Assigning Suppliers to Meet a Deadline

Abstract

Most real-world project have a deadline and consist of completing tasks. In our setting, each task needs to be executed by a single supplier, chosen from a subset of suppliers that have the required proficiency to handle that task. The suppliers differ in their execution times, which are stochastically taken from known distributions. The Supplier Assignment for Meeting a Deadline (SAMD) problem is the problem of assigning a supplier to each task in a manner that maximizes the chance to meet the overall project deadline. We propose an A*-based approach, along with an efficient admissible heuristic function, that guarantees an optimal solution for this problem. Experimentally, compare our A*-based approach to an exhaustive brute-force approach and several heuristic methods. The results show that our A*-based approach compares favorably with the heuristic methods, and is orders of magnitude faster than the exhaustive alternative.

1   Introduction

A fundamental challenge in project management is to finish projects in a timely manner. A common representation of such problems is of a ighted digraph, termed the tasks graph, with vertices depicting the states of the project, and ighted edges expressing tasks and their respective completion times. The objective is usually of finding the shortest path to project completion.

When the execution times of tasks are stochastic, i.e. represented by random variables, the term "shortest" (optimal) becomes ambiguous, leading to several different notions of optimality (Loui 1983). The most trivial notion is that of minimal expected time, in which one can consider the expectations of the task execution times and simply solve the resulting deterministic problem using Dijkstra's algorithm (Dijkstra 1959). Hover, the identified path can be potentially "risky". For example, if the project has a deadline, such a path may have a high chance of not meeting the deadline, despite of its optimal expectation. An alternative optimality notion is to maximize of meeting the deadline (Frank 1969).

Unfortunately, only exhaustive methods are known to solve the resulting problem.

Orthogonal to the above classic line of research, recent studies from the Mechanism Design field consider settings in which there are several suppliers (agents) that can perform a task (Porter et al. 2008; Conitzer and Vidali 2014). These studies aim at solving mechanism design problems, in which the agents may lie about their true distributions. Nevertheless, their main motif of having multiple alternative suppliers for executing a task is very relevant in many real-world projects. Consequently, in contrast to the setting of Loui that assumes a general graph, it is interesting to examine projects that consist of a set of known tasks, but enable choosing the supplier that will perform each task.

In this work, introduce the SAMD problem. The problem setting includes a project that needs to be completed before some predefined deadline and a set of suppliers (agents) that can perform tasks within the project. The tasks that form the overall project are fully ordered, and the execution of a task cannot be initiated before the completion of its preceding task. Each task needs to be executed by a single supplier, chosen from a subset of suppliers that have the required proficiency to handle that task. The suppliers differ in their execution times, which are stochastically taken from known distributions. We apply the optimality notion of Frank, which fits problems with deadlines. Thus, the SAMD problem is of assigning a supplier to each task in a manner that maximizes the chance to meet the overall project deadline.

Computing of some assignment of suppliers to meet the project deadline is equivalent to an existing NP-hard problem (Cohen, Shimony, and Weiss 2015). Therefore, as a first step, devise two suboptimal heuristic approaches. Both these approaches create deterministic versions of SAMD, which are easily solvable, but naturally do not guarantee finding the optimal solution of the original stochastic SAMD problem. We then proceed to our main contribution in the form of a complete algorithm, which is based on the A* algorithm (Hart, Nilsson, and Raphael 1968). The SAMD problem has some unique properties that require making several interesting adjustments to the A* algorithm. As part of

the A* algorithm use an efficient domain-specific heuristic, which is composed of two main types of computations, both admissible.

We perform an extensive experimental evaluation that compares beten four alternatives – the two suboptimal heuristics, the A*-based approach, and a brute-force approach. We use in our experiments several types of distributions for the execution times of the tasks. The results show that, on the one hand, the A*-based approach finds better solutions than the heuristic methods in many instances, and on the other hand, it is orders of magnitude faster than the brute-force alternative.

2 Problem Definition

A project is defined by a sequence of tasks $T = \{t_1, \ldots t_N\}$. We assume that the tasks must be completed sequentially, i.e., if $i < j$ it means that t_i must be completed before starting t_j.

Every task t can be performed by a supplier. Let S be the set of suppliers, and let S_t be the subset of S that consists all suppliers that can perform the task t. For ease of presentation, assume that every supplier can perform exactly one task. The suppliers differ in their task execution times, represented by real-valued random variables with values in $[0, \infty)$ with known distributions. For A supplier s, denote by X_s its corresponding random variable. A supplier assignment is a function $\varphi : T \to S$ such that $\varphi(t) \in S_t$.[1] For a given project T, supplier assignment function φ, and a project deadline $d > 0$, denote by $M(T, \varphi, d)$ that all tasks in T will be completed in their respective order by the suppliers assigned to do them according to φ before time d. Formally,

$$M(T, \varphi, d) = Pr\left(\sum_{t \in T} X_{\varphi(t)} \le d\right) \qquad (1)$$

Hence, in accordance with the optimality notion of Frank, the problem at focus is defined as follows.

Definition 1 (The SAMD problem). An SAMD problem Π is defined by a tuple $\langle T, S, X, d \rangle$, where T is an ordered set of tasks, S is a set of suppliers, X is the set of random variables, one for each supplier, that represents the task completion time of this suppliers, and d is the project deadline. A solution to an SAMD problem is a supplier assignment φ for T and S. An optimal solution to an SAMD problem is a supplier assignment φ^* that maximizes $M(T, \varphi^*, d)$, that is, for every other solution φ it holds that $M(T, \varphi^*, d) \ge M(T, \varphi, d)$.

Example 1. As an example of an SAMD problem Π, consider a project T with tasks $T = \{T_1, T_2\}$ and suppliers $S = \{s_1, s_2, s_3, s_4\}$. Suppliers s_1 and s_2 can perform task T_1 and suppliers s_3 and s_4 can perform task

---

[1]The observant reader will notice that slightly abuse the mathematical notation, since T is a sequence and not a set. Hover, believe this is clear from the context.

T_2. The corresponding completion-time distributions of the tasks are as follows:

$$Pr(X_{s_1} = v) = Pr(X_{s_3} = v) = \begin{cases} 1/4 & v = 1 \\ 3/4 & v = 2 \end{cases}$$

$$Pr(X_{s_2} = v) = Pr(X_{s_4} = v) = \begin{cases} 3/4 & v = 1 \\ 1/4 & v = 3 \end{cases}$$

Let $\varphi_{i,j}(t)$ denote the supplier assignment function defined as follows:

$$\varphi_{i,j}(t) = \begin{cases} s_i & t = T_1 \\ s_j & t = T_2 \end{cases}$$

If the deadline is 2, then the optimal solution to problem Π is $\varphi_{2,4}$ since $M(T, \varphi_{2,4}, 2) = 9/16$. If the deadline is 3, then the optimal solution to problem Π is either $\varphi_{1,4}$ or $\varphi_{2,3}$, both of which meet the deadline with probability $3/4$. If the deadline is 4, then the optimal solution to problem Π is $\varphi_{1,3}$ since $M(T, \varphi_{1,3}, 4) = 1$.

Corollary 1. The computation of $M(T, \varphi, d)$ is NP-hard.

The above corollary refers to the complexity of assessing a single supplier assignment. We expect that in order to find the supplier assignment that optimizes to meet the deadline, i.e., to find an optimal SAMD solution, one will need to assess a set of possible supplier assignments and compute their M values. Thus, expect that the SAMD problem will be at least as hard as the the computation of an M value, i.e., at least NP-hard.

Having said that, computing M is still feasible in small instances and mainly relies on the support sizes of the completion-time distributions.[2] The support size grows exponentially when summing a series of random variables (Cohen, Shimony, and Weiss 2015). Thus, when the number of tasks is small and when the completion-time distributions have limited support size, computing M is feasible.

3 Heuristic Suboptimal Approaches

Following our conjecture about the hardness of SAMD, propose two heuristic approaches for finding effective supplier assignments. Both approaches rely on identifying a special case of SAMD that is easy to solve optimally, and on showing how to map an arbitrary SAMD problem to an easy-to-solve SAMD problem.

Definition 2 (Deterministic SAMD). An SAMD problem $\langle T, S, X, d \rangle$ is deterministic iff for every supplier $s \in S$ there is a value v_s such that $Pr(X_s = v_s) = 1$.

In words, a deterministic SAMD problem is an SAMD problem in which the task completion time of every supplier is deterministic and known a-priori. For convenience, define the set of distributions X for a deterministic SAMD problem as simply a set of values, one for each supplier, specifying its task completion time.

---

[2]The support size is the number of values of the random variable for which is non-zero.

To find the optimal solution of a deterministic SAMD problem, all that is needed is to locate for each task the supplier with the minimal completion time for that task. This is because tasks are executed sequentially, and thus choosing a supplier to perform one task does not prevent it from performing a different task later. Next, present two SAMD algorithms based on solving deterministic SAMD problems.

3.1 Sampling-based Approach

The first heuristic approach propose for solving SAMD samples possible task completion times, generates corresponding "optimal" supplier assignments, and chooses the best resulting supplier assignment. We refer to this algorithm as Sampling.

Sampling is an iterative algorithm. In every iteration, sample for every supplier $s \in S$ a possible task completion time x_s according to its distribution X_s. Then, create a deterministic SAMD problem Π_D that has the same set of tasks, suppliers, and deadline as the original problem, and has the deterministic task completion times sampled for every supplier. Next, use the simple algorithm described above to find an optimal solution φ for this deterministic SAMD problem. After that, compute that this supplier assignment φ will meet the deadline in the original problem, i.e., compute $M(T, \varphi, d)$. If φ yielded an M value that is higher than the M value of all previously found supplier assignments, then set φ as the best supplier assignment found so far.

Sampling accepts a parameter K that indicates the number of iterations that the algorithm will run. Running more iterations of Sampling can never result in returning a worse solution, and it can result finding better solutions. Thus, Sampling is an anytime algorithm, i.e., an algorithm "whose quality of results improves gradually as computation time increases" (Zilberstein 1996). This is a desirable property for an algorithm, as it gives more control over its execution to its user. Sampling is also easy to implement and it has virtually constant space requirements. The dominant factor in the runtime of Sampling is the computation of the M value for every generated solution φ. This is done exactly once in each of the K iterations, and thus the runtime of Sampling is K times the runtime of computing the M value.

3.2 Minimize Expected Completion Time

The second heuristic approach propose for solving SAMD is similar to Sampling, except that it considers the expected task completion times instead of performing multiple iterations and sampling the task completion time in every iteration. Like Sampling, Expectation also has minimal space requirement. In terms of runtime, Expectation is much faster, as it has only a single iteration and it does not need to compute M for any supplier assignment. Hover, Expectation does not consider the given deadline d, and can consequently return bad solutions. Loui refers to such a phenomenon as not taking account of "risk" (of not meeting the deadline).
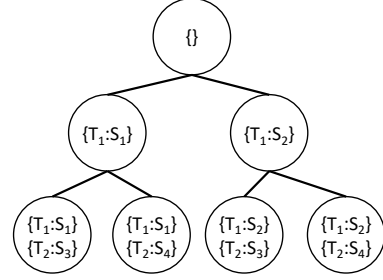

Figure 1: An example of part of the SAMD search space.

Consider the example problem Π from Section 2. The expected completion time of suppliers s_1 and s_3 is 1.75, whereas for suppliers s_2 and s_4 it is 1.5. So Expectation will choose the supplier assignment $\varphi_{2,4}$ regardless of the deadline d. For a deadline 2, this is indeed the optimal solution. Hover, it is not optimal for deadlines 3 or 4. In general, both Sampling and Expectation do not guarantee that the solution they return is optimal. Next, propose a heuristic search algorithm that guarantees optimality.

4 Finding Optimal Solutions

A brute-force solution to find an optimal solution to an SAMD problem is to enumerate all possible supplier assignments, compute the value of $M(\Pi, \varphi, d)$ for each assignment φ and return the assignment that yields the maximal value. Observe that the number of possible supplier assignments is exponential in the number of tasks ($|T|$), and for each assignment need to compute $M(\Pi, \varphi, d)$, which, according to Corollary 1, is a computationally hard problem on its own.

Next, propose an algorithm that is guaranteed to find an optimal solution to any SAMD problem and is significantly more efficient than the aforementioned brute-force approach. This algorithm is based on the the ll-known A* algorithm (Hart, Nilsson, and Raphael 1968). As a preliminary, formalize the SAMD problem as a graph search problem.

4.1 SAMD as a Graph Search Problem

A graph search problem is defined by a graph (V, E), a node $s \in V$, and a set of nodes $G \subseteq V$, where a solution is a path in the graph from s to a node in G. The graph can be explicitly given as input, or implicitly given by defining an initial node and a set of transition functions that map a node to its neighbors in the graph.

SAMD can be represented as a search problem in the following graph. A node in the graph represents a partial supplier assignment function, which is function that assigns suppliers to a subset of the tasks in T. The initial node s is the empty supplier assignment function, that is, a partial supplier assignment function that does not assign any supplier to any tasks. Every child node of s represents a possible assignment of a supplier to T_1. Every child node of these nodes represents a possible

assignment of a supplier to T_2, and so forth. Note that this search space is a tree. Figure 1 illustrates two levels of this tree. The depth of this tree is the number of tasks N, and every leaf node represents a complete supplier assignment.

4.2 A* for SAMD

Algorithm 1: Psuedo-code of A* for SAMD.

Input: $\Pi = \langle T, S, X, d \rangle$, an SAMD problem
Output: φ, an optimal supplier assignment

1   $M_{best} \leftarrow 0; \varphi_{best} \leftarrow \text{null}; \varphi_{init} = \{\}; \text{Open} \leftarrow \emptyset$
2   $n_{init}.\text{task} \leftarrow 0; n_{init}.\varphi \leftarrow \varphi_{init}$
3   Add n_{init} to Open with key $U(n_{init})$
4   **while** Open is not empty **do**
5     $n_{best} \leftarrow$ pop from Open a node with maximal $U(n)$
6   **if** $U(n_{best}) \leq M_{best}$ **then**
7   ⌊ return φ_{best}
8     $i \leftarrow n_{best}.\text{task}$
9     $\varphi \leftarrow n_{best}.\varphi$
10   **foreach** supplier s for task T_{i+1} **do**
11       $n_{child}.\text{task} \leftarrow i + 1$
12       $n_{child}.\varphi \leftarrow \varphi \cup (T_i \rightarrow s)$
13   **if** $n_{child}.\varphi$ is a complete assignment **then**
14         $M \leftarrow M(n_{child})$
15   **if** $M \geq M_{best}$ **then**
16           $M_{best} \leftarrow M$
17   ⌊ $\varphi_{best} \leftarrow n_{child}.\varphi$
18   **else**
19   ⌊ Add n_{child} to Open with key $U(n_{child})$

20 return φ_{best}

To search this search space for the supplier assignment φ that maximizes $M(T, \varphi, d)$, use the A* algorithm (Hart, Nilsson, and Raphael 1968) with some adjustments. A* is a best-first search algorithm that maintains two lists of nodes: an open list (denoted Open) that contains all nodes that re generated and not expanded yet, and a closed list (denoted Closed) that contains all nodes that re already expanded. Initially, Open contains the initial node (in our case, the node representing an empty supplier assignment). Then, in every iteration one node is moved from Open to Closed, and all its children nodes are added to Open. The search halts when a goal node is popped from Open.

A* considers two values for every node n: the cost of the best-known path from the initial node to n, and a heuristic estimate of the cost of the optimal path from a given node to a goal. The former is denoted by $g(n)$ and the later by $h(n)$. A* pops from Open in every iteration the node with the smallest $g(n) + h(n)$. A heuristic h is called admissible if its estimate is never larger than the real cost of the optimal path from n to the goal. Given an admissible heuristic, A* is guaranteed to have found the optimal solution when a goal node is expanded. Moreover, under certain conditions, all other algorithms must expand at least

the same set of nodes as A* (Dechter and Pearl 1985; Holte and Zilles 2019).

A* is commonly used for minimization problems, but (Stern et al. 2014) pointed out how to adapt it to solve maximization problems. Two changes are required: (1) for a heuristic function to be admissible it must return an estimate that is never smaller than the real cost to reach a goal, and (2) the search can only halt after the cost of the best solution found so far is smaller than or equal to $g(n) + h(n)$ for all n in Open.

To apply A* as is to solve SAMD, need to define $g(n)$ and $h(n)$. Hover, the notion of paths and costs of paths is not natural in our search space, since can only compute the actual value – $M(T, \varphi, d)$ – of a complete supplier assignment. Instead of defining $g(n)$ and $h(n)$, define for every node n a single value, denoted $U(n)$, that is an upper bound over the cost of all goal nodes in the subtree of the search space rooted by n. Then, in every iteration of our modified A*, pop from Open the node with the highest $U(n)$ value and insert its children to Open. A leaf node represents a complete supplier assignment. So, for every generated leaf node n compute its corresponding M value, and store the leaf with the highest M seen so far. We refer to this leaf node as n_{best} and its M value as M_{best}. The search halts when either Open is empty or the expanded node n is such that $U(n) \leq M_{best}$. When the former condition occurs, know that all supplier assignments have been checked, in which case n_{best} is indeed optimal. If the latter condition occurs, know that for all nodes $n \in$ Open it holds that $U(n) \leq M_{best}$, and thus n_{best} is optimal. Consequently, given that an admissible $U(\cdot)$ is implemented, completeness and optimality are guaranteed (Stern et al. 2014; Holte and Zilles 2019).

rithm 1 provides a complete pseudo-code for our A*-based algorithm. For convenience, for a node n denote by $n.\varphi$ and $n.\text{task}$ the partial assignment that n represents and the index of the last task assigned by $n.\varphi$, respectively. Hence, for the initial node n_{init} have that $n_{init}.\varphi$ is an empty supplier assignment and $n_{init}.\text{task}$ is zero (line 2 in rithm 1). Note that due to the tree structure of the search graph, there is no need to maintain Closed.

4.3 Computing an Upper Bound

Our A* can be used with any implementation of $U(\cdot)$ as long as it is admissible for maximization problems, i.e., returns a valid upper bound over all the goals in its sub-tree. Next, propose a concrete admissible $U(\cdot)$ that works ll in practice.

Let n be the node wish to compute $U(n)$ for, and assume that n represents a partial assignment φ_n that assigns a supplier to tasks $t_1, \ldots t_i$ for some $i < N$. We compute $U(n)$ by assuming that in every unassigned task $t \in \{t_{i+1}, \ldots, t_N\}$ all suppliers that can perform this task do so simultaneously, so that the task is completed when the fastest one finished. Formally, for the computation of $U(n)$, assume that the completion time

of every unassigned task t to be $\min_{s \in S_t} X_s$. This is a random variable that can be computed easily given all the X_s random variables. Since in our setting only a single supplier performs each task, the real completion time of any single supplier must be larger than this value (or at least equal). Thus, it can be used as an admissible heuristic.

To summarize, define $U(n)$ as follows:

$$U(n) = \sum_{t \in \{t_1, \ldots t_i\}} X_{\varphi_n(t)} + \sum_{t \in \{t_{i+1}, \ldots t_N\}} \min_{s \in S_t} X_s \quad (2)$$

While the above $U(\cdot)$ is clearly admissible, its computational effort is equivalent to that of the $M(T, \varphi, d)$ value of some complete supplier assignment φ, which is NP-hard (Corollary 1). For small problems it may be feasible to apply such a computation for a limited number of times, as is done in Sampling. Hover, $U(n)$ needs to be computed frequently for every node n that is added to Open, which renders its exact computation inapplicable. Hence, turn to an approximate computation of $U(\cdot)$.

Recall that the problem with computing the M value lies in the exponential growth of the support size (Cohen, Shimony, and Weiss 2015). Consequently, after every addend in the computation of $U(\cdot)$ apply the OptApprox (X, m) operator (Cohen, Grinshpoun, and Weiss 2018), which for a given random variable X and a requested support size m returns a new random variable X' with a support size of at most m in polynomial time. By restricting the support size to a given m after every addend, prevent the exponential growth of the support size.

The resulting X' of OptApprox (X, m) is actually the best approximation (optimal) of X given the requested support size m according to the "one-sided" version of the Kolmogorov distance. The "one-sided" approximation means that the cumulative distribution function (CDF) $F_{X'}$ of X' is greater or equal to the CDF F_X of X, i.e., for every value v of X, $F_X(v) \leq F_{X'}(v)$. This, in turn, results in a pessimistic computation of to meet the deadline. Consequently, use an inverse version of OptApprox that gives $F_X(v) \geq F_{X'}(v)$ for all v, i.e., an optimistic computation of to meet the deadline. We term the computation of $U(\cdot)$ with the inverse version of OptApprox after every addend as $U_{\text{OptApprox}}(\cdot)$.

Lemma 1. $U_{\text{OptApprox}}(\cdot)$ is admissible.

Proof. $U(\cdot)$ is trivially admissible, since it assumes that all suppliers that can perform a future task will do so simultaneously, which is obviously not worse than any single supplier performing the task. Also, the inverse version of OptApprox that is applied in $U_{\text{OptApprox}}(\cdot)$ is optimistic, hence also admissible. $\square$

Corollary 2. A* for SAMD is complete and optimal.

Note that in order to maintain the optimality of the algorithm cannot use OptApprox in the computation of M for complete supplier assignments (line 14 in

rithm 1). Theoretically, this computation is performed an exponential number of times. Nonetheless, in practice our A* approach manages to refrain from expanding most of the leaves (complete assignments), as is evident from the run-time results of our experimental evaluation (Section 5).

5 Experimental Evaluation

In what follows experimentally evaluate the performance of four algorithms – two sub-optimal algorithms, Sampling with 100 samples (Subsection 3.1) and Expectation (Subsection 3.2), and two optimal algorithms, the A*-based algorithm (Subsection 4.2) and the brute-force approach that was presented at the beginning of Section 4. We compare beten the four algorithms in terms of run-time performance and solution quality. For evaluating the solution quality of the sub-optimal methods use two measures – the average error, i.e., the difference from the optimal probability to meet a deadline, and the percentage of instances in which the optimal solution is found. Obviously, the optimal approaches have no error and always find the optimal solution.

We consider various experimental setups of both synthetically-generated completion-time distributions and distributions extracted from real software projects. The problem instances are in the form of a graph with different number of layers (tasks) and different number of nodes in each layer (suppliers). We also compare three deadline types that are derived from $maxd$, where $maxd$ is the maximal possible completion time of a project given the maximal values of the distributions of the suppliers (excluding the value of failure, Subsection 5.1) and the number of tasks. The deadline types are early ($0.25 \cdot maxd$), expected ($0.5 \cdot maxd$), and late ($0.75 \cdot maxd$). We use various deadlines in order to examine their effect on the different algorithms, especially on Expectation. Intuitively, presume that Expectation will perform ll for expected deadlines.

5.1 Synthetic Distributions

We consider three types of synthetically generated distributions – random, structural, and structural with failure. We use a support size of 4 in all the distributions. In the random distribution generate possible completion times for each supplier in the range [0...4] with uniform distribution. The distribution is also randomly generated and then normalized to sum up to one. An example of a random distribution may look like [0.58:0.23, 3.24:0.32, 3.87:0.17, 3.44:0.28]. In the structural distribution uniformly generate a single possible completion time beten every two consecutive integers, and the distribution is generated as in the random distribution. An example of a structural distribution may look like [0.58:0.15, 1.24:0.59, 2.87:0.01, 3.44:0.25]. In the structural with failure distribution generate the possible completion times as in the structural distribution, except for the last value that is always set to 10^6. This large value represents in practice a situation

in which the supplier can never complete the task, or in other words – fail. Indeed, in real-world situations, some suppliers may fail at performing a task. An example of structural with failure distribution (or failure for short) may look like [0.58:0.14, 1.24:0.41, 2.87:0.33, 10^6:0.12]. Note that in our experiments do not consider normal distributions, since such problems are much easily solvable, as the sum of normal distribution can be computed in polynomial time (Schattenberg 2005; Glaubius et al. 2009; Olya and others 2014).

We vary the number of tasks (layers) in a project. Due to the obvious limitations of the exponential search space, used a 10 minute time-out for each instance, which was sufficient to run up to 8 layers with the brute-force approach. For every setting composed of the number of layers, the distribution type, and the deadline duration, run 30 different problem instances in order to gather significant statistics. For every task consider two possible suppliers, each with a different distribution. In addition, use a support size of 4, since the brute-force approach failed to solve problems with larger support size due to lack of memory.

5.2 Distributions from Software Development

The next type of distributions use is aimed to simulate a SAMD problem of a software project, in which the suppliers are software developers and the tasks are user stories that need to be implemented by a developer. A user story in software development is "a small piece of functionality of the final software perceived from the end-user point of view" (Rees 2002), and is one of the most common ways to specify requirements (Kassab 2015). We extracted information about implemented user stories from three open-source projects: (1) JBoss Developer,[3] (2) Lsst DM,[4] and (3) Spring XD.[5] Specifically, extracted the list of user stories that re assigned to a developer and marked as done. For each user story, extracted the developer assigned to implement it and the number of sprints it took to implement this user story. A sprint corresponds to a fixed time frame, which is either two or three eks in most projects. Using this data, created for every developer a distribution of the number of sprints that developer needs to implement a user story. In total obtained the distributions of user story completion time for 39 developers. Here is examples for a developers distributions [1:0.95, 2:0.045, 3:0.005] or [1:0.62, 2:0.3, 3:0.07, 4:0.01]. We note that this information is publicly available from JIRA, a popular issue tracking system used by all our projects to manage the development. See the footnotes above for links to this publicly available data.

With these distributions, generate three types of SAMD problems as follows. The first has four tasks (Developer, DM1, DM2, XD), with each task having all the possible developers from that task's developer pool. This project is unique, and examine to meed the deadline for different deadlines, up to 20 sprints. The second type is of randomly generated projects with 4 tasks (Developer, DM1, DM2, XD), where for each task randomly select (without repetitions) developers from the relevant developer pool. We examine how the different number of suppliers in each task (up to 7 suppliers) affects the run-time and solution quality. In the third type the number of developers is set to 4 for every task, i.e., randomly select (without repetitions) 4 developers from the relevant developer pool. The parameter vary here is the number of layers, up to 7 layers. We randomly generate 30 instances for each of the latter two problem types.

5.3 Results

We start with presenting the results for the experiments with synthetic distributions. Note that since the A* algorithm and the brute-force approach are both optimal, their error is always zero and thus do not present their error results.

Figure 2 compares the run-times of the four algorithms. In all conducted experiments (in all of the settings) the run-time of the A* algorithm outperforms that of the brute-force algorithm. In addition, one can also see that the run-time of A* is smaller than the run-time of Sampling, which is not optimal in most cases. In all experiments, the run-time of the Expectation heuristic algorithm is considerably shorter than all other approaches, hover its solution quality is low and in some cases even returns an error of 1. In other words, the A* algorithm guarantees optimality in reasonable run-time. Expectation and Sampling heuristic algorithms result with an error and do not return the optimal assignment. For instance, the error comparison results for 6 layers (of the experiment in Figure 2) shows that an error of $1.6 \cdot 10^{-5}$ for Expectation and 0.006 for Sampling. For 7 layers the respective errors are $7 \cdot 10^{-5}$ and 0.004, and for 8 layers they are $4 \cdot 10^{-5}$ and 0.003.
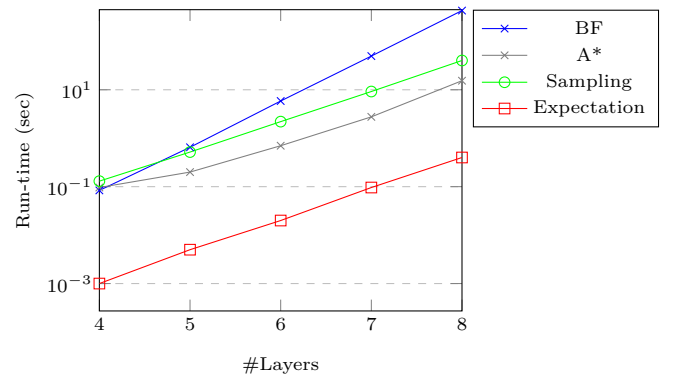

Figure 2: Run-time comparison of all algorithms for deadline of size $0.25 \cdot maxd$ and random distribution

It is interesting to illuminate the fact that the Expectation heuristic can not handle correctly some types

[3]https://issues.jboss.org
[4]https://jira.lsstcorp.org
[5]https://jira.spring.io

of distributions.

Example 2. Let us observe the following Failure distribution $[1{:}0.99, 10^6{:}0.01]$ and the following uniform distribution $[100{:}0.5, 200{:}0.5]$. For every task, assume two suppliers, one supplier has the uniform duration distribution (call it u), and the other supplier has the Failure duration distribution (call it f). For a project with two tasks and a deadline of 10 seconds, the Expectation heuristic algorithm will choose for both tasks supplier u because its expectation is smaller. Eventually the Expectation heuristic algorithm returns zero chance to meet the deadline where in fact there is a chance of almost 100% to succeed.

The errors get when using the Expectation heuristic algorithm in our experiments with Failure distribution are 0.51-0.7, depending on the number of layers; note that such errors are high (the range is 0-1). We witness such high errors because the Expectation heuristic algorithm returns that there is no chance to meet the deadline where in fact there is. The other algorithms executed in a setting with this Failure distribution resulted with an error equal to 0. In addition, for all other approaches got 97%-100% correct optimal assignment for every number of layers, but using the Expectation heuristic algorithm did not get any right assignment for any number of layers.

Another concern regarding the Expectation heuristic is the chosen deadline. As long as the deadline is around the average duration, it is more likely that the Expectation heuristic algorithm will return solid results. Hover, the deadline is not strict, and in the general case, where the deadline can be very small or very large compared to the expectation, this heuristic may produce substantially sub-optimal results. For a deadline such as $0.5 \cdot maxd$ can observe good results produced by the Expectation heuristic algorithm due to the comfortable deadline. For 4, 5, 6, 7, 8 layers the results are 93%, 80%, 70%, 90%, 90% optimal assignments, respectively.

As the number of layers grows, the Sampling heuristic is less accurate. Even though, the error is somewhat small, the number of optimal assignments is reducing as a function of the number of layers, see Table 1. Since the number of possible assignments is growing (exponentially) as the number of layers grows, the Sampling heuristic needs more samples in order to return good solution, which increases run-time. Here used 100 samples for each experiment.

The run-time plots are mostly similar, and as described before, the A* heuristic algorithm presents the second best performance in aspect of run-time, see Figure 2. The best run-time is presented by Expectation heuristic algorithm, hover, the solution quality of this method is poor.

Now report the results of our experiments on the software development distribution types described earlier. Figure 3 shows how the deadline (x-axis) effects the error (y-axis). As expected, for later deadlines the error is smaller, since many different assignments will meet the

Deadl.		#Layers				
		4	5	6	7	8
0.75	A*/BF	100%	100%	100%	100%	100%
	Sampl.	93%	97%	87%	70%	47%
	Exp.	57%	57%	50%	30%	17%
0.25	A*/BF	100	100%	100%	100%	100%
	Sampl.	97%	100%	83%	77%	57%
	Exp.	33%	30%	27%	33%	10%

Table 1: Number of optimal assignments comparison (in percentage out of 30) of all algorithms for deadline of size $0.75 \cdot maxd$ and $0.25 \cdot maxd$ with structural distribution

deadline with certainty. Also, for early deadlines where there in no chance to meet the deadline (omitted from the figure). For different deadlines, even on the same instance, the error may change dramatically. Contrary to that, the A* algorithm is not sensitive for such deadline changes and always returns the optimal result.
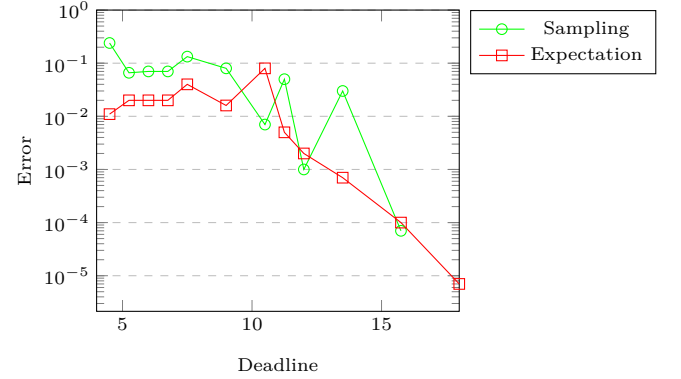

Figure 3: Error comparison of all algorithms for different deadlines and

The next results present are from the setting where the number of tasks is constant (4 tasks) and the number of possible suppliers to execute the task is changed. Here can see how the error grows when add more suppliers, mostly in the case of the Sampling heuristic algorithm. Similarly, the number of optimal assignments decreases as add more optional suppliers, see Table 2. In this setting increase the search space when add more options for suppliers. The first heuristic to be affected by it is the Sampling heuristic algorithm. As the search space grows more samples are needed in order to maintain accuracy which also affects run time. The run-time results for this setting continue the same trend as before, where the brute force algorithm takes the longest run-time and the Expectation heuristic algorithm takes the shortest time (with an error). In the middle of the run-time range are Sampling and A*, where A* is slightly better, taking shorter time.

Finally, vary the number of layers (tasks). The error results are 0.04-0.06 for Sampling and 0.007-0.01 for Expectation. These algorithms detect only a few optimal assignments, Sampling less than 7% and Expectation

		#Suppliers				
		3	4	5	6	7
Error	A*/BF	0	0	0	0	0
	Sampl.	0.009	0.05	0.09	0.08	0.09
	Exp.	0.009	0.008	0.008	0.008	0.01
#Opt	A*/BF	100%	100%	100%	100%	100%
	Sampl.	40%	3%	0%	0%	0%
	Exp.	30%	30%	23%	20%	13%

Table 2: Error comparison and number of optimal assignments (in percentage out of 30) of all algorithms for different number of suppliers in each layer, deadline of 4 seconds and

between 13%-30%, which is even less than what we

6   Conclusion and Future Work

We introduced the SAMD problem of assigning suppli posed. Two suboptimal algorithms, Sampling and Expectation, and an optimal A*-based algorithm. Exper trends for our A*-based approach that on the one -time. We believe that the presented SAMD problem is to see how SAMD can be extended to allow in many real-world projects. For future work, one may consider using approximations of computations of $M(\cdot)$. but will enable using the (now suboptimal) A*-based

References

[Cohen, Grinshpoun, and Weiss 2018] Cohen, L.; Grinshpoun, T.; and Weiss, G. 2018. Optimal approximation of random variables for estimating the probability of meeting a plan deadline. In Thirty-Second AAAI Conference on Artificial Intelligence.

[Cohen, Shimony, and Weiss 2015] Cohen, L.; Shimony, S. E.; and Weiss, G. 2015. Estimating the probability of meeting a deadline in hierarchical plans. In Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI).

[Conitzer and Vidali 2014] Conitzer, V., and Vidali, A. 2014. Mechanism design for scheduling with uncertain execution time. In Twenty-Eighth AAAI Conference on Artificial Intelligence.

[Dechter and Pearl 1985] Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of a. Journal of the ACM 32(3):505–536.

[Dijkstra 1959] Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. Numerische Mathematik 1(1):269–271.

[Frank 1969] Frank, H. 1969. Shortest paths in probabilistic graphs. Operations Research 17(4):583–599.

[Glaubius et al. 2009] Glaubius, R.; Tidwell, T.; Gill, C.; and Smart, W. D. 2009. Scheduling design with unknown execution time distributions or modes.

[Hart, Nilsson, and Raphael 1968] Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics 4(2):100–107.

[Holte and Zilles 2019] Holte, R., and Zilles, S. 2019. On the optimal efficiency of cost-algebraic A*. In Thirty-Third AAAI Conference on Artificial Intelligence.

[Kassab 2015] Kassab, M. 2015. The changing landscape of requirements engineering practices over the past decade. In IEEE international workshop on empirical requirements engineering (EmpiRE), 1–8.

[Loui 1983] Loui, R. P. 1983. Optimal paths in graphs with stochastic or multidimensional weights. Communications of the ACM 26(9):670–676.

[Olya and others 2014] Olya, M. H., et al. 2014. Applying dijkstra's algorithm for general shortest path problem with normal probability distribution arc length. International Journal of Operational Research 21(2):143–154.

[Porter et al. 2008] Porter, R.; Ronen, A.; Shoham, Y.; and Tennenholtz, M. 2008. Fault tolerant mechanism design. Artificial Intelligence 172(15):1783–1799.

[Rees 2002] Rees, M. J. 2002. A feasible user story tool for agile software development? In Asia-Pacific Software Engineering Conference, 22–30. IEEE.

[Schattenberg 2005] Schattenberg, B. 2005. Project planning under temporal uncertainty. Planning, Scheduling and Constraint Satisfaction: From Theory to Practice 117:189.

[Stern et al. 2014] Stern, R.; Kiesel, S.; Puzis, R.; Felner, A.; and Ruml, W. 2014. Max is more than min: Solving maximization problems with heuristic search. In Symposium on Combinatorial Search (SoCS).

[Zilberstein 1996] Zilberstein, S. 1996. Using anytime algorithms in intelligent systems. AI magazine 17(3):73.