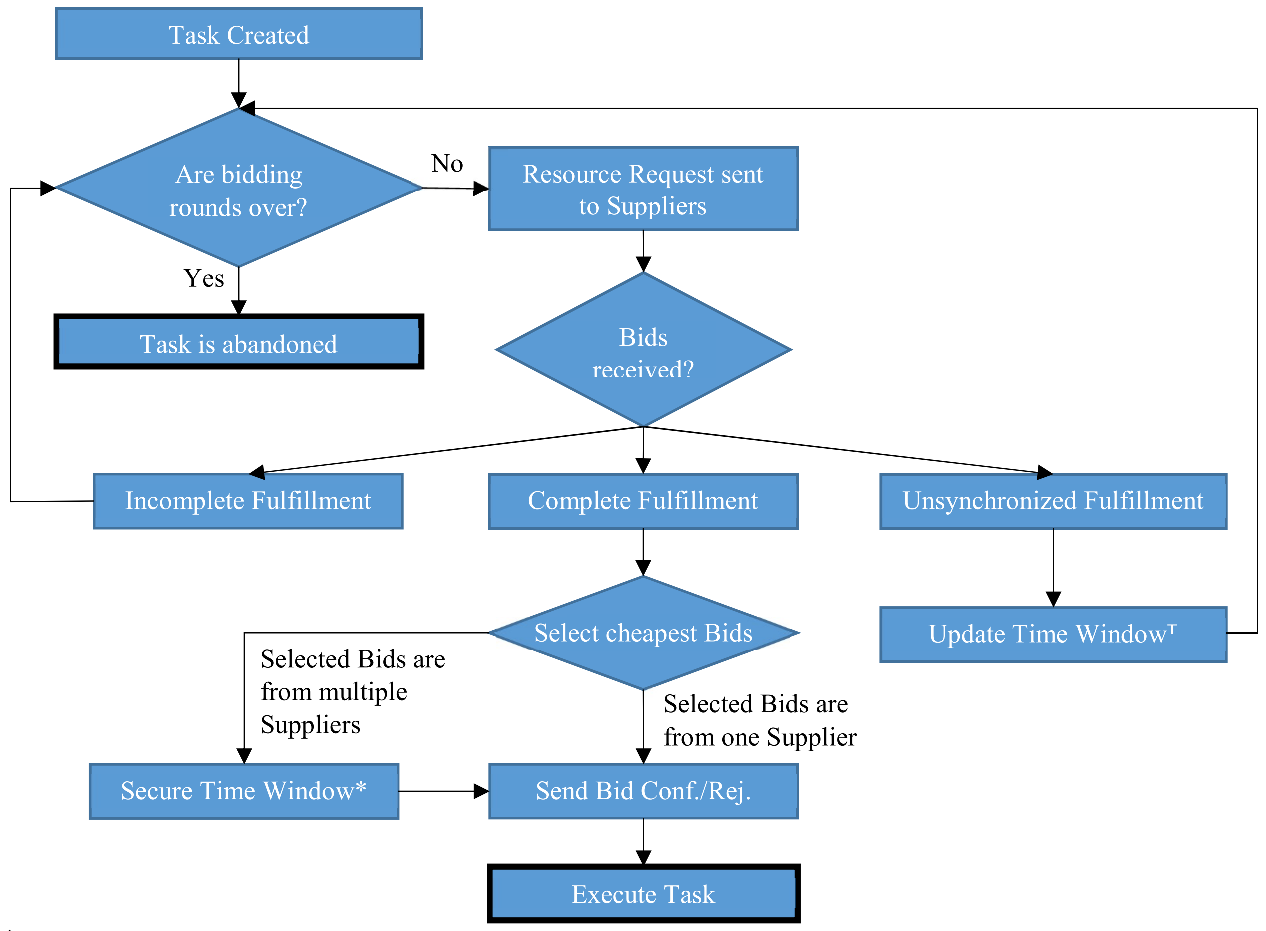
Orient – Which fulfillment status does a task fall under.

Decide – How to handle these fulfillment statuses.

Act – Send Bid Confirmation/Rejection and begin the next round of Resource Requests.

The Consumer Decision Process is outlined in Figure 5.

**Figure 5: Consumer Decision Process**



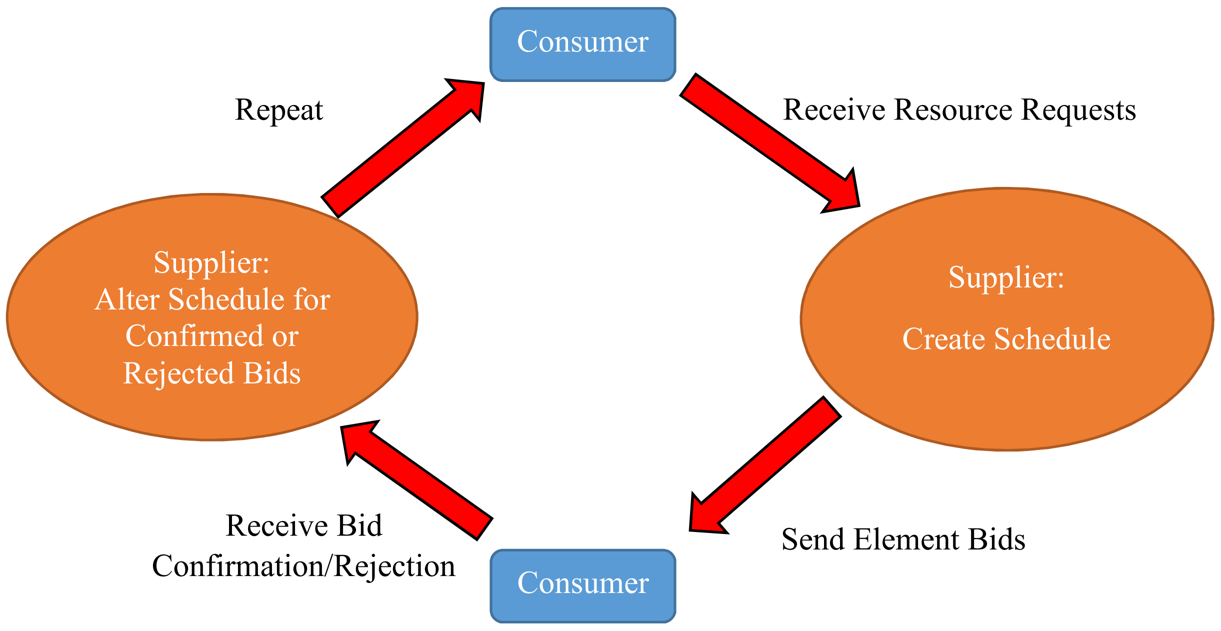
* See Section 2.2.2 for more information.

ᵀ See Section 2.2.3 for more information.

### 2.2.1 Incomplete Fulfillment

Element Bid outcome one is relatively straightforward to manage in an e-commerce network. For Incomplete Fulfillment, the consumer rebroadcasts the original task request. This request is made in the hopes that a previously busy supplier now has room in its schedule. Suppliers are constantly altering schedules through bidding rounds, as seen in Figure 6, as scheduled tasks receive bid rejections.

**Figure 6: Supplier Schedule Decision Cycle**


Consumer
Repeat
Receive Resource Requests
Supplier: Alter Schedule for Confirmed or Rejected Bids
Supplier: Create Schedule
Receive Bid Confirmation/Rejection
Send Element Bids
Consumer

### 2.2.2 Complete Fulfillment

For element bid outcome two, Complete Fulfillment, the consumer will select which combination of bids they need at the cheapest cost. The consumer shall notify suppliers of unselected bids that their bid was rejected and should thus be dropped from that supplier's schedule. As mentioned in the beginning of Chapter 2, consumers are assumed to be honest agents that do not hoard unnecessary resources. Consumers will only accept the bids they need to fulfill a task's resource requirements. Should an allocated resource suddenly become unavailable, a consumer can always re-submit a resource request in the next bidding round.

Of note in the Consumer Decision Process, Figure 5, is the route of "Selected Bids are from one supplier." When a single supplier is providing all the resources for a task, they have control over the tasks start and end time, as long as it remains within the original time window. Even if a consumer accepts a bid (or bids) for a specific start time, the supplier can change that start time as they see fit. If this previously agreed upon start time were made rigid, the supplier may drop this commitment in favor of a new more valuable task request. The supplier's flexibility to move start times, only if it is allocating all the resources needed, is advantageous to consumers as it lowers the risk of bids being withdrawn. This is also advantageous to the system as suppliers have flexibility in their schedule to open up slots for tasks that may be denied from schedules made in earlier bidding rounds.

When a consumer selects resources from multiple suppliers, the “Secure Time Window” stage is necessary. This simply alters the task to begin and end only at the time all element bids overlap. This is necessary as the original resource request specified a time window with a range of possible arrive and depart times for suppliers to offer resources in. Once a set of bids that completely fulfill the task are found, consumers do not want these resources’ time slots to be move around by suppliers, resulting in the task becoming unsynchronized. To prevent this, consumers secure the task’s time window to one start and end time that all element bids overlap. Suppliers are updated of this change to the task’s information through the bid confirmations sent.

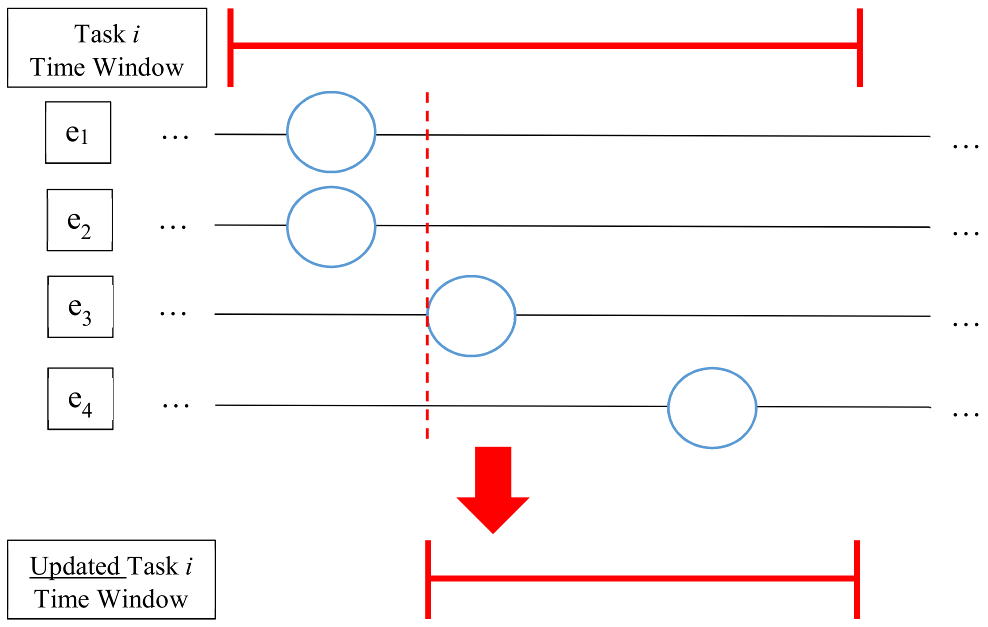
### 2.2.3 Unsynchronized Fulfillment

Element bid outcome three, Unsynchronized Fulfillment, presents the greatest challenge for consumers. As suppliers are not communicating with each other, consumers rebroadcasting the same request leaves synchronization to chance. Progressing to complete fulfillment requires direction from the consumer to coordinate scheduling across suppliers.

This “Decide” phase of the OODA loop in the Consumer Decision Process attempts to balance task feasibility with synchronization. The wider the range of the time window for the task, the greater the likelihood a resource request will be answered as suppliers have many options to fit servicing the task into their schedule. The drawback of these numerous options for suppliers is the decreased probability Element Bids will then be synchronized with other suppliers whose schedules are unknown. To push diverse suppliers towards coordination, a consumer updates the time window.

The time window is updated by pushing the task’s early time window to the second earliest bid arrival time. This heuristic is based on ICAARUS pushing all tasks to be serviced as early as feasible. This means that when a bid is received, the consumer knows the resource cannot be moved any earlier without altering the schedule. However, the resource could be moved back and the schedule remains feasible with idle time in the later part of the resource path. An example of updating the time window is shown in Figure 7:

**Figure 7: Update Time Window**



Updating to only the second earliest bid arrival time is crucial to synchronizing suppliers without forcing schedules to sub-optimal solutions. As can be seen in Figure 7, even with the new shortened time window, all four Element Bids are not synchronized. While suppliers may push bids $e_1$ and $e_2$ to have the same arrival time as $e_3$, $e_4$'s arrival time may not be moved up. In that case the task will again reach Unsynchronized Fulfillment status and the time window will be shortened to $e_4$'s arrival time, and then the task is expected to reach synchronization across all Element Bids.

The task's early time is not immediately updated to the latest arrival time, $e_4$'s arrival time, as that may tighten the task's time window to an infeasible time range. Suppose one supplier was providing bids $e_1$ and $e_2$, and it has a highly valuable task it is servicing right after task *i* that it cannot move. If the time window were drastically tightened then task *i* would lose two of its Element Bids. By gradually updating the time window through multiple bidding rounds, the following rounds could have the supplier of $e_4$ move its arrival time up or new suppliers become able to service task *i* in place of $e_4$'s supplier.

If the updating time window process results in a supplier dropping its Element Bid for the task with no other supplier filling its bid, then the task would become an Incomplete Fulfillment case. This would result in the consumer re-submitting its resource requests with the task's original time window, giving maximum flexibility to suppliers again.

# Chapter 3

## Model Formulation and Development

This chapter develops a mathematical representation that addresses the Supplier Scheduling Problem (SSP). The mathematical representation begins with a description of the input and output variables that define the problem. The mathematical structure of the SSP is visualized through multiple graphical representations to highlight the network nature of this problem.

The mathematical formulation allows us to identify similar problems in the literature. Research into the *Traveling Salesman Problem* and its *Time Window* variants help us understand the challenges of SSP and provide techniques for finding solutions through exact and heuristic methods. The *Team Orienteering Problem* is also a useful variant of the *Traveling Salesman Problem* for understanding and current methods used for organizing a team to maximize prize collection as the SSP organizes multiple resources for maximum mission completion.

At the end of the chapter, a Mixed Integer Programming (MIP) model of the SSP is presented. This MIP utilizes binary and continuous decision variables to build a schedule for a set of resources that satisfies the constraints of the SSP.

### 3.1 Supplier Scheduling Problem (SSP)

This section describes the SSP and defines the inputs and outputs of the problem. It presents assumptions that were made on the capabilities of the resources, and explains any other assumptions made to simplify operational constraints for the purpose of this mathematical model.

### 3.1.1 Inputs to Supplier Solver

3.1.1.1 Supplier Inputs

| | |
|---|---|
| *supplierID* | Integer value identifying which supplier is making this schedule. |
| *pos* | Position of supplier's base location. |

| | |
|---|---|
| *resourceList* | Set of-which resources and what quantity of those resources a supplier has control over. |
| *schedule* | Assignments of  which task elements a supplier has committed particular resources too, and what time they are assigned to service these requests. |
| *unconfirmedTasks* | Set of which tasks the supplier has on its schedule that it bid to fulfill for a consumer, but has not received a status notification of yet. |
| *confirmedTasks* | Set of which tasks the supplier has on its schedule that it has bid on and an acceptance of selection has been received by a consumer. |

The *supplierID*, *pos*, and *resourceList* are unchanging for each supplier through all rounds of bidding. Initially the *schedule*, *unconfirmedTasks*, and *confirmedTasks* are empty until the supplier begins bidding on task requests.

The *resourceList* can have duplicate entries to express a supplier's capacity of multiple resources for that one type. For example: a *resourceList* of {*A,B,C*} means a supplier has one resource of type *A,* one resource of type *B,* and one resource of type *C.* Meanwhile a *resourceList* of *{A,A,A}* means a supplier has three resources of type *A.*

3.1.1.2 Task Inputs

| | |
|---|---|
| *consumerID* | Integer value identifying which consumer the task request is from. |
| *taskID* | Integer value a consumer gives to identify the task. |
| *pos* | Position of the task's location. |
| *value* | Rated importance of the task from [1-100]. |
| *early* | Beginning of time window for the task. |
| *late* | End of time window for the task. |
| *minDur* | Minimum time duration the task needs to be serviced. |

| | |
|---|---|
| *resourceCount* | Number of elements for which the task is requesting resources. |
| *resourceReq* | Set of resources the task is requesting. |

A task request has two forms of identification, *consumerID* & *taskID*. From here onward a task may be referred to as simply task *i* or task *j*, but this *i* or *j* is not a task's complete ID, it is merely a shorthand to index individual tasks. For example: *task i* could be referring to *task (2,3),* which is a task from consumer 2 and labeled as task 3 from that consumer.

The *resourceReq* can have duplicate entries to express the consumer needing multiple resources of that one type for a task. For example: a *resourceReq* of *{A,B,C}* means a task is requesting three elements with one resource of type *A,* one resource of type *B,* and one resource of type *C*. Meanwhile a *resourceReq* of *{A,A,C}* means a task is requesting three elements with two resources of type *A,* and one resource of type *C.*

### 3.1.2 Outputs of Supplier Solver

As mentioned in the motivation for this research, the model is decentralized in nature for its bidding structure. So, the outputs from a supplier that a consumer would see are bids for task elements. However, the solution to the SSP is a schedule of supplier resources accommodating tasks, which remains visible only to that individual supplier. A supplier's schedule is described as a list of *composites*:

| | |
|---|---|
| *composite* | A single *resource* and a *path* plan |

The *path* of a *composite* has the following characteristics:

1. *Task Order*: The order in which this resource will accommodate its assigned tasks.
2. *Arrival Times*: The time at which the resource will be able to start each task assigned.
3. *Departure Times*: The time at which the resource will need to end each task assigned.
4. *Travel Times*: The time required to travel from the previous task to the subsequent task.

Example 1 displays a supplier's schedule with composites.

*Example 1: This example describes a schedule with two composites, meaning two resources and two path plans. Here there is a set of five tasks, {$t_1$,$t_2$,$t_3$,$t_4$,$t_5$}, to be completed by a set of two resources, {A,B}. The following is a table of a selection of the parameters for this set of tasks:*

**Table 1: Example 1 Task Parameters**

| Task | Early | Late | Min. Duration | Resource Req. |
|---|---|---|---|---|
| **$t_1$** | **0.0** | **4.0** | **1.0** | **{A}** |
| **$t_2$** | **2.0** | **6.0** | **1.0** | **{B}** |
| **$t_3$** | **4.0** | **8.0** | **1.0** | **{A}** |
| **$t_4$** | **6.0** | **10.0** | **1.0** | **{A}** |
| **$t_5$** | **8.0** | **12.0** | **1.0** | **{A,B}** |

Only task $t_5$ requires multiple resources and needs synchronization across resources. The following provides path plans for the composite example:

**Composite 1.** Resource *A* departs the base station at time 0, and then performs the following tasks. The resource returns to the base station directly after completing $t_5$:

**Table 2: *Example 1* Composite 1: Path Plan for Resource A**

| Task | Order | Arrival Time | Departure Time | Travel Time |
|---|---|---|---|---|
| **$t_1$** | **1** | **1.0** | **2.0** | **1.0** |
| **$t_3$** | **2** | **4.0** | **5.0** | **1.0** |
| **$t_4$** | **3** | **7.0** | **8.0** | **2.0** |
| **$t_5$** | **4** | **11.0** | **12.0** | **3.0** |

Note that the while resource *A* can begin $t_1$ at time 0, it takes a travel time of 1 hour to get to that task (from starting base) so its arrive time is 1.0. Also, while resource *A* can leave $t_1$ at time 2.0, and it only requires a travel time of 1 hour to get to *$t_3$* at time 3.0, the resource cannot begin servicing the task till *$t_3$'s* Arrive Time of time 4.0.

**Composite 2.** Resource *B* only requires a travel time of 1 hour to get to *$t_2$* from the base station, so it does not depart till time 1.0, and then performs the following task. The resource returns to the base station directly after completing *$t_5$:*

**Table 3: *Example 1* Composite 2: Path Plan for Resource B**

| Task | Order | Arrival Time | Departure Time | Travel Time |
|---|---|---|---|---|
| **$t_2$** | **1** | **2.0** | **3.0** | **1.0** |
| **$t_5$** | **2** | **11.0** | **12.0** | **2.0** |

Note that while resource *A* can leave *$t_2$* at time 3.0, and it only requires a travel time of 2 hours to get to $t_5$ at time 5.0, the resource does not begin servicing the task till time 11.0 even though the time window for servicing the task is time 8.0. This is because *$t_5$* requires synchronization of both resources, so resource *B* must wait for resource A to arrive to begin servicing *$t_5$* together.

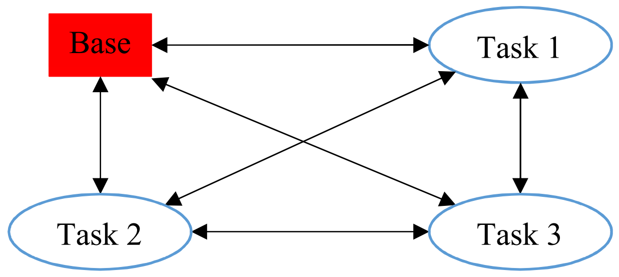
## 3.2 Network Representation

This section describes how to transform the physical real-world problem with complex parameters, into a graphical structure easier to visualize. This formulation of the problem into a graph will provide insight into methods to solve the problem. First introduced is a simplistic static graph of the problem, then a more complex time-space graph, and finally a placement-space graph.

### 3.2.1 Static Graph Representation

To begin applying analytical techniques to the SSP, the physical system should be mapped to the proper theoretical framework. Large-scale transportation problems, such as the V*ehicle Routing Problem*, are often represented using a directed graph, *G(N,A)*, consisting of a set of nodes, *N*, and a set of arcs, *A* [15]. In this network the nodes represent task and base locations, and arcs represent the path to get from one node to another. A two-dimensional map can be rearranged to a simpler network to visualize with weights on those arcs representing the distance between node pairs. A static graph representation for the operations of a single resource and three tasks is shown in Figure 8:

**Figure 8: Static Graph Representation**

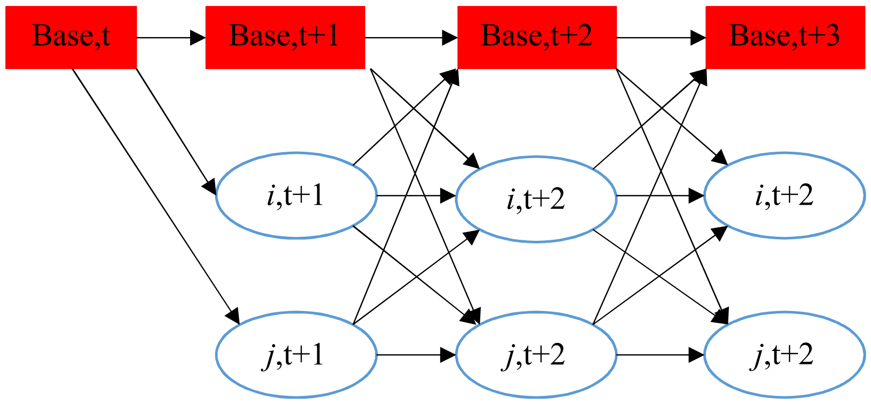


### 3.2.2 Time-Space Graph Representation

The static graph representation presented in the previous section is unable to model the time dimension of resource scheduling. To model time and position, the time-space graphical representation can incorporate time by creating nodes that include the location of the resource and the time that the resource is at the task. Therefore, a node most be created for each task at every time period over the planning horizon. Task-time nodes cannot be limited to only be the time periods for which the task's time window is active as the network needs to capture where the resource is if the resource is idly waiting between two time windows.

The time-space graph assists in scheduling resources, because it records the location of every resource at each point in time. Each node in the time-space graph is indexed by task and time period. For example, index ($i$,t) corresponds to task $i$ at time t. However, now the arcs must become directed in the network as travelling back in time is prohibited in reality. below we show Figure 9 of a time-space graph with two tasks:

**Figure 9: Time-Space Graph Representation**



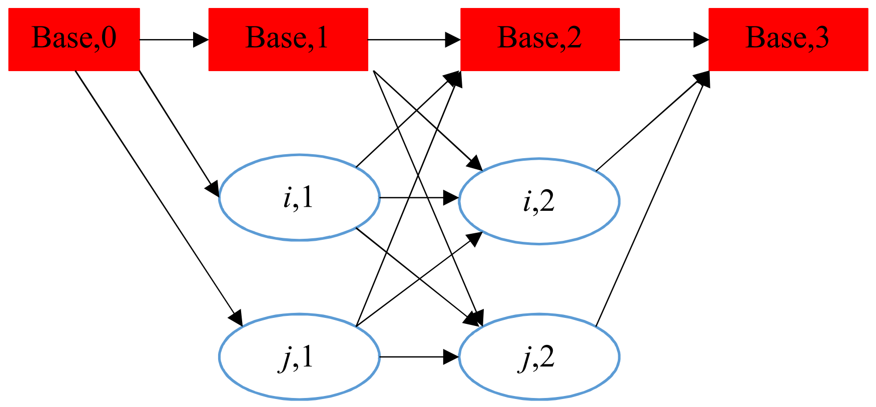
While this network enables visualizing a resources position in space and time, it creates many unnecessary nodes and arcs due to task nodes existing across the whole planning horizon. Another issue this graph representation encompasses is the need to discretize time. The drawback of high precision with shrinking time intervals comes at the cost of rapidly growing the number of nodes.

### 3.2.3 Placement-Space Graph Representation

The time-space graph representation in the previous section suffers in the trade-off of precision of time vs. graph size. When a continuous variable such as time is binned into discrete segments, the ease of interpretation comes at the risk of suboptimal solutions. To keep time of arrivals and departures as continuous variables, the placement-space graphical representation incorporates order by creating nodes that include the location of the resource and the placement that the resource is in the service list. This requires a node to be created for each task at every index on the resource's service list. This is expected to create a graphical representation with fewer nodes as the number of tasks to accommodate should be significantly smaller than the number of time periods in the planning horizon**.**

Each node in the placement-space graph is indexed by task and placement. For example, index (*i*,2) corresponds to task *i* being the second task the resource visits. The arcs must also be directed as the time-space graph, as traveling back in order is prohibited in reality. Below we show Figure 10 of a placement-space graph with two tasks:

**Figure 10: Placement-Space Graph Representation**



A path through this network reveals which tasks the resource will be able to service and in what order the tasks will be accommodated. However, like the time-space graph, the time window and synchronization constraints are not implicitly expressed and will require additional constraints on the network's arcs. The placement-space graph does require less arcs and nodes than the time-space graph, while maintaining continuous time variables.

## 3.3 Problem Classification

Now that the parameters of the problem are defined and a visual representation can be created for the placement-space dimension of the problem, similar problems in literature are compared to lend more insight to solution formation. The *Vehicle Routing Problem* (VRP) has many similarities to a supplier attempting to find an optimal routing of its resources. After all a vehicle is just a specific type of resource. The VRP has been studied since 1959's work by Dantzig and Ramser, and so many variations and formulations exist to draw inspiration from [16]. This section aims to classify previous research methods similar to the SSP's goals of maximizing benefit received while handling elements of:

- Time
- Multi-resource Synchronization
- Resource classes

In addition to accomplishing these features, there is also the desired feature of quickly finding a solution for a large number of tasks.

### 3.3.1 Mathematical Programming Background

A proven field of research for solving the VRP is known as Linear Programming. Linear Programming makes use of abstract mathematical models to represent real-world problems. The name "linear" comes from the linear functions of the decision variables that form the objective and all constraints. By formulating the rewards and costs of a real-world problem into an objective function, linear programming can find an exact solution that maximizes or minimizes this value. It does this under a set of constraints that represent the real-world conditions that limit the problem [12].

The classical linear programming problem necessitates all variables are continuous non-negative real numbers. This is not the case for a problem such as resource scheduling that requires binary decision variables, and a "yes" or "no" answer to the question of allocation. This is because a resource, such as a vehicle, cannot be cut in half and used to service half of two requests. A resource must be sent in its integral entirety or not at all. Fortunately, this a well-studied problem in the realm of Mixed Integer Programming (MIP), which is classified as a subset of mathematical programming using integer and binary variables. The unique types of problems faced in the SSP also fall under a more general class of problems known as the *Traveling Salesman Problem* (TSP) [17].

### 3.3.2 Travelling Salesman Problem

Dantzig  describes the *Traveling Salesman Problem* as "Find the shortest route for a salesman starting from a given city, visiting each of a specified group of cities, and then returning to the original point of departure" [18]. It is a mathematical conjecture that the complexity class of the TSP is Nondeterministic Polynomial-time Complete or NP-Complete [19]. This means that there is no known algorithm that can always give an optimal solution in a polynomial time variation. The best that can be done currently is finding a solution to an NP-Complete problem that has solution times grow exponentially with the number of nodes [20].

#### 3.3.2.1 Mixed Integer Programming Formulation for Traveling Salesman Problem

The TSP can be modeled as a MIP with binary variables that indicate the decision whether or not to traverse an arc in the network. The variable, $x_{ij}$, will take on the value of one if the arc between node *i* and node *j* is traveled, and zero otherwise. The variable, $d_{ij}$, is the distance from node *i* to node *j*. Dantzig, Fulkerson, and Johnson give the following MIP model:

$$Minimize \quad \sum_{(i,j)\in N} d_{i,j} x_{i,j}$$

$$Subject\ to: \quad \sum_{i\in N} x_{i,j} = 2 \qquad \forall j \in N$$

$$\sum_{i,j\in S} x_{i,j} \le |S| - 1 \qquad \forall S \subset N, S \neq \emptyset$$

$$x_{i,j} \in \{0,1\}$$

The set *N* is the set of all nodes in the network, and *S* is a subset of *N*. The first constraint ensures that each node in the network is visited by forcing the traveler to travel into and out of each node. The second constraint is a sub-tour elimination constraint; it ensures that the solution is a single tour. The size of the problem increases exponentially with the addition of the sub-tour elimination constraint, adding $2^N$ constraints.

### 3.3.2.2 Exact Solution Methods

All known solution methods to TSP run in non-polynomial time, and in the worst case the entire solution space might have to be searched to confirm an optimal solution. However, there are methods that attempt to intelligently search the solution space and reduce the solve time for the model. Two of the most common methods are *branch and bound* and *cutting planes*.

The *branch and bound* method is a divide and conquer method to find an optimal solution. This method begins by solving the *linear programming relaxation*, which removes the constraints that ensure variables are integral and binary [19]. This means the $x_{ij}$ variable can take on continuous values between zero and one and will provide a "lower bound" to the solution that may or may not be attainable. Next, this method solves sub-problems in an attempt to find integer solutions. The lower bound is used to discard certain subsets of the feasible set from consideration [20].

The *cutting planes* method begins similarly to *branch and bound* by first solving the *linear programming relaxation*. Next, sub-tour elimination constraints are added to force fractional solutions towards integer solutions, as well as to get rid of any sub-tours in the relaxation solution [18]. In the right circumstances, only a few intelligently chosen constraints are needed to find an optimal solution to all constraints.

### 3.3.2.3 Heuristic Solution Methods

When an exact solution method is relaxed to only need near-optimal solutions, but in a shorter search time, heuristic solution methods can be used to generate quick solutions. Heuristics do not search the entire set of solutions, but find the “best” solution in a reasonable amount of time. “Best” is defined as a threshold the designer creates for the acceptable gap from optimality.

Laporte categorizes heuristic methods into two classes: 1. *Tour construction procedures* to efficiently build feasible routes by adding nodes one at a time, and 2. *Tour improvement procedures* to improve an already existing route [21]. Most heuristic algorithms that solve the TSP incorporates both tour construction and tour improvement procedures in what is called a *composite algorithm* [21]. Note here the name “composite” refers to the algorithm being a combination of multiple procedures, and is not an algorithm of *composites* as defined in Section 3.1.2.

For example, Flood suggests the *nearest neighbor* algorithm is a straightforward *tour construction procedure* to find a solution to the TSP [22]. This algorithm begins at an arbitrary node, and proceeds to add the node that is closest to the present node. It repeats this nearest neighbor addition to the route until all nodes are included in the path. The last node is then connected to the origin to create a complete tour.

Another class of *tour construction procedures*, known as *insertion algorithms*, follow these basic steps [21]:

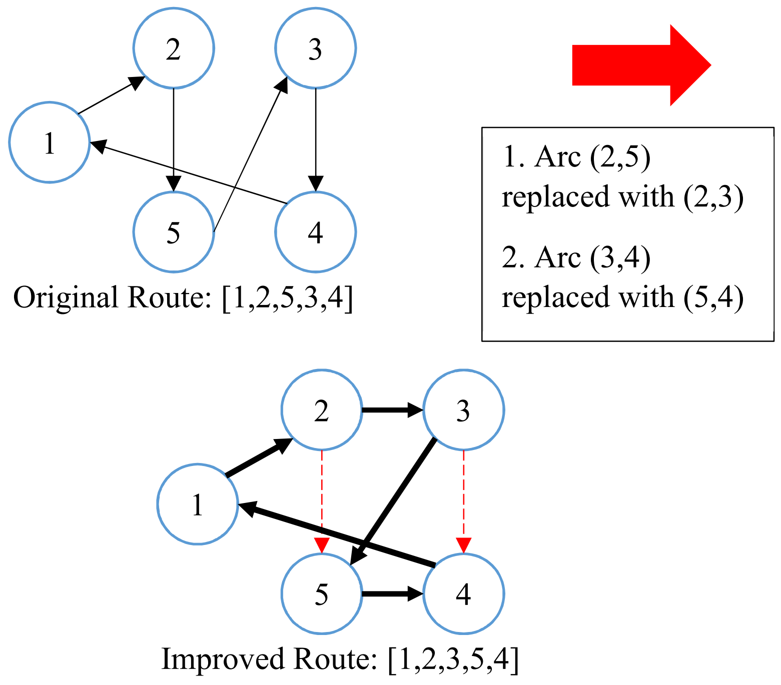
Step 1: Construct a simple tour with only two nodes

Step 2: Consider each node not in the tour. Insert the node that meets a specific criterion.

Common criteria that are used to measure which node is most appropriate to be added next to the network include: 1. Adding the node that is closest to the two nodes in the current selected tour, 2. Adding the node that is furthest from the two nodes in the current selected tour, and 3. Adding the node that produces the least increase in distance for the current path. While these examples of criteria are not exhaustive, other criteria include combining multiple metrics with weights assigned to each method [23].

*Tour improvement procedures* already start with a route, built by a simple or complex construction procedure. These procedures then aim to improve this given route by some routine. Flood noticed that if a path crosses itself at any point during the tour, then the tour could be improved by switching the order of nodes so that the tour does not cross [22]. Croes proposed a similar idea, known as *inversion*, where the order of two nodes is switched in a tour to see if the resulting route is improved [24]. Lin and Kernigan expanded upon these methods to bound the scope of searching for tour improvements in the *k-opt algorithm* [25]. The algorithm goes through all subsets of *k* arcs and attempts to reconnect the tour with a set of *k* new arcs. If an improvement is found, the *k* old arcs are deleted in favor of the new better arcs, and the algorithm continues down the route to the next set. An example of a *k-opt algorithm* with *k* = 2 is shown in Figure 11:

**<u>Figure 11: 2-Opt Algorithm</u>**


1
2
3
4
5
Original Route: [1,2,5,3,4]
1. Arc (2,5) replaced with (2,3)
2. Arc (3,4) replaced with (5,4)
Improved Route: [1,2,3,5,4]

Another group of *tour improvement procedures* are *metaheuristics*. These methods include subroutines that select which scope of the solution set to explore. These heuristics include *tabu search*, which records which solutions have already been generated to avoid repeated searches, *ant colony*, which records which previous solutions led to improvements and should be explored more, and *simulated annealing*, selecting solutions with a probability based on a measured criterion of optimality [26]. S*imulated annealing* is used extensively in ICAARUS as detailed in Chapter 4.

### 3.3.3 Traveling Salesman Problem with Time Windows

The TSP with Time Windows (TSPTW) is very similar to the basic TSP with extra constraints on when the salesman can visit a city. Each city has a time window in which the salesman can visit the city. Arriving in the city before or after that time window does not count as a visit for the salesman. The objective remains to visit $N$ nodes at least once (within their time windows) at a minimum travel cost.

In linear programming these time windows would be expressed as a constraint the Salesman must visit the city after the time window's lower bound, $l_i$, and before the time window's upper bound, $u_i$. Baker proposed a model with decision variables, $t_i$, that specified the time the Salesman visited city $i$. The shortest travel times between each node pair is already found and expressed by $d_{ij}$ .The decision variable $t_{n+1}$ specifies the time the salesman returns to the start node, and it is the difference between this time and the start time, $t_0$, that is trying to be minimized [28].

$$
\begin{array}{lll}
\textit{Minimize} & t_{n+1} - t_0 & \\
\textit{Subject to}: & t_i - t_1 \geq d_{i1} & i = 2, \dots, n \\
& |t_i - t_1| \geq d_{ij} & i = 3, \dots, n \qquad 2 \leq j \leq i \\
& t_{n+1} - t_i \geq d_{i1} & i = 2, \dots, n \\
& t_i \geq 0 & i = 1, \dots, n+1 \\
& l_i \leq t_i \leq u_i & i = 2, \dots, n
\end{array}
$$

In scenarios with many cities for the salesman to visit, a large $|N|$, this solution scales poorly and can take a long time to solve. Another exact algorithm to solve the TSPTW was introduced by Mingozzi et al. through use of precedence constraints [27]. Precedence constraints in a route ensure a node is visited no earlier than the proceeding node is visited. These constraints ensure that in tour construction, the next node added in the route is always in time order. A faster, but inexact, approach was proposed by Gendreau et al. that utilizes the *nearest neighbor* algorithm to construct tours [29]. This method requires at each iteration, that the time window bounds be checked for the added city to ensure the solution constructed is feasible.

### 3.3.4 Team Orienteering Problem

The Orienteering Problem (OP), or generalized TSP, is described as: given *n* nodes, each node *i* has a non-zero score $s_i$. The arc between node *i* and *j* has an associated cost of $c_{ij}$. In this problem the travel time between each node represents the cost and each node can be visited at most once. The objective of the OP is to maximize the score of a path that consists of a subset of nodes beginning at node 1 and ending at node n without violating the max cost (travel time) constraint T.

Golden et al. demonstrate this model for the sport of orienteering has useful application to the VRP and production scheduling [30]. They also proved that the OP is NP-hard, warning of the computational limitations of exact methods and encouraging a focus on heuristic procedures for problems of these classification. Golden et al. present a *composite algorithm* where the first stage constructs routes through a cost-benefit analysis and the second stage improves the initial route by 2-opt method similar to Lin-Kernigan and then a center-of-gravity method. Golden, Wang and Liu produced a more efficient algorithm where the algorithm learns the most effective *route improvement methods* and adapts through the course of improvements [34]. Tsiligrides approached the OP with a two stage heuristic, building initial routes through a Monte Carlo approach, and improving routes through a local search space heuristic method that performs route optimization similar to Lin-Kernigan's 2-opt method [33]. Ramesh and Brown solve the OP by iterating through four phases [35]:

1. Construct an initial route by a cost-to-benefit analysis to see which node is best to add next on the route.
2. Use Lin-Kernighan 2-opt method to improve the route.

3. Select nodes to delete from route that can be replaced with more valuable nodes in their places.
4. Repeat until the marginal improvement of a round falls below a specified threshold.

The Team Orienteering Problem (TOP) extends the OP by creating multiple tours of the network for multiple Orienteers to maximize the score collected. In the sport or orienteering a team of Orienteers attempt to coordinate together to collect as many waypoints as possible in a given amount of time. This aspect of coordination adds a great deal of complexity to the problem. Tang and Miller-Hooks used the heuristic of *tabu search* to overcome these complexities in TOP [31]. Following their work, Archetti et al. compared *tabu search* with *variable neighborhood search* and found that *Variable Neighborhood Search* outperformed two *tabu search* heuristics [32]. *Adaptive Large Neighborhood Search*, an extension of *Variable Neighborhood Search*, is explored further in Chapter 4 as a solution to the SSP.

## 3.4 Mixed Integer Linear Programming Model

This section discusses a Mixed-Integer Linear Programming (MILP) formulation that can be readily formatted to optimization software. The mathematical model provides the ability to find an exact optimal solution. Although previous research has shown that exact methods might not be practical, the optimal solution will give insight to compare the quality of a solution generated through heuristic methods. In addition, the *linear programming relaxation* of this model, while inexact, will give a quick theoretical upper limit to the "best solution." This upper limit can be useful to evaluate the gap between LP and heuristic methods in a time efficient manner.

This work draws heavily from Miller's reformulation of the Team Orienteering Problem with Time Windows (TOPTW) to solve his Unmanned Surface Vessel Observation-Planning Problem (USVOPP) [12]. The MILP developed by Miller took advantage of integer decision variables to create of placement-space nodes in network which greatly inspired this particular model. Miller's binary variable $x_{ikt}$takes a value of 1 if node $i$ is visited by USV $k$ in the $t$-th placement on the route, encompassing multiple decisions in a single variable. Also instrumental to the development of this model was Negron's work in creating a MILP model to make schedule sorties (schedules with multiple tasks on a route for a resource). Negron linearized travel time constraints by introducing linearization variables that are calculated a priori, which is key to formulating this particular model [7].

### 3.4.1 Supplier MILP Model Formulation

This mathematical formulation is used by individual suppliers to find the optimal schedule for a supplier's resources for all the task requests received. First the notations for sets, decision variables, and input variables are defined. Then the objective function and constraints are introduced. Finally variations of this model are introduced to improve synchronization of element bids for a particular task.

#### 3.4.1.1 Set Definitions

The following sets are used in the formulation:

T = set of all tasks

U = set of all resources

U(e) = Set of all resources of type *e*

P = set of all placements in a path

The set P is the set of placements of tasks in a path for a resource. A placement denotes the order of a task in the path. For example, if a task is in placement three, then the task is the third task that will be performed by the resource. The set of placements contain the placement for each task in the path. *For Example: The set associated with a path of five tasks, $T=\{t_1,t_2,t_3,t_4,t_5\}$, will contain the first five natural numbers, $P=\{1,2,3,4,5\}$.*

#### 3.4.1.2 Decision Variables

| | |
|---|---|
| $accommodate_{i,u,p}$ | A binary decision variable of 1 if task *i* is accommodated by resource *u* in placement *p*, 0 otherwise. |
| $travel_{i,j,u}$ | A binary decision variable of 1 if the arc from task *i* to task *j* is travelled by resource *u*, 0 otherwise. |
| $arrive_{i,u}$ | A continuous decision variable that assigns the time that resource *u* will arrive at task *i*. |

|  |  |
|---|---|
| $depart_{i,u}$ | A continuous decision variable that assigns the time that resource $u$ will depart from task $i$. |

Note where Negron and Miller make use of a decision variable named *perform*, this work uses the decision variable *accommodate*. This is to reflect that even though a resource of a supplier may be assigned to a specific task, it is not guaranteed to perform that task by the consumer, thus it has only been allocated to "accommodate" a task at this stage.

3.4.1.3 Input Variables

|  |  |
|---|---|
| $early_i$ | Beginning of time window for task $i$. |
| $late_i$ | End of time window for task $i$. |
| $minDur_i$ | Required time to complete task $i$. |
| $horizon$ | Planning horizon. |
| $travelTime_{i,j,u}$ | Length of time for resource $u$ to travel from location of task $i$ to location of task $j$. |
| $travelToBaseTime_{i,u}$ | Length of time for resource $u$ to travel from location of task $i$ to location of $u$'s supplier. |
| $resourceCount_{i,e}$ | The number of elements that task $i$ requests for a resource type $e$. |
| $value_i$ | The value to complete task $i$. |

3.4.1.4 Objective Function

The objective is to maximize the total value of all tasks completed. Each task has a varying number of resources request, some tasks may only request one resource, and others may request multiple resources. Thus, the value of a task is divided by the number of resources requested so that the full value can only be achieved if all the resources requested are accommodated. The objective function is:

$$Max \qquad \sum_{i \in T} \frac{value_i}{resourceCount_i} \sum_{u \in U, p \in P} accomodate_{i,u,p}$$

3.4.1.5 Constraints

The model has twelve constraints that are categorized as resource type constraints, network constraints, or time window constraints. The constraints ensure that the capacities of suppliers and capabilities of resources are not exceeded, so that the resulting schedule for resources is feasible for all suppliers to fulfil. The resource type constraints ensure that a resource of one type does not accommodate a task element request of a different type. The following is the resource type constraints:

(1) Ensure a Task, i, cannot be accommodated by a resource, u, it does not request.

$$accomodate_{i,u,p} = 0 \qquad \forall i \in T, e \in i, u \notin U(e), p \in P$$

The network constraints ensure that the resulting operations scheduling creates a feasible path for the resources' schedule. The following are the network constraints:

(2) Each Task, i, can only be assigned one placement per resource.

$$\sum_{p \in P} accomodate_{i,u,p} \leq 1 \qquad \forall i \in T, u \in U$$

(3) Each placement, p, can only be assigned one task per resource.

$$\sum_{i \in T} accomodate_{i,u,p} \leq 1 \qquad \forall u \in U, p \in P$$

(4) Ensure each Task, i, does not get more resources, u, per element type of its request.

$$\sum_{u \in U(e)} \sum_{p \in P} accomodate_{i,u,p} \leq resourceCount_{i,e} \qquad \forall i \in T, e \in i$$

(5) Ensure that the tasks are assigned in successive placements on the resource path.

$$\sum_{i \in T} accomodate_{i,u,p+1} - \sum_{i \in T} accomodate_{i,u,p} \leq 0 \qquad \forall u \in U, p \in P-1$$

(6) Force a travel arc to exist between two tasks performed successively.

$$accomodate_{i,u,p+1} + accomodate_{j,u,p} - 2 * travel_{i,j,u} \leq 1 \quad \forall i \in T, j \in T,$$

$$\forall u \in U, p \in P-1$$

The time window constraints limit the resulting operations scheduling to performing tasks within the desired time window. The following are the time window constraints:

(7) Resource must arrive after the beginning of the time window if it is accommodating task $i$.

$$early_i \times \sum_{p \in P} accomodate_{i,u,p} \leq arrive_{i,u} \qquad \forall i \in T, u \in U$$

(8) Resource must exit before end of time window if it is accommodating task $i$.

$$horizon - (horizon - \ late_i) \times \sum_{p \in P} accomodate_{i,u,p} \geq depart_{i,u}$$

$$\forall i \in T, u \in U$$

(9) Resource can depart the task only after the minimum required duration of time.

$$arrive_{i,u} - minDur_i \times \sum_{p \in P} accomodate_{i,u,p} \leq depart_{i,u} \quad \forall i \in T, u \in U$$

(10) Resource must begin schedule at location of supplier's base.

$$travelToBaseTime_{i,u} \leq arrive_{i,u} \qquad \forall i \in T, u \in U$$

(11) Resource must end schedule at location of supplier's base.

$$depart_{i,u} \leq horizon - travelToBaseTime_{i,u} \qquad \forall i \in T, u \in U$$

(12) Ensure sufficient travel time between tasks. $a_{i,j,u}$must be calculated a priori as explained in Subsection 3.4.1.6

$$depart_{i,u} + travelTime_{i,j,u} - a_{i,j,u}(1 - travel_{i,j,u}) \leq arrive_{j,u}$$

$$\forall i \in T, j \in T, u \in U$$

### 3.4.1.6 Linearization of Constraint

The LP model needs to constrain the time that a resource arrives at a subsequent task to be greater than the time that it takes to arrive at that following task, which is the time it departs the previous task plus the travel time between the two tasks. The intuitive way to write this constraint would be:

$$\left(depart_{i,u} + travelTime_{i,j,u}\right) \times travel_{i,j,u} \leq arrive_{j,u}$$

However, this constraint in not linear, because decision variables are being multiplied together. Therefore Ropke, Cordeau, and Laporte developed the following way to linearize the constraint ensuring sufficient travel time between two tasks [14]. The value, $a_{i,j,u}$ , which must be calculated before solving the model, is:

$$a_{i,j,u} = \max\left(0, late_i + minDur_i + travelTime_{i,j,u} - early_j\right)$$

This constraint, (12), will now either be redundant with the non-negativity constraints (if the resource does not travel between the two tasks) or will constrain the departure time of the last task plus travel time to be less than the arrival time to the next task.

$$(12)= \begin{cases} \left(depart_{i,u} - \left(late_{i,u} + minDur_i\right)\right) \leq \left(arrive_{j,u} - early_{i,u}\right) & , \ travel_{i,j,u} = 0 \\ depart_{i,u} + travelTime_{i,j,u} \leq arrive_{j,u} & , \ travel_{i,j,u} = 1 \end{cases}$$

### 3.4.2 Maximum Time Window Model Formulation

This e-commerce structure relies on two decision stages in each round of bidding, 1. the supplier deciding which tasks to service with which resources and when to service them and 2. the consumer deciding which resource bids have time windows overlapping that the consumer can accept to synchronize all the elements of a task. In stage 1, the supplier is trying to maximize the number of task elements accommodated which can result in very tight arrive and depart time windows.

As the supplier is trying to pack as many tasks into its sortie as possible to maximize the objective function, resources' time windows usually only last for the required minimum duration even if they had idle time between that task and its following task. This becomes a problem in stage 2 when a consumer may have received bids for all its task elements, but they are all in different sections of the task's initial time window. While these resources may be surrounded by idle time in which they could have stayed on the task longer and still had enough time to travel to their next task, this is not reported in the current MILP model. To address this inefficiency the concept of Maximum Time Window (MTW) is added to the model to increase the likelihood of task elements bids overlapping for consumer synchronization.

### 3.4.2.1 MTW Objective Function

The new objective function is very similar to the original model's objective function of maximizing the total value of tasks completed, but it also aims to maximize the time window that the resource schedules for a task. The MTW objective function is:

$$Max \quad \sum_{i \in T} \frac{value_i}{resourceCount_i} \sum_{u \in U, p \in P} accommodate_{i,u,p}$$
$$+ w_{timeWindow} \sum\nolimits_{i \in T, u \in U} (depart_{i,u} - arrive_{i,u})$$

This objective function maximizes the time windows for bids by maximizing the difference between the depart and arrive time for a resource to a task. An important parameter is the weight for stretching the time windows, $w_{timeWindow}$, as it controls the relationship of how valuable accommodating a task is to maximizing the time window for accommodated tasks. Reminder: *accommodate* is a binary decision variable so can be {0,1}, while *depart* and *arrive* are continuous decision variables that are constrained by [0,$late_i$ - $early_i$] (which for the context of this thesis is [0,25]).

In the most aggressive case for a supplier filling its sorties with as many tasks as possible, it will want to always choose to accommodate a task rather than maximize the time window of another task. For the particular parameters of this thesis, the weight should be set to:

$$w_{timeWindow} = \frac{accommodate_{MAX}}{value_{MAX} \times \Delta time\ window_{MAX}} = \frac{1}{100 \times 25} = 0.0004$$

In more conservative cases where a supplier may want to allocate larger time windows for high value tasks at the expense of not accommodating low value tasks, $w_{timeWindow}$can be increased depending on the desired ratio of large time windows to number of tasks accommodated.

### 3.4.2.2 Maximum Time Window Constraints

A constraint will need to be added to the original model as well to control the difference in depart and arrive times for non-accommodated tasks. Constraints (7) and (8) only restrict $arrive_{i,u}$ and $depart_{i,u}$ when $\sum_{p \in P} accomodate_{i,u,p} = 1$, so when inactive the difference in *depart* and *arrive* can become as large as the *horizon* value. Thus, the difference between *arrive* and *depart* times needs to be pushed to zero when a task is not being accommodated by that resource.

(13) Force the time windows of non-accommodated task elements to zero.

$$depart_{i,u} - arrive_{i,u} \leq horizon \times \sum_{p \in P} accomodate_{i,u,p} \qquad \forall i \in T, u \in U$$

Constraint (9) ensures $arrive_{i,u} \leq depart_{i,u}$, so no additional constraints are needed to prevent negative time windows from occurring.

### 3.4.3 Supplier MILP w/ Synchronization Model Formulation

This section address adding decision variables and constraints to ensure that resources a supplier is bidding on for a task are synchronized to the same arrival times. While altering the original MILP model to a MTW model increases the chance of overlapping time windows, it does not guarantee that the resources will be synchronized to arrive on task at the same time. To ensure the supplier is coordinating its resources, overarching task arrival and departure variables are introduced with three new constraints. These additions are to the nominal MILP model, as incorporating both overarching time variables and the MTW objective function is redundant.

Synchronization Decision Variables

| | |
|---|---|
| *Arrive<sub>i</sub>* | A continuous decision variable that assigns the time that all resources requested will arrive at task *i*. |
| *Depart<sub>i</sub>* | A continuous decision variable that assigns the time that all resources requested will depart from task *i*. |

#### 3.4.3.1 Synchronization Constraints

These super arrival and departure variables safeguard that for all resources servicing task *i* that they must be scheduled to service *i* for an overlapping time window of at least the minimum duration. The resources can arrive before the synchronized task servicing, and can stay after the synchronized task servicing. These synchronization constraints are:

(14) Overarching variable $Arrive_i$ is the latest arrival time of any resource servicing task *i*.

$$arrive_{i,u} - horizon \times \left(1 - \sum_{p \in P} accommodate_{i,u,p}\right) \leq Arrive_i$$

$$\forall i \in T, u \in U$$

(15) Overarching variable *Depart<sub>i</sub>* is the earliest departure time of any resource servicing task $i$.

$$depart_{i,u} + horizon \times \left(1 - \sum_{p \in P} accommodate_{i,u,p}\right) \geq Depart_i$$

$$\forall i \in T, u \in U$$

(16) Resources can depart the task only after the minimum required duration of time.

$$Arrive_i + minDur_i \leq Depart_i \qquad \forall i \in T$$

With constraint (16) constraint (9) becomes redundant and can be removed from the model.

This model addition of synchronization as a constraint rather than a desired feature, as in MTW, does constrict scheduling to sub-optimal solutions. suppliers may be forced to drop accommodating a task, because one element cannot be synchronized with the remaining elements for that task. This is highlighted in Example 2.

*Example 2: A consumer is requesting three resources for a task, {A,B,C}. This resource request goes to two suppliers.*

*Supplier 1 has three resources, {A,B,C}. But it is very busy and its resources are already committed to servicing other tasks.*

*Supplier 2 has two resource {B,C}. It has a very open schedule as it is not servicing any other tasks.*

*Supplier 1 can synchronize its B and C resources, but due to its tight schedule this aligned service time for the resource request is not compatible for any of the windows A is available. Under MTW supplier 1 would still bid to service A,B, and C, and supplier 2 would bid for servicing the task's B and C resources. After a few rounds of bidding supplier 1 and 2 could find a mutual arrival time and the Task could be fulfilled. However, with synchronization being* *required as a constraint, supplier 1 will always choose to send a bid that synchronizes B and C's arrival time, rather than a bid that can service only A. This will result in the consumer never getting a bid on A, and thus the task goes unaccommodated.*

Synchronization as a constraint in the first stage of the bidding ensures more matches that are desirable to the consumer, but there may be a decrease in total number of resource bids for the second stage. These dropped resource bids, while having a time window that did not align with other resources from the supplier, could have been aligned with the time window of bids from other suppliers. In summary, the synchronization constraints may lead to sub-optimal results for consumers, but allows for task requests to be filled and synchronized much faster.

### 3.4.4 Consumer MILP Model Formulation

This mathematical formulation is used by individual consumers to find the optimal selection of Elements Bids for a consumer's tasks. First the notations for sets, decision variables, and input variables are defined. Then the objective function and constraints are introduced.

#### 3.4.4.1 Set Definitions

The following sets are used in the formulation:

T = set of all tasks

B = set of all element bids

B(i,e) = set of all bids for a specific task and specific resource type *e*

#### 3.4.4.2 Decision Variables

| | |
|---|---|
| $bidSelect_b$ | A binary decision variable of 1 if element bid *b* is selected by the consumer, 0 otherwise. |
| $perform_i$ | A binary decision variable of 1 if task *i* is to be performed, 0 otherwise. |
| $Arrive_i$ | A continuous decision variable that assigns the time that bids will begin task *i*. |

| | |
|---|---|
| $Depart_i$ | A continuous decision variable that assigns the time that bids will end task *i*. |

3.4.4.3 Input Variables

| | |
|---|---|
| $early_b$ | Earliest time bid *b* can begin servicing its task. |
| $late_b$ | Latest time bid *b* can end servicing its task. |
| $cost_b$ | Cost of the element bid *b.* |
| *horizon* | Planning horizon. |
| $minDur_i$ | Required time to complete task *i.* |
| $resourceCount_{i,e}$ | The number of elements that task *i* requests for a resource type *e*. |

3.4.4.4 Objective Function

The objective is to maximize the number of tasks completed with minimal cost. When a supplier has multiple bids for a single element, and thus has choices on which bid to accept, this model choses the bids that have the lowest total cost. Cost being defined by the supplier. The objective function is:

$$Max \quad \sum_{i \in T} perform_i - \sum_{b \epsilon B} cost_b \times bidSelect_b$$

3.4.4.5 Constraints

These constraints ensure resource type and temporal conditions are respected. All element bids selected must respect one overarching arrival and departure time that lasts the task's minimum duration.

(17) Ensure that each task, i, does not get more bids for a resources type, *e*, per element type of its request.

$$\sum_{b \in B(i,e)} bidSelect_b \leq resourceCount_{i,e} \qquad \forall i \in T, e \in i$$

(18) A task, i, can only be performed if it has all resources requested.

$$perform_i \times \sum_{e \in i} resourceCount \leq \sum_{b \in B} bidSelect_b \qquad \forall i \in T$$

(19) Overarching variable *Arrive*$_i$ is the latest arrival time of any bid, *b*, selected for task $i$.

$$early_b \times bidSelect_b \leq Arrive_i \qquad \forall i \in T, b \in B(i)$$

(20) Overarching variable *Depart*$_i$ is the earliest departure time of any resource servicing task $i$.

$$late_b + horizon \times (1 - bidSelect_b) \geq Depart_i \qquad \forall i \in T, b \in B(i)$$

(21) A task can only be performed if it can be serviced for the required minimum duration of time.

$$Arrive_i + minDur_i \times perform_i \leq Depart_i \qquad \forall i \in T$$

The Consumer MILP is much simpler than the Supplier MILP as the consumer does not have to worry about network or travel time constraints. The consumer does not concern itself with routes or order of resources servicing its tasks. This results in the Consumer MILP for bid selection having much shorter runtimes when compared to the Supplier MILP for scheduling.

## 3.5 Implementation

While a multitude of optimization software could be used to solve this MILP, this project used Gurobi Optimizer 9.0.2 with Julia programming language through the extension Julia for Mathematical Optimization (JuMP) 0.18.The results and performance of this implementation are discussed in detail in Section 5.2.

# Chapter 4

## Formulation of Algorithm

The Mixed Integer Linear Programs introduced at the end of Chapter 3 provides an exact solution to the SSP. However, due to the complexity of the MILP, with a small increase in the number of tasks comes an exponential increase in runtimes for suppliers to find solutions. This motivates the development of an algorithm that uses heuristics to schedule resources in a reasonable time, as was shown in the literature review of Chapter 3.

This chapter introduces the *Infeasibility Cooling Adaptive Allocation for Resource United Scheduling* (ICAARUS). ICAARUS is a composite algorithm that draws its name from its three major components. Initially a *Simulated Annealing* criterion is used to schedule tasks with resources that may violate the task's time window. The allowed *infeasibility* is *cooled* as the algorithm progresses to force a solution to feasibility. In the improvement phases, multiple route improvement heuristics are applied to the schedule and the algorithm learns which methods are most effective, and then *adapts* its *allocation* method to the specific problem at hand. As this work focuses on tasks that require multiple resources, this algorithm handles synchronization across all of a supplier's *resources* for *united scheduling*.

### Notation

| Symbol | Description |
|---|---|
| $s$ | Schedule of tasks for a supplier's resources |
| $v(s)$ | Value of all tasks in schedule $s$ |
| $f(s)$ | Cost of all tasks in schedule $s$ |
| $u(s)$ | Utility of schedule $s$, $u(s) = v(s) - f(s)$ |
| $\alpha,\beta$ | Schedule cost parameters |
| $\varphi$ | Weight adjustment parameters |
| $y$ | Percentage of task list to be removed in removal phase |
| $q$ | Count of tasks to be removed in removal phase |
| $\psi,\omega$ | Related removal parameters |

| | |
|---|---|
| $p$ | Randomness parameter in related and worst removal |
| $DL$ | List of dropped tasks |
| $\Omega$ | Cross Synchronization Matrix |
| $\lambda$ | Lambda-insertion parameter for task insertion index search |
| $\Gamma_1$ | Set of removal methods |
| $\Gamma_2$ | Set of insertion methods |
| $w_i$ | Weight of removal and insertion methods |
| $\pi_i$ | Score of removal and insertion methods |
| $\kappa$ | Weight adjustment parameter |

## 4.1 Confirmation Lists

The bidding system outlined in Section 2.2 depends on a supplier remembering which tasks it has bid on servicing. This bookkeeping is done by each supplier storing two lists:

1. Unconfirmed Task List – A list of all tasks that the supplier had in its previous schedule, and has not yet received a confirmation of acceptance or rejection of the element bid from the issuing consumer.

2. Confirmed Task List – A list of all tasks for which the supplier has received an element bid acceptance, from its issuing consumer.

To improve the runtime of ICAARUS, these lists also contain which resource is supposed to service this task and at what time. This information is used in the construction phase of future schedules to intelligently insert old tasks into what was once the optimal resource path position. This “hot start” to scheduling allows the current round of bidding to utilize the progress made in previous bidding rounds. While a schedule will change when tasks are removed from a consumer rejecting a bid, the memory of this skeleton schedule greatly improves future bidding round runtimes as seen in Chapter 5.

In the first round of bidding, these two lists are empty as no bids have been sent by the supplier. After the first round of bidding, suppliers must manage updates to their schedule and changes to the Confirmation Lists. After the supplier solver has found its new schedule, tasks in the schedule are compared to the two confirmation lists and three possible scenarios can occur:

1. If the task is not on either confirmation list:

   The task is added to the Unconfirmed Task List and the consumer is sent a notification of the supplier's bid.
2. If the task is already on the Unconfirmed Task List:

   The task's arrival and departure times are updated on the Unconfirmed Task List in case of changes in the new schedule. The consumer is sent a notification of the supplier's bid again.
3. If the task is already on the Confirmed Task List:

   The task stays on the Confirmed Task List, and no message is sent to the consumer, as they have already confirmed notification of the bid(s).

## 4.2 Construction Phase

In the construction phase, initial paths are created for the resources of the supplier. The first part to any *composite algorithm* is the construction of an initial route, the second being the improvement of those routes. These two phases are utilized by the construction phase making potentially flawed scheduling assumptions for the sake of speedy construction. This is tolerable as the improvement phase will later correct these errors. Specifically, ICAARUS tolerates time window violations initially, as a later phase will correct these infeasibility issues.

The general challenge of the construction phase is striking a balance between construction speed vs. solution optimality. If the construction phase creates a schedule with only a few tasks in its task list, the improvement phase's search space is decreased with less possible combinations for a schedule. This limited task list is only one neighborhood of the entire possible search space, and is not guaranteed to house the global optimal solution. Since it is hard to know which tasks will be in the optimal solution from the beginning, adding as many tasks as possible in the initial construction widens the scope of the search space. This increases the chances of finding the global optimal schedule, but at the cost of exponentially increasing the search runtime.

First, the Construction Phases for Negron's *Composite Operations Planning Algorithm* (COPA) and Herold's Dual-Collections are analyzed for insight into striking a balance between construction speed and solution optimality. Then ICAARUS's construction phase is presented and how it uses *Simulated Annealing* to schedule resources in a united manner.

### 4.2.1 COPA Construction Phase

COPA's construction phase takes input on UAV types, tasks, and task locations to select initial paths. The input notation is defined here:

| | |
|---|---|
| $value_t$ | Value of task $t$ |
| $obsTime_t$ | Required time to complete task $t$ |
| $idleTime_t$ | Length of time that the UAV will have to wait before starting task due to a time window constraint |
| $late_t$ | End of time window for task $t$ |
| $travelTime_{(t'),(t),u}$ | Length of time for a UAV to travel from task $t$ to $t'$ |

below in Equation 1 is the cost-benefit ratio that was used by Negron to find the best next task to be assigned to a UAV's path [1]:

**Equation 1: Negron Cost-Benefit Ratio**

$$CBR_t = \frac{value_t}{w_1 \times obsTime_t + w_2 \times idleTime_t + w_3 \times late_t + w_4 \times traveltTime_{(t'),(t),u}}$$

Parameters $w_1$, $w_2$, $w_3$, and $w_4$ are the weights used in the cost-benefit ratio to manage each factor's importance to scheduling. Tuning these parameters must be learned for the system beforehand.

The following steps describe COPA's construction procedure. The procedure is repeated for each UAV type, and produces the set $P_{i,k}$, the set of ordered tasks for the path of UAV $k$ of type $i$:

(1) Obtain list of all tasks that can be serviced by UAV of type $i$. This is set $T_i$
(2) Set $currentTime_k = 0$; current task is starting at home base.
(3) For each UAV of Type $i$:
    a. For each feasible tasks in $T_i$ calculate $CBR_t$

b. Find $\max\{CBR_t\}\, \forall t \in T_i$; this is $t*$

c. Add $t*$ to $P_{i,k}$

d. Remove task $t*$ from $T_i$

e. Update *currentTime*$_k$. If *currentTime*$_k$ > *horizon, start path for next UAV;* $k = k + 1$

This procedure provides an initial path, in the form of an ordered set of tasks, for each UAV. As previously noted, the ordered set of locations for UAV $k$ of type $i$ is represented by $P_{i,k}$. $P_i$ represents the set of paths for UAVs of type $i$ ($P_i = \{P_{i,1}, P_{i,2}, \ldots\}$). $P$ denotes the entire set of paths and corresponding resources, which is also the set of composites. However, by creating each path independently, COPA's construction phase is unable to service tasks that require collaboration among resources.

### 4.2.2 Dual-Collection Construction Phase

Herold extends COPA's construction procedure to handle dual-collection tasks by creating two separate task requests for each dual-collect [9]. Initially there is no method used to synchronize these two identical tasks, it is left for a later phase to correct. This keeps the runtime for the construction phase as reasonable as COPA's as the set of Tasks increases linearly by the number of dual-collections, $|T_{total}| = |T| + |T_{DualCollections}|$.

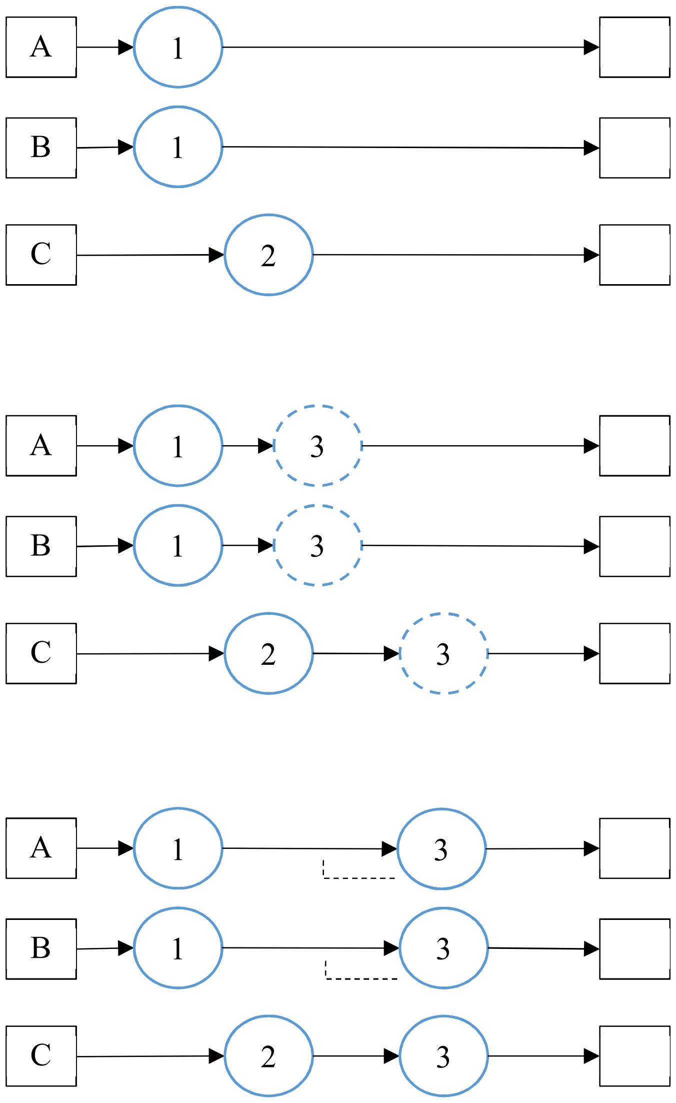
However a simple construction phase that lacks attention to synchronize resources can ultimately have expensive runtime costs in the long run. Herold's method to align two initially separate tasks is computationally expensive in worst-case scenarios. There is also the potential error that only one of the dual tasks will be added to the set of paths. Herold's work revealed that scheduling multiple versions of a task leads to few task pairs advancing in the solution after the construction phase. To overcome this handicap, tasks being scheduled in the construction phase have their cost-benefit ratio inflated by doubling the value of the task. This increases the likelihood of dual collections being possible, by increasing the request's attractiveness to the route construction procedure.

Herold's construction phase operates under the assumption that tasks only need to be synchronized across at most two resources, and only a few of the total tasks may require this difficult temporal alignment. Thus, these prior methods do not scale well for this thesis's goal of schedule creation where a majority of tasks require synchronization across a variable number of resources ({1,2,3, or 4 resources}).

### 4.2.3 ICAARUS Construction Phase

ICAARUS constructs initial paths for all resources simultaneously, uniting the resource scheduling specifically for handling task synchronization. This is done by adding tasks in their entirety to the schedule, with all elements of that task maintaining synchronization across resources. If a new task is added that requires one element of a task arriving delayed, all elements of that task are delayed. This synchronization is shown in Figure 12:

**Figure 12: Example of Resource United Scheduling**



*In the example of Figure 12 a supplier has three resources, {A,B,C}. Task 1 requests resource A & B, and Task 2 only requests resource 2. Task 1 is closer to the supplier's home base then Task 2, so requires less travel time, thus Task 1 is scheduled earlier than Task 2.*

*Task 3 request comes in asking for three resources, {A,B,C}. The earliest each resource could arrive at Task 3 is shown in the middle diagram. Servicing Task 3 at the earliest possible times for each resource would result in all three resources not being synchronized as resource C cannot service Task 3 until after resource A and B would service it. For resource united scheduling, Task 3 is pushed back in the schedule for Resource A and B to the time the earliest time that* <u>*all*</u> *requested resources can synchronize their services. This may result in inefficient idle time as A and B arrive at Task 3, but are inactive till the beginning of the synchronized service time for Task 3.*

Figure 12 shows the simplest case of a supplier only possessing one resource of each type, but when a supplier possesses multiple resources of the same type, a decision must be made which resource to allocate to the task. This is done with speed and fairness in mind through the following resource element selection procedure:

```
(1) For each resource $u$ requested by task $t$;
    a. $rc$ is the number of elements needed of type $u$
    b. For each element $e$ that the supplier has of $u$
        i. Find earliest arrival time for $e$ to $t$; This is $arriveTime_e$
    c. Sort all $arriveTime_e$ into $arriveTimeList$
    d. Assign first $rc$ elements to service task $t$; $arriveTimeList[1:rc]$
```

ICAARUS construction phase takes input on resource types, tasks, and the planning map to select initial paths. The input notation is defined below:

| | |
|---|---|
| $value_t$ | Value of task $t$ |
| $resourceCount_t$ | Number of resources requested by task $t$ |
| $idleTime_t$ | Length of time that resource $u$ will have to wait before starting task $t$ due to a time window or synchronization constraint |

| | |
|---|---|
| *$minDur_t$* | Minimum time required for resources to service task *t* |
| *$late_t$* | End of time window for task *t* |
| *horizon* | Planning horizon |
| *$travelTime_{(t'),(t),u}$* | Length of time for resource *u* to travel from task *t* to *t'* |
| *mapDistance* | Time it would take to travel the longest distance possible of the planning map. |

below in Equation 2 marks the benefit-to-cost ratio, BtoC, that is used to find the best next task to be allocated resources in the schedule:

**<u>Equation 2: Benefit-to-Cost Ratio</u>**

$$BtoC = \frac{value_t * resourceCount_t}{\frac{\max\limits_{u \in U} idleTime_{t,u}}{minDur_t} + \frac{late_t}{horizon} + \frac{\max\limits_{u \in U} travelTime_{t',t,u}}{mapDistance}}$$

The numerator is multiplied by *$resourceCount_t$* to increase the likelihood of synchronization tasks being selected as used effectively by Herold dual-collection tasks [9]. The maximum *$idleTime_{t,u}$* and *$travelTime_{t,t',u}$* reflects the fact that resources, selected for task *t*, may be coming from different tasks and have differing idle and travel times. The three variables in the denominator *$idleTime_t$*, *$late_t$*, and trav*$elTime_{(t'),(t),u}$* are normalized by *$minDur_t$*, *horizon*, and *mapDistance* respectively to capture their relative weight for BtoC analysis. Such as idle time increasing by one minute should penalize a task more than travel time increasing by one minute as productive travel is acceptable in a large planning map, while unproductive idle time for a task of short duration is undesirable.

Negron's COPA construction phase achieves its speed by advancing linearly through the planning horizon [7]. Once a task is added to a path, COPA does not look back to see if tasks can be inserted earlier in the spaces left by idle time. This is a problem for resource united scheduling as it is prone to create schedules with increased idle time. The reason being it pushes many elements back in a task's time window to accommodate synchronizing across all resources. With more idle time comes less available time to schedule a task within its allotted time window, running the risk of infeasibility.

*Simulated Annealing* is utilized to manage infeasibility by scheduling tasks outside of their requested time windows. This is to maintain the speed of linear path planning, while becoming more accommodating of resource united scheduling's idle drawbacks. The variable *temp* controls the threshold for accepting tasks in an infeasible time assignment. These infeasible task placements in the path are expected to be remedied in the improvement phase or dropped from the schedule all together.

The following steps describe the ICAARUS construction procedure:

```
(1) Collect requests into one task list. This is set T
(2) Collect Unconfirmed and Confirmed Task List into one task list; This is set C
(3) For each task t in T
    a. If a task requests a resource from the supplier, but the supplier does not have
       that resource, remove the element request from the task list.
    b. If a task's element request list is made null, remove task from T.
(4) While $T \neq \emptyset$:
    a. best_BtoC = -∞; best_t = ∅; best_pathList = ∅
    b. For each task t in T:
        i. Resource element selection procedure finds best paths to add t on;
           $pathList_t$
        ii. Find BtoC and departTime for t
        iii. If $departTime_t \leq late_t$
             or $rand() < e^{\frac{departTime_t - late_t}{horizon * temp}}$
            1. If best_BtoC < BtoC
                a. best_BtoC = BtoC; best_t = t; best_pathList = $pathList_t$
    c. Quit if best_BtoC = -∞
    d. Add best task to earliest available resources on the schedule; Add best_t to
       path(s) in best_pathList
    e. Calculate $cTime_u \forall u \in U$; $cTime_u$ = departTime of the last task on u's path
```

```
        f. scheduleUpdate = TRUE
        g. While scheduleUpdate:
            i. scheduleUpdate = FALSE
            ii. Find earliest task, $t_e$ ,in C where $arriveTime \leq cTime_u \forall u \in U'$;
                U' being the subset of U that $t$ is allocated
                1. Add $t_e$ to the schedule
                2. Remove $t_e$ from C
                3. Update altered paths $cTime_u$
                4. scheduleUpdate = TRUE
    (5) Add any tasks still in C to end of schedule
```

The ICAARUS construction procedure relies heavily on the improvement phase to correct infeasibility in the schedule, as seen by tasks from set C only being added to the new schedule after their original scheduled time is violated surpassed. This method of adding sections of a previous schedule to a new schedule is to allow new task requests filling potential holes in the previous schedule. These potential holes are where resources have open windows of inactivity. This addition method aims to keep task placement patterns similar to what previous rounds of ICAARUS had found as the optimal order of tasks to service.

## 4.3 Improvement Phase

The improvement phase of ICAARUS improves the paths created by the construction phase. This phase uses five removal methods to improve the paths: *Related Removal*, *Worst Removal*, *Synchronized-Services Removal*, *Route Removal*, and *Random Removal* with two insertion methods: *Best Insertion* and *Regret Insertion*.

The improvement phase continuously repeats removing tasks and re-inserting them to create new improved paths. These methods are known as Large Neighborhood Search (LNS) for searching individual domains of the search space, known as neighborhoods [13]. The methods effect several tasks at a time, dealing with a larger scope of the search space than some methods that only tweak two task positions at a time. Hendel improved their LNS heuristic for MIP by a reward function to learn to distinguish between successful and unproductive methods calls, this is known as Adaptive LNS (ALNS) [13]. ICAARUS proposes a reward function for tracking if these methods create improved or novel schedules, and adapts the likelihood of that method's future use accordingly.

In order to encourage searching new neighborhoods of the solution space, the current schedule for improvement is not always the best schedule found so far. While ICAARUS stores the best solution found so far, the improvement phase can alter the current iteration's schedule into one of lesser value. This is done in order to prevent solutions becoming trapped in a local optimum.

### 4.3.1 Infeasibility measurement

The ICAARUS improvement phase takes schedules with tasks assigned arrival times outside of their time windows, and may reschedule them to arrival times that are still infeasible. This is necessary, as mentioned in Section 4.2.3, because resource united scheduling commonly results in large gaps of idle time. If a scheduler were to only assign tasks to resources that can arrive within their time window, then a greedy heuristic would create a schedule with very few tasks and make finding high value schedules impossible. A final solution can not have any infeasible task assignments, therefore a S*imulated Annealing* criteria is used to guide the improvement phase out of infeasibility.

Initially, the temperature variable that controls the degree of acceptance of infeasibility is set high. This variable is then brought down through iterations to force the improvement methods to either create a feasible schedule or drop tasks from the schedule. This corresponds with schedules that are originally greedily created with many tasks to not rule out potential solutions too early. However, after iterations of the improvement phase the schedule should begin to approach an optimal solution and know which tasks cannot be included in a feasible solution.

What quantifies one schedule as more infeasible than another is measured through the cost of the schedule, *f(*s*)*, defined below:

$$f(s) = c(s) + \alpha P_1(s) + \beta P_2(s)$$

c(s) = Travel Time costs of all resources on their paths

$P_1$(s) = Sum of all differences between late and depart times

$P_2$(s) = Sum of all differences between horizon and depart times

If a solution is feasible, then $P_1(s) = P_2(s) = 0$. The variables of α and β adjust the penalty of violating these thresholds of feasibility. The values of these parameters are increased or decreased at the end of every improvement iteration. This helps facilitate exploration throughout the search space. The rules for adjusting are given below:

$$\alpha = \begin{cases} \max\left(\alpha_{Min}, \dfrac{\alpha}{1+\varphi_1}\right), & If\ solution\ is\ feasible \\ \min(\alpha_{Max}, \alpha \times (1+\varphi_2)), & If\ solution\ is\ infeasible \end{cases}$$

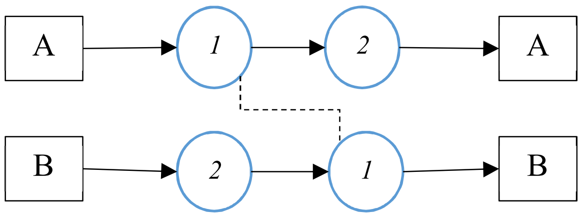
$$\beta = \begin{cases} \max\left(\beta_{Min}, \dfrac{\beta}{1+\varphi_1}\right), & If\ solution\ is\ within\ planning\ horizon \\ \min(\beta_{Max}, \beta \times (1+\varphi_2)), & If\ solution\ violates\ planning\ horizon \end{cases}$$

The bounds ($\alpha_{min}$, $\alpha_{max}$) and ($\beta_{min}$ , $\beta_{max}$) serve to control the interval of the infeasibility measuring weights. A streak of infeasible or feasible solutions for too long may make these weights too massive or too insignificant to be corrected when a new feasible or infeasible solution, respectively, is found. Thus, these bounds constrain the weights to stay within effective ranges.

### 4.3.2 Avoiding Cross Synchronization

Due to some tasks the supplier is servicing requiring synchronization across resources, insertion methods (Section 4.3.4) must avoid cross-synchronization in the resulting schedule. Cross-synchronization is when two synchronization tasks become entangled, and makes synchronization impossible. Figure 13 illustrates this problem:

**Figure 13: Example of Cross-Synchronization**



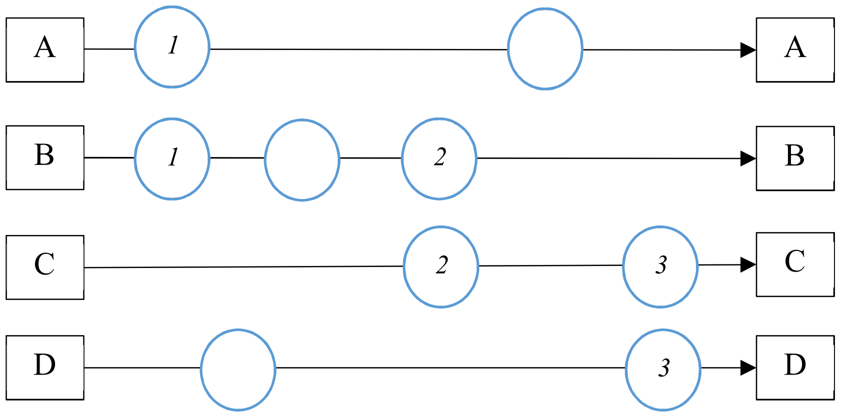
*In the example of Figure 13, there are two tasks, task 1 and task 2, that each require synchronization across resources. For resource A, the path services task 1 first then task 2. For resource B, the path services task 2 first then task 1. This entanglement makes synchronization of both tasks infeasible. Synchronizing either task puts the other task simultaneously both before and after the original task, making synchronization of the other task impossible.*

Cross-synchronization is not a problem for tasks that request only a single service from the supplier. Thus, avoiding cross-synchronization only requires monitoring tasks requesting multiple resources from a supplier. Insertion methods are prevented from creating a cross-synchronization scenario through use of a cross-synchronization matrix, Ω.

### 4.3.2.1 Checking for Cross-Synchronization

To prohibit cross-synchronization cases, the use of a $|N_{syn}| \times |N_{syn}|$ square matrix Ω records the visit sequence of each pair of synchronization tasks. $N_{syn}$ being the number of tasks in the schedule that require synchronization. An example of this matrix is shown in Figure 14:

**Figure 14: Example of Cross-Synchronization Matrix**



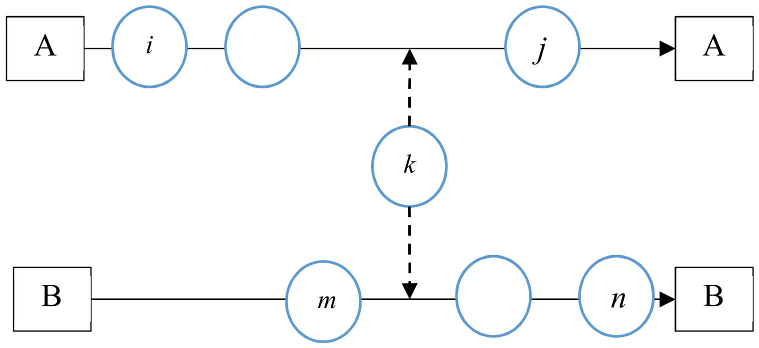
| *Ω* | *1* | *2* | *3* |
|---|---|---|---|
| *1* | 0 | 1 | 1 |
| *2* | 0 | 0 | 1 |
| *3* | 0 | 0 | 0 |

*In the example of* Figure 14*, a schedule with 3 synchronization tasks, {1,2,3} are shown on the right with its corresponding cross-synchronization matrix on the right. There are no rows or columns in Ω for single service tasks. On resource B's path, Task 1 comes before Task 2, so consequently Ω[1,2] equals 1. On resource C's path, Task 2 comes before Task 3, so Ω[2,3] also equals 1.*

*Despite no resource path servicing both Task 1 and Task 3, they are linked together through Task 2. Servicing sequences can be transferred between various routes, so as a result Ω[1,3] equals 1. All other values in the matrix equal 0.*

With a cross-synchronization matrix, ICAARUS can check if inserting a synchronization task into multiple routes simultaneously will result in the problem of cross-synchronization. To do this, the insertion method must find the closest synchronization tasks before and after the proposed insertion spot on the selected resource paths. This can be seen in Figure 15. Tasks immediately before or after the desired insertion spot are not necessarily synchronization tasks, they could be just single resource tasks. This requires ICAARUS to search sequentially outwards on the resource path until it finds the first synchronization tasks both before and after the desired task insertion spot. In addition, the proposed new task could be getting inserted into a spot that has no synchronization task before or after it. In that case there is one less task to check for cross-synchronization issues.

**Figure 15: Checking for Cross-Synchronization when inserting a Synchronized Task**



In the above example, the synchronization task *k* is being considered for insertion into the paths of resource A and B. *i* is the first synchronization task on A's path preceding the insertion spot of *k*, and *j* is the first synchronization task on A's path following the insertion spot of *k*. In addition, $m$ is the first synchronization task on B's path preceding the insertion spot of $k$, and n is the first synchronization task on B's path following the insertion spot of $k$. To find if this insertion would cause a cross-synchronization, ICAARUS will check $\Omega[j,m]$ and $\Omega[n,i]$. If both of them equal 0, then this is a valid insertion. However, if either one of them equals 1, then this insertion is removed from consideration as it would cause cross-synchronization.

#### 4.3.2.2 Constructing Cross-Synchronization Matrix

At the beginning of every insertion method, the current schedule $s$ will need a corresponding cross-synchronization matrix Ω. Initially Ω is an all zero $|N_{syn}| \times |N_{syn}|$ square matrix. Then Ω is constructed with Algorithm 1. In steps 2-5, task precedence in each route is addressed separately. In steps 6-15 the servicing sequence among each pair of synchronized tasks that depend on different paths is addressed, until all relationships disseminate throughout Ω.

Algorithm 1: Construct Cross-Synchronization Matrix

```
1    Create list of all synchronized tasks, T
2    For k=0; k ≤ |Supplier Resource Count|; k++
3        Select each pair of synchronized tasks, (i,j), in path k
4            If i is serviced before j Then Ω[i,j] = 1
5    End For
6    Do
7    For i=0; i ≤ |T|; i++
8        For j = 0; j ≤ |T|; j++; i ≠ j
9            If Ω[i,j] = 0
10               For k = 0; k ≤ |T|; k++; k ≠ i; k ≠ j
11                   If Ω[i,k] = 1 and Ω[k,j] = 1 Then Ω[i,j] = 1
12               End For
13       End For
14   End For
15   Until Ω is not updated
```

#### 4.3.2.3 Updating Cross Synchronization Matrix

After every synchronized task insertion, matrix **Ω** must be updated to reflect the updated schedule $s$. This update is done by Algorithm 1 steps 6-15, to disseminate the new servicing sequence pairs across paths.

*For example, from Figure 15, we update* Ω *as follows. First, because in A's path the immediate synchronization task following task k is j, the tasks serviced after j must now be serviced after k. Thus, row k of* Ω *is updated, i.e., for each column x of* Ω*, if* Ω*[h, x] is 1 then set*

*Ω[k, x] equal to 1. This is repeated for every path the task is being inserted into. So for visit sequence in B's path, for each column x of Ω, if Ω[n,x] is 1 then set the value of Ω[k, x] equal to 1.*

*Next, because in A's path the immediate synchronization task preceding k is i, the tasks serviced before i must also be serviced before k. Thus, column k of Ω is updated, i.e., for each column x of Ω, if Ω[x, i] is 1, then set Ω[x,k] equal to 1. This is repeated for every path the task is being inserted into. Therefore, for visit sequence in B's path, for each row x of Ω, if Ω[x,m] is 1 then Ω[x,k] is also set equal to 1.*

When a removal operator has been selected and tasks are removed from the solution, the cross-synchronization matrix is updated using the construction procedure for Ω. Whether Ω is updated after each individual task or a set of tasks are removed is specified for each removal method.

### 4.3.3 Removal Methods

The improvement phase faces a trade-off of speed versus optimality when it comes to search size. If it removes a few tasks, then less time is required to re-insert them, and thus the improvement phase can have more iterations. However, more iterations are unhelpful as the solution becomes trapped in a local optimum of the search space. While removing numerous tasks increases the time to re-insert them, but it also increases the likelihood of re-inserting them in an order that improves the schedule's utility. To balance this trade off, each iteration of the improvement phase calculates selects a random percentage, *y*, from a uniform distribution range of [0.15,0.30], which is then used to calculate $q$, the number of tasks to be removed:

$$q = y \times |task\ list|$$

By varying the number of tasks to be removed, and consequently re-inserted, ICAARUS balances speed with search space depth.

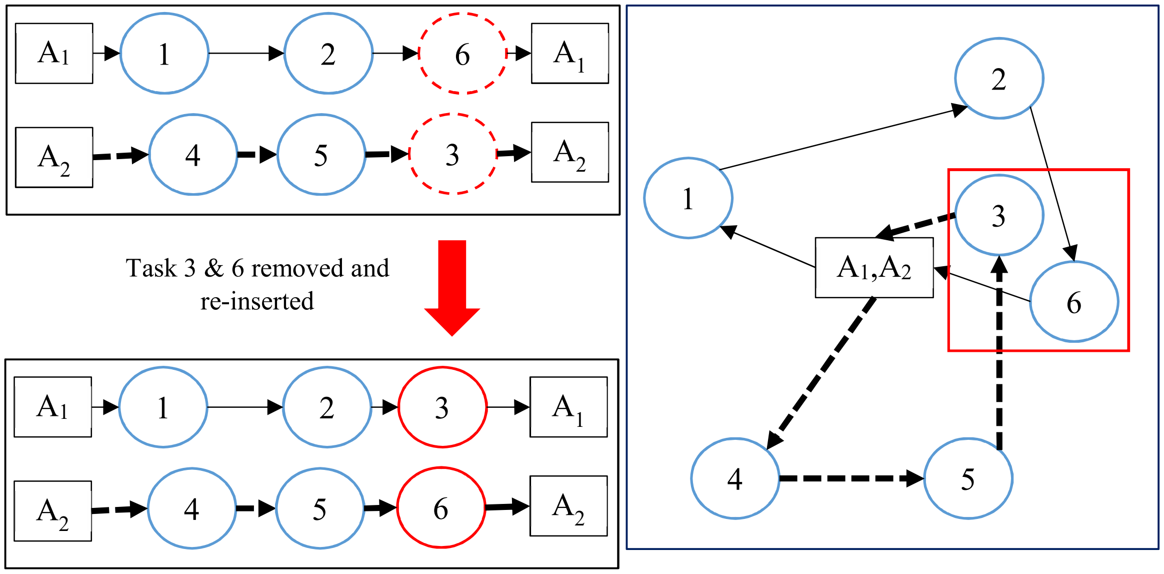
#### 4.3.3.1 Related Removal

The related removal is inspired from the "Shaw Removal" operator [10]. This removal method searches all tasks for pairs that are related to each other, because removing and later re-inserting similar tasks has a higher probability they will be reshuffled into feasible and better solutions. Two tasks i and j's relatedness is evaluated based on their distance and arrive times and measured by a Relatedness Score:

$$R(i,j) \ = \psi * distance_{i,j} + \ \omega * |arriveTime_i - arriveTime_j|$$

Ψ and ω are weight parameters.

**<u>Figure 16: Example of Related Removal</u>**



*In the example of* Figure 16 *there are two resources, $A_1$ and $A_2$. $A_1$'s current schedule is servicing tasks {1,2,6} and $A_2$'s current schedule is servicing tasks {4,5,3}. Task 3 and 6 is the closest pair to each other in both time and space, as seen in the left and right diagrams respectively. If Task 3 were removed, its most related task would be Task 6, and would be chosen as the next task to be removed by the Related Removal method. An ideal re-insertion would switch 3 and 6 assignments and create more efficient paths.*

Below is Algorithm 2 that utilizes this removal method:

Algorithm 2: Related Removal

```
1   Create list L of all tasks in schedule s
2   Randomly select a task i from L
3   D = {i}
4   Remove task i from L
5   **While** |D| < q
6       Randomly select a task j from D
7       Sort L in decreasing order according to the relatedness to i => R(i,j)
8       Choose y, a random number from uniform distribution in [0,1]. Set $E = y^p \times |L|$
9       Select task L[E] from set L. Insert it into set D and remove from set L.
10  **End While**
11  Remove tasks in D from the current schedule s
12  Synchronize(s)
```

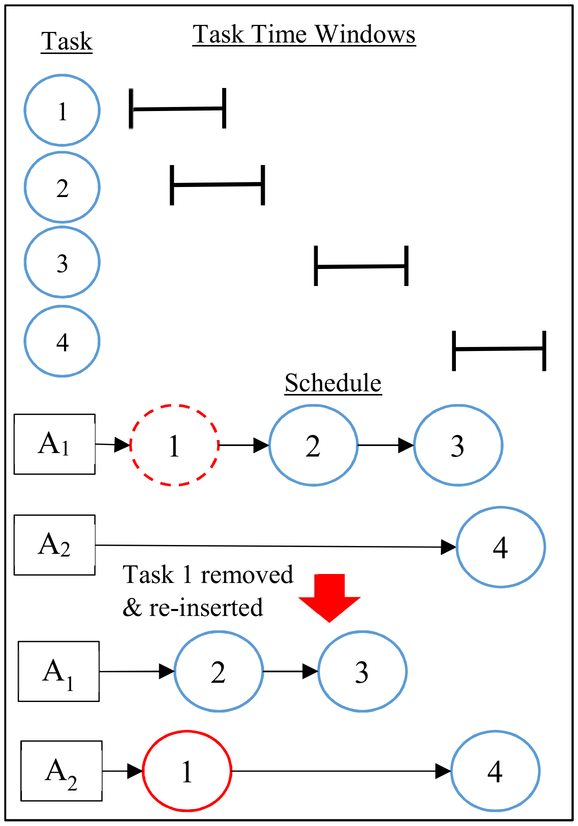
Parameter $p$, Line 8, introduces randomness in the selection of related customers. This is used in Line 9 for the effect of decreasing the likelihood that the same related pairs are constantly chosen, allowing for greater search of the solution space.

Synchronization of the schedule after each removal iteration could make some tasks have similar relatedness scores when their original arrival times were actually quite different. To prevent this mistaken rating, synchronization only occurs after all tasks are removed. This saves time as only one synchronization check is needed to send an accurate schedule to the subsequent task insertion method.

4.3.3.2 Worst Removal

The worst removal removes the tasks that would result in the greatest savings for the schedule. The greatest savings are measured by what creates the greatest decrease in the schedule utility u(s), thus removing tasks that do not contribute much to v(s) and are costly in c(s), $P_1$(s), or $P_2$(s). Figure 17 presents a scenario that showcases worst removal:

**Figure 17: Example of Worst Removal**



*In the example of* Figure 17 *the schedule has four tasks, {1,2,3,4}. In the top schedule resource* $A_1$ *services Task 1 first, this results in Tasks 2 & 3 being serviced outside of their time windows. Thus, the top schedule is infeasible. The Worst Removal method would remove Task 1, which allows a schedule to be created with Task 2-4 in feasible time windows. A resulting insertion may assign Task 1 to resource* $A_2$ *which has plenty of room in its schedule.*

Below is Algorithm 3 that utilizes this removal method:

Algorithm 3: Worst Removal

```
1   Create list L of all tasks in schedule
2   D = {}
3   **While** |D| < q
4       Sort L in decreasing order according to schedule utility u(s) if task i is removed
5       Choose y, a random number from a uniform distribution in [0,1].
6       Set $E = y^p \times |L|$
7       Select task L[E] from set L. Insert it into set D.
8       Remove task L[E] from schedule s and set L.
9       Synchronize(s)
10  **End While**
```

Parameter p for randomness in selection of worst customers is used in a similar manner as described in related removal for increased search of the solution space. Differing from related removal is the need for synchronization after each task removal. A task i may appear as "costly" because earlier task j forces it to be late, but after the troublemaker task j is removed then task j no longer significantly increases f(s).

#### 4.3.3.3 Synchronized-Services Removal

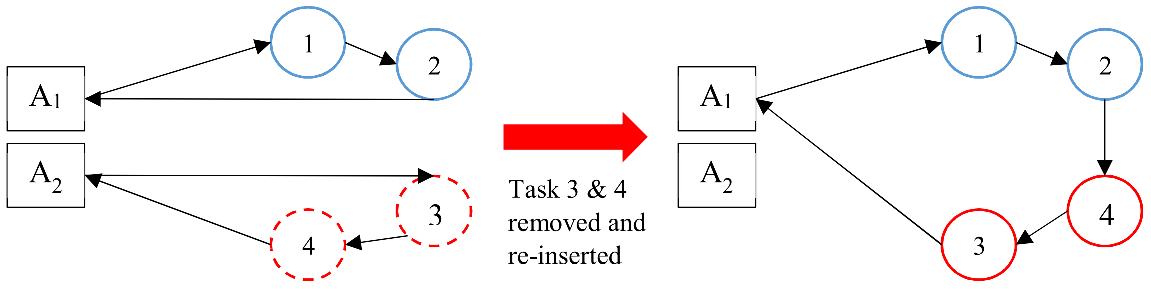
This removal method targets tasks that have two or more resource requests. tasks requiring multiple resources to be synchronized can appear as very valuable, but become choke points for the whole schedule. These complicated tasks, when put in a sub-optimal position can force numerous following tasks to be outside of their feasible time windows. Meanwhile the task that is the source of the backup remains feasible and thus has an apparent "good" score.

Which synchronized-services tasks to be removed are selected randomly. Because the synchronized tasks are selected randomly, not based off of schedule features, re-synchronization can be done just once. This re-synchronization adjustment of the schedule is done after the q tasks are removed.

#### 4.3.3.4 Route Removal

This removal method randomly selects one resource of the supplier at a time, and removes all tasks on its path, clearing as many resource paths as required to remove q tasks from the schedule. This is done to see if groupings of tasks similar in distance can be better served by another resource that may be servicing other tasks in that geographic area. This is depicted in Figure 18:

**Figure 18: Example of Route Removal**



*In the example of* Figure 18 *the schedule has four tasks, {1,2,3,4}. The initial schedule has resource $A_1$ service Task 1 & 2, and resource $A_2$ service Task 3& 4. In this map the four tasks are all close spatially, but far from the supplier that houses $A_1$ and $A_2$. The Route Removal method selects $A_2$'s path to eliminate, and results in Task 3 & 4 being reinserted to $A_1$'s path. Assuming this new path is feasible with all tasks time windows, this results in lower total cost as $A_1$ has already traveled a long distance to service Task 1 & 2, it would be a waste to have $A_2$ travel that far as well.*

The re-synchronization adjustment of the schedule can be done just once, after the q tasks are removed.

4.3.3.5 Random Removal

To avoid becoming stuck in local optimum solutions, this method removes randomly selected tasks. This helps push ICAARUS to explore a wide breadth of the search space. Because the tasks are selected randomly, not based off of schedule features, re-synchronization can be done just once. This re-synchronization adjustment of the schedule is done after the q tasks are removed

### 4.3.4 Insertion Methods

Once q tasks have been removed from the schedule, those q tasks are added to a list of all tasks dropped in previous removal phases. This dropped list, DL, gives the system the memory to hold onto tasks through multiple improvement phase iterations. These insertion methods are not restricted to insert all tasks in the DL. An iteration may produce a schedule where certain tasks appear impossible to allocate resources, therefore for feasibility the tasks must be removed.

However, a later iteration could service those tasks to produce a new best schedule. ICAARUS inherently accepts that the schedule from the construction phase may contain tasks that make feasibility impossible. ICAARUS will continue to evaluate these dropped tasks for possible insertion in order to maximize schedule value.

#### 4.3.4.1 Best Insertion

The best insertion inserts the tasks that would result in the greatest reward for the schedule. The greatest reward is measured by what creates the greatest increase in the schedule utility u(s), thus inserting tasks that contribute greatly to v(s) and are not costly in c(s), $P_1(s)$, or $P_2(s)$. Below is the algorithm that utilizes this removal method:

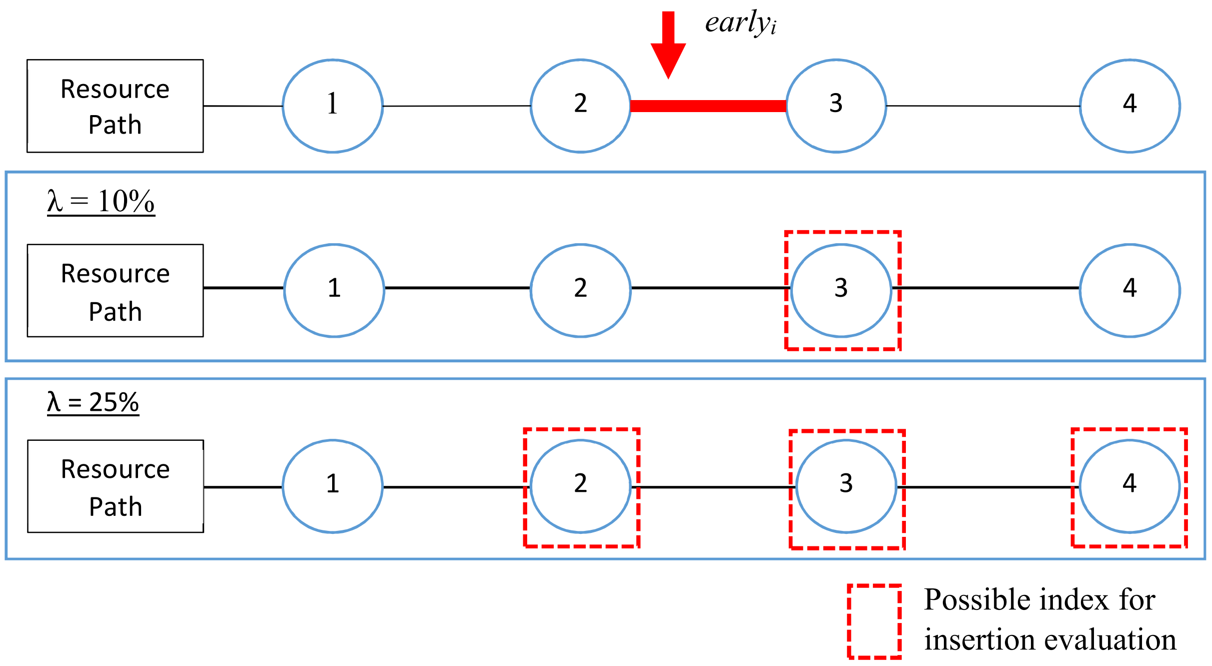
Algorithm 3: Best Insertion

```
1   $s_{curr} = s$
2   D = DL
3   Ω = Cross Synchronization Matrix from current schedule s and DL
4   While |D| > 0
5       $s_{best} = s_{curr}$
6       $u_{prev} = u(s_{curr})$, $u_{best} = -\infty$
7       $task_{best} = \emptyset$
8       For task i in D
9           $s_{tmp}, u_{tmp} = taskInsertSearch(s_{curr}, i, \Omega, \lambda)$
10          If $u_{tmp} > u_{best}$
11              $s_{best} = s_{tmp}$, $u_{best} = u_{tmp}$, $task_{best} = i$
12      If $u_{best} < 0$
13          Exit insertion, all possible tasks to insert only lower utility of schedule
14      Else If $u_{best} > u_{prev}$
15          $s_{curr} = s_{best}$
16          update Cross Synchronization Matrix, Ω, for $task_{best}$ added to schedule
17      Else If $e^{\frac{u_{best}-u_{prev}}{u_{prev}\times temp}} \geq rand()$
18          $s_{curr} = s_{best}$
19          update Cross Synchronization Matrix, Ω, for $task_{best}$ added to schedule
20      remove $task_{best}$ from D
```

The new schedule is accepted as an update to the current schedule when it has a greater utility than the previous schedule, as seen by Lines 14-16. However, when it is not an improvement, simulated annealing is used to encourage state space search. Line 17 shows the criteria for deciding if the lower scoring schedule should be accepted based on the magnitude of its utility difference from the previous schedule.

For taskInsertSearch() in Line 9, a straightforward "simple algorithm" may try inserting task i's resource requests in all indexes of the path of the current schedule, to find which insertion spot causes the least increase in the cost of the overall schedule. However, because the tasks are constrained by time windows, many infeasible insertion spots would be evaluated despite being far out of the tasks time window. Miller's improvement phase makes uses of a λ parameter to represent a percentage of the route that the algorithm will search for insertion locations [12]. As can be seen in Figure 19 below, by finding the index of the task on the path that is the first task after task i's early time. This reference index +/- λ% × |path length| create the range of indexes of the path where insertion is evaluated, saving a large amount of unnecessary calculation.

**Figure 19: Example of Lambda Insertion**



*In the example of* Figure 19, *Task i is trying to be inserted into a path of one resource that has four tasks. The beginning of Task i's time window is known to be time early*$_i$*. Based off of the arrive times of the* 2nd *&* 3rd *tasks on this resources path, it is known early*$_i$ *occurs between these two tasks, so the* 3rd *task is the reference index for lambda insertion:*

$$index_{reference} \pm \lambda\ |path\ length|$$

*For this example index$_{reference}$ = 3, and as the resource only has four tasks on its path, path length = 4. The resulting range of insertion indices for λ = 10% is* $[3 + 0.10 \times 4, 3 - 0.10 \times 4]$ *or* $[3,3]$*, so index 3 is the only index Lambda Insertion will attempt to schedule task i.*

*The resulting range of insertion indices for λ = 25% is* $[3 + 0.25 \times 4, 3 - 0.25 \times 4]$ *or* $[2,4]$*, so there are three indexes for Lambda Insertion to attempt to schedule task i.*

4.3.4.2 Regret Insertion

This insertion method selects a task for insertion that will be most regretted if it is not inserted immediately into its best insertion location in the schedule. The *regret-k* heuristic chooses the task *i* that maximizes:

$$i = \max_{i \in DL} \sum_{j=1}^{k} (f_i^j - f_i^1)$$

$f_i^j$ denotes the cost of inserting task *i* in the *j*th cheapest insertion position. This method sorts the DL according to their regret value, an adaption of the *regret-k* method [14]. In this thesis, the regret value is *regret-2*, which is the difference in cost of insertion for a task's best insertion position (as measured in Section 4.3.4.1) and its second best position. While Ropke and Pisinger only considered one best insertion position in each path, this thesis measures multiple potential insertion positions in each path as selected by λ-insertion. After each synchronized service task is inserted, the cross-synchronization matrix is updated as in best insertion.

### 4.3.5 Removal and Insertion Weights

Each method *i* is associated with a score $\pi_i$ and a weight $w_i$. Removal and Insertion methods performances are tracked, so that the more effective methods can be intelligently implemented. Each method deals with different challenges in scheduling, so no one pair of removal and insertion weights should be chosen and the rest removed from possibility. The probability of the *i*th removal method being chosen is $\frac{w_i}{\sum_{i \in \Gamma_1} w_i}$ where $\Gamma_1$ is the set of removal methods. The insertion method is chosen in the same manner with $\Gamma_2$.

Scores record the methods performance in an iteration. If a new best overall solution is found, the score of that removal and insertion pair increases. However, even if the current iteration's solution is not a new best solution, but an improvement from the previous iteration's value, the methods' scores are increased. This is necessary as ICAARUS allows improvement iterations to alter schedules to lesser value, but the system still rewards heading in an improving direction. A minor score boost is also given if solutions are novel and do not match any of the previous solutions found, to encourage solutions from a wide swath of the search space. The exact calculation of score improvement is shown below:

$$\pi_i = \begin{cases} \pi_i + \varphi_3, & If\ new\ best\ solution\ is\ found \\ \pi_i + \varphi_4, & ElseIf\ improved\ solution\ is\ found \\ \pi_i + \varphi_5, & ElseIf\ new\ solution\ is\ found \\ \pi_i, & Else \end{cases}$$

Improvement phase iterations are sectioned into segments, with 50 improvement iterations per segment. Initially the weight of each method is set the same at 1.0. The weights remain unchanged throughout the segment, while the scores are recorded after each iteration. At the end of a segment, the weights are updated with, $\phi_i$ , the number of times the method has been called during the last segment, and κ controlling the inertia of the weight adjustment:

$$w_i = (1 - \kappa)w_i + \frac{\kappa\,\pi_i}{\phi_i}$$

Keeping track of the number of times a method is used to find the average solution improvement made by that method. Otherwise a methods score can be inflated by being called numerous times, but only producing marginal solution improvements. To maintain methods performance across segments, weight adjustment is controlled by inertia value κ. This inertia value prevents a weight from dropping rapidly if it performs poorly in the current segment, but is historically helpful.

# Chapter 5

## Tests and Analysis

This chapter tests and analyzes two methods to solve the SSP: the Mixed Integer Linear Program (MILP) presented in Chapter 3 and the *Infeasibility Cooling Adaptive Allocation for Resource United Scheduling* (ICAARUS) presented in Chapter 4. While several MILP models were presented in Chapter 3, this chapter will only be testing the Supplier MILP with Synchronization model. The MILP method is an extension of previous work that guarantees solution optimality, while ICAARUS is a novel method to find near optimal solutions in shorter runtimes.

This chapter begins with outlining how the test scenarios are created for evaluating performance. This section describes how each test set generates its unique map and task parameters. This section also includes summarizing the parameter settings for ICAARUS used in testing.

This chapter then focuses on quantitative performance of ICAARUS. As the objective of this research is to find an efficient method to use for coordinating suppliers with consumers, the runtime and optimality is measured and compared. The ability of ICAARUS to handle large-scale cases of the Supplier Scheduling Problem is tested to determine if it is able to solve problems intractable to the MILP method. Also measured is the gap between ICAARUS's solutions and theoretical "best" solutions. Finally, the fulfillment rates of different synchronization tasks are analyzed throughout the bidding rounds.

### 5.1 Test Datasets and Parameters

Each test set requires a large amount of information to describe the input into the simulator. This section outlines how the data that specifies task input is generated for the test sets. While the positions of consumers on the planning map are irrelevant, the location of supplier's bases and tasks has a crucial impact on which tasks are serviceable and those that are unreachable.

The exact parameters for ICAARUS, as covered in Chapter 4, are also explained. Altering these parameters can affect runtime and optimality of solutions found by ICAARUS, but the sensitivity of ICAARUS solutions to changes in these parameters is left for future research.

### 5.1.1 Test Set Development

To analyze MILP and ICAARUS, twenty pseudo-random test sets were created. The two methods for allocating resources to tasks are tested through five rounds of four test sets in each round. For tasks, the set of possible resource requirements is: {1,2,3,4}, where {1} is a single service task, and {4} is a task requiring synchronization across four resources. In each test set, the number of tasks requiring each level of synchronization are equal. In other words, if a test set has five single service tasks, there are also five dual collect tasks, and so on. The five testing rounds are distinguished by their consumer task counts. The set of task counts being: 4, 8, 12, 16, 20.

Most data for test sets is created by a random number generator within specified bounds. The planning map is confined to a two-dimensional size of 10 Kilometers x 10 Kilometers. Thus, the latitude and longitude of task positions are randomly generated pairs of numbers within the range [0,10]. The values of tasks are randomly selected from a uniform distribution of [1,100]. The planning horizon to complete all tasks is 100 minutes. All tasks have a required minimum duration of 5 minutes to complete servicing. The arrival times of tasks are selected from a uniform distribution of [0,75] minutes with all late times being 25 minutes after the selected arrival time.

For each resource a task requests, its type is randomly selected from the set {A,B,C,D}. Therefore, a task requesting four resources could be given any combination of this set, such as {A,A,A,A} or {D,C,B,A}. All resources perform the same, i.e. travel distances at the same speed and have the same endurance. The differing characteristic is only a resource of type X can fulfill a resource request for type X. Each resource type is exclusive and non-mutable.

Supplier base positions are randomly assigned longitudes and latitudes just as the task positions are assigned. All resources controlled by a supplier begin the planning horizon at the supplier base's position, and must end the planning horizon at that same supplier base. Suppliers are randomly assigned four resources from the resource type set, {A,B,C,D}. Thus supplier's resources can range from {A,A,A,A} to {D,D,D,D} and all permutations in-between.

### 5.1.2 Parameter Selection

ICAARUS's performance depends on the below set of correlated parameters.

| Symbol | Description | Value |
|---|---|---|
| $\alpha, \alpha_{Min}, \alpha_{Max}$ | Initial, minimum, and maximum values of $\alpha$ | 10,0.01,200 |
| $\beta, \beta_{Min}, \beta_{Max}$ | Initial, minimum, and maximum values of $\beta$ | 10,0.01,200 |
| $\varphi_1$, $\varphi_2$ | Weight adjustment parameters for $\alpha$ & $\beta$ | 0.5,0.25 |
| $\varphi_3$, $\varphi_4$, $\varphi_5$ | Weight adjustment parameters for Removal and Insertion | 35,10,15 |
| $y$ | Percentage of task list to be removed in removal phase | [0.15,0.30] |
| $\psi$,$\omega$ | Related removal parameters | 1.0,1.0 |
| $p$ | Randomness parameter in related and worst removal | 5 |
| $\lambda$ | Lambda-insertion percentage | 0.10 |
| $\kappa$ | Weight inertia parameter | 0.4 |

The sensitivity of ICAARUS's performance to alterations in these parameters is left to future research.

## 5.2 Evaluation Tests

This section focuses on the performance of a Linear Programming method versus ICAARUS's heuristic method. The consumer-supplier network explored in these tests is two suppliers to three consumers. Each consumer and each supplier has tasks and resources generated from the test dataset, explained in Section 5.1.1. To determine which method is better suited to schedule resources in an operational context, the methods are compared by analyzing two characteristics: 1) the runtime of the method and 2) the value of the best schedule solution.

The software discussed in Section 3.5 is used in the analysis of ICAARUS. Gurobi optimization software is used through JuMP for the Supplier MILP with Synchronization Model (Section 3.4.3) and the Consumer MILP Model (Section 3.4.4). ICAARUS is implemented in Julia and was designed to have a much shorter runtime, while remaining close to MILP optimality. ICAARUS uses a heuristic method that can find a "best" optimal solution quicker than an algorithm that evaluates the entire set of solutions. However, this also results in the "best" solution not being guaranteed to be a global optimum solution. Much of ICAARUS's runtime is spent in the improvement phase taking suboptimal steps to search the solution space and avoid becoming trapped in a local optimal solution.

The MILP analysis utilizes Gurobi 9.0.2 to solve the linear programs. In the worst case the MILP searches the entire solution space, which can scale exponentially in runtime for Mixed-Integer Programs that rely on Branch and Bound methods. However, the Consumer MILP has LP-tight bounds on its solution space, removing the need for lengthy Branch and Bound steps. The Consumer MILP is used by consumers to select element bids, regardless of whether the suppliers generated bids by MILP or ICAARUS. The Consumer MILP requires less than a second to find solutions in these test trials. This offers little room for improving runtime compared to the greater runtime of the Suppliers. A heuristic based consumer solver is left for future research as it is not the focus of this thesis.

### 5.2.1 MILP Comparison

When testing the MILP, the *Linear Programming (LP) relaxation* is also tested to give a theoretical upper limit. The LP relaxation removes the binary constraints on the decision variables. This means the variables $accomodate_{i,u,p}$ and $travel_{i,j,u}$ can take any value on the interval [0,1], rather than only {0,1}. This relaxation does not guarantee a feasible schedule. The relaxation can produce solutions where resources are not servicing tasks for their full minimum duration or resources are unsynchronized when servicing a task. However, it does provide an upper bound for the MILP's performance that can be useful for cases in which the MILP cannot find the optimal solution in a feasible runtime.

To accurately find the theoretical "best" upper limit of resource allocation to tasks, the supplier MILP is assumed to be an omniscient centralized planner. While ICAARUS has three suppliers that allocate four resources each in a decentralized manner to fulfill tasks, the supplier MILP will control all twelve resources together. Thus, there will only be one round of bidding for the MILP as the omniscient supplier has found its optimal resource allocation. This coordinated supplier will give a consumer's task only one set of resources requested, so no bids will be rejected that might alter the schedule and create a need for follow on resource bidding rounds. These results will show the performance gap of ICAARUS for coordinating across decentralized suppliers.

### 5.2.2 Runtime Comparison

The aim of this runtime comparison is to determine whether the MILP or ICAARUS method is able to generate a schedule that solves the Supplier Scheduling Problem in the smaller amount of time. To evaluate this aspect, the test rounds discussed in Section 5.1.1 were solved using each method: ICAARUS, MILP, and the LP-relaxation. Each test round has four test sets and the results across all twenty sets are presented in Table 4.

It is expected for ICAARUS to be the faster method compared to MILP, as it does not search the entire solution space in worst case scenario as with MILP. While ICAARUS's solution is not guaranteed to be optimal, its methods do push its scope to be wide ranging across the solution space. ICAARUS eventually cools to finding a local optimum in a neighborhood of the solution space, while MILP may search the entire solution space till a best schedule is selected.

**Table 4: Runtime Comparison**

| Number of Tasks per Consumer | ICAARUS Runtime (s) | MILP Runtime (s) | LP-relaxation$^+$ Runtime (s) |
|---|---|---|---|
| 4 | 2.88 | 6.46 | 0.10 |
| 4 | 10.83 | 10.70 | 0.15 |
| 4 | 2.54 | 8.94 | 0.13 |
| 4 | 5.45 | 10.35 | 0.10 |
| 8 | 24.88 | 3174.74 | 1.14 |
| 8 | 11.52 | 2520.81 | 1.20 |
| 8 | 14.23 | 1539.23 | 1.19 |
| 8 | 27.59 | 3276.85 | 1.32 |
| 12 | 791.03 | 5479.36 | 29.02 |
| 12 | 1412.87 | 4077.86 | 17.11 |
| 12 | 1671.79 | 4192.52 | 21.65 |
| 12 | 1713.51 | * | 28.31 |
| 16 | 823.96 | 5785.59 | 969.25 |
| 16 | 1053.25 | * | 1400.61 |
| 16 | 1705.54 | * | 988.71 |
| 16 | 837.74 | * | 1390.19 |
| 20 | 1782.53 | * | * |
| 20 | 1587.44 | * | 3279.02 |
| 20 | 1221.87 | * | 3321.86 |
| 20 | 1366.05 | * | * |
| * Runtime exceeds limit of 7200 s<br>$^+$ Solutions not guaranteed to be feasible | | | |

As expected, in Table 4, for all but one test set ICAARUS solves the test set in less time than the MILP. This outlier is due to the Gurobi solver pre-solving much of the MILP model to eliminate many redundant constraint columns that normally drive up runtime. Meanwhile ICAARUS unnecessarily explored many schedules that were not improvements, in order to ensure it was not becoming stuck on a local optimum solution.

In general, as the size of test sets increases, the MILP method becomes unable to find the optimal solution in under two hours. Meanwhile ICAARUS was able to find solutions consistently within a reasonable time frame. Despite the MILP method not needing multiple bidding rounds like ICAARUS, the MILP method becomes intractable after eight tasks per consumer. Average runtimes for each round of testing is displayed in Table 5. In many of the sixteen and twenty task size sets, the MILP was unable to find a solution, so the average MILP runtimes are artificially low. In reality, these testing rounds have longer average runtimes.

**Table 5: Average Runtime**

| Number of Tasks per Consumer | ICAARUS Avg. Runtime (s) | MILP Avg. Runtime (s) | LP-relaxation $^{+}$ Avg. Runtime (s) |
|---|---|---|---|
| 4 | 5.43 | 9.11 | 0.12 |
| 8 | 19.56 | 2627.91 | 1.21 |
| 12 | 1397.30 | 4583.25 | 24.02 |
| 16 | 1105.12 | 5785.59 | 1187.19 |
| 20 | 1489.47 | * | 3300.44 |
| * Runtime exceeds limit of 7200 s $^{+}$ Solutions not guaranteed to be feasible | | | |

When runtimes are averaged across tests sets of each round, a clear runtime superiority of ICAARUS is seen over MILP. One outlier from expectations is the average runtime of sixteen tasks is less than the average runtime of twelve tasks. This might appear to suggest as the number of tasks increases the schedule is easier to find. However, this is most likely due to the specific test sets generated having different complexities in other factors, such as task position and time window parameters. Some twelve task test sets in this analysis are randomly more complicated than the four sets generated for sixteen tasks, despite the sixteen tasks having a greater task count.

Of note is the runtime for ICAARUS in the last three testing rounds becoming capped at 1800 seconds, while the MILP runtime becomes infeasible. For an operational commander, having a solution in a reasonable time frame of 30 minutes can be a very attractive feature. However, ICAARUS is not inhibited from taking longer than 30 minutes to find a solution. It very well may take more than half an hour if the number of tasks were increased.

## 5.2.3 Optimality Comparison

The aim of this optimality comparison is to determine how far the ICAARUS method's generated schedule is from the optimal solution. The MILP method is guaranteed to find the optimal solution, even if it requires searching the entire solution space. However, when the MILP method requires an infeasible runtime, the LP relaxation can provide insight into the upper bound of solution value.

To evaluate optimality, the same test sets evaluated in Section 5.2.2 were solved using each method: ICAARUS, MILP, and the LP-relaxation. The value of a solution is the addition of all the values of tasks fulfilled minus the travel cost of the resources. The optimality performance of the different methods is compared in Table 6.

Table 6 contains two new columns of information, optimality gap and integrality gap. Optimality gap shows the difference between the optimal solution, as found by the MILP method, and ICAARUS's solution. However, as noted in the previous section, the MILP method is not always able to find the optimal solution in a feasible time frame. Thus, the LP-relaxation is used as an upper bound on performance. While this upper bound may not be feasible, its solution value still lends insight into ICAARUS's performance. Integrality gap shows the difference between the LP-relaxation and the best-known solution, this is the MILP solution when one is found and the ICAARUS solution otherwise.

**Table 6: Optimality Comparison**

| Number of Tasks per Consumer | ICAARUS Solution Value | MILP Solution Value | Optimality Gap | LP-relaxation$^+$ Solution Value | Integrality Gap |
|---|---|---|---|---|---|
| 4 | 214 | 214 | 0 % | 249 | 14.06 % |
| 4 | 172 | 172 | 0 % | 208 | 17.31 % |
| 4 | 290 | 290 | 0 % | 290 | 0.00 % |
| 4 | 249 | 249 | 0 % | 249 | 0.00 % |
| 8 | 352 | 352 | 0 % | 464 | 24.14 % |
| 8 | 455 | 455 | 0 % | 480 | 5.21 % |
| 8 | 348 | 348 | 0 % | 362 | 3.87 % |
| 8 | 315 | 315 | 0 % | 432 | 27.08 % |
| 12 | 536 | 560 | 4 % | 655 | 14.50 % |
| 12 | 627 | 627 | 0 % | 815 | 23.07 % |
| 12 | 620 | 620 | 0 % | 700 | 11.43 % |
| 12 | 624 | * | ** | 954 | 34.59 % |
| 16 | 674 | 674 | 0 % | 930 | 27.53 % |
| 16 | 766 | * | ** | 1065 | 28.08 % |
| 16 | 757 | * | ** | 1302 | 41.86 % |
| 16 | 948 | * | ** | 1504 | 36.97 % |
| 20 | 905 | * | ** | * | ** |
| 20 | 788 | * | ** | 1567 | 49.7 % |
| 20 | 996 | * | ** | 1923 | 48.2 % |
| 20 | 732 | * | ** | * | ** |
| * Runtime exceeds limit of 7200 s<br>** Gap incalculable<br>$^+$ Solutions not guaranteed to be feasible | | | | | |

The data in Table 6 shows that ICAARUS provides solutions with values that are close to the optimal solution. In the twelve cases that the MILP method could find an optimal solution in under two hours, in only one case was the ICAARUS method also unable to find the optimal solution. Furthermore, in that one case the optimality gap, the difference between optimal solution and ICAARUS solution; was only 4%.

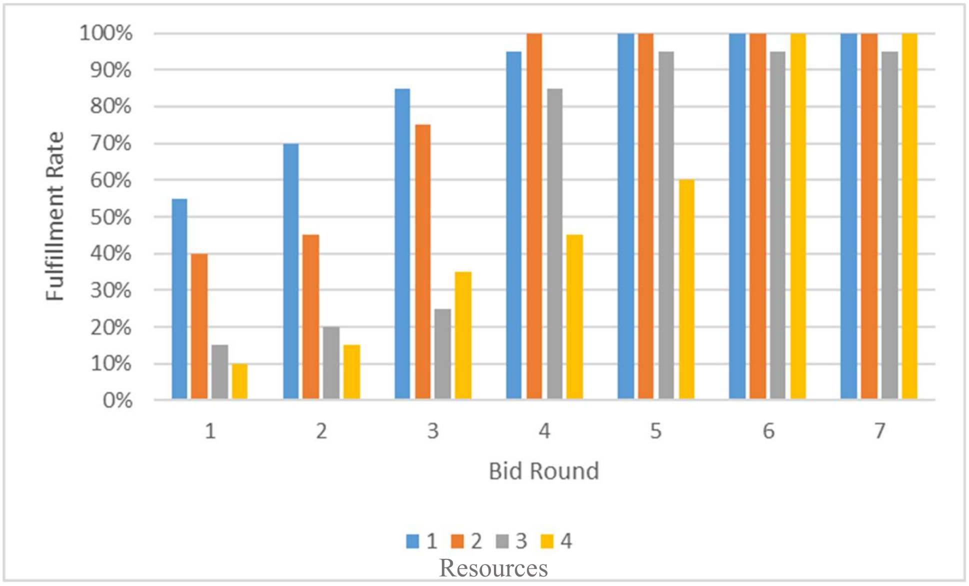
Despite the small optimality gap seen for ICAARUS, its performance for higher task count rounds may not be as promising as the lower task counts rounds suggest. To evaluate this difference the LP-relaxation is analyzed, understanding this may be an infeasible solution. The average integrality gap of the two smallest rounds is 11.46%, while for the two largest rounds it is 38.72%. This would suggest for later testing rounds, should the MILP have found the feasible optimum solution, the ICAARUS solution may have a larger optimality gap than in the earlier rounds.

## 5.3 Synchronization Coordination

An important characteristic for consumers entering a decentralized market is how long they will have to wait for task fulfillment. As this thesis aims to improve scheduling of tasks that require a variable number of resources, fulfillment rates differ on the level of coordination required. Figure 20 measures average task fulfillment rates for test sets requiring 20 tasks as they are the most complicated test sets that push suppliers to make the most tradeoffs.

Fulfillment is measured after each round of bidding as that is when the Consumer will know if they have collected the necessary requested resources or need to continue bidding. It is expected that tasks requiring less resource coordination, tasks with {1} resource requested, will be the quickest to be fulfilled. This is expected because they require no synchronization across suppliers to be fulfilled. In contrast, it is expected the tasks that require the most resource coordination, tasks with {4} resources requested, will be the slowest to be fulfilled. Figure 20 only shows the results of the first seven rounds of bidding despite there being ten rounds, this is for convenience sake as the last three rounds do not change any supplier schedules.

**Figure 20: Average Fulfillment Rate vs. Bid Round for 20 Tasks per Consumer**



As expected, the single resource tasks have the highest fulfillment rate in the earliest three rounds. Over half of the tasks are fulfilled after the first round of bidding, and all are completed by the fifth round of bidding. In the fourth round of bidding, the second simplest tasks to synchronize are the first to achieve 100% fulfilment rate. This most likely comes from the high likelihood of suppliers, all with four resources, having the resources to completely fulfill these dual synchronization tasks without coordinating with another supplier.

The most complicated tasks, those requiring four resources, undergo a large jump in fulfilment rate of 60% to 100% in the sixth round. This is likely from 97.5% of single and dual synchronization tasks being settled by round four, and two additional rounds are needed for a consumer to update the task time windows, as outlined in Section 2.2.3, to synchronize across suppliers. It also appears that the fulfillment rate of tasks requiring three resource synchronization is stuck at 95% fulfillment rate. This is most likely from the specific test sets generated having an infeasible task accommodation.

# Chapter 6

## Conclusions

This chapter summarizes the work presented throughout the previous five chapters. This summary begins with presenting the contributions made by this thesis, specifically the methods and implementation for consumer-supplier task fulfillment. The next section proposes future work to improve the methods presented in this thesis. These improvements were only briefly explored, but show great promise for follow-on research. The end of this chapter has concluding thoughts on the methods and results.

### 6.1 Summary of Contributions

This section reviews the contributions made by this thesis, as stated in Chapter 1. The purpose of this research was to explore decentralized resource allocation between mission commanders and resource suppliers. In particular, this thesis focuses on the challenge of task synchronization in SSP. The contributions of this thesis are summarized in the following paragraphs.

- **An e-commerce bidding structure to coordinate multiple consumers with multiple suppliers asynchronously and decentralized.** Chapter 2 outlined a system for coordinating multiple bidding rounds where task creators are consumers and resource providers are suppliers. This three way hand shake method was then tested in conjunction with ICAARUS and the performance results were presented in Chapter 5.
- **A MILP model to solve the SSP.** Chapter 3 presented several candidate MILP formulations that can provide exact solution to the Supplier Scheduling Problem, with Section 3.4.3 outlining a model with synchronization constraints. This MILP was implemented using JuMP v0.18 and Gurobi 9.0.2 solver.
- **A MILP model to select cheapest resource bids.** Chapter 3 presented a MILP formulation that can provide an exact solution to the Consumer Element Bid Selection Problem. This MILP was implemented using JuMP v0.18 and Gurobi 9.0.2 solver.
- **The development and implementation of ICAARUS, an algorithm to schedule multiple resources for tasks requiring time and spatial synchronization.** Chapter 4

presented a novel Adaptive Large Neighborhood Search algorithm for solving the Supplier Scheduling Problem. ICAARUS is a composite algorithm with construction and improvement phases that maintain resource synchronization for tasks throughout the optimal schedule search. ICAARUS is implemented in Julia v1.0.5. It was shown that this implementation can develop a schedule for 3 decentralized suppliers and 40 tasks (with synchronization requirements ranging 1-4 resources) in less than 30 minutes.

- **Testing and analysis of the MILP and ICAARUS.** Chapter 5 examined the advantages and shortcomings of both methods under varying test scenarios. The MILP method was able to find optimized schedules for suppliers, but at increasingly lengthy runtimes. Test sets with more than 32 tasks total were intractable for the MILP method. Meanwhile the heuristic method, ICAARUS, generated schedules in a significantly shorter runtime, with almost all single and dual synchronization tasks scheduled in the first four rounds of bidding. Furthermore, ICAARUS was able to generate solutions for every test, while the MILP was not. These faster solutions are valuable for the consumers and suppliers, but were not always the most valuable solution possible.

## 6.2 Future Work

While this thesis implements two methods for solving the SSP as outlined in Chapter 2, numerous modifications could have been made to improve these methods. These areas of improvement include adapting ICAARUS's improvement phase for faster runtimes and making the solutions robust to changes in the task parameters.

### 6.2.1 Adaptive Drop List

A majority of the runtime of ICAARUS is spent by the improvement phase's insertion methods, discussed in Section 4.3.4. The insertion methods operate by testing the insertion of each task on the Drop List, DL, individually. If the DL were shorter, the runtime of ICAARUS could be improved. By attempting to insert fewer tasks, the cross-synchronization matrix – a time-expensive operation -- fewer times.

To review, the DL is the set of tasks removed from the initial schedule made in the construction phase. These are the tasks removed in the most recent iteration's removal method, as well as tasks removed by earlier removal methods that did not get re-inserted. This means the insertion methods will continuously attempt to re-insert tasks from the DL, despite a task on the DL being repeatedly found infeasible by the insertion methods.

Currently ICAARUS is purposely designed to be overly optimistic; for example, the construction phase accepts tasks that are known to be infeasible with the current schedule. This is done to give the improvement phase the ability to create a schedule with maximum value, by repeatedly removing and re-inserting tasks till an optimal schedule is found. ICAARUS is also designed to avoid becoming stuck in a local optimum by not always inserting the most valuable task in the insertion phase. Sometimes the $n^{th}$ best task is inserted, to allow ICAARUS to evaluate a variety of schedules. So, even when a task is removed and not re-inserted in the improvement iteration, that does not mean it should be permanently dropped from future consideration. ICAARUS presently stores a long DL to have the best chance as feasible at maximizing its schedule. However, while this lengthy task list helps improve solution optimality, it severely hampers runtime performance.

A dropped task learning method could be implemented to intelligently shorten the DL, while maintaining limited infeasibility allowances. This could be done in a manner similar to adapting method selection, as discussed in Section 4.3.5. Features this learning method could find as important characteristics to score are:

- **Task Value**. The learning method should penalize low value tasks, as they do not offer the total schedule as much utility as tasks with high values.
- **Insertion Rejection Rate**. Tasks that are removed and continue through the improvement rounds staying on the DL, are tasks that are unlikely to be in a feasible solution. Rather than repeatedly test their insertion to find they would only again result in an infeasible schedule, an intelligent DL should drop them permanently from insertion consideration.

This pruning of poor performing task from the DL at the end of every segment offers great runtime improvements at potentially little impact to optimality performance. The exact weights of task features and cut-off lines for when to drop the task would require future analysis to tune for optimal performance.

### 6.2.2 Robust Optimization for Duration Changes

The setup of scenarios in this thesis tested resource request fulfillment for tasks that never change their required minimum durations. However, this is an unrealistic assumption for scheduling. Real-world missions often experience alterations in service time from original requests as situations change. The methods presented in this thesis, however, are oriented at maximizing value, and thus create exquisite, yet fragile, schedules. A task that overestimated service duration is typically harmless. That error would just result in more idle time as resources depart the task early. However, a task that has its service time extended can force resources to either abandon the task, and leave it as incomplete, or stay and cancel later tasks on their paths.

The use of Robust Optimization (RO) could be applied to the MILP model. This would create plans that remain feasible even after a change in the duration of tasks. A naive method to create robust schedules would substitute all minimum durations with the maximum duration changes expected by the mission commander. However, this conventional solution would significantly lower the value of solutions, compared to the nominal solver. Schedules will conservatively allocate tasks substantially more service time than necessary, and thus limit the total number of tasks that a resource can service in its path. RO intelligently deals with uncertainty, under the assumption not every task will experience the maximum duration changes.

The rest of this section presents a possible implementation of RO for dealing with uncertainty in task minimum duration, although it is certainly not the only implementation possible. Adapting the MILP model presented in Chapter 3 to robust optimization could be done with the introduction of uncertainty variables *robustMinDur*$_i$ and $\delta_i$ with the following constraints:

(22) Substitute for constraint (9). Resource can depart the task only after the required duration of service time.

$$arrive_{i,u} - robustMinDur_i \times \sum_{p \in P} accomodate_{i,u,p} \leq depart_{i,u} \quad \forall i \in T, u \in U$$

(23) Lower bound limit on *robustMinDur*$_i$'s uncertainty

$$minDur_i \leq robustMinDur_i \quad \forall i \in T$$

(24) Upper bound limit on *robustMinDur*$_i$'s uncertainty

$$robustMinDur_i \leq minDur_i + \delta_i \times durDelta \qquad \forall i \in T$$

(25) Upper bound limit on percentage of change added to minimum duration

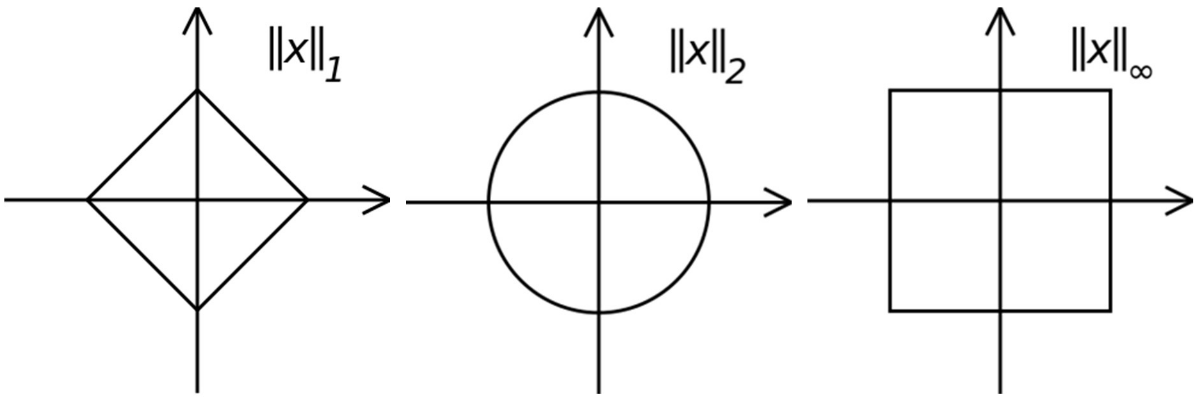
$$\delta_i \leq 1 \qquad \forall i \in T$$

(26) Upper bound limit on total duration change expected by mission commanders

$$\left|\left|\delta_i\right|\right|_p \leq \Gamma \qquad \forall i \in T$$

The mission commanders express their expected maximum duration change with the constant *durDelta*. The uncertainty variable $\delta_i$ controls the percentage of duration change for $robustMinDur_i$, to be in the range of [$minDur_i$, $minDur_i + durDelta$]. The constant Γ controls how much uncertainty the mission commander expects across all the tasks. As mentioned in the previous paragraph, robust optimization intelligently manages uncertainty rather than scheduling all tasks to experience worst-case scenario. Γ is the limit on $\delta_i$ within the *p*-norm. Visual representations of the Manhattan norm ($p$=1), Euclidean norm ($p$=2), and Infinity norm ($p$=∞) are given in the figure below:

**Figure 21: *p*-norms**



What is the correct *durDelta* if left for consumers to specify and what is the optimal Γ & *p* is left for future research based on specific scenarios.

The use of RO in this way also has the desirable outcome of increasing the likelihood of element bids overlapping their service time windows. Since tasks are being serviced in a time window that is greater than the minimum duration, consumers face the challenge of coordinating synchronization. The Maximum Time Window approach, presented in Section 3.4.2, aims to extend element bid service times so suppliers increase the likelihood of their offered service time overlapping with other suppliers offered service times. This could remove synchronization constraints that increase runtime of the supplier-based MILP method, but may increase the number of bidding rounds to align task time windows.

### 6.2.3 Robust Optimization for Value Changes

RO can also be used to intelligently manage value changes in tasks. As mentioned in the previous section, current methods create exquisite yet fragile paths. A supplier may only service a low valued task because it is spatially and temporally close to another high value task. However, should the high value task have its value decreased, the exquisite schedule can suffer great drops in optimal schedule value.

The rest of this section presents a possible implementation of RO for dealing with uncertainty in task value, although it is certainly not the only implementation possible. Adapting the MILP model presented in Chapter 3 to RO could be done by introducing the uncertainty variables $robustValue_i$ and $\delta_i$. $robustValue_i$ replaces $value_i$ in the objective function and requires the following constraints:

(27) Lower bound limit on $robustValue_i$'s uncertainty

$$value_i - \delta_i \times \sigma_i \leq robustValue_i \qquad \forall i \in T$$

(28) Upper bound limit on $robustValue_i$'s uncertainty

$$robustValue_i \quad \leq value_i + \delta_i \times \sigma_i \qquad \forall i \in T$$

(29) Upper bound limit on percentage of value change

$$\delta_i \leq 1 \qquad \forall i \in T$$

(30) Upper bound limit on total value change expected by mission commanders

$$||\delta_i||_p \leq \Gamma \qquad \forall i \in T$$

The uncertainty variable $\delta_i$ controls the percentage of value change for $robustValue_i$ within its specified variance, $\sigma_i$. Γ is the limit on $\delta_i$ within the $p$-norm. The optimal Γ & $p$ is left for future research based on specific scenarios.

## 6.3 Conclusions

This thesis has addressed the challenge of coordinating resources in a decentralized and asynchronous environment. A three-part bidding phase was introduced to coordinate consumer needs across multiple suppliers in an e-commerce inspired bidding structure. An optimization model was presented that can be solved by a MILP to optimally schedule a supplier's resources. A novel ALNS algorithm, ICAARUS, was presented to solve the SSP in an operationally-feasible runtime. It uses a benefit-to-cost formula to select promising tasks and quickly generate a schedule in its construction phase. ICAARUS's improvement phase then uses simulated annealing to cool the infeasibility in the schedule to a solution that is within time windows and synchronized. Five candidate removal, and two insertion, methods are presented for removing and inserting synchronized tasks in a schedule to improve its utility.

It is concluded that ICAARUS is a viable algorithm for decentralized resource allocation. We have demonstrated the algorithm is capable of generating near-optimal schedules in a computationally tractable manner. While this work focused on a specific scheduling scenario, we believe that it presents a promising approach to schedule multiple-resource synchronization tasks, a problem that arises in many important applications.

# References

[1] Rogoway, Tyler. "USS Racine gets pummeled to death during RIMPAC 2018 sinking exercise." The Warzone. July 2018

[2] "A Design for Maintaining Maritime Superiority." Version 2.0. December 2018

[3] TRADOC Pamphlet 525-3-1. "The U.S. Army in Multi-Domain Operations 2028." 06 December 2018

[4] BAA. "Secure Advanced Framework for Simulation and Modeling (SAFE-SiM)." DARPA Adaptive Capabilities Office. HR001120S0007. 16 January 2020

[5] Spears W. "A Sailor's take on Multi-Domain Operations." War on the Rocks. 21 May 2019

[6] Jamieson V., Calabrese M.. "An ISR Perspective on Fusion Warfare." The Mitchell Forum. October 2015

[7] Nanehkaran, Y.A.."An Introduction to Electronic Commerce." International Journal of Scientific & Technology Research. Vol 2, Issue 4. April 2013.

[8] Leake Negron, B.. "Operational Planning for Multiple Heterogeneous Unmanned Aerial Vehicles in Three Dimensions". Master's Thesis, Massachusetts Institute of Technology. 2009.

[9] Herold, T.. "Asynchronous, Distributed Optimization for the Coordinated Planning of Air and Space Assets". Master's Thesis, Massachusetts Institute of Technology. 2008.

[10] Ropke, S. and Pisinger, D., 2006. "An Adaptive Large Neighborhood Search heuristic for the pickup and delivery problem with time windows". Transp. Sci. 40, 455–472.

[11] Liu R., Tao Y., Xie X., 2018. An adaptive large neighborhood search heuristic for the vehicle routing problem with time windows and synchronized visits. Computers & Operations Research Vol 101, 250-262.

[12] J.V. Miller. "Large-Scale Dynamic Observations Planning for Unmanned Surface Vessels". Master's Thesis, Massachusetts Institute of Technology. 2007.

[13] Hendel, G. "Adaptive Large Neighborhood Search for Mixed Integer Programming." ZIB Report 18-60. 18 December 2018.

[14] Ropke, S., Cordeau, J., and Laporte, G.. "Models and Branch-and-Cut Algorithms for Pickup and Delivery Problem with Time Windows," Networks 49(4), 258-272, 2007.

[15] Tzoreff, T., Granot, D., Granot, F., and Sosic, G.. “The vehicle routing problem with pickups and deliveries on some special graphs.” Discrete Applied Mathematics, Volume 116, Issue 3, 193-229.

[16] Dantzig, G. and Ramser, J.. “The Truck Dispatching Problem.” Management Science. 6. 80-91

[17] Bodin, L., Golden, B., Assad, A., and Ball, M., “Routing and schedule of vehicles and crews: The state of the art.” Computational Operations Research, 62-212., 1983.

[18] Dantzig, G., Fulkerson, R., and Johnson, S., "Solution of a Large-Scale Traveling Salesman Problem," Operations Research 2(4), 393-410, 1954.

[19] Bertsimas, D. and Tsitsiklis, J. N., Introduction to Linear Optimization, Athena Scientific, Belmont, MA, 1997

[20] Lawler, E., Lenstra, J., Rinnooy Kan, A., and Shmoys, D.. The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. John Wiley and Sons, Inc. New York. 1985.

[21] Laporte, G.. "The Traveling Salesman Problem: Overview of Algorithms." European Journal of Operational Research 59(2). 231-247. 1992.

[22] Flood, M., "The Traveling Salesman Problem," Operations Research 4(1), 61-75, 1956

[23] Rosenkrantz, D., Stearns, R., and Lewis, P.. "An Analysis of Several Heuristics for the Traveling Salesman Problem." Society of Industrial and Applied Mathematics Journal of Computing 6(3). 563-581. 1977.

[24] Croes, G.. "A Method for Solving Traveling-Salesman Problems." Operations Research 6(6). 791-812. 1958.

[25] Lin, S. and Kernighan, B.."An Effective Heuristic Algorithm for the Traveling-Salesman Problem." Operations Research. 21 (2). 498–516.

[26] Hove, J..“An Integer Program Decomposition Approach to Combat Planning.” Doctoral Dissertation. Air Force Institute of Technology. September 1998.

[27] Mingozzi, A., Bianco, L., and Ricciadelli, S.. “Dynamic Programming Strategies for the Traveling Salesman Problem with Time Window and Precedence Constraints.” Operations Research 45. 365-377. 1997.

[28] Baker, E. "An Exact Algorithm for the Time Constrained Traveling Salesman Problem." Operations Research 31(5). 938-945. 1983.

[29] Gendreau, M., Hertz, A., Laporte, G., and Stan, M.. "A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows." Operations Research 46(3). 330-335. 1998.

[30] Golden, B., Levy, L. and Vohra, R.. “The Orienteering Problem.” Naval Research Logistics, Vol. 34, Issue 3, Pages 307-318. June 1987.

[31] Tang, H. and Miller-Hooks, E.. “A TABU search heuristic for the team orienteering problem.” Computers & Operations Research 32, 1379-1407. 2005.

[32] Archetti, C., Hertz, A., and Speranza, M.. “Metaheuristics for the team orienteering problem.” Journal of Heuristics 13(1), 49-76. 2007.

[33] Tsiligrides, T. "Heuristic Methods Applied to Orienteering." Journal of the Operational Research Society 35, 797-809. 1984.

[34] Golden, B., Wang, Q., Liu, L.. "A Multifaceted Heuristic for the Orienteering Problem." Naval Research Logistics 35 (3), 359-366. 1988.

[35] Ramesh, R. and Brown, K.."An Efficient Four-Phase Heuristic for the Generalized Orienteering Problem." Computers and Operations Research 18, 151-165. 1991.

[36] “Illustration of unit circles in different norms.” Norm, Wikipedia. https://en.wikipedia.org/wiki/Norm_(mathematics).