Fabiola

The culinary experience you deserve

AI final project

Group 17

Miriam Sasson 346465230, Oz Matlaw 329603674, Guy Levy 208917609,

Daniel Alfia 318808540

Abstract

Eating outside is usually a nice experience, especially when we are in the company of friends or family, but picking a good restaurant or cafeteria can be hard considering all the factors that might influence the experience: quality of the food, costs, waiting times, etc. In particular, the clientele of cafeterias that are in specific locations, such as industrial parks (where most of us software engineers will find ourselves eating most of our meals) or universities, has a restricted amount of time for breaks and is very serious about waiting times.

In this project, the restaurant ordering system is approached as a combinatorial optimization problem. The solution proposed is an AI that can work online or offline. The offline solution receives all data (a list of orders with the time they were placed) in advance and finds the optimal scheduling. The online solution looks for the optimal solution at all given time intervals using heuristics, obtaining results that exceed 2-approximations of the optimal solution.

Introduction

Want to study, in an environment that meets your culinary standards?

That is where we come in. Ever made your way to the fine dining establishment of Fabiola just to find out that the line is as long as your Valgrind report? If you are as good at coding as we are then this can be a major issue! We are here to put an end to the awkward stomach growling heard all around the Huji campus. With our agent, you will be able to sneak in a quick coffee or nutritious meal, with minimum waste of precious study time, and be able to steer clear of all the pointless small talk with your classmates (like a true computer scientist). We will work hard to get you and your friend's nutrition tailored to your needs.

Problem Description

In our project, we chose to solve the Fabiola restaurant problem. The Fabiola problem is a problem of combinatorial optimization; given a set of orders, each with an expiration time and a value, determine the order of moves to fulfill the orders so that the total score is as large as possible.

Our problem is one of a broad range of scheduling and prioritizing jobs. This problem applies to many fields in our society, such as prioritizing customers based on specific needs, managing a service environment with many moving parts, scheduling shipments of online orders, and most important of all providing the proper nutrition to our students, the future leaders of our industry. While our agents are built for managing the Fabiola cafeteria, the implementation can easily be modified to solve various problems.

Let us notice that this problem is characterized by an exponential runtime: the number of actions required to fulfill the orders is exponential in the number of orders and in the available ingredients (a more thorough explanation can be found in the examination of our search space section)

We believe that a problem of this sort is an excellent example of the capabilities of Artificial Intelligence. Whereas an optimal solution is almost always unachievable in real-time, due to the enormous number of possible solutions.

Goal

Our goal is to optimize Fabiola's order scheduling and fulfillment. We will examine and compare the performances of agents that perform offline and online searches aiming toward an optimal solution, with and without applying a heuristic, and a trivial solution, to better understand the impact of Artificial Intelligence on the outcome of our problem.

As demonstrated in the studies of algorithmic behavior, a 2-approximation of the optimal solution can be achieved in polynomial time. We are aiming to provide an online agent using search and heuristics that will exceed the 2-approximation solution. We will investigate this by running the game for different numbers of orders to observe the relations between the agents.

Solution Background

In our solution, we have implemented online and offline search agents that will allow us to find the best scheduling and prioritizing methods for optimizing Fabiola's service hours. Both agents will use state space search algorithms to reach a solution. According to this family of algorithms, the solution consists of a sequence of legal moves of the simulator from the initial state to a goal state, that is a state in which all orders are served. In our case, we look for a solution that maximizes the agent's total score, by scheduling and fulfilling the provided orders in the desired time.

Our approaches to the problem are:

1. A* search – search agents that approach the problem using several heuristics to modify their search and provide solutions in an exponential search space
    a. Online search – receives incoming orders in real-time.
    b. Offline search – receives a full record of all orders in advance and performs a single search to provide an optimal solution.

2. Scheduling prioritization algorithms:
    a. Shortest Job First (SJF)
    b. First Come First Served (FCFS).

The online and offline search agents will be compared to two common non-preemptive scheduling algorithms, Shortest Job First (SJF) and First Come First Serve (FCFS), that fit the task at hand, seeing as a worker in the order fulfillment industry will most likely prioritize his task in one of these ways. This will give us a perspective of the quality of our solutions. Comparison with preemptive scheduling algorithms would produce poor results since it does not fit the problem description (such algorithms would not consider the current order once they receive a new one, ultimately disregarding most of the orders).

Trade-off: TIME vs. OPTIMALITY

After extensive research, we concluded that even with the help of an A* search and an optimal heuristic, the problem is still exponential given an input of over 20 orders, to the point where solving it offline is unrealistic. So, we came up with a solution that tries to tackle this problem with a trade-off of runtime vs optimality.

To deal with exponential search space, we integrated an element of search space partitioning into our solution. We partitioned the search space for the online agent by updating the goal state to no orders left to serve, instead of the end of service hours. Therefore, at any given time our online agent, due to the lack of knowledge of what orders lie ahead, will search the current space that is available to it which is bounded by the depth of the last expiration time of the current orders. Hence, we get an upper bound of 10^{15} for up to fifteen orders in a partition, and in reality, much less. From our experience, we get a run time per move of under 10 seconds.

Overall, with this approach, each move in the online agent is the outcome of an optimal solution of a time frame, instead of the entire space. Seeing as we are dealing with an online agent that does not get the entire list of orders at the beginning anyway, we believe that this approach provides an excellent solution, which doesn't deviate much from the optimal solution.

Methods and Solutions

Implementation: Simulator and Agents

Our implementation is based on a running simulator that receives orders from customers and tries to fulfill them in the provided expiration time to satisfy the customers' requests. Customers place orders at random times and orders are composed of random ingredients; thus, the simulator must choose what move to make at each time interval to maximize its score. To do so, we used different agents: we implemented online and offline search agents that solve the problem above while aiming for an optimal solution.

The online agents, online search agent, FCFS, and SJF, will receive orders during runtime and will try to calculate our next move in a reasonable time and obtain the best score according to the current state. The offline search agent will receive all orders in advance and will calculate the optimal solution. All agents look for the best move running through a graph that represents Fabiola's ordering system at that time, where nodes are states of the system that describe the current orders and waiting times, the order in process and the current score, and nodes' successors represent consecutive states reachable by one of the legal moves available for the agent.

To measure the performance of the simulator, each served order is assigned a personal score that reflects the amount of work needed to fulfill the order. The total score of the simulator is a cumulative sum of the served customers' scores.

Modeling the Problem as a Search Problem

States

Our states are defined by simulator objects that represent configurations of the simulator:

- Orders – set of currently available orders that the simulator can choose from
- Missed orders – set of orders that have expired and can no longer be served
- Time – current time of the simulator
- Active order – the order that the simulator is currently working on
- Total score – the accumulated score from all the orders the simulator served
- Stock – a dictionary representing the current ingredient stock. The number of servings of each ingredient that are available to the simulator.

Initial state:

Our initial state is defined as the start of Fabiola's service time 0, where no orders have been received, and we are fully stocked.

Goal state:

A goal state is a state where there are no pending orders left

Actions

The agent can perform one of these moves at any given time:

- If currently working on an order:
  - Take available ingredients
  - Restock missing ingredients
  - Drop order
  - Serve order (if ready)
  - Wait
- If not currently working on an order:
  - Start one of the currently available orders
  - Wait

Evaluation Function

The evaluation of each state is defined as the points lost until this state is reached. Furthermore, the heuristic used evaluates a state by estimating how many points we will lose from this state until the end of the run.

Optimality: a goal state is considered optimal if it has the smallest score deficit.

Let us examine the evaluation function:

$$\forall n \in V: \quad f(n) = g(n) + h(n)$$

Where:

- $G =< V, E >$: the search graph that represents the problem
- $g(n)$: the points lost until the state represented by node n is reached
- $h(n)$: the sum of scores of jobs that are impossible to complete represented by node n is reached
- $f(n)$: the estimated points we will lose from the state represented by node n

Examining the Search Space

The precise size of our search space cannot be calculated, since there is no specific number of successors to every given state because it is dependent on the number of available orders to choose from, the number of ingredients in the chosen order, and the additional capabilities. Therefore, we will try to give an upper bound to provide a perspective of the magnitude of our search space.

First, let us estimate the upper bound of the number of successors states for each state in our search tree.

- In the case where we are currently working on an order, given that each order consists of 1 to 5 ingredients, and the additional options of Restock, Wait and Drop we arrive at an upper bound of 8.
- In the case where we aren't currently working on an order, we get an upper bound that consists of the number of optional orders to start at the given time and to wait a turn, therefore is dependent on the number of optional orders which is unbounded in theory but in our project, we chose to bound it to MAX_ORDERS with a default value of 10.

Second, let's estimate the depth of the search tree. Given that the number of moves is bounded by Fabiola's working hours or the completion of the last order. In our project we set Fabiola to close at 17:00 (like in the real Fabiola, but in our implementation, the time is in time units so a default value of 50), therefore we can receive the last order at time 50 and the orders expiration time can be up to 15 time-units, giving us an upper bound of 65.

Overall, we get an upper bound for our search space of approximately 10^{65}. Just to put this in perspective the number of atoms in the universe is estimated to be between 10^{78} and 10^{82}. For a typical computer, it would take 3.16 x 10^{49} years to iterate over our search space.

Search Agents

The offline agent

The offline agent receives all orders in advance and at any given time it is aware of all upcoming orders. This agent uses an A* search algorithm with an admissible heuristic o find the optimal solution.

The online agent

The choices of the online agent reflect those of a waiter in the real world. At every given time, the agent is not aware of upcoming orders, but only of those that have been placed already. This agent

chooses the next move according to the A* search algorithm as well, ensuring all current options are evaluated and the optimal one is picked.

Heuristic

In our project, we tested many heuristics and evaluated their outcomes. Here are a few that gave the best results:

Admissible:

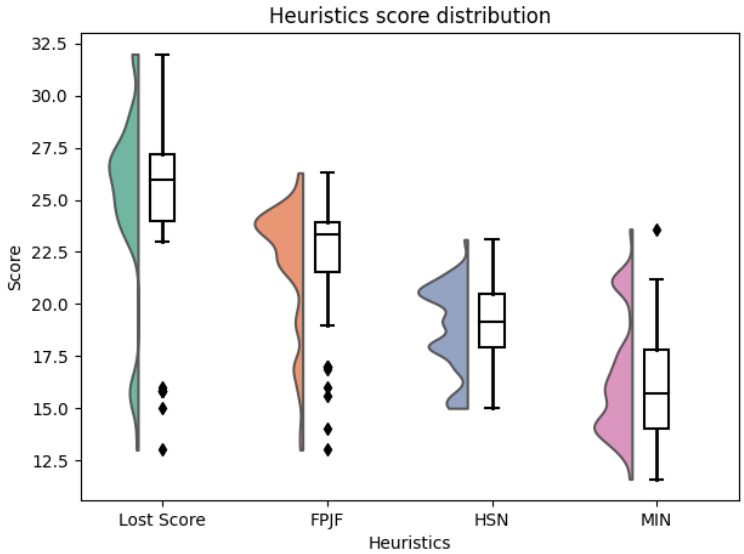
- Lost Score
- Fastest Possible Job First

Inadmissible:

- Highest Score Next
- Most Ingredients Next

After a hundred runs of each of these heuristics, we concluded that the Lost Score heuristic was the most capable and provided the best results on average.

In the graph below you can see a comparison of the score distribution of different heuristics.



Distribution of the heuristics score

The cloud plots are smoothed versions of histograms, that give an idea of the distribution of each score. The boxplots show the median, quartiles, and outliers of each distribution; in particular, they help understand that, on average, the Lost Score heuristic reaches the highest scores.

Lost Score heuristic

Our heuristic bases its approach on the fact that to maximize our score we need to minimize losses. It evaluates each state predicting what losses lie ahead. This approach allows us to avoid paths that lead to losing points. We use our advantage that at any given time we can calculate the preparation time needed for each order offline efficiently enough that it will not hinder the runtime of the program drastically. We do so by considering the current state and the time remaining to collect all the relevant ingredients, given that we devote our full attention to the task.

Formally, for each state S, we define the function $h(S)$, h will iterate over all the current orders and determine if it can be completed before it expires. It will then sum up all the scores of all the orders that cannot be completed in time and add the cumulative value of all the points that have been lost up until now for the given state.

We define orders that cannot be completed as orders whose time of expiration is less than the number of ingredients +2 (one to start the order if not currently active and one to serve).

To minimize the loss of points, we defined the priority of each state to be :

$$priority(S) = S.lost_{score} + h(s) =$$

$$= \sum_{\substack{order \in \\ S.\text{missed}}} order.score + \sum_{\substack{order \in \\ S.\text{cant\_be\_completed}}} order.score$$

Proof of admissibility

We will show that $h(S) \leq opt(S)$ where opt(S) is the optimal solution – where we lose the minimal number of points. Notice that in each state our heuristic calculates the score of all the orders that cannot be completed, without taking into consideration the consequences of future actions that may cause overlapping orders to get missed. To demonstrate this point here is an example:

Current state - no active order and all ingredients are fully stocked (4 of each ingredient)

Order 1 – coffee with milk and ice, expiration time – 4 seconds

Order 2 – coffee with oat milk and ice, expiration time – 4 seconds

Order 3 – salad with lettuce, tomato, avocado, and cucumber, expiration time – 3 seconds

According to our prerequisites, to complete an order when not currently working on it, we need to perform the task of selecting it (considered a move) and then gather all the necessary ingredients (one move per ingredient if in stock otherwise 2) and another to serve.

Therefore, we get for each order the preparation time of:

order 1 – take the order, add milk, add ice, serve = 4 moves

order 2 – take the order, add oat milk, add ice, serve = 4 moves

order 3 – take the order, add lettuce, add tomato, add avocado, add cucumber, and serve = 6 moves

since each move takes one-time unit, we get that we can finish orders 1 or 2 but not both, and no matter what can't finish order 3. Therefore, our heuristic will return the sum of all points that have been lost already plus the score of order 3 since it can't be fulfilled in time, but it won't take into consideration the fact that orders 1 and 2 can't both be fulfilled. Therefore, since the optimal solution does take into consideration this fact, we get that it will also add the loss of one of the other orders, ultimately providing a priority with a higher loss of points than our heuristic.

$$h(S) \leq opt(S)$$

Formally, define:

$$S.cant\_be\_completed\_in\_time =$$

$$= \{S.order_i \ \mid \ S.order_i \ \ expiration\ time \ \leq \ S.order_i \ \ preparation\ time\}$$

Then

$$h(S) = \sum_{\substack{order \in \\ \text{S.cant\_be\_completed\_in\_time}}} order.score$$

Notice that in the optimal solution there are more orders we cannot complete due to the time we need to spend between two orders.

Formally,

$$S.cant\_be\_completed =$$

$$= \{S.order_i \ \mid \ S.order_i \ \ cannot\ be\ completed\}$$

It is easy to see that *S.cant_be_completed_in_time* is contained in *S.cant_be_completed*. Hence,

$$h(S) = \sum_{\substack{order \in \\ \text{S.cant\_be\_completed\_in\_time}}} order.score \leq \sum_{\substack{order \in \\ \text{S.cant\_be\_completed}}} order.score = opt(S)$$

■

<u>Proof of consistency</u>

To prove that this heuristic is consistent, for each state S and its successor S' it follows that:

Let S be some state and S' be his successor, notice that

$$S.cant\_be\_completed \leq S'.\ cant\_be\_completed$$

Formally, denote $S.order_i$ to be the i'th order.

Define:

$$S.cant\_be\_completed =$$

$$= \{S.order_i \mid S.order_i \ \ expiration\ time \leq S.order_i \ \ preparation\ time\}$$

Then

$$h(S) = \sum_{\substack{order \in \\ \text{S.cant\_be\_completed}}} order.score$$

Notice that:

$$S'.cant\_be\_completed =$$

$$= \{S'.order_i \mid S'.order_i \ \ expiration\ time < S'.order_i \ \ preparation\ time\}$$

Hence,

$$
\begin{aligned}
| \{S'.order_i \ \mid \ s.t\ S'.order_i.expertion\ t \ \leq \ 0\} \mid = \\
&= | \{S.order_i \ \mid \ s.t\ S.order_i.expertion\ t \leq \ 1\} \mid \leq \\
&\leq |\{S.order_i \ \mid \ s.t\ S.order\_i\ .expertion\ t \ \leq \ 0\}|
\end{aligned}
$$

We can conclude that $\mathrm{S'.missed\ orders} \ \leq in\ \mathrm{S.missed\ orders}$.

From here we obtain:

$$
\begin{aligned}
priority(S) = & \sum_{\substack{order \in \\ S.\mathrm{cant\_be\_completed}}} order.score \\
& \leq \sum_{\substack{order \in \\ S'.\mathrm{cant\_be\_completed}}} order.score \ = priority(S')
\end{aligned}
$$

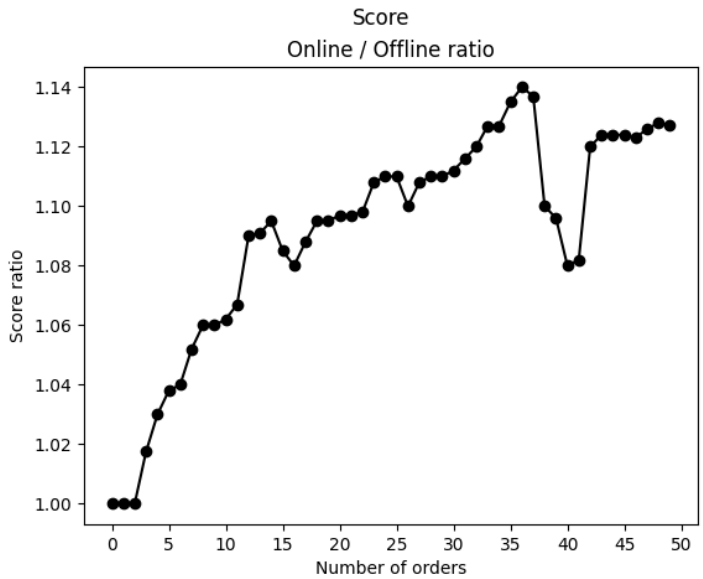
And because the cost from S to S' is always greater than 0, ***S'****.missed_orders*, that is the number of orders we can lose in successor ***S'*** is at least the number of points we can lose is in state *S*. Thus, $h(S) \ \leq \ h(S')$ and $h(S) < \ h(S') + \ c(S, S')$.

■

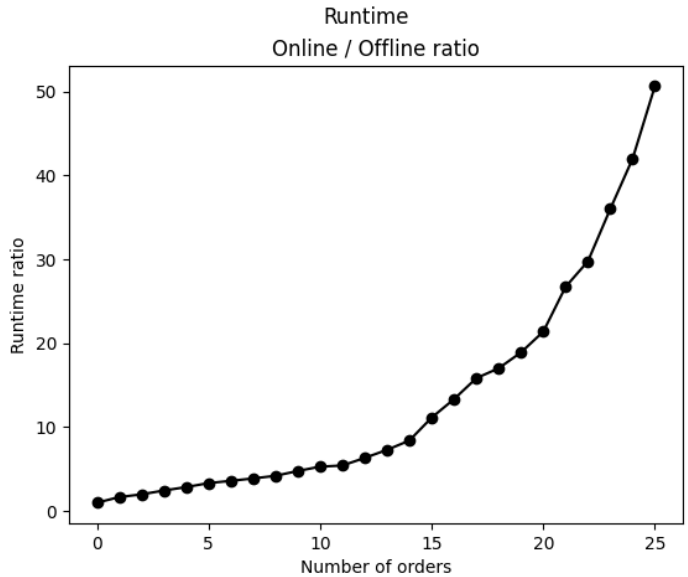
Evaluation and Results

Online vs Offline

We compared the offline and online search agents based on their final score and on their runtime. The graphs below show the average ratio between offline and online score and runtime on 25 service days, provided the same inputs for each service.



We can deduct from the first graph that the online solution obtains results that do not deviate much from the optimal ones, given by the offline agent.



The second graph shows that the runtime of the offline solution is exponential in comparison to the online agent, that calculates the next move in much less time.

Tactics

To better understand the impact of our agents, we will attempt to uncover their reasoning and examine their tactics. We will approach this task from different angles.

Base tactics: our agents when given the Lost Score heuristic, strive to minimize losses. We started with the assumption that this would lead to close-sighted tactics aiming to minimize losses in the short term. To our surprise, our agents were cleverer than expected and proved to be much more long-term score driven. As demonstrated in *Figure1* below. Where our online agent chooses to drop a current order restock and miss out on some low-value orders to fulfill a higher value order in the future.

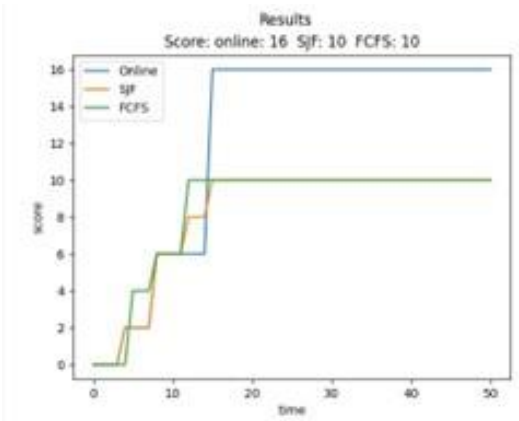


Figure1

To further examine this phenomenon, we isolated the situation by feeding orders to match the specific situation and arrived at the same results, as shown in the graph in *Figure2:*

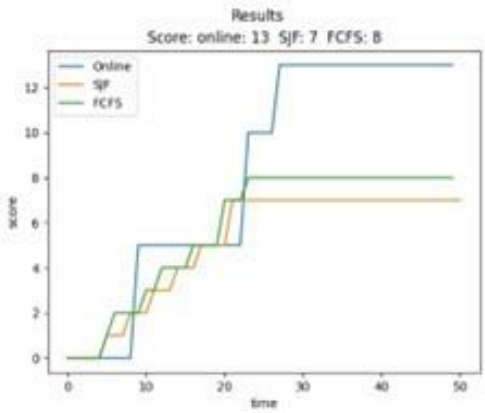


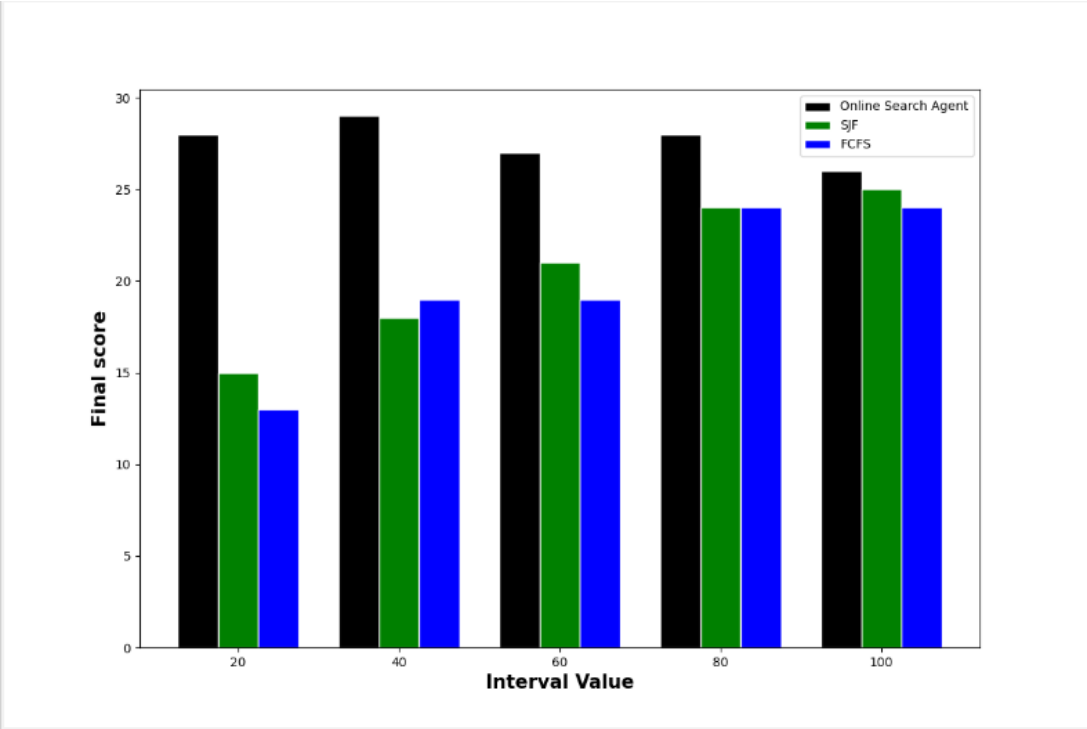
Figure2

Environment-influenced behavior

To examine the impact of the environment on our search agents, we played with the simulator's characteristics. Here are some of our findings and assumptions made from examining the results.

Order distribution:

In our original hypothesis, we assumed that order distribution would play a major role in the outcome of the simulations. Basing our assumption on the idea that when the orders are more evenly distributed, there are fewer overlaps where the agents need to choose which order to fulfill and which to disregard. This leads to a more even playing field where time-based prioritizing algorithms have a good chance of providing optimal solutions, hence artificial intelligence provides little to no benefit. Therefore, we chose to investigate the more interesting situations where orders aren't distributed evenly, and each decision has a big impact on which orders will be fulfilled and ultimately affects the final score. To demonstrate this phenomenon, we played with the time intervals in which the orders arrive, which will cause more overlapping between orders. And arrived at these results.



As we can see the smaller the intervals the larger the difference between our online agent and the time-based prioritizing algorithms SJF and FCFS. This came as no surprise to us, due to the nature

of the time-based prioritizing algorithms decision making. It did outperform our expectations and emphasized the strength of a good heuristic. That will take into consideration more than just time.

Summary

In our project we looked for a way to demonstrate the impact and capabilities of artificial intelligence in real-life scenarios. We strived to provide a solution that can easily be modified to suit a wide range of problems and provide a new approach to known exponential problems where real-time systems with classical computing are never an option. We used the technique of state space search with the added complexity of state space partitioning to approach an incomputable task.

In the process of implementing our solution, we were surprised again and again by the almost magical power of a well-thought-out heuristic, as if it was guiding us through an infinite space to a pinpoint solution while outperforming well-known prioritizing algorithms at their own game. Furthermore, we were reminded how important it is to properly formalize a problem, for it to have any chance at achieving its goal.

That being said, we strongly believe that given the proper devotion and expertise a problem of this sort can be optimized drastically to a point where it can be implemented into our daily lives to further the reach of humanity.