Planet: A Spatial and Temporal Task Planner

Larkin Flodin, Avery Johnson, Maraki Ketema, Abby Lewis, Will Schifeling, Alex Trautman

Introduction:

Everyone has tasks they must accomplish. Many people use a calendar or schedule to plan out their lives. However, these tools rarely account for both time and space. For example, Google Maps can plan out spatially where you should go, but not when to go based on the tasks you must complete. Scheduling applications can plan out when you have time to complete tasks, but they do not account for where you have to complete them. Our goal was to explore different algorithms to maximize the amount of value of all tasks that get completed to accomplish both temporal and spatial goals.

The Problem:

Given a set of *n* tasks we will output a schedule for a single person that maximizes the amount of value gained from the tasks that have been scheduled. Each task will have eight characteristics. First, each task will have a location (grocery shopping must be done at Econo Foods). The locations we use are simply coordinates on a plane, and thus all distances are Euclidean. Second, each task must have a duration (a meeting might last one hour). Third, each task has a release time (a homework assignment cannot be completed until it has be handed out on Wednesday at 5:00PM). Each task will also have a deadline (the assignment is due on Friday at 2:00PM). Next, each task will have time windows, or spans of time in which a task can be completed (grocery shopping can only happen when Econo Foods is open, 6:00 AM - 11:00 PM). Next, each task will be either optional or required (grocery shopping is required but watching Netflix is optional). Similarly, tasks will have value. The more important tasks - the required tasks - will have a higher value than less important tasks (we value going to class more than watching Netflix). Finally, tasks may have release tasks, or dependencies. These are tasks that must be completed before the task trying to be scheduled can be completed (you must buy groceries before you can eat dinner). Given all of these constraints, our algorithms aim to create a feasible schedule with the most value.

Relevant Problems:

Below we discuss some well-known research areas that are closely related to our problem. Most of our early research was concerned with these areas. A better understanding of these scheduling problems allowed us to make informed decisions as to which specific algorithms to implement.

Knapsack Problem:

In the traditional Knapsack Problem (KP), we are given a knapsack with a particular capacity and a set of objects, each of which has both a size and a value. The goal is to maximize the total value of items placed in the knapsack without exceeding the capacity of the knapsack. KP does not account for route-planning or spatial elements of our problem, and as a result, it was not particularly applicable to the algorithms that we chose to implement.

Orienteering Problem:

In the most basic version of the Orienteering Problem (OP), there is an individual holding a compass and a map. The player is given a list of locations, each of which has an associated reward and

duration. She must get from the starting point to the ending point within the time limit while maximizing her reward.[1] We have abstracted this to fit our own problem. The user has a time limit of the waking hours of the day, and wants to spend as much of the day as possible working on completing those tasks. We have also extended the rewards from the orienteering problem to include user-defined values.

The Orienteering Problem has several generalizations that were useful to us. One of these is the Team Orienteering Problem (TOP), where each person in a team has a path. We used this idea in our problem by making each path correspond to the tasks assigned to the user for a single day.

Another generalization is the Multi-Period Orienteering Problem with Time Windows (MuPOPTW). In this version of the OP, each node has some time window during which it must be completed, and there are set periods in which the agent can work.[2] For us this translated into the time of day the tasks are available to be completed (e.g. I can only return this shirt to Macy's while Macy's is open) and the concept of "waking hours", or the time of day in which a user can do their tasks (e.g. 9-5).

Traveling Salesperson Problem (TSP):

The TSP is the most basic routing problem. Given a list of cities to be visited, the goal of TSP is to evaluate the shortest possible route to visit each city once. The application of the TSP to our problem definition becomes relevant as we think about each of the extensions that we would have implemented given more time.

There are several classifications of the TSP. These classifications vary depending on the arrangement of the cities to be visited. In the definition of the Symmetric TSP, the distances between cities or nodes are the same in both directions, thus forming an undirected graph.[3] Our base problem fits into the symmetric variant of Traveling Salesman Problems.

There are several extensions to our base problem that we could have approached using generalizations of the TSP. There are many different approximation algorithms that can solve these different classifications of the TSP. These generalizations and their potential applications to our problem are discussed in the extensions section below.

Algorithms:

We ultimately implemented five different sets of algorithms: Brute Force, Pulse, an Integer Program, Greedy Algorithms, and Variable Neighborhood Search (VNS). Of those, Brute Force, Pulse, and the Integer Program produce optimal solutions if allowed to run without a time limit; the other algorithms do not guarantee an optimal solution.

Brute Force:

The first and most basic algorithm we implemented was a simple brute force algorithm. The algorithm takes in a set of tasks, and creates a schedule corresponding to each permutation of tasks.

Given a particular permutation of tasks, a schedule is constructed by taking the first task in that permutation and scheduling it as early as possible in the earliest available time slot satisfying all

---

[1] Vansteenwegen et al., 2009.
[2] Tricoire et al., 2009.
[3] Goyal, 2010.

constraints, then doing the same with each successive task in the permutation. If at any point some task in the permutation cannot be inserted anywhere into the schedule, we abandon that task. The reason this never causes the algorithm to lose the optimal solution is that in some other permutation of the tasks, the abandoned task will come earlier in the permutation and thus be schedulable.

Similarly, scheduling each task as early as possible does not pose a problem for finding the optimal solution. If a better schedule could be created by scheduling task 'A' later, that implies it is blocking some task 'B'. But then there is another permutation in which task 'B' precedes task 'A', and in that permutation task 'B' will be scheduled first.

While this algorithm is optimal, it is very slow, as there are $n!$ permutations of n tasks. To account for this, we implemented two alternative methods of Brute Force which, while not optimal, can be run with a time limit. The first simply runs Brute Force on every permutation in order and terminates after a time limit is reached. The second, which we referred to as Timed Random Iteration, tries permutations at random until the time limit is reached. As the number of permutations becomes very large, it turns out to often be better to try the permutations randomly, as this avoids getting stuck in one small portion of the solution space.

Pulse:

Another algorithm we researched and implemented was Pulse. Pulse is an optimal algorithm that can be run in parallel, and it is recursive. Intuitively, we imagine the "tree" of possible orderings of tasks. Pulse explores this tree and uses pruning methods to discard the portions of the solution set that we know will not include the optimal solution. Pseudocode for the algorithm follows:

```
Define the bounds matrix
Pulse(current ordering):
        Prune: by Feasibility, Soft Dominance, Detour, Bounds
        For each unscheduled task i:
                Pulse(current ordering + unscheduled task i)
```

Each iteration of Pulse begins with an ordering of tasks and does two things. First, it checks to see if the ordering should be "pruned" based on the four pruning methods. This checks to see whether or not this current ordering is worth further exploration. If not, it terminates and this branch of the recursive tree perishes. Otherwise, it calls Pulse on each new ordering that can be created by adding an unscheduled node to this current ordering. The pruning methods guarantee that we will never remove the optimal ordering from the solution set, and thus the algorithm is optimal.

The first pruning method is pruning by feasibility. This determines whether all the tasks in the current ordering can be scheduled in the specified order such that all tasks are scheduled within their time windows, there is sufficient time to travel between tasks, and the tasks can all be completed within the predetermined time limit. If an ordering is not feasible, then it should not be explored any further, and we prune that branch of the tree.

The next pruning method is pruning by soft dominance. The goal of this method is to determine whether any *similar* orderings will give us a better solution. If they will, then we know a better solution will be found somewhere else in the tree, and we can prune the branch based off this ordering. Pruning by soft dominance accomplishes this by creating new schedules; it takes the last task in the ordering and swaps it with each other task in the ordering to create a new ordering. For an ordering with five tasks,

pruning by soft dominance will create four new schedules to consider, one with the last task in the first position, one with the last task in the second position, and so on.

To determine which schedule is better, we look at the time at which the last task is completed. Since all the schedules that pruning by soft dominance considers are comprised of the same tasks, the most important difference is the completion time. If pruning by soft dominance determines that the same five tasks can be completed more quickly if we use a different ordering, then the current ordering is not worth exploring, and we may prune the current branch of the tree.

These methods help us intelligently disregard a large portion of the solution set (for details regarding pruning by detour and pruning by bounds, the other pruning methods, see Lozano et al.). The algorithm only requires two global variables: the bounds matrix, and a current best ordering, so it may be run in parallel. If an ordering manages to make it through the pruning stage of pulse without being pruned, we update our current best ordering (if we made an improvement), and call Pulse again, on orderings based upon this one, with a new task added to the end of the ordering. Pulse performed poorly compared with other algorithms in our implementation, although it was heavily lauded in the literature. This poor performance was largely the algorithm was designed to work with tasks that have just one time window, while our tasks had multiple time windows.

Integer Program:

The third optimal algorithm we implemented was a mixed-integer linear program. Very generally, in a linear program there is an objective function, along with a set of variables and constraints. The goal is then to find the setting of the variables such that the value of the objective function is maximized (or minimized), but the variables must be set in a way such that no constraints are violated. The "linear" portion is the additional restriction that the constraints must be linear; since the constraints are written in terms of variables, that means each constraint can only involve sums of constants multiplied by variables, and variables cannot be multiplied together. Furthermore, the "integer" part of the name means that the variables can only be assigned integer values. This avoids strange situations such as only scheduling half of a task.

The particular conversion of our problem into a set of constraints and variables is adapted from Tricoire et al.'s formulation used to solve the MuPOPTW, as the two problems are very similar. We create decision variables representing whether or not a particular task is in the schedule and variables from which the particular time the task takes place at can be determined, then aim to maximize the sum of the values of all scheduled tasks.

We used the PuLP library to set up the integer program, as it provides an easy way to interface with a variety of different industrial solvers used to solve integer programs. Though we experimented with some alternatives, we eventually settled on using the Gurobi Optimizer for the actual solving. The Gurobi Optimizer, like most other integer program solvers, uses an algorithm that exploits the fact that linear programs, in which the variables are not required to be integers, are much easier to solve than integer programs. The intuition behind this is that the set of constraints defines a high-dimensional polytope, and in a linear program, only the vertices of the polytope need to be checked. In an integer program, potentially every integer point within the polytope could represent the optimal solution.

At a high level, the Gurobi Optimizer operates by relaxing the integer program into a linear program where variables can have non-integer values, solving that linear program, then rounding a variable from that solution either down or up by imposing a constraint on the variable. Since it cannot be known in advance how to optimally round that variable, the Optimizer branches into two new integer

programs, one with an added constraint rounding the variable up, and another with a constraint rounding it down. This reduces the size of the search space in each of the two new programs, and then the process continues on each of the two programs separately.[4]

This process is sped up significantly by a number of complicated techniques which allow large portions of the tree created by this branching process to be pruned. However, solving these programs optimally still takes prohibitively long lengths of time for inputs with more than 30-40 tasks. To counteract this, the Gurobi Optimizer includes a time limit feature which allows it to be stopped after a particular length of time, and returns the best solution it has found thus far.

Greedy Algorithms:

Greedy algorithms are ones that lack foresight. They sometimes require some preprocessing, but in general they are very simple, and we found that they produce competitive results. Greedy-by-present-choice was the first greedy algorithm we implemented. The pseudocode follows:

While we can still schedule tasks:
    Select the task that can be finished as early as possible
    Schedule the selected task as early as possible

Due to the fact that this greedy algorithm fails to consider each task's value, it performed poorly relative to the other greedy algorithms.

Greedy-by-order was the second greedy algorithm we coded. The pseudocode for this class of greedy algorithms follows:

Given an ordered list of tasks:
For task i in the ordered list:
    For position j in the schedule from the earliest position to the latest position:
        If task i can be scheduled at position j such that it doesn't delay the following task:
            Schedule task i at position j (break out of the "position" for loop)

We tried several different orderings: ordering by deadline, value, a combination of value and availability, and ordering by a modified value (related to release tasks). All performed similarly, but when the number of release tasks increased our "ordering by a modified value" approach was the best.

Variable Neighborhood Search:

Variable Neighborhood Search (VNS) is a non-optimal algorithm useful for many different applications. The version of the algorithm that we worked most closely with came from Tricoire et al, 2009, but a more general version of the algorithm is outlined in Hansen and Mladenovic, 1997. The basic steps are as follows:

Generate an incumbent solution (we used a greedy algorithm)
While the stopping condition is not met:
    Shaking: Modify the solution in some way
    Iterative improvement: Increase the feasibility of that solution
    Feasibility: Find out if the solution is feasible

[4] "MIP Basics."

If the new solution is feasible and better than the old solution, replace the old solution.

The strategy is to modify a known feasible solution in some small way to find a new, possible solution. If no better solution is found, the old solution is changed in a different way. This is where the name Variable Neighborhood Search comes from. In the solution space, a "neighborhood" of some solution is the set of solutions that can result from modifying that solution in some set way. A "small" neighborhood is one in which the solutions are very similar to the original, while a "large" neighborhood contains solutions with bigger differences.

By varying the neighborhood sizes, it is possible for us to escape some local maxima within the solution space in our search for the global maximum. We generated neighborhoods using "Shaking". One example of a shaking operator is Optional Exchange 1, in which some tasks are removed from the schedule and replaced with some unscheduled tasks[5].

The schedule may no longer be feasible after the shaking step, so the next operation is called Iterative Improvement, and it decreases the infeasibility of the new schedule, while increasing its value. Finally, the schedule is sent through the Feasibility Check. This final check returns an empty schedule if the schedule was infeasible, or it returns the version of this schedule with the least possible duration if the schedule was feasible.

One way to think about the feasibility check is to realize that our algorithm only cursorily considers time windows while shaking and iteratively improving. The feasibility check actually makes sure every task can be scheduled within a time window, and finds the best time window for each task, such that this ordering of tasks takes the least possible amount of time.

VNS Visualizer:

In order to visualize and debug our VNS algorithm we created a visualizer that can step through the algorithm, displaying the changes that have been made in the schedule. We created the Visualizer using Python Mode for Processing. The visualizer consists of two main steps. First, when the Visualizer is run, it runs VNS. VNS then returns the final schedule, but also a list of schedules that appeared along the way. This list is created in VNS. Whoever wants to run the Visualizer decides when in the VNS algorithm they want to see the schedule. For instance, we used the Visualizer to debug each of the shaking and iterative improvement operators. Thus, we appended to the list of schedules before, after and sometime during these operators. When VNS is then called, these and only these schedules are appended. Thus, the user has full control of how many and which schedules will be displayed.

[5] Tricoire et al., 2009.

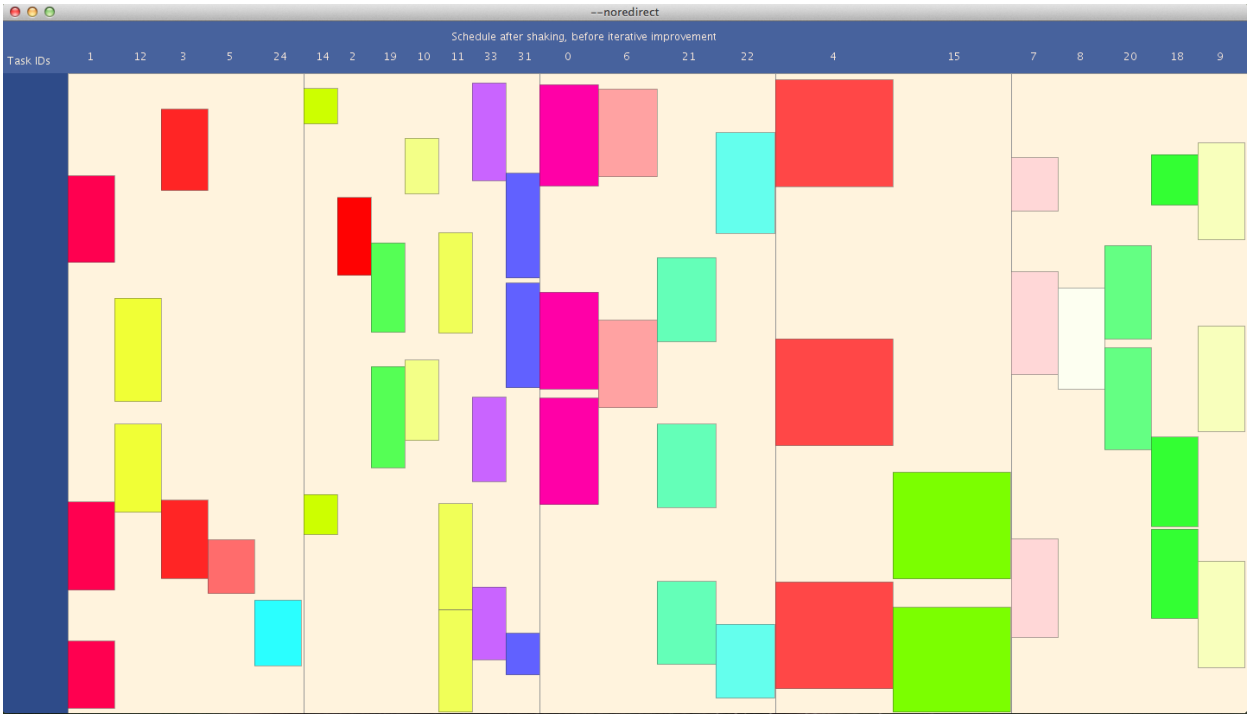

Figure 1: VNS Visualizer

The next step of the Visualizer is displaying the schedules that are in the list received from VNS in an informative way. In Figure 1, at the top of the screen we have listed the task IDs and also a string representing where in the VNS algorithm this schedule came from. The user adds these strings when appending the schedule in the VNS code. The five large beige rectangles represent days in our schedule. In each day, the colorful blocks represent the time windows for the task ID displayed above. Once tasks have been scheduled within the feasibility check we represent them with a grey block, which appears within one of the task's time windows. The user can then step through the schedules using the arrow keys.

Being able to view the tasks' time windows and scheduled times was immensely helpful in debugging the quite complicated VNS algorithm. We could easily see when schedules were infeasible because a bug has resulted in overlapping tasks. The Visualizer was also a great way to show off the functions of the different VNS operators. Creating the visualizer streamlined our coding and debugging processes immensely.

GUI:

We created a graphical user interface (GUI) to visualize our schedules and debug our algorithms. To do so we utilized Python Mode for Processing. The user begins by selecting an algorithm, a test file, and a time limit, and clicks "Plan it!" The GUI then runs the algorithm for the selected amount of time and displays the completed schedule (see figure 2).

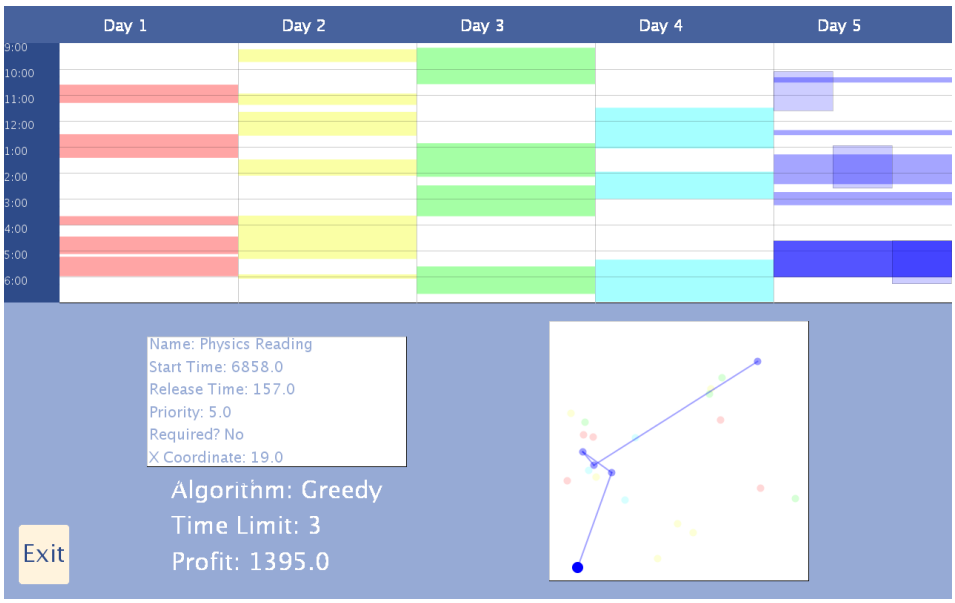

Figure 2: GUI displaying schedule

Once a schedule has been generated the GUI displays all the tasks scheduled on each day. There is a map in the lower right that displays the location of each task – represented by a dot on the x-y plane. When you mouse over a task on the map or a task on its scheduled day, both representations are highlighted; the task's time windows on that day appear as translucent boxes. Additionally, the route is displayed on the map and is represented by lines connecting consecutive task dots. The task information is displayed in a window in the lower left of the screen: the task name, start time, release time, priority value, required value, and coordinates.

Interfacing the GUI with the algorithms was non-trivial. For the Greedy Algorithms, Brute Force, and VNS we were able to create a schedule simply by importing the algorithm's schedule-solving function. However, with Pulse and the Integer Program we needed a different approach. We instead implemented these algorithms to accept time limits and test files as command line arguments. We ran the algorithms through the operating system and wrote the schedule as a comma separated value file. Finally, we read this file back into the GUI and created the visual schedule.

Testing and Comparison:

To compare our algorithms, we decided to randomly generate to-do lists. We created a baseline set of tests that had 64 tasks each. These tasks had the following properties: 8 out of 64 were required, all had up to 2 time windows per day, all optional tasks had a value between 1 and 10, all were located in a 60-by-60 grid, all had durations between 1 and 120, and no tasks had dependencies. The overall schedule itself had 3 days each with 1440 minutes to replicate an actual person's day.

Because we wanted to run each algorithm with a five minute time limit, it wasn't possible for us to test every possible combination of different parameters. Thus, we opted to vary only a single parameter at a time.

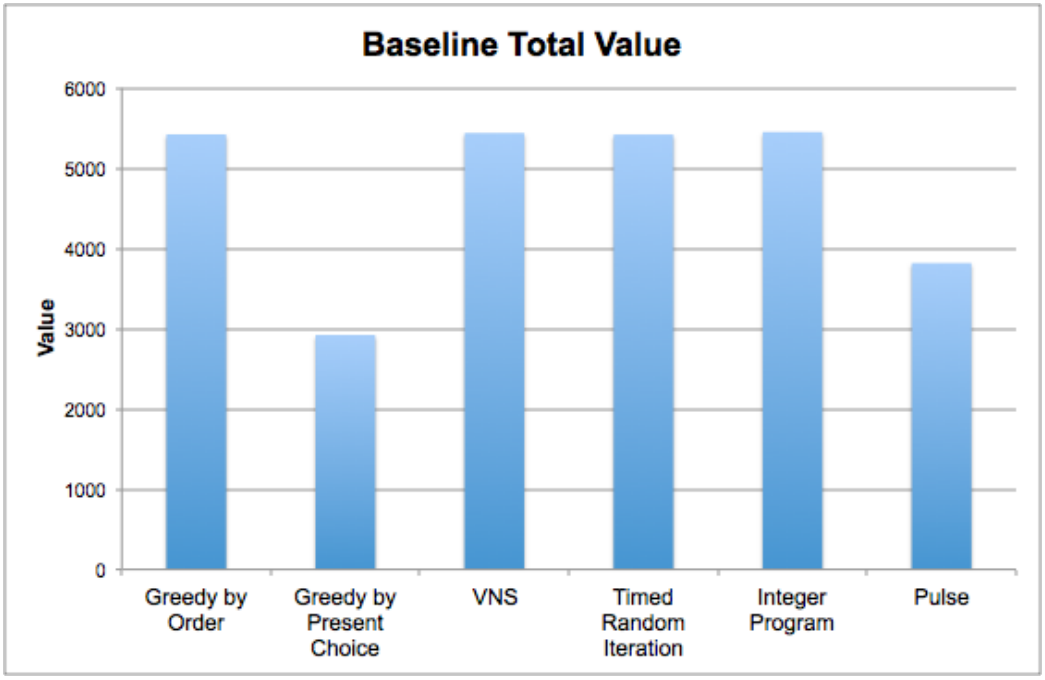

Figure 3

Figure 3 shows the average performance of each of our algorithms on the baseline set of tests described above. In this graph four algorithms appear to perform similarly, while Greedy by Present Choice and Pulse clearly underperform the others. This large discrepancy is due to those algorithms failing to schedule required tasks, which are modeled by assigning required tasks very inflated values. Greedy by Present Choice does poorly at accounting for high differentials in value between tasks, so its poor performance is expected. Figure 4 gives a more in-depth look at the four top performing algorithms.

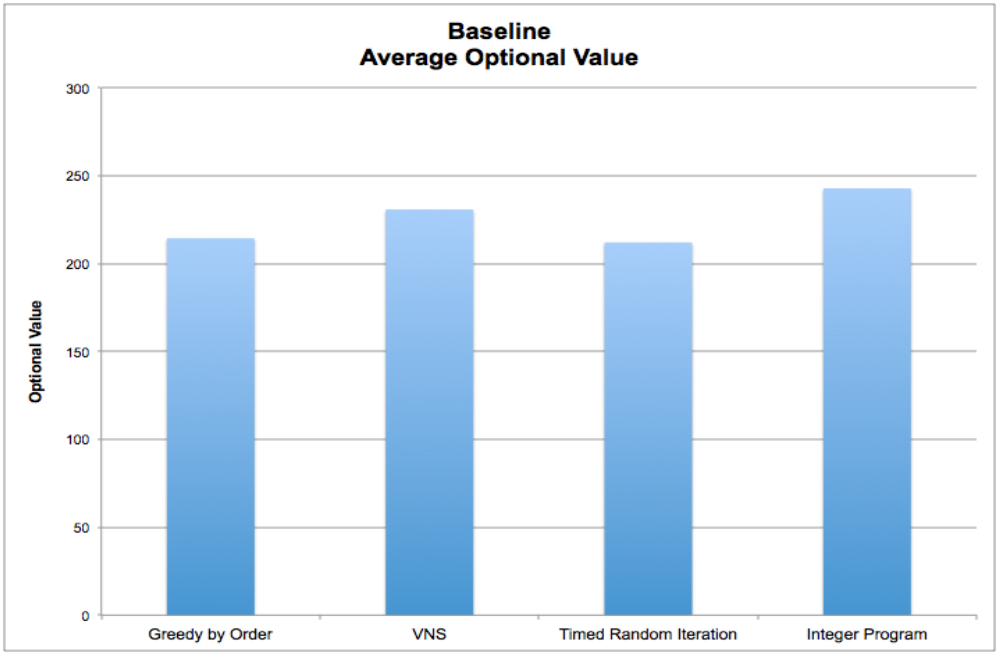

Figure 4

All four of the algorithms shown in Figure 4 successfully scheduled all 8 required tasks on every baseline test. Thus the value they obtained from scheduling optional tasks gives a more accurate picture of their performance. VNS is able to significantly improve the greedy result when running for only five minutes, but is still on average slightly outperformed by the integer program. As expected, Timed Random Iteration does not perform especially well, as it is very simplistic.

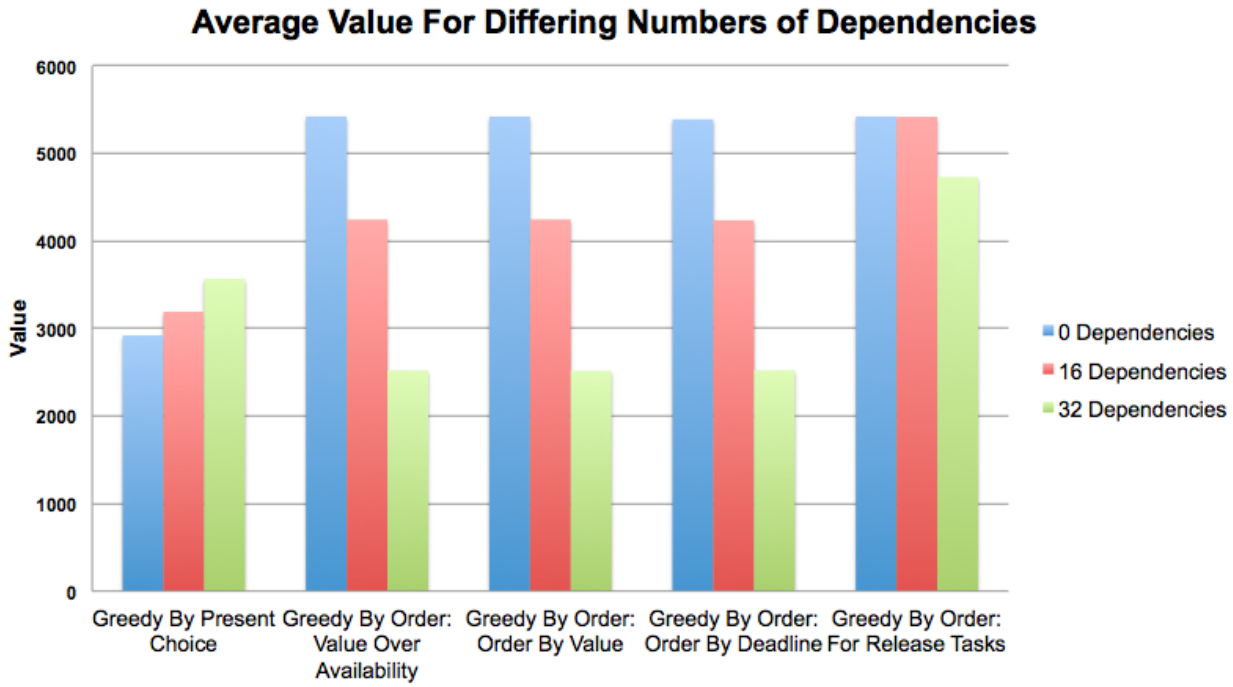

Figure 5

Figure 5 displays a comparison of the performance of our different greedy algorithms as we varied the number of dependencies; that is, the number of tasks that require another task to be completed before they can start. We expect to see a decrease in performance as more dependencies are added, as this represents a more complex schedule. This is generally the trend that is observed, though Greedy by Present Choice's increased performance as dependencies increased is somewhat confounding. Additionally, note that our greedy algorithm designed to handle release tasks (another term for dependencies) performs significantly better than our other greedy algorithms as the number of dependencies increases.

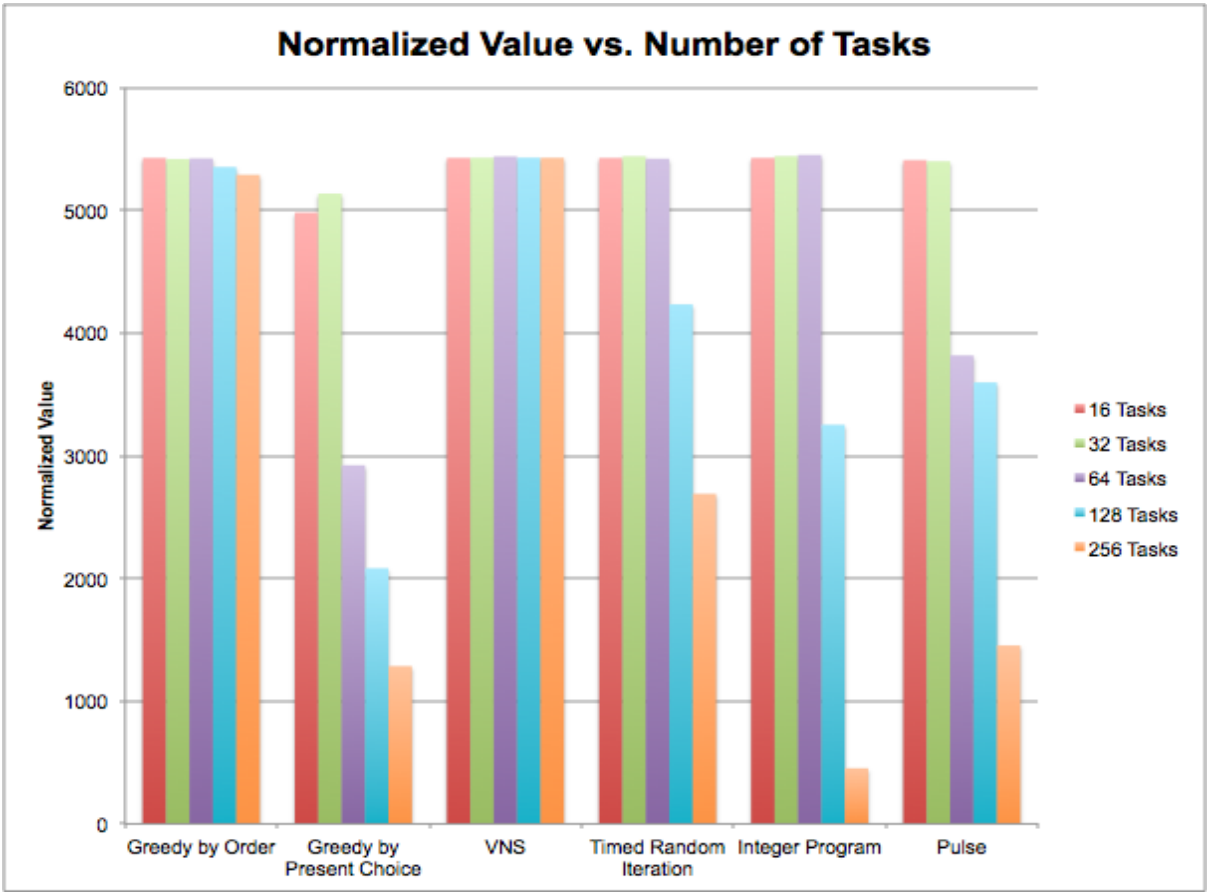

Figure 6

In Figure 6 we display the results from varying the number of tasks available to be scheduled. The value is normalized, because the value provided by required tasks becomes artificially inflated as more tasks become available. Ideally we would have normalized by scaling the values based on the optimal value, but since that cannot be computed in a reasonable length of time, we instead scaled the values based on the value of the VNS incumbent solution (the best performing greedy algorithm).

Greedy by Order and VNS appear to perform quite consistently as the number of tasks is increased; this is likely due to the fact the greedy algorithm can compute the schedule very quickly regardless of the number of tasks, and VNS uses that solution as an incumbent, though the data may be slightly skewed from the normalization process. Timed Random Iteration, Integer Program, and Pulse, though, all experience significant decreases in performance as the number of tasks available becomes larger. Pulse starts to perform worse with only 64 tasks, while the other two algorithms fair well until 128 and 256 tasks are available.

Given More Time:

In this section we discuss the extensions that we would incorporate into our project if we had more time. Note that this list of extensions is much smaller than the one that we began with. All the extensions that were implemented have been discussed in previous sections.

Google Maps:

All of our algorithms currently use locations based on an x-y plane. If incorporated, the Google API would allow us to work with real world locations and distances. The implementation of this extension would also enable the incorporation of the other extensions to be described below.

Location Groups:

The Generalized Traveling Salesman Problem (GTSP) is an extension of the TSP. In GTSP, nodes (representing task locations) are put into categories based on the classification of services provided at each location. The goal of this problem is to find a minimum cost path that allows for one node from each category to be visited; such a path is called a minimum-weight hitting path. A variant of GTSP is GTSPP (Generalized Traveling Salesman Path Problem), which considers when the traveler must start and end the schedule at specific locations.

GTSPP is an important extension to our problem. Breaking errand locations down into categories is critical; there could potentially be many locations at which one task can be completed. For example, I might be able to go grocery shopping near my home or near my work, and one might be much more convenient. There are several algorithmic approaches for solving GTSPP, but the one that caught our attention is an approximation algorithm outlined by Rice and Tsotras.[6]

Dynamically Changing Schedule:

Another generalization we could consider would be allowing tasks to be added after the schedule has already been generated and some tasks may have already been completed. If the time required for schedule generation is small, it may be feasible to simply recalculate the schedule with the new tasks added in and completed tasks removed. In a sparse schedule, it may be possible to insert the new tasks into empty space between other tasks, using an approach similar to the Knapsack Problem.

Alternatively, if there are many new tasks to add and the time required to compute a new schedule is too great, we could look into adapting our algorithm into an online algorithm. There is a considerable amount of literature available for the online TSP,[7] which our problem generalizes. The online TSP is much like the prize-collecting TSP, but the jobs (or prizes) are released while the traveling salesman is en route.

Unfortunately, there is not yet a lot of literature available about the online Orienteering Problem,[8] which our problem more closely resembles. It seems possible we could generalize some existing algorithms for online TSP to work with our algorithm,[9] though we would need to do more research on how exactly we could accomplish this.

Dynamic Edge Weights:

One interesting variation of the base problem is TSP with dynamic edge weights. This setup represents any situation where the distances or times to travel between cities are changing. An algorithm that could handle dynamic edge weights could design a schedule that avoided driving during rush hour. This approach would help the user make the most efficient use of her time.

This extension could be relatively easy to implement from our base algorithm. Further research into existing approaches to the dynamic TSP needs to be performed; however, there exists an algorithm

---

[6] Rice and Tsotras, 2013.
[7] Jaillet and Wagner, 2006.
[8] Luteyn, 2013.
[9] Jaillet and Lu. *Online Traveling Salesman Problems with Service Flexibility*, 2009.

closely related to the Ant Colony System (ACS) that incorporates these dynamic edge weights. It is yet to be determined what other algorithms exist to approach the dynamic TSP.[10]

Conclusion:

The main focus of this project was developing algorithms. Rather than focusing on solving many different problems, we created many algorithms that work well on solving a more specific set of problems. We also created a graphical user interface both to show off our algorithms and to provide an in-depth look at when and where tasks were scheduled. We hope you enjoyed our project. Thanks for reading!

[10] Wen et al., 2012.

References:

Baptiste, Philippe, Laurent Peridy, and Eric Pinson. *A branch and bound to minimize the number of late jobs on a single machine with release time constraints*, 2003.

Chekuri, Chandra and Sanjeev Khanna. *A Polynomial Time Approximation Scheme for the Multiple Knapsack Problem*, 2006.

Chen, Bo, Chris Potts, and Gerhard Woeginger. *A Review of Machine Scheduling: Complexity, Algorithms and Approximability*. Kluwer Academic Publishers. 1998.

Coffman. *Computer and Job-Shop Scheduling Theory.* New York: John Wiley & Sons. 1976.

Cook. *In Pursuit of the Traveling Salesman.* Princeton: Princeton University Press. 2012.

Dauzère-Pérès, Stéphane. *Minimizing late jobs in the general one machine scheduling problem*, 1995.

Dijkstra. *A note on two problems in connexion with graphs,* 1959.

Duque, Daniel, Leonardo Lozano, and Andrés L. Medaglia. *Solving the Orienteering Problem with Time Windows via the Pulse Framework*, 2015.

Goyal. *A Survey on Travelling Salesman Problem,* 2010.

Hansen, Pierre, and Nenad Mladenovic. *Variable Neighborhood Search*, 1997.

Jaillet, Patrick and Michael Wagner. *Online Routing Problems: Value of Advanced Information as Improved Competitive Ratios*, 2006.

Leonardo Lozano, Daniel Duque, Andres L. Medaglia. *An Exact Algorithm for the Elementary Shortest Path Problem with Resource Constraints*. Transportation Science, 23 Jan 2015.

Luteyn, Corrinne. *The UAV-Mission Planning Problem with Time Windows and Stochastic Fuel Consumption*, 2013.

Martello and Toth. *Knapsack Problems: Algorithms and Computer Implementations.* West Sussex: John Wiley & Sons. 1990.

"MIP Basics." *Gurobi Optimization.* Web. Accessed 10 March 2015.

Phol. *Bidirectional Heuristic Search in Path Problems,* 1969.

Rice and Tsotras. *Exact graph search algorithms for generalized traveling salesman path problems,* 2012.

Rice and Tsotras. *Parameterized Algorithms for Generalized Traveling Salesman Problems in Road Networks,* 2013.

Tricoire et al., *Heuristics for the multi-period orienteering problem with multiple time windows*, 2009.

Vansteenwegen et al., *The orienteering problem: A survey*, 2009.

Wen, Xingang, Yinfeng Xu, and Huili Zhang. "Online Traveling Salesman Problem with Deadline and Advanced Information." *Computers & Industrial Engineering* 63 (2012): 1048-053. Web.