# Optimal Task Scheduling in Collaborative Software Development

Emily Sheehan, Jack Heuberger, Nick Crispino, and Jake Browning

Washington University in St. Louis

April 22, 2023

## Abstract

Before work can begin on any software development project, features must be broken up into smaller tasks and delegated across team members. In the case of one-person teams, not much thought is put into how these tasks are delegated; the solo developer will complete every task at their discretion. In the case of multi-person teams, however, deciding how to delegate tasks is not trivial; different team members may take longer to complete specific tasks, some tasks may be dependent on the completion of another task, and some tasks may only be able to be completed by a subset of the team members. This research analyzes multiple known methods and algorithms to optimize parallel task scheduling problems in collaborative software development. As a proof of concept, we build an example scenario and compute an efficient task schedule under different constraints. These computed schedules are then compared to schedules determined by a human subject, and the quality and time spent to arrive at the solutions are compared.


## 1. Introduction

Efficient scheduling is an important issue that emerges in many applications. In general, scheduling problems involve solving for an optimal schedule under various objectives, different environments, and specific problem characteristics [1]. In practice, the most challenging part about finding an efficient schedule is first abstracting the real-world issue (the objective and the constraints) into a form that you can apply existing algorithms to solve it.

Take as an example the problem of hiring applicants to fill several jobs. Each applicant has a subset of the jobs they are willing to fill, each job can only accept a single applicant, and each applicant can only work a single job; these are the constraints. To abstract this problem, we first define the objective function we want to maximize as the total number of filled jobs. We can then encode the constraints as a bipartite graph, with edges going from applicants to jobs if an applicant is willing to fill that role. A solution, then, is finding the most significant subset of edges such that no two edges are incident to each other. Luckily, this is a well-known construction named Maximum Bipartite Matching, and efficient algorithms exist to solve this problem [2]. Other typical applications include job scheduling and resource sharing in digital circuits [3], flow-shop scheduling [4] which models assembly lines, and weighted task scheduling where tasks are given priority [5].

In this paper, we focus on the issue of delegating tasks to team members, with various constraints. First, in software development, there are often tasks that only a subset of the team members can complete. For example, executing SQL commands on a production database should only be performed by Database Administrators. Second, different team members may work at different speeds. For instance, while both a senior developer and a junior developer may be able to complete a particular task, the senior

developer may do so more quickly. Third, it is often the case that tasks have dependencies or an order in which they need to be completed. Consider two tasks, one is creating a database table, and the other is filling this table with data. There is a dependency between these two tasks, as one cannot fill the table with data if it does not exist.

One last aspect we implicitly model is that each task has a length associated with it. In other words, the lengths of the tasks are not necessarily equal. This length can be encoded by the number of hours it is expected to take to complete, but frequently in software development, the difficulty of a task is measured in story points, especially when considering the Agile methodology [6]. A *story point* is an integer associated with a specific task. Team members agree on a relative weighting system, where higher story points indicate a more extensive workload. This definition of task length is better, as it allows us to encode the differing work rates across team members. For example, a senior developer may have a work rate of 1 story point per hour, whereas a junior developer may work at a rate of 0.5 story points per hour.

Our objective to minimize in this case is the time the team member with the most work takes. This is known as the makespan and is commonly used in parallel computing contexts to refer to the processor that takes the longest time to finish executing; this is analogous to our problem, where each team member is a processor that runs tasks. With this definition, constraints, and application-specific details, we can begin analyzing different approaches to model and solve this parallel task scheduling problem.

## 2. Background

The problem of delegating tasks to a team of software engineers is a parallel task scheduling problem. Many variants of this problem exist and we will consider a few relevant to us and our constraints. First, it is important to note that these parallel scheduling problems, in general, are NP-hard [7]. Therefore, the algorithms we discuss either find an approximate solution or return the optimal solution in exponential time. To reiterate, in the case of scheduling problems, the optimal solution is the schedule with the minimum makespan.

In the literature, the notation used for the parallel task scheduling problem is as follows. The objects that complete the tasks are called processors, and are notated by a set $\{P_1, \ldots, P_m\}$, that is, there are $m$ processors, with the $\mathtt{i}$-th processor notated by $P_i$. Similarly, there are $n$ tasks $\{T_1, \ldots, T_n\}$, with the $\mathtt{j}$-th task notated by $T_j$. Note that in all of our scenarios, the tasks are scheduled non-preemptively, meaning once a processor starts a task they must finish it before starting a different task.

The simplest case of such variants is known as identical-machine scheduling. In this scenario, the constraints of our original problem are relaxed. Each $P_i$ executes

tasks at the same rate (they are identical), any $T_j$ can be completed by any $P_i$, and every $T_j$ can be completed at any point in time (i.e., there are no dependencies).

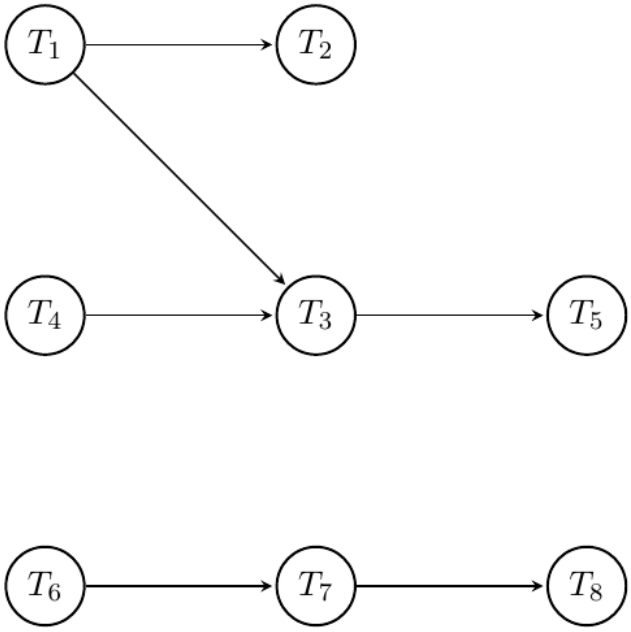
In this case, it can be shown that a simple greedy algorithm known as *list scheduling* achieves a $(4/3 - 1/(3n))$-approximation [7]. The algorithm works by first constructing any list $L := (T_{k_1}, \ldots, T_{k_m})$ of the tasks, where the subscript $k_a$ represents the task that is $a$-th in the list. Then, we let $P_{min}$ be the processor with the least amount of scheduled work, and for every task $T_{k_a} \in L$ (in order), we add $T_{k_a}$ to $P_{min}$, and calculate the new $P_{min}$. This loop repeats until there are no tasks left in the list. We notice a few benefits with this approach. First, the runtime is linear in the number of tasks, making the algorithm very efficient for problems with a large $m$. We also notice that the algorithm produces a solution that is not that far away from the optimal solution. While a $(4/3 - 1/(3n))$-approximation isn't the tightest bound we might hope for, it at least gives us some guarantees as to the quality of the solution we should expect; it is not always the case where a simple algorithm produces a reasonable approximation. However, there are some negative properties about this formulation. We recall that in software development, some tasks may be dependent on the completion of other tasks, some tasks may only be able to be completed by a subset of the developers, and some developers may work at different rates. These three constraints are not modeled by this formulation; we have sacrificed the accuracy of the model for efficiency.

Next, we look at a more complicated variant known as uniform-machine scheduling. In this formulation, everything is the same as in identical-machine scheduling, except the machines are no longer identical. That is, each machine $P_i$ has a *speed factor* $s_i$. If we let task $T_j$ have length $\ell_j$, then the time it takes some processor $P_i$ to complete task $T_j$ is $\ell_j / s_i$. As you can see, larger speed factors produce lower completion times. Luckily, as with identical-machine scheduling, there is a greedy algorithm that produces an approximate solution. The difference here are that tasks from $L$ are not assigned to the machine with the least amount of scheduled work. Instead, we assign a task $T_{k_a} \in L$ to $P_i$ if and only if the finish time of $T_{k_a}$ is minimized. This scheduling scheme is known as longest-processing-time-first (LPT), and it has also been shown that this algorithm produces a solution that is a $(4/3 - 1/(3n))$ -approximation [8]. The pros in this case are the same as in identical-machines, however, now we are also modeling the developers' speed.

While these formulations are helpful, we are still missing out on some of the constraints we are interested in. In this next formulation, we consider dependency constraints. A dependency between two tasks $T_a$ and $T_b$ is a directed relationship. Suppose that $T_b$ cannot be completed until $T_a$ is finished. We can represent this dependency by encoding these tasks as nodes in a directed graph, and constructing an edge from $T_a$ to $T_b$. Doing this with every relationship between tasks, we get a directed

graph. More specifically, we get a directed acyclic graph (DAG) which defines a partial-ordering on the tasks. To visualize this, consider the following example.

Suppose that $L := (T_3, T_1, T_2, T_4, T_6, T_5, T_7, T_8)$. Then, the following DAG is one possible example of the dependencies of these tasks:




In this case, we notice that $T_5$ cannot be processed until $T_1, T_4,$ and $T_3$ are processed. This example also shows that the DAG need not be connected; we see that tasks $T_6, T_7,$ and $T_8$ create their own independent chain of dependencies.

This partial ordering, encoded by a DAG, can be injected into the algorithms we have seen previously in order to produce solutions which do not violate any dependency. Specifically, take the greedy LPT list scheduling algorithm given for uniform-machines and make the following modifications. Before assigning some task $T_{k_a}$ to a processor $P_i$, first query the DAG and observe whether there are any incoming edges to task $T_{k_a}$. If there are no incoming edges, schedule $T_{k_a}$ as usual. However, if there are incoming edges, this means some tasks that $T_{k_a}$ are dependent on have not yet been completed. In this case, the algorithm moves onto the next task in the list, $T_{k_{a+1}}$, and repeats.

Once a processor has completed a task, that task is then removed from the DAG. This possibly removes incoming edges to some task, allowing it to be scheduled the next time the algorithm considers it. Lastly, one crucial edge case to handle is when all remaining tasks in the list have incomplete dependencies. In this case, the algorithm waits until the next processor finishes some task and goes through the list again from the beginning.

The runtime for this algorithm is clearly polynomial, and it can be shown to return a schedule that is a $2 - 2/(n+1)$-approximation. Unlike the previous two algorithms which produced a solution that, in the limit, was a $4/3$-approximation, this algorithm converges to a $2$-approximation for large $n$. This is clearly worse from a quality perspective, but we are now modeling the dependencies of the tasks, which may be a tradeoff we are willing to make depending on the number of dependencies in our problem.

Note that to formalize these combinatorial optimization problems, we could frame them as integer programming problems [11], [12], [13], for which we could use an ILP solver. Additionally, there are many other ways [14] to solve scheduling problems like the ones we presented. The branch and bound algorithm can solve problems precisely by using a search tree, keeping track of bounds, and pruning when called for, but it can be slow. Therefore, many instead choose to use approximation algorithms. For example, one could think of using variations of local searches, such as a form of simulated annealing or a genetic algorithm, or different formulations of greedy algorithms. We chose the latter due to their simplicity and decently reasonable bounds.

## 3. Problem Formulation

Staying consistent with the notation introduced in the previous section, we will have $n$ developers and $m$ tasks that we wish to delegate across them. We continue to write $P_i$ for *person* $i$ and $T_j$ for task $j$. Also, recall that in the uniform-machines variant, each processor has a speed $s_i$ and each task has a length $\ell_j$. In the context of software development, however, it is unclear what the *length* of a task is or what the *speed* of a developer is. Instead, we define a *work rate* $r_i$ and a *difficulty* $d_j$. The difficulty $d_j$ is in terms of story points, and naturally the work rate $r_i$ is in terms of story points per hour. Before, we had written that it takes a processor $P_i$ a time of $\ell_j/s_i$ to complete a task $T_j$, and this is still the case with our new definitions; it takes a person $P_i$ a time of $d_j/r_i$ to complete a task $T_j$.

Dependencies of tasks has a natural analog to the software development tasks that the developers work on; not much changes in this aspect. We continue to encode our dependencies as DAGs in order to get a partial ordering of the tasks.

## 4. Application

We will construct an example to illustrate the algorithms and formalization discussed in previous sections. In this scenario, we seek to optimize task management and scheduling assignments for Company A, a software start-up based in St. Louis, Missouri. The company, founded in 2020, has 75 employees, including 20 software engineers, and develops day-trading algorithms for financial institutions. The software

branch of the company is composed of 4 divisions, traditional frontend and backend teams, as well as both security and data teams.

Regarding constraints unique to Company A, the company allows its employees to work remotely, so many software engineers are scattered throughout the continental United States. The team holds daily scrum meetings at 2 PM Central Time every day, but otherwise, employees are free to work at their discretion so long as their work is completed in due time. Furthermore, Company A's board of investors is growing increasingly anxious and demands that the start-up releases a fully-functional beta by the end of 2023. The company must produce code quickly and assign weight and importance to features. Additionally, as with all software development, dependencies must be considered. A software dependency is a relationship between components where one relies on another to function correctly. Since Company A is constantly pushing new features, the order in which components are built must be carefully selected to optimize software production and prevent bugs and delays.

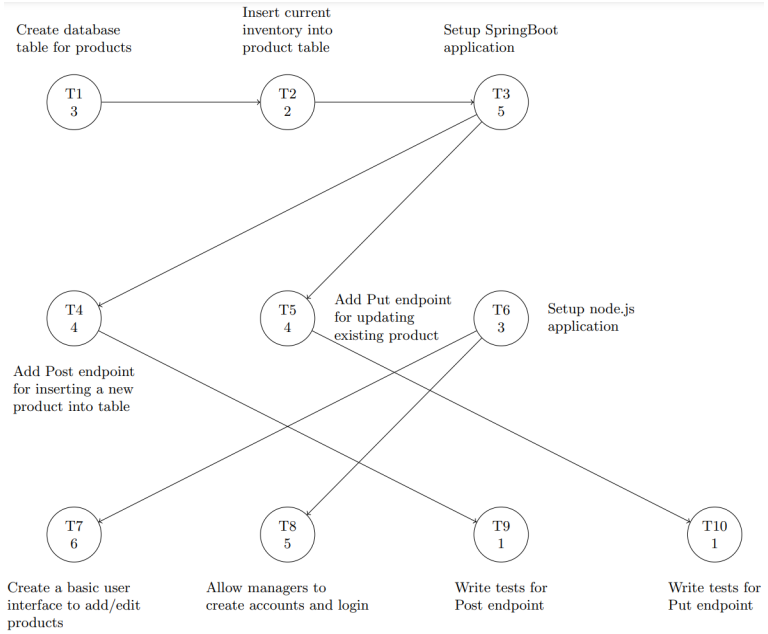
The software segment of the company consists of 5 senior developers and 15 junior developers. The company has determined that senior developers produce four story points worth of work per hour, while junior developers complete one story point per hour. The more points a story is assigned, the more complicated it is estimated to be. In this example, the team has 200 tasks that need to be assigned across developers. Below is a table containing the task and point breakdown, along with real-world examples that would apply to each category.

| ***Tasks*** | ***Points*** | ***Example*** |
|---|---|---|
| *1-75* | *1* | Writing tests, creating API endpoints |
| *76-100* | *2* | Inserting inventory into a product table |
| *101-110* | *3* | Writing database-dependent endpoints |
| *111-125* | *4* | Simple database query operations |
| *126-150* | *5* | Creating an admin panel to manage inventory |
| *151-175* | *6* | Implementing role-based authentication |
| *176-185* | *7* | Creating scripts to export and analyze data, CI/CD Pipelines |
| *186-196* | *8* | Editing pipelines to include report generation and sale data visualization |
| *197-200* | *9* | Exploratory tasks for new technologies, such as Large Language Models |

For the dependency algorithm, we define the following relationships between tasks. If a task group is not listed, they have no dependencies and no tasks depend on them.

| ***Tasks*** | ***Dependencies*** | ***Example Scenario*** |
|---|---|---|
| *1-20* | *None* | Creating a database table |
| *76-80* | *1-20* | Endpoints that perform database operations |
| *125-133* | *None* | Initializing a SpringBoot application |
| *111-117* | *125-133* | Writing endpoints for a SpringBoot API |
| *53-69* | *111-117* | Writing tests for SpringBoot endpoints |
| *101* | *None* | Build a Node.js App |
| *123-125* | *101* | Build an admin page UI |
| *170-175* | *101* | Edit endpoints that allow for admin control |

The following is a simplified example with only 10 tasks. Each task has an ID (1-10), name, and story point assignment (inside the node below the ID) as well as incoming/outgoing edges indicating dependency relations.



Due to its complexity, we will not provide a graph for our entire example; however, it would be a natural extension of what's provided above.

## 5. Results

Using our example scenario from the previous section (with the different work rates, different task difficulties, and task dependencies) we produced a scheduling according to the identical-machines problem, the uniform-machines problem, and the scheduling with dependencies problem. We index each developer with a number 1 through 20, and each task by a number 1 through 200. Additionally, the 5 senior developers are indexed by the numbers 1 through 5. Below are the results.

Our algorithms, in the form of a Python script, are also accessible via our GitHub repository and the results are displayed in a .txt file.

Makespan Comparison

| | Identical-Machines | Uniform-Machines | Dependencies |
|---|---|---|---|
| Makespan | 38 hr | 22.5 hr | 23 hr |

Identical-Machines

| Person | Task Sequence |
|---|---|
| 1 | 1, 21, 41, 61, 81, 101, 121, 141, 161, 181 |
| 2 | 2, 22, 42, 62, 82, 102, 122, 142, 162, 182 |
| 3 | 3, 23, 43, 63, 83, 103, 123, 143, 163, 183 |
| 4 | 4, 24, 44, 64, 84, 104, 124, 144, 164, 184 |
| 5 | 5, 25, 45, 65, 85, 105, 125, 145, 165, 185 |
| 6 | 6, 26, 46, 66, 86, 106, 126, 146, 166, 186 |
| 7 | 7, 27, 47, 67, 87, 107, 127, 147, 167, 187 |
| 8 | 8, 28, 48, 68, 88, 108, 128, 148, 168, 188 |
| 9 | 9, 29, 49, 69, 89, 109, 129, 149, 169, 189 |
| 10 | 10, 30, 50, 70, 90, 110, 130, 150, 170, 190 |
| 11 | 11, 31, 51, 71, 91, 111, 131, 151, 171, 191 |
| 12 | 12, 32, 52, 72, 92, 112, 132, 152, 172, 192 |
| 13 | 13, 33, 53, 73, 93, 113, 133, 153, 173, 193 |
| 14 | 14, 34, 54, 74, 94, 114, 134, 154, 174, 194 |
| 15 | 15, 35, 55, 75, 95, 115, 135, 155, 175, 195 |
| 16 | 16, 36, 56, 76, 96, 116, 136, 156, 176, 196 |
| 17 | 17, 37, 57, 77, 97, 117, 137, 157, 177, 197 |
| 18 | 18, 38, 58, 78, 98, 118, 138, 158, 178, 198 |
| 19 | 19, 39, 59, 79, 99, 119, 139, 159, 179, 199 |
| 20 | 20, 40, 60, 80, 100, 120, 140, 160, 180, 200 |

## Uniform-Machines

| Person | Task Sequence |
|---|---|
| 1 | 1, 6, 11, 16, 36, 41, 46, 51, 71, 76, 81, 86, 101, 111, 116, 121, 126, 141, 151, 156, 161, 176, 186, 191, 196 |
| 2 | 2, 7, 12, 17, 37, 42, 47, 52, 72, 77, 82, 87, 102, 112, 117, 122, 127, 142, 152, 157, 162, 177, 187, 192, 197 |
| 3 | 3, 8, 13, 18, 38, 43, 48, 53, 73, 78, 83, 88, 103, 113, 118, 123, 128, 143, 153, 158, 163, 178, 188, 193, 198 |
| 4 | 4, 9, 14, 19, 39, 44, 49, 54, 74, 79, 84, 89, 104, 114, 119, 124, 129, 144, 154, 159, 164, 179, 189, 194, 199 |
| 5 | 5, 10, 15, 20, 40, 45, 50, 55, 75, 80, 85, 90, 105, 115, 120, 125, 130, 145, 155, 160, 165, 180, 190, 195, 200 |
| 6 | 21, 56, 91, 131, 166 |
| 7 | 22, 57, 92, 132, 167 |
| 8 | 23, 58, 93, 133, 168 |
| 9 | 24, 59, 94, 134, 169 |
| 10 | 25, 60, 95, 135, 170 |
| 11 | 26, 61, 96, 136, 171 |
| 12 | 27, 62, 97, 137, 172 |
| 13 | 28, 63, 98, 138, 173 |
| 14 | 29, 64, 99, 139, 174 |
| 15 | 30, 65, 100, 140, 175 |
| 16 | 31, 66, 106, 146, 181 |
| 17 | 32, 67, 107, 147, 182 |
| 18 | 33, 68, 108, 148, 183 |
| 19 | 34, 69, 109, 149, 184 |
| 20 | 35, 70, 110, 150, 185 |

## Dependencies

| Person | Task Sequence |
|---|---|
| 1 | 1, 21, 26, 31, 36, 73, 78, 83, 103, 108, 120, 140, 145, 150, 158, 170, 175, 62, 67, 178, 198 |
| 2 | 2, 22, 27, 32, 37, 74, 79, 84, 104, 109, 121, 141, 146, 151, 111, 166, 171, 53, 63, 68, 179, 199 |
| 3 | 3, 23, 28, 33, 38, 75, 80, 85, 105, 110, 122, 142, 147, 152, 112, 167, 172, 54, 64, 69, 180, 200 |
| 4 | 4, 24, 29, 34, 39, 76, 81, 101, 106, 118, 138, 143, 148, 156, 168, 173, 55, 65, 176, 196 |
| 5 | 5, 25, 30, 35, 40, 77, 82, 102, 107, 119, 139, 144, 149, 157, 169, 174, 56, 66, 177, 197 |
| 6 | 6, 41, 86, 123, 153, 181 |
| 7 | 7, 42, 87, 124, 154, 182 |
| 8 | 8, 43, 88, 125, 155, 183 |
| 9 | 9, 44, 89, 126, 113, 57, 184 |
| 10 | 10, 45, 90, 127, 114, 58, 185 |
| 11 | 11, 46, 91, 128, 115, 59, 186 |
| 12 | 12, 47, 92, 129, 116, 60, 187 |
| 13 | 13, 48, 93, 130, 117, 61, 188 |
| 14 | 14, 49, 94, 131, 159, 189 |
| 15 | 15, 50, 95, 132, 160, 190 |
| 16 | 16, 51, 96, 133, 161, 191 |
| 17 | 17, 52, 97, 134, 162, 192 |
| 18 | 18, 70, 98, 135, 163, 193 |
| 19 | 19, 71, 99, 136, 164, 194 |
| 20 | 20, 72, 100, 137, 165, 195 |

# 6. Analysis

As tasks in large software engineering teams are usually divided by humans rather than algorithms, we assigned the following problem to a human scheduler in order to test the viability of our algorithms. Vinay Viswanathan is a CS student at Washington University in St. Louis. He received the task number and point value table from Section 4 and the following prompt:

*You are tasked with scheduling 200 software development tasks for 20 developers -- 5 senior engineers and 15 junior engineers. Tasks are assigned story points, 1 point representing the easiest tasks and 9 points being the most difficult. The point values for each task are described in the attached table. Senior engineers produce 4 points worth of work per hour, while junior engineers produce 1 point per hour. Assign tasks to each of the engineers to produce the most efficient schedule.*[1]

Viswanathan's result was naive and not optimized. Viswanathan utilized a greedy algorithm to assign tasks to engineers — all tasks with story points above a certain cutoff were assigned to senior engineers, while junior engineers were only assigned tasks worth a number of points below the cutoff.

It took Viswanathan 37 minutes to develop this greedy solution after trying various other arbitrary approaches. His first solution set a cutoff of 5 points, leading to a makespan of 69 hours to complete all tasks, or an error of over 200% when compared to the optimal uniform machine makespan of 22.5 hours for the same problem. Upon further analysis by Viswanathan, he realized that his attempt was non-optimal. At this point, we ended the experiment and analyzed his algorithm.

We have developed a simulation of this greedy approach using Python (linked in our repository). Through this simulation, we determined that the optimal cutoff for the greedy solution is for point values of 7, clocking in at 38 hours. This is an error of 69% compared to the optimal solution of 22.5 hours. Therefore, at best, Viswanathan could have produced a solution that was at best 69%[2] worse than the optimal solution using the greedy algorithm he derived.

# 7. Conclusion

For companies, optimal task scheduling directly delivers more money and higher margins. Across industries, people have been trying to find a solution for their teams for

[1] It is important to note that this is the uniform-machine variant of the problem. Because the nature of the experiment was to quickly determine a naive human solution, the human tester was not provided with task dependencies.

[2] Due to the nature of the experiment and the time it took the subject to develop the algorithm, the subject did not have time to determine the optimal cutoff of the implementation. Because of this, we feel as though it's fair to compare the optimal version of the subject's solution, as it disregards the constraints placed on the subject by the experiment.

decades, but this challenge has yet to be answered within the software industry. This paper is an attempt to tackle such a problem, for which we provide three algorithms and solutions that account for common scheduling optimization hurdles encountered by software engineers. The aforementioned algorithms, which increase in complexity, address the following constraints: variable productivity, varied task lengths, and component dependencies. Identical machines assumes equal productivity across engineers and ignores dependencies, uniform machines weighs productivity against a work rate for each developer, while the dependencies variant introduces and accounts for dependencies. Our paper, and the algorithms included within it, should be considered and evaluated by software teams of all sizes who hope to increase their productivity.

Viswanathan posed an interesting question to us during the human trial experiment — he asked whether he was one of the engineers. Generally on software engineering teams, one of the task schedulers is a developer on the team they are scheduling for. This means that the act of scheduling itself has an associated cost and time impact, and could be represented as a required dependency before tasks can be completed. Processes such as Agile attempt to use large group meetings and discussions to prioritize tasks and split them accordingly among team members. Yet, due to the time taken by these meetings, Agile is significantly more costly than running a task scheduling algorithm.

In the future, we hope to further evaluate this scenario by incorporating additional constraints. With the advent of artificial intelligence and programs that can create code, many tasks in software engineering have the potential to be automated away, leading to great improvements in efficiency. GitHub Copilot, the coding assistant developed by OpenAI, has been shown to increase programmer happiness and speed. In fact, in a case study done by GitHub, 90 engineers were tasked with writing a webserver in JavaScript; 45 used Copilot and 50 did not. In the former group, 78% finished in an average of 1 hour 11 minutes, whereas in the control group 70% finished in an average of 2 hours 41 minutes. These results were statistically significant at the 1% level, meaning that such AI assistants have already proven to be a great help to programmers [9]. Additionally, frameworks like AutoGPT [10] have shown it is possible to prompt an AI with simple instructions, such as a name, task, and goal, and have the AI continuously prompt itself to accomplish its goal. One could see how this may apply to simpler tasks like building a basic website or getting a domain-specific framework to work. Even if few in a company have a deep knowledge of tasks or languages (e.g., the aforementioned SQL developer), AI could essentially relax the constraints that require only certain developers to tend to certain tasks. We see the need to integrate this and all of other constraints derived from this new approach within our model; for example, we could assign probabilities to the efficiency gains an AI assistant may provide a junior and senior developer respectively; we could also add assistants as their own workers that

complete tasks in a relatively short time and place them between jobs in our DAG, adjacent to certain developers focused on prompt engineering. As these models are so novel, many efficiency gains are likely unrealized. More research needs to be done on this topic that allows for both individuals and companies to profit off the large potential of this new technology.

Another potential future feature could account for a developer's fundamental ability to contribute to specific tasks; limiters could include one's technical skillset, knowledge of the codebase, and/or security clearances. For example, assigning a DevOps engineer to a job that includes developing a metrics dashboard using a front-end framework like React may not be a valuable use of a company's resources. Similarly, for federal contractors, for example, assigning a Top-Secret project to an engineer with only a Secret level clearance is impossible. Future iterations of this project must consider such intricacies. Another constraint could revolve around breaking apart tasks into smaller stories, allowing for scheduling preemptively, or assigning multiple engineers to a single task. A solution to this challenge could incorporate artificial intelligence, such as training a model to determine how a task could be delegated, and furthermore, which parts of a task could be completed using a program like GitHub Copilot or AutoGPT.

Ultimately, there are many potential extensions to the optimal task scheduling problem introduced here. Our paper serves to provide a baseline for future computer scientists to build on and adjust to best suit their engineering team's requirements.

## 8. Team Contributions

Jake
- Literature review of parallel task scheduling algorithms
- Abstract, Introduction, Background, and Problem Formulation sections
- Implemented Dependencies Algorithm

Emily
- Application & Conclusion sections
- Implemented Identical Machines Algorithm & Uniform Machines Algorithm
- Analysis

Nick
- Application & Conclusion sections
- Implemented Dependencies Algorithm
- Reviewed and summarized alternative problem formulations

Jack
- Conducted real-person scheduling experiment
- Developed Python greedy algorithm implementation
- Results section, Analysis

# 9. References

[1] Hochbaum, Dorit S. “The Scheduling Problem.” *RIOT*, UC Berkeley, 1998, https://riot.ieor.berkeley.edu/Applications/Scheduling/index.html.

[2] “Maximum Bipartite Matching.” *GeeksforGeeks*, GeeksforGeeks, 20 Feb. 2023, https://www.geeksforgeeks.org/maximum-bipartite-matching/.

[3] Giovanni De Micheli. 1994. Synthesis and Optimization of Digital Circuits (1st. ed.). McGraw-Hill Higher Education.

[4] Karlowitsch, Elmar. “What Is Flow Shop Scheduling vs Job Shop Scheduling?” *Just Plan It - Production Scheduling Software*, NETRONIC Software GmbH, https://www.just-plan-it.com/smb-production-scheduling-blog/what-is-flow-shop-scheduling-vs-job-shop-scheduling.

[5] “Weighted Job Scheduling.” *GeeksforGeeks*, GeeksforGeeks, 23 Mar. 2022, https://www.geeksforgeeks.org/weighted-job-scheduling/.

[6] Radigan, Dan. “What Are Story Points and How Do You Estimate Them?” *Atlassian*, Atlassian, https://www.atlassian.com/agile/project-management/estimation.

[7] Ellis Horowitz and Sartaj Sahni. 1976. Exact and Approximate Algorithms for Scheduling Nonidentical Processors. J. ACM 23, 2 (April 1976), 317–327. https://doi.org/10.1145/321941.321951

[8] Gonzalez, Teofilo & Ibarra, Oscar & Sahni, Sartaj. (1977). Bounds for LPT Schedules on Uniform Processors. SIAM J. Comput.. 6. 155-166. 10.1137/0206013.

[9] Kalliamvakou, Eirini. “Research: Quantifying Github Copilot's Impact on Developer Productivity and Happiness.” *The GitHub Blog*, GitHub, 7 Sept. 2022, https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/.

[10] Richards, Toran Bruce. “Auto-GPT: An Autonomous GPT-4 Experiment.” GitHub, Mar. 2023, https://github.com/Significant-Gravitas/Auto-GPT. Accessed Apr. 2023.

[11] “11.4 Multiprocessor Scheduling.” *11.4 Multiprocessor Scheduling - MOSEK Fusion API for Python 10.0.40*, MOSEK, 2023, https://docs.mosek.com/latest/pythonfusion/case-studies-multi-proc-scheduling.html.

[12] Nash, Stephen G. “Nonlinear Programming.” *INFORMS*, June 1998, https://www.informs.org/ORMS-Today/OR-MS-Today-Software-Surveys/Nonlinear-Programming-Software-Survey/Nonlinear-Programming.

[13] Pablo E. Coll, Celso C. Ribeiro, Cid C. de Souza (2004). Multiprocessor scheduling under precedence constraints: Polyhedral results.

[14] Lively, Thomas & Long, William & Pagnoni, Artidoro. (2019). Analyzing Branch-and-Bound Algorithms for the Multiprocessor Scheduling Problem.