Efficient Algorithms for Robust Procurements

Shawn Levie

January 8, 2013

Studentnr. 344412

E-mail: shawnlevie@gmail.com

Supervisors: Dr. A. F. Gabor and Dr. Y. Zhang

Erasmus School of Economics

Abstract

Starting with a project and several agents willing to execute it, the goal is to find a sequence of agents which can work on the project if the previous agent fails. Each sequence has a total cost and probability of finishing within the deadline. Two objectives for optimization are introduced: reliability and cost. We model a bi-objective optimization problem for solving these procurements efficiently. First, we propose an alternative for the traditional weighted sum method to find a Pareto front. Furthermore, an adapted branch and bound algorithm is proposed to compete with the adaptive weighted sum method in terms of quality and computation time. In the experiments we present the running times on which the number of agents has huge influence. The discussion handles the shape of the Pareto front and discusses its dependence on the parameters. In terms of both cost and time efficiency, our algorithm offers a useful tool for solving procurement problems.

1 Introduction

If a project to allocate services or goods is started, there can be several parties willing to execute the job. An auction is a frequently used method to assign a project to an agent, which will be the winning company to execute the job. Auctions give transparency and provide the auctioneer with the best possible offers. The goal of this paper is to provide a tool for the auctioneer that will help finishing the project within time against minimum cost.

The auctioneer imposes a deadline D before which the agents should finish the project. Each agent participating in the auction provides information on the cost of executing a task (c_i), the deadline within the task can be executed (d_i) and the backup cost for standby(b_i) while reliability (r_i) is available to the auctioneer based on historical information from previous contracts. Reliability is the probability that an agent will finish before the deadline d_i. The main issue in procurement, which happens way too often, is that an agent might fail to finish within the deadline d_i. Then, usually a new auction is started to determine the next possible agent to execute the task. These practices are very time consuming and do not contribute to the efficiency of our economy.

In this paper we aim to develop a method to determine a sequence of agents in advance where the next agent can takeover if one might fail. Such a sequence of winners provides a high reliability of finishing within time (D) although, since backup fees are payed, a higher total cost is incurred compared to hiring one agent. When two objectives are optimized simultaneously, cost and reliability in this case, there might not be one single optimal solution to both objectives. The goal is to find a number of solutions which give a trade-off between total cost and reliability.

In Section 2 we give a brief introduction of the variables used and the goal of our research. Section 3 discusses the previous research in this field and bring other methods to our attention. Then we present a new method called the adaptive weighted sum method in Section 4. The main technique of this method

is based on varying the weights of the objectives to obtain different optimal solutions. The actual algorithms are carefully explained in Section 5 and 6. Both the qualitative and quantitative results appear in Section 7 and Section 8, after which we conclude with our final remarks.

Own contribution The field of multi-objective optimization is well explored, however, it is not very often applied to procurement problems. Compared to existing approaches, where mostly single-objective optimization is performed, we want to include multiple objectives in the problem. Our aim is to develop one or more methods to solve a bi-objective procurement problem. We lay emphasis on the understandability of the methods so that they can be easily used or explored for further research.

2 Detailed problem description

When procurement of a new project occurs, several agents apply to complete the task. All agents provide the auctioneer with parameters such as deadline, reliability and cost for the specified project. The probability of finishing in time should be verifiable by the auctioneer by checking historical data on the agent. The auctioneer can pick one agent of choice, however, this might be difficult due to large variation of the parameters. It is even more difficult to determine a sequence of agents in which the following agent in line can take over if the previous agent fails. Our problem of finding an optimal sequences of agents has the following parameters.

The parameters of this problem are cost (c_i), deadline of agent (d_i), backup cost (b_i), reliability (r_i) and project deadline (D). The execution cost c_i of an agent is incurred when the agent starts working on his assignment. Since it might happen that this agent fails to finish within his proposed deadline d_i, another agent is put standby which yields him the backup cost b_i. Based on historical data, one can assign a probability r_i of finishing within time d_i to each agent.

Two objectives are present, J_1 which correspond to one minus the reliability of a sequence and J_2 which is the total cost of a sequence and . The aim is to use bi-objective optimization methods to minimize (J_1, J_2), this means finding a sequence of agents that maximizes the total reliability while the total cost is minimized. The constraint on the optimization is that the optimal sequences can finish the project within the deadline D.

The set $X \subset \mathbb{R}^n$ of all feasible solutions is called the decision space. Since there are two objective functions to be minimized, J_1 and J_2, the objective space is two-dimensional. This is mathematically represented as $J_i : \mathbb{R}^n \to \mathbb{R}^2 (i = 1, 2)$. A graphical representation of the relation between decision and objective space is given in Figure 1.

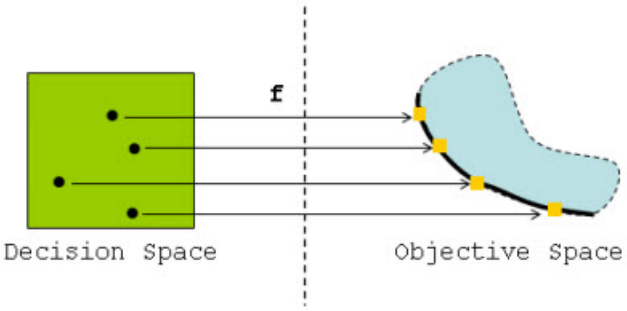


Figure 1: Relation between decision and objective space

The goal is to find a sequence of agents for which cost is minimized and reliability is maximized. When optimizing two objectives, an optimal solution

for both at the same time is mostly not feasible. What happens is that several optimal solutions can be found, each having a trade-off between cost and reliability. This is depicted by the dots in the objective space in Figure 1, each representing an optimal solution to the bi-objective optimization. In Section 3 we will consider several possible methods for this optimization.

3 Literature review on bi-objective optimization

The area of multiobjective optimization, well explored by Ehrgott (2008), has grown a lot over the past decades. The goal in multiobjective optimization is to find the so-called Pareto Front, defined below, consisting of non-dominated solutions. A solution is dominated if better values for at least one objective can be found. For a non-dominated solution it is not possible to improve one objective without worsening another. A solution strictly dominates another solution if all objective values are better, this is written as $y \prec y'$ which means that $y_1 \leq y'_1$ and $y_1 \leq y'_2$ where at least one inequality is strict. All non-dominated solutions constitute the Pareto Front which is formally defined below.

Definition 1 (Pareto optimality from Marco Laumanns and Zitzler (2005)**).** *Let $f : X \to F$ where X is called decision space and $F \subseteq R^m$ objective space. The elements of X are called decision vectors and the elements of F objective vectors. A decision vector $\mathbf{x}* \in X$ is Pareto optimal if there is no other $\mathbf{x} \in X$ that dominates $\mathbf{x}*$. For $\mathbf{x}$ to dominate $\mathbf{x}*$, denoted as $\mathbf{x} \succ \mathbf{x}*$, it must be true that $f_i(\mathbf{x}) \leq f_i(\mathbf{x}*)$ for all $i = 1, \ldots, m$ and $f_i(\mathbf{x}) < f_i(\mathbf{x}*)$ for at leaste one index i. Note that the inequalities are inverted if a maximization instead of minimization problem is present. The set of all Pareto-optimal decision vectors $X*$ is called the Pareto set. $F* = \mathbf{f}(X*)$ is the set of all Pareto-optimal objective vectors and is denoted as the Pareto front.*

Finding the Pareto front is our main goal and can be done in many different ways. We select the most popular and interesting methods to compare their features and advantages. One of the oldest but widely used methods to solve a problem with multiple objectives is to transform the objectives into one single objective which is easier to handle. The weighted sum method does this by assigning weights to each objective function and varies them to obtain multiple solutions. Although this method is very insightful and easy to understand, it is not able to find solutions on non-convex parts of the frontier. Another disadvantage of the weighted sum method is its inability to find Pareto efficient solutions that are equally spaced. It is common that solutions are mostly found around the inflection points of the frontier. The even distribution of Pareto efficient solutions is desirable because each solution contains its own piece of valuable information about the problem. If all solutions are grouped together on parts of the frontier, there will be only information on those groups but not on the parts of the frontier where no solutions are present.

Marglin (1967) developed the ϵ-constraint method where a chosen objective is minimized with a constraint placed on the other objective functions, for example $J_k \leq \epsilon$ where J_k is the k-th objective function, hence the name epsilon-constraint method. The idea of this method is iteratively decreasing the constraint bound ϵ by a pre-defined constant δ. Since only one solution can be found in each iteration, δ should be chosen small enough to avoid missing any Pareto-optimal solutions. The necessity to choose such a constant is the main difficulty and drawback of this method. The advantage of the ϵ-constraint method is that it is capable of finding Pareto efficient solutions in non-convex regions. To get around the difficulty of finding proper δ, Marco Laumanns and

Zitzler (2005) proposed an adaptive scheme for the epsilon constraint method with the idea to make use of the information about the objective space as soon as it is available, which is during the search process. In this way, δ can be adjusted according to the solutions already found. Unfortunately, they indicate that the running times would exponentially increase with the number agents which makes it less useful for our problem.

Another popular method in optimization is the evolutionary approach. Evolutionary algorithms can also be used for generating a Pareto front such as developed by Shukla and Deb (2005) and Bentley and Wakefield (1997). The steps performed in multi-objective evolutionary algorithms are similar to the steps performed in the traditional single objective variant. In practice, the evolutionary method stores the optimal solutions, assigns fitness values based on Pareto dominance and removes solutions that are almost similar to reduce the number of non-dominated solutions without losing characteristics of non-dominated points. The advantage is an evenly distributed Pareto front, however, a drawback is the much longer running time compared to the other methods.

Normal Boundary Intersection (NBI) is developed by Das and Dennis (1996). Unlike the weighted sum method, NBI is capable of finding a uniform spread of Pareto points by using a scalarization scheme. The original technique is based on finding Pareto points on a continuous Pareto front. Opposite to that, the Pareto front containing solutions to our problem will be discrete. The discontinuity of our Pareto front causes that only customizations of NBI can be used, which are complex to implement for our specific case.
Several methods have been briefly discussed, each with its own advantages and disadvantages. Despite the drawbacks of the weigthed sum method, we choose an adaptive variant which is actually able to find solutions on concave parts of the Pareto front but retains its simplicity. The concept and generic steps of this approach are explained in Section 4.

4 The adaptive weighted sum method for bi-objective optimization

Finding a set of Pareto efficient solutions is the goal of multiobjective optimization. The definition of Pareto optimality is given in *Definition 1* in Section 3 but will be explained here for the bi-objective case.

A solution is Pareto efficient if it is non-dominated. For a solution to be non-dominated, no other solution can improve one objective without worsening another. Our objective functions are J_1 and J_2 for reliability and cost respectively. Mathematically, if there is no $x' \in X$ such that $J_1(x') \leq J_1(x)$ and $J_2(x') < J_2(x)$ or that $J_2(x') \leq J_2(x)$ and $J_1(x') < J_1(x)$, the solution is Pareto efficient. We define $x*$ as a Pareto efficient solution which implies that $z* = J(x*)$ is a non-dominated point. The set of all $z*$ is called the Pareto front or efficient frontier, which is exactly what we are looking for.

Generating the Pareto front can be done using several methods which are listed in Section 3. One of the simplest among them is the weighted sum method. This method uses weight functions to reflect the importance of each objective: $\sum_{i=1}^{2} \lambda_i J_i(x)$, where λ_i are the weights assigned to each objective function. Due to its simplicity it has some drawbacks such as the need to choose weights subjectively, inability to find solutions in concave regions, unevenly distributed solutions and the fact that small changes in λ may cause enormous changes in the objective vectors. That is why this method is not suitable for our purposes; it cannot generate a complete Pareto front.

To overcome the inability of finding solutions in concave regions, we propose the adaptive weighted sum (AWS) method developed by Kim and de Weck (2005). In the AWS method the weights are not predetermined but they evolve according to the Pareto front of the problem. In Figure 2 we depicted the difference between the weighted sum method and the adaptive weighted sum method.

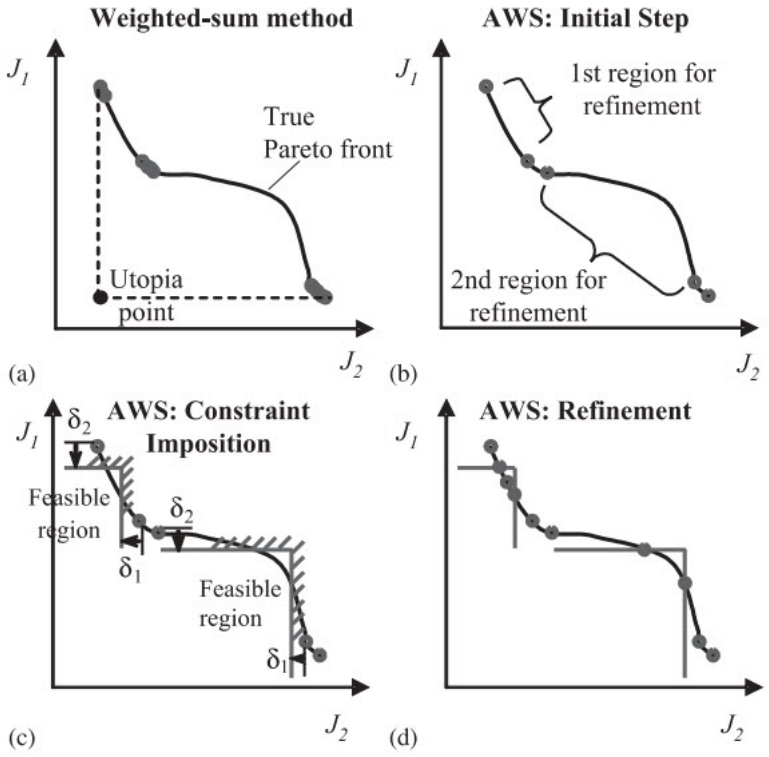


Figure 2: Comparison of WS with AWS method

As shown in Figure 2a, the weighted sum method primarily generates solutions near the inflection and anchor points which clearly visualizes a disadvantage of this method.

The AWS method in Figure 2b defines regions by setting each pair of two consecutive Pareto points as the limits of the region. In 2c two additional inequalities to define the search space are added to each region. By optimizing for different λ we obtain new solutions in Figure 2d. From here, the process is repeated iteratively until a given termination criterion is met.

As stated above, the AWS method does not have the drawbacks mentioned earlier. It is capable of finding solutions on concave parts of the frontier which are more evenly distributed than with the traditional weighted sum method. We cannot guarentee equally spaced Pareto points since the Pareto front is discontinuous in our case. However, the AWS method removes nearly overlapping solutions and focusses on regions where no solutions have been found yet. All steps of the AWS method will be explored in detail in the next section.

4.1 Steps for the adaptive weighted sum method

The AWS method consists of 7 steps which are discussed in the section. In these steps we make use of the constants $n_{initial}$,C,ϵ and δ_J which will be explained at the time of use.

Step 1: We define x_i^* as the optimal value for objective i. Then, the utopia point J^U is defined as:

$$J^U = [J_1(x_1^*), J_2(x_2^*)]$$

The nadir point is exactly the opposite of the utopia point and it is defined as:

$$J^N = [J_2(x_1^*), J_1(x_2^*)].$$

Now that the utopia and nadir point are defined, we can normalize the objective functions in the objective space by:

$$\overline{J_i} = \frac{J_i - J_i^U}{J_i^N - J_i^U}.$$

The normalization is required to ensure that both objective functions have the same order of magnitude. For convenience, we will use J_i as the notation for our objective functions and introduce sf_i as a scaling factor defined as:

$$sf_i = \frac{J_i}{\overline{J_i}}.$$

Step 2: To obtain initial solutions to start the AWS method we use the traditional weighted sum method with different λ to minimize $\frac{\lambda}{sf_1} \cdot J_1 + \frac{(1-\lambda)}{sf_2} \cdot J_2$ under the deadline constraint. Our problem is described in detail in Section 5.1. We solve the problem using branch and bound which is further explained in Section 5.2.1. The weighting factors λ are based on $n_{initial}$ which is the step size that should be small, 3 for example, such that not too many iterations are performed. Then, the weighting factors are defined by

$$\triangle\lambda = \frac{1}{n_{initial}},$$

which gives $\lambda = \{0, \triangle\lambda, 2\triangle\lambda, \dots 1\}$.

Step 3: In step 2 we have acquired a few Pareto points using the traditional weighted sum method. We sort these Pareto points in ascending value for cost and draw a straight line between each two consecutive points. The straight line drawn constitutes a segment which is included in set L_A. Solutions nearly overlap if the euclidian distance between two solutions is smaller than the a chosen constant ϵ. If this occurs we delete all neighbouring solutions and their corresponding segments except for the one with lowest cost. The remaining segments constitute set L_F.

Step 4: The Pareto points that form the remaining segments L_F in step 3 also define the vertices of regions, for example points P_1 and P_2 in Figure 3. Each search region has a rectangular shape with sides parallel to the axis and two vertices defined by two consecutive Pareto points the forms a segment. Each region has to be refined a number of times depending on the length of the segment that defines the region. If a segment is large, it needs to be refined more. The number of refinements is given by:

$$n_i = round(C\frac{l_i}{l_{avg}})$$

for the i-th segment. Here, l_i is the length of segment i, l_{ave} is the average length of all remaining segments in set L_F and C is a constant that determines the number of refinements. If n_i is less then or equal to one, no further refinements are required on this segment. Otherwise, go to the next step.

Step 5: The size of the regions from step 4 to search for new solutions is decreased in this step. A piecewise linearized line connects the two endpoint solutions P_1 and P_2 of segment i. the angle to the horizontal shown in Figure 3b is computed by:

$$\theta = tan^{-1}(-\frac{P_1^y - P_2^y}{P_1^x - P_2^x})$$

where P_1^x is the first coordinate of the end point P_1. Now we introduce the offset distance δ_J (shown in Figure 3b); only Pareto points with an euclidian distance to another Pareto points that is larger than constant δ_J are considered. Then, $\delta_1 = \delta_J cos\theta$ and $\delta_2 = \delta_J sin\theta$. From this we can compute the new bounds for the region to be refined. The lower and upper bounds on J_1 are given by $y_L = P_1^y + \delta_2$ and $y_U = P_2^y - \delta_2$. The bounds on J_2 are given by $x_L = P_2^x + \delta_1$ and $x_U = P_1^x - \delta_1$.

Step 6: Using the lower and upper bounds from step 5, we add new inequality constraints to the original problem and conduct sub-optimization with the weighted sum method. The problem is then stated as:

$$min \;\; \lambda\frac{J_1(x)}{sf_1(x)} + (1-\lambda)\frac{J_2(x)}{sf_2(x)}$$

s.t.

original constraints (as defined in Section 5.1)

$$x_L \leq J_1(x) \leq x_U$$

$$y_L \leq J_2(x) \leq y_U$$

Here, sf_1 and sf_2 are scaling factors for J_1 and J_2 respectively. These scaling factors ensure that each objective has the same order of magnitude in a region

and are defined in step 1. The weighting factors equal $\lambda = \{0, \Delta\lambda, 2\Delta\lambda, \ldots 1\}$ by using n_i from step 4 and $\Delta\lambda = \frac{1}{n_i}$. The unique solutions from the optimization are added to the set Pareto points.

Step 7: Sort the Pareto points by ascending cost and create segments between each two consecutive Pareto points. If all segments have a length less than δ_J, terminate. Otherwise go to step 3.

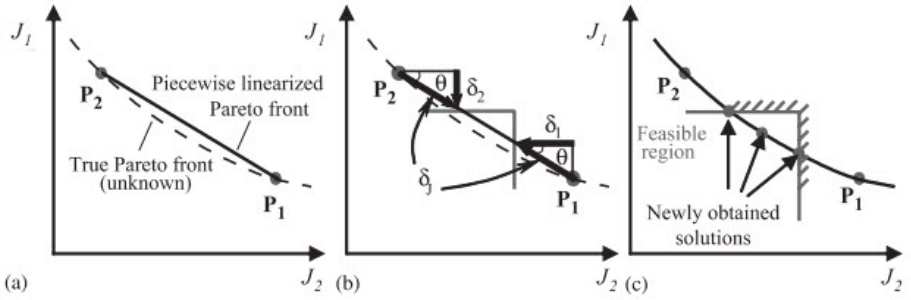


Figure 3: Identify and refine region by AWS method

The adaptive weighted sum method described above is a general method for solving a bi-objective optimization problem. In Section 5 we give a detailed problem description, propose a branch and bound method to solve the optimization in step 6 and apply the AWS method to our problem.

5 Adaptive weighted sum method for procurement

5.1 Integer program for finding an optimal sequence of agents

In step 3 and step 6 of the adaptive weighted sum method in Section 4.1 we perform an optimization to find an optimal sequence of agents. In this Section we propose a method to solve this optimization problem. All input parameters for this problem, namely c_i, d_i, b_i and r_i, are described in Section 2. The variables and objectives are declared below.

The decision variable x_{ij} is a binary variable that determines whether an agent is used in the sequence and at what place. m is the total number of agents we can put in a sequence. To vary the importance of each objective function, we use the weights λ which are defined in Section 4.1.

In Section 4 we have used J_i for the i-th objective. For our specific problem we explicitly define two objectives J_1 and J_2 which are aggregated into one objective function: $\frac{\lambda}{sf_1} \cdot J_1 + \frac{(1-\lambda)}{sf_2} \cdot J_2$.

Objective J_1 constitutes the probability of finishing within the deadline D of a sequence. Note that maximizing the probability of succes is equal to minimizing the probability of failure. The probability of failure for agent i is equal to $(1 - r_i)$. Therefore, the probability of failure of a sequence of agents, that is not finishing within the deadline, is defined as the product of $(1 - r_i)$ for all agents i in the sequence. Consequently, the probability of failure of a sequence is given by:

$$J_1 = \prod_{j=1}^{m}\prod_{i=1}^{m}(1 - r_i x_{ij}). \tag{1}$$

To determine the expected cost of a sequence, we propose the second part of the objective function. The value

$$\sum_{j=1}^{m}\sum_{i=1}^{m}(x_{ij}c_i + x_{ij+1}b_i)$$

defines the sum of all execution and backup costs incurred by a feasible solution x. However, there is only a probability of

$$\prod_{p=2}^{j}\sum_{k=1}^{m}x_{k,p-1}(1 - r_k)$$

that the j-th agent will actually provide service. The product of the cost and chance of execution gives us the total expected cost. This is given by:

$$J_2 = \sum_{j=1}^{m}\sum_{i=1}^{m}(x_{ij}c_i + x_{ij+1}b_i)\prod_{p=2}^{j}\sum_{k=1}^{m}x_{k,p-1}(1 - r_k). \tag{2}$$

A representation of these objectives in the objective space is given in Figure 3, the costs are on the x-axis while (1 - reliability) is on the y-axis.

In step 6 of Section 5 we perform the following problem optimization problem.

$$min \quad \frac{\lambda}{sf_1} \prod_{j=1}^{m} \prod_{i=1}^{m} (1 - r_i x_{ij}) + \frac{(1-\lambda)}{sf_2} \sum_{j=1}^{m} \sum_{i=1}^{m} (x_{ij} c_i + x_{ij+1} b_i) \prod_{p=2}^{j} \sum_{k=1}^{m} x_{k,p-1} (1 - r_k)$$

$s.t.$

$$\sum_{i=1}^{m} d_i x_{ij} \leq D \quad for \quad j = 1, \ldots, m \tag{3}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \quad for \quad j = 1, \ldots, m \tag{4}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \quad for \quad i = 1, ..., m \tag{5}$$

$$\sum_{i=1}^{m} x_{ij} \geq \sum_{i=1}^{m} x_{i,j+1} \quad for \quad j = 1, ..., m-1 \tag{6}$$

$$x_{ij} \in [0, 1] \quad for \quad i = 1, ..., m \quad and \quad j = 1, ..., m \tag{7}$$

$$y_L \leq J_1(x) \leq y_U \tag{8}$$

$$x_L \leq J_2(x) \leq x_U \tag{9}$$

Constraint 3 ensures that the total deadline is not exceeded. The 4-th limits the number of agents on a place in the sequence to one. The 5-th constraint makes sure that an agent cannot be used more than once in the sequence. Constraint 6 is there to fill up the sequence from the start. Constraint 7 sets the variable x_{ij} as a binary value.

In Section 5.2.1 we propose a branch and bound algorithm for solving the optimization problem above. Both constraint 8 and 9 are the bounds on the region defined in step 6 of Section 4.1. These constraints assure that the branch and bound algorithm in Section 5.2.1 is only examining the current regions search space.

5.2 Algorithm

5.2.1 Branch and bound

To solve the integer program in Section 5.1, we use a branch and bound algorithm. Each node of our branch and bound tree corresponds to a sequence of agents which is depicted in Figure 4. The starting node is an empty sequence S of zero agents, while all nodes one level below contain one agent. All steps of the branch and bound algorithm are carefully explained below.

1. Branch and bound (for pseudocode see Algorithm 4 in the Appendix).

   (a) *Compute general lower bound and general upper bound.* The bounds from constraint 8 and 9 are used to define the search space. The general lower bound is defined as $LB_g = \lambda \frac{y_U}{sf_1(x)} + (1-\lambda)\frac{x_L}{sf_2(x)}$, while the general upper bound is defined as $UB_g = \lambda \frac{y_L}{sf_1(x)} + (1-\lambda)\frac{x_U}{sf_2(x)}$.

   (b) *Start with a set S containing all sequences with one agent, that is*: $\{1, 2, 3, \ldots, m\}$. If a sequence contains n agents, the depth level in the branch and bound tree is n (see Figure 4).

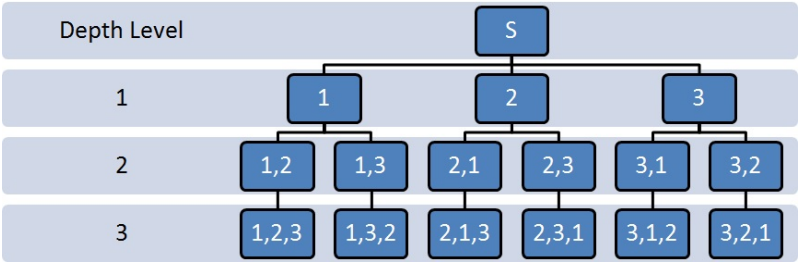


Figure 4: Branch and bound tree for 3 agents

   (c) First we define the set B which contains the possible lower and upper bounds for a node. For each sequence at current depth level, that is, at each node:

      i. *Calculate lower bound (LB) of the node* (for pseudocode see Algorithm 5 in the Appendix). Let s_c be the sequence of agents at the current node and add this sequence to the set S. From all agents not in s_c we select the one with minimum cost, append it to s_c and call the new sequence s_{lbcost}. Then, we create a sequence starting with the agents in s_c and append agents to this sequence. Adding these agents is a problem equal to the knapsack problem where you choose, from a given set of objects with given volume and profit, the most profitable subset that fits into a knapsack of finite capacity. Many ways of solving this particular problem are known, we chose to use dynamic programming which is shown in Algorithm 2 in the Appendix. We set d_i as volume, $-ln(1-r_i)$ as profit and D as capacity constraint to find the sequence, starting with s_c, with maximum reliability within D. The natural logarithm is necessary since we want to minimize $\prod_{j=1}^{m}\prod_{i=1}^{m}(1-r_i x_{ij})$ which is equal to $exp(\sum_{j=1}^{m}\sum_{i=1}^{m} ln(1-r_i x_{ij}))$. The sequence which is the result of the knapsack algorithm is denoted by s_{lbrel}. The lower bound is equal to $\lambda \frac{J_1(s_{lbrel})}{sf_1(x)} + (1-\lambda)\frac{J_2(s_{lbcost})}{sf_2(x)}$ where $J_i(s)$

is the value of objective J_i for sequence s. There are two reasons why this is a lower bound for the current node. First, the probability of failure for sequences starting with s_c is minimized. Consequently, there cannot be a child of the current node with lower failure probability (= higher reliability). Secondly, the cost of a sequence starting with s_c is minimized; there cannot be a child of the current node with lower cost. Finally, we add the lower bound to the set B.

ii. *Calculate possible upper bound (UB) for this node*, that is, the objective value $\lambda \frac{J_1(x)}{sf_1(x)} + (1-\lambda)\frac{J_2(x)}{sf_2(x)}$ for the sequence of agents at this node s_c. Add this value to the set B.

(d) For all solutions at this 'depth level', *find a solution that satisfies the deadline constraint and has minimum cost.* This solution will give the upperbound at this node and is denoted by UB_{min}

(e) For all new solutions, if a solution satisfies: LB $> UB_{min}$ or LB $> UB_g$(the general upper bound) or deadline $> D$, then:

i. *prune this node including its children delete and remove the corresponding values from S and B*

(f) *Add new sequences to set S.* For each sequence at current 'depth level', append an agent which is not yet in the sequence. New sequences are created for all nodes on which is branched.

(g) If any new sequences are added, go to step (c). Else go to step (h).

(h) Find the minimum upper bound, this is the optimal value of the branch and bound tree. The corresponding cost and reliability are the optimal solution to this iteration of the adaptive weighted sum method.

The proposed branch and bound algorithm finds one solution at a time. Therefore, it is necessary to run the branch and bound algorithm for different λ and different regions. This is exactly what the adaptive weighted sum method does in Section 5.2.2.

5.2.2 Adaptive weighted sum with branch and bound

The adaptive weighted sum method solves the branch and bound algorithm for all regions and varying weights such that the Pareto front will be generated. The two steps below explain how the solutions are obtained.

1. Calculate the sequence with lowest cost and sequence with highest reliability within D.

   (a) Lowest cost: find agent with lowest cost whose deadline is within D.

   (b) Highest reliability: execute dynamic programming algorithm (see Algorithm 1 in the Appendix) to solve the knapsack problem with d_i

as volume, $-ln(1-r_i)$ as profit and D as capacity constraint to find the sequence with maximum reliability within D. The natural logarithm is necessary since we want to minimize $\prod_{j=1}^{m}\prod_{i=1}^{m}(1-r_i x_{ij})$ which is equal to $exp(\sum_{j=1}^{m}\sum_{i=1}^{m} ln(1-r_i x_{ij}))$.

(c) The pseudocode of the knapsack algorithm is given in Algorithm 2 in the Appendix.

(d) The cost and reliability of these sequences are added to the set of Pareto points.

2. Adaptive weighted sum method, generally explained in Section 4.1 (for pseudocode see Algorithm 3 in the Appendix).

(a) Calculate scaling factors to ensure that both reliability and cost are in the same order of magnitude. They are defined in step 1 in Section 4.1.

(b) Sort Pareto points for increasing cost and calculate the lengths l_i between each consecutive set of points. If $l_i < \epsilon$ (nearly overlapping), delete Pareto point with higher cost.

(c) Calculate average segment length l_{ave}.

(d) Calculate the number of refinements on each segment $n_i = round(C\frac{l_i}{l_{ave}})$.

(e) Calculate the corresponding lambda's for this segment according to $\triangle\lambda = \frac{1}{n_i}$.

(f) According to step 5 in the AWS algorithm in Section 4.1, calculate the bounds to define the search space.

(g) For each λ in $\lambda = [0, \triangle\lambda, 2\triangle\lambda, \ldots, 1]$, run the branch and bound algorithm described in Section 5.2.1.

(h) If no new solutions are found, terminate. Else go to step 2(b).

All steps introduced above have also been written in detailed pseudocode for easy reference. In these pseudocodes, several standard functions are used. The function **RL** is defined by equation 1, **expcost** by equation 2 and **deadline** is given by $\sum_{j=1}^{m}\sum_{i=1}^{m} d_i x_{ij}$. The function **knapsack** is a dynamic programming algorithm to find the best combination of weights to achieve the highest reliability satisfying the capacity constraint.

The method described above is able to find a Pareto front in every case. However, the running time of the algorithm is an important factor in practice. Every different lambda creates a corresponding branch and bound tree which has to be searched through all the way. Depending on the density of Pareto points on the frontier the algorithm can be very slow. We provide a solution for the slow running time by proposing a new method of solving the problem in the next section. This method is also able to find the complete Pareto front, but searches the branch and bound tree only once.

6 A new branch and bound algorithm for finding the Pareto front

In this section we propose to find the Pareto front by using a modification of the original branch and bound algorithm (see Section 5.2.1) instead of the AWS method from Section 5.2.2. The difference between the two methods is that the AWS method searches the branch and bound tree several times depending on $\triangle\lambda = \frac{1}{n_i}$ (Section 5.2.2, step 2e) while the new branch and bound algorithm searches the tree only once. It uses a global upper bound to find out if nodes can be optimal and discards them if necessary.

The new branch and bound algorithm operates by constructing nodes which correspond to sequences of agents. The starting node is an empty sequence of zero agents, while all nodes one level below contain one agent. We will show that all Pareto solutions are found within one exploration of the branch and bound tree. The two steps below describe this method in detail.

1. Modified branch and bound (for pseudocode see Algorithm 6in the Appendix).

   (a) Start with a set S containing all sequences with 1 agent, that is: {1,2,3...,k}. If a sequence contains n agents, the depth level in the branch and bound tree is n (see Figure 4). The total number of agents is m.

   (b) Calculate global upper bound. We use the knapsack algorithm (for pseudocode see Algorithm 2 in the Appendix) to determine a sequence of agents s_{maxr} that has highest reliability satisfying the deadline constraint. Now that we know the combination of agents with highest possible reliability, we want to get the order with the lowest cost. Since the cost of a sequence does depend on the order of agents, we use another branch and bound algorithm to find the optimal order. Solving the new branch and bound algorithm only for the agents in sequence s_{maxr} yields a lowest cost for using all agents. The lowest cost is set as the minimum upper bound UB_{min}, a global bound, in the algorithm.

   (c) For each sequence of agents at current depth level:

      i. Calculate lower bound LB, that is, the objective value J_2 which equals the expected cost for the sequence at this node. Each child has more agents than the current node and more agents implies higher cost. Therefore, the cost of the current sequence is a lower bound for the current node.

      ii. Calculate the deadline of the sequence. That is the sum of all deadlines of the individual agents.

   (d) For all nodes explored in step (c) that satisfies $LB > UB_{min}$ from step (b) or deadline $>$ D, delete corresponding sequence from set S and prune this node and subnodes.

(e) Add new sequences to set S. For each sequence at current 'depth level', append an arbitrary agent which is not yet in the sequence. New sequences are created for all nodes on which is branched.

(f) If any new sequences are added, go to step 1(c). Else go to step 2.

2. Calculate cost and reliability for all sequences from the nodes we branched on.

   (a) For all sequences in S, calculate the corresponding solutions containing cost, reliability and deadline.

   (b) Get Pareto points from these solutions:

      i. Calculate the values of J_1 and sort them.

      ii. For all solutions the share the same reliability, determine lowest cost and discard other solutions.

      iii. If the solution from step (ii) satisfies the deadline and the cost is smaller than that of the previously added Pareto point, add this solution to the set Pareto points. If the set Pareto points is empty, add the solution obtained in step (ii).

      iv. Go to step (i) until all solutions are examined.

6.1 Correctness of the algorithm

Next we will prove that this algorithm finds all the Pareto points.

An important fact is that the reliability of a sequence of agents does not depend on their order, however the total cost does. Upper bounds and lower bounds are computed according to step 1(b) and 1(c) in the algorithm above. The lower bound is the cost of the current sequence. The upper bound is a global bound which is equal to the lowest cost for a sequence with maximum reliability under the deadline constraint.

Firstly, the knapsack algorithm with deadline constraint (for pseudocode see Algorithm 2 in the Appendix) is used to find the optimal combination of agents in terms of reliability called S_{maxr}. Subsequently the branch and bound algorithm finds an optimal ordering of the agents in S_{maxr}. This ordered set will be named S_{opt}. The obtained lowest cost of S_{opt} is set as upper bound, UB_{min}, for all nodes in the tree.

Now notice that each sequence in the tree can have at most a reliability equal to that of S_{opt} since this has been optimized by the knapsack algorithm. Therefore, if the cost of a sequence X is higher than the cost of sequence S_{opt}, sequence X and all sequences starting with X can be safely neglected in our search (see step 1(d) in the algorithm). Since all other points are examined and tested for domination, we are sure that the non-dominated points found actually constitute the complete Pareto front.

For easy reference, we have included the pseudocode for the new branch and bound method in Algorithm 6 in the Appendix.

7 Numerical results

7.1 Data description

In this section we will explain how we generate data sets to test our algorithms with. Before executing the algorithms explained in Section 5 and Section 6, we define the paremeters D and $info$.

D is the total deadline and $info$ is the set of data containing the cost (c_i), deadline (d_i), backup cost (b_i) and reliability (r_i) for each agent. The data set is randomly generated and reliability is dependent on the execution cost and deadline. The pseudocode for generating data is given in Algorithm 7 in the Appendix. This algorithm generates the four parameters mentioned above for one agent at a time.

To see how the Pareto front differs among different data instances, we have plotted the solutions of five randomly generated intances (by Algorithm 7 in Appendix) in Figure 5 with $D \gg \sum_{i=1}^{m} d_i$ where m is the total number of agents in the system.

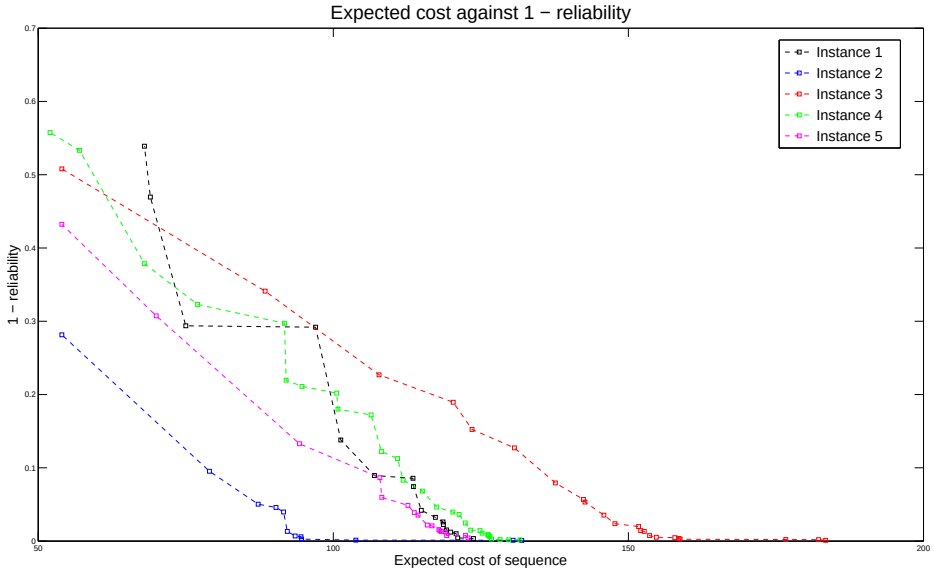


Figure 5: Pareto fronts for five different instances generated with the new branch and bound method

It is clear that the graphical representation of the Pareto fronts heavily depends on the original data. However, they show the similarity that many Pareto points are found in areas with high reliability which will be explored in detail later on.

7.2 Effect of the deadline constraint on running times

Running times of an algorithm are always interesting to users since large running times may make an algorithm not useful in practice. The running times are particularly dependent on the number of agents and the deadline constraint. For varying D, the running times for both the AWS method and new branch and bound algorithm are shown in Figure 6. To test the new branch and bound method we use a data set containing 100 different data instances with 10 agents each. For the adaptive weighted sum method we use only 10 different data instances with 10 agents each due to the large running time. The deadline constraint D is varied from 100 to 1000 with increments of 100.

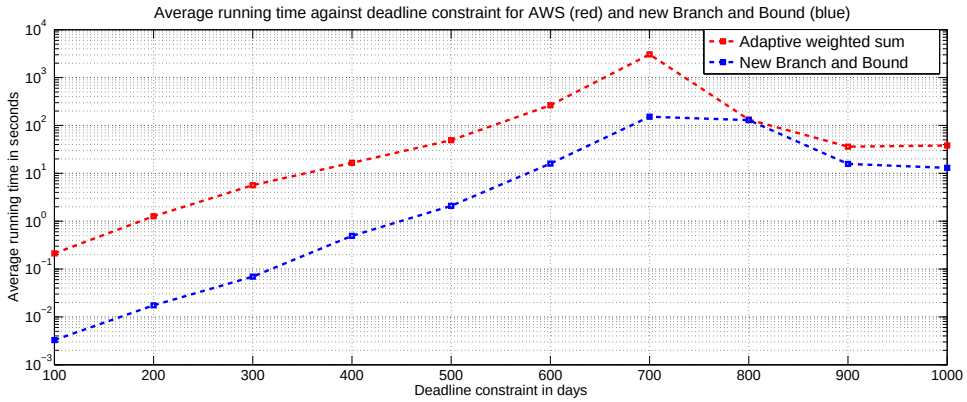


Figure 6: Average running time for the AWS method and new branch and bound method with $D = 100, 200, \ldots, 1000$

To see the large difference in running times, the y-axis has a logarithmic scale. In terms of performance, the modified branch and bound method is on average more than ten times faster than the adaptive weighted sum method. Due to the way the Pareto front is computed, the solution quality for the new branch and bound method is always equal to or better than that of the adaptive weighted sum method. This is shown for an abitrary data instance in Figure 7.

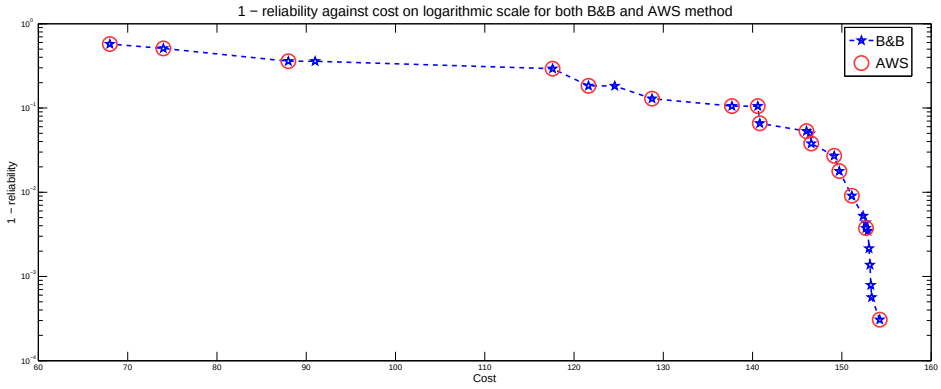


Figure 7: Pareto front on logaritmic scale for both the AWS and B&B method

Combining the results obtained above, we can conclude that the adaptive weighted sum method performs worse in both running time and solution quality. Therefore, we will discard the adaptive weighted sum method for now and continue using the modified branch and bound algorithm for our experiments.

7.2.1 Numerical experiments with the new branch and bound algorithm

As can be seen in Figure 6, the variation in running time for different D is more than four orders of magnitude among the different values of the deadline constraint. This could be due to the number of Pareto points that are found for a specific D. To explore this hypothesis, a graphical representation of the number of Pareto points against deadline constraint is given in Figure 8.

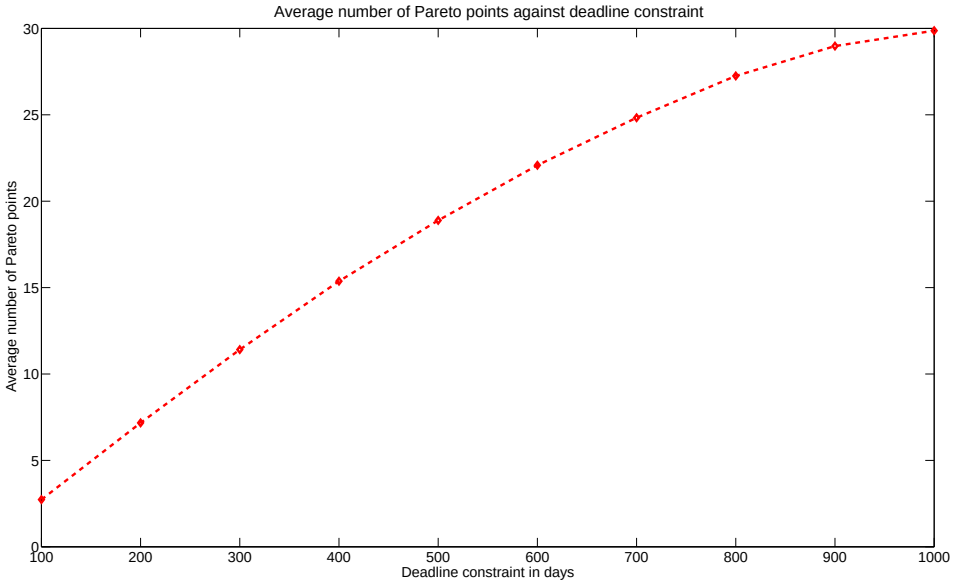
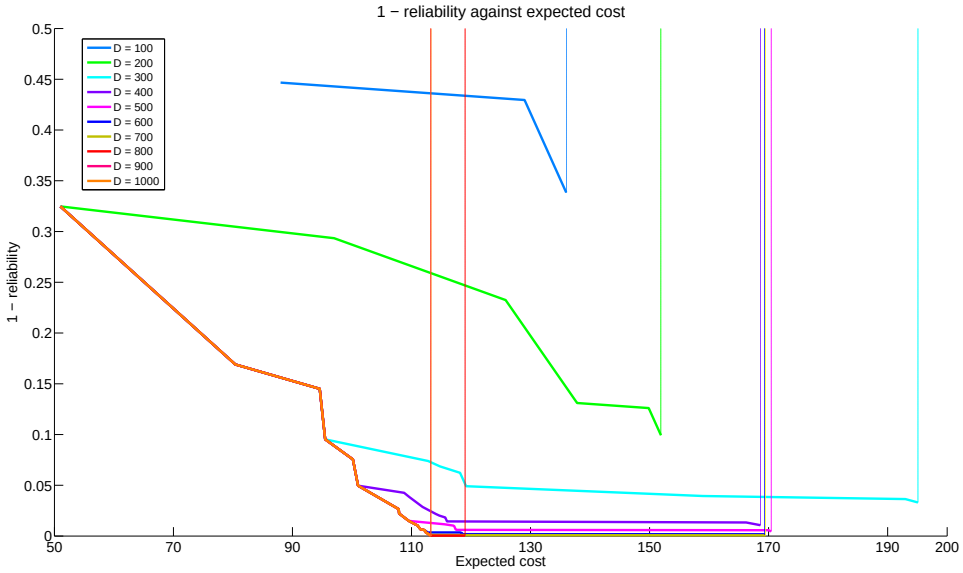


Figure 8: Average number of Pareto points for 100 data instances with $D = 100, 200, \ldots, 1000$

From the comparison of Figure 6 and 8 we can conclude that the number of Pareto points to find has influence on the running time of the algorithm. However, two characteristics are remarkable. Firstly, the absolute number of Pareto points on Pareto front grows slower when the deadline is higher, while the running times increase almost exponentially. Secondly, the running times eventually decrease again when the deadline constraints gets large but the number of Pareto points is always increasing. The concave shape of Figure 8 is easily understood by taking note of the fact that each set of agents has a maximum number of Pareto points. When the deadline constraint satisfies $D > \sum_{i=1}^{m} d_i$ for all data sets, the slope will be zero and the graph is completely horizontal. To explain the peak in Figure 6, we should look at Figure 9 which is generated using one data set with ten agents. Figure 9 contains the ten different Pareto fronts for each D with particularly high running times.



D	100	200	300	400	500	600	700	800	900	1000
seconds	0.0053	0.0163	0.2435	0.6256	16.61	225.4	5708	43.78	0.5915	1.238
nodes explored	54	218	2543	4878	24538	102706	383565	32559	1703	1837

Figure 9: 10 Pareto fronts for equal agents with $D = 100, 200, \ldots, 1000$

Not many Pareto points are present for a small deadline constraint since only a few agents are included in the optimal sequences. As D increases, the reliability also increases. For D between 500 and 800, this results in quite a large expected cost for the maximum reliability. Since the expected cost for maximum reliability is the upper bound for the modified branch and bound algorithm, it means that all points with cost below this bound have to be explored. This is the cause for a large running time; it is due to a high upper bound which in turn is the result of a high deadline constraint. For $D = 900$ and 1000 the upper bound is smaller and thus the running times are considerably less.

The region where the reliability is high is of most interested to us, since high reliability is desirable in practice. In Figure 10 we take a closer look at the region where the reliability is high, that is, a probability of finishing of 98% or higher.

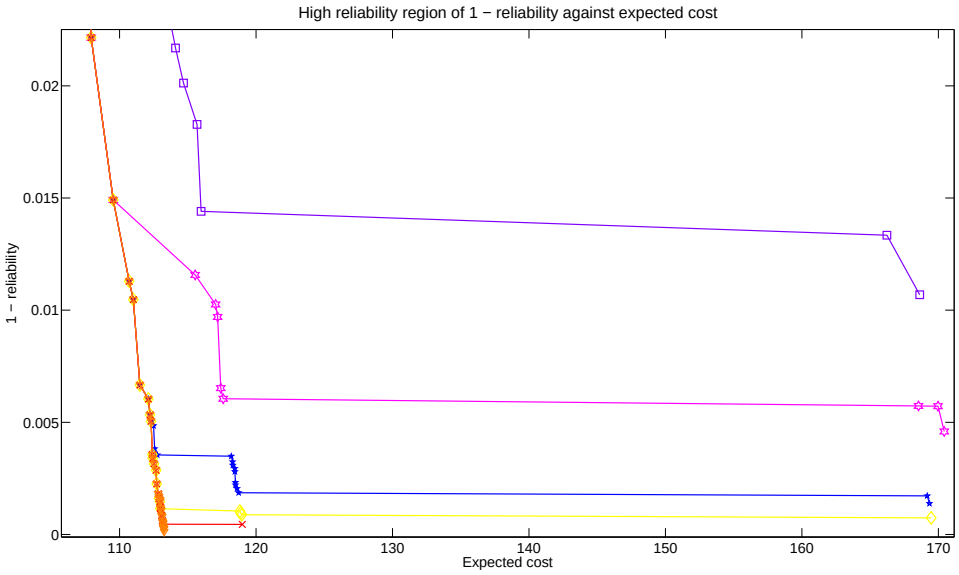


Figure 10: High reliability region of Figure 9 of ten Pareto fronts with $D = 100, 200, \ldots, 1000$

A lot of Pareto points can be found in the region where the reliability is high. Actually, for the Pareto fronts in Figure 10, 138 out of 219 Pareto points have a reliability higher than 98%. That is more than 63% of the Pareto points for this specific data instance. The fact that most Pareto points are found in regions with high reliability is due to the nature of the initial problem. The advantage of chosing a sequence of agents over one agent becomes readily apparent with the following example. Imagine a sequence of just four different agents with a propability of finishing within d_i of 0.7 each, the result is an overal reliability of 99.19%.

7.3 Effect of the number of agents on running times

The speed of the Branch and Bound algorithm depends heavily on the number of agents in the data set. Table 1 gives us an insight in the increase in running time. These running times are tested for only one instance of agents with $D \gg \sum_{i=1}^{m} d_i$. The deadline constraint is chosen to be large to see the effect of the number of agents independently.

Number of agents	Running time in seconds
10	0.8143
15	17.459
20	32.51
30	653.3
40	$8.541 \cdot 10^3$

Table 1: Running time against number of agents without deadline constraint

It is also interesting to test the dependency of the running time on the number of agents for a fixed deadline constraint. For this case we use 1 to 15 agents per instance. Since running times for an instance with 15 agents can go up to ten hours, we chose to examine only 20 different instances for each number of agents. In Figure 11 we display the average, median, minimum and maximum running time against the number of agents in the instance.

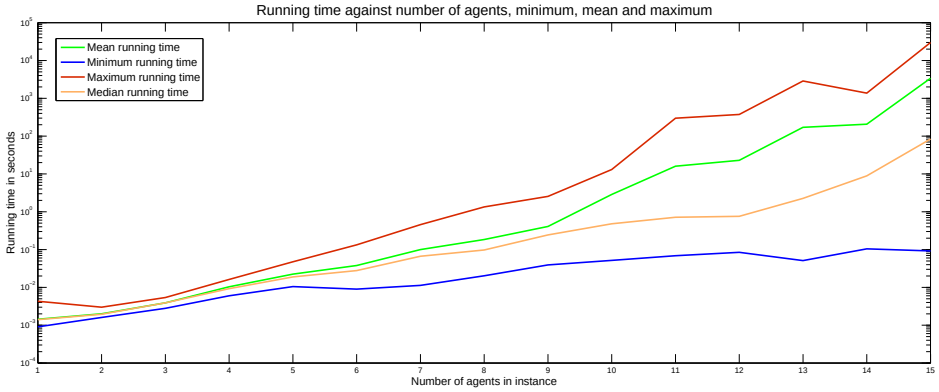


Figure 11: Running time on a logarithmic scale against number of agents with $D = 500$

For larger amounts of agents, the curve seems to follow an exponential increase. It is also remarkable that the minimum running time remains short although the number of agents goes up. Clearly, the probability that running time is high depends on the number of agents. However, the difference between running times increases for more agents per instance.

7.4 Effect of the cost and reliability parameters on running times

To see how the combination of cost and reliability effect the running time, we computed the running times for nine different instances combining different cost and reliability. All data instances contain 10 agents and the running time is an average of 1000 different instances for $D = 400$ (Figure 12) and 100 different instances for $D = 500$ (Table 2).

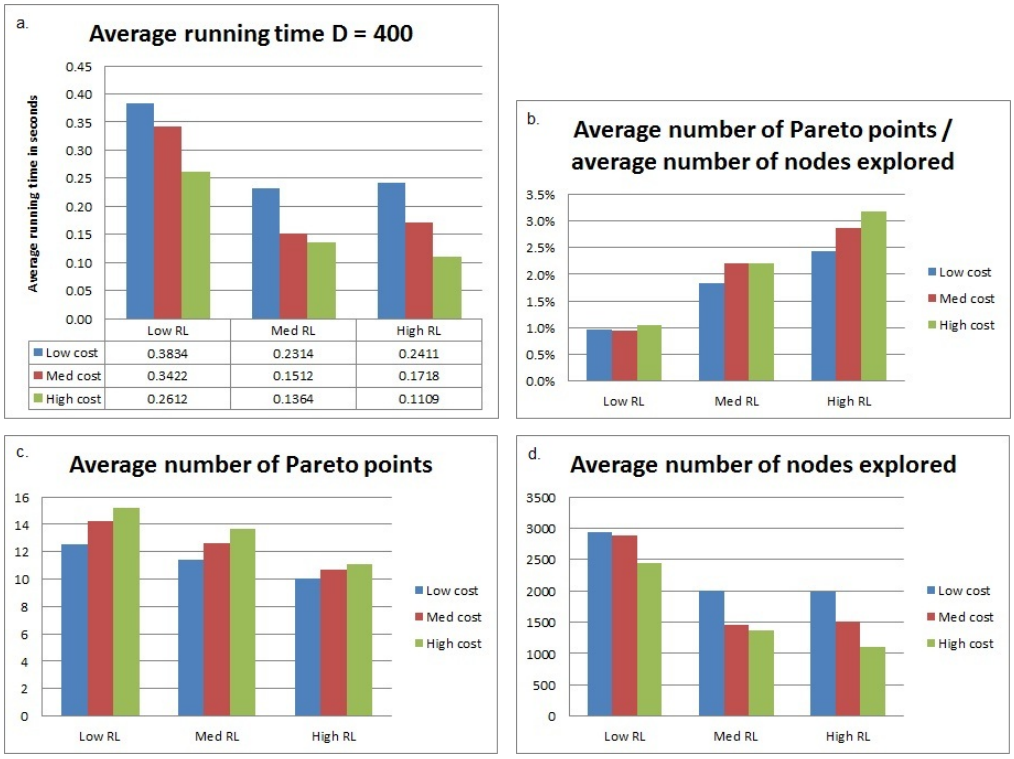


Figure 12: Analysis on running times with different parameters for cost and reliability with $D = 400$

	Low RL ($\bar{r} = 0.45$)	Medium RL ($\bar{r} = 0.65$)	High RL ($\bar{r} = 0.80$)
Low cost ($\bar{c} = 50$)	2.1179 | 0.3436	1.4963 | 0.2266	13.9389 | 0.2566
Medium cost ($\bar{c} = 100$)	2.6270 | 0.3258	1.0012 | 0.1092	1.4115 | 0.1295
High cost ($\bar{c} = 150$)	0.8482 | 0.3354	0.2017 | 0.0980	0.6386 | 0.0863

Table 2: Average and median running times in seconds for different instances with $D = 500$

The cost parameters are defined as a random discrete number between 1 and 100 plus a, with a = 0, 50 or 100 for low, medium and high cost respectively. Reliability is defined as a random continuous number between 0 and 0.4 plus b, with b = 0.25, 0.45 or 0.6 for low, medium and high reliability respectively. The average $d_i = 100$, so the average total deadline is 1000 for all 10 agents.

It is clear that an overall higher cost decreases the computation time significantly. This behaviour is explained by backup costs remaining constant while execution costs change in the experiment. When execution costs get higher, backup costs will have less impact wherefore the algorithm needs less time to

determine the agents in the solution.

Another remarkable feature of this data is that the running time is lowest for medium reliability. The standard deviation of the reliability is equal, namely 0.115 for each testing case, low, medium and high so this cannot be the cause. However, for the high reliability case the standard deviation is percentually smaller than that of the low reliability case. Therefore, it might be harder to pick the best agent if their reliabilities do not differ that much.

Figure 12a and 12d clearly show a strong correlation between running time and the number of nodes explored by the algorithm. This result is intuitive since running time is directly dependent on the number of calculations.

Figure 12b shows the percentage of Pareto points found against the number of nodes explored. It is noteworthy that the algorithm seems to perform better in terms of found Pareto points per node explored for high reliability. Since Figure 12b depends directly on 12c and 12d we should seek an explanation in these graphs.

In Figure 12c we observe that the average number of Pareto points increases as the cost increases but decreases as reliability increases. Unfortunately there is no indication that more optimal solutions should be present at higher cost or lower reliability.

For another deadline constraint, $D = 900$, we also computed the running times for the same 100 instances as used in Table 2.

	Low RL ($\bar{r} = 0.45$)	Medium RL ($\bar{r} = 0.65$)	High RL ($\bar{r} = 0.80$)
Low cost ($\bar{c} = 50$)	353.64	174.26	285.49
Medium cost ($\bar{c} = 100$)	32.991	9.8734	201.79
High cost ($\bar{c} = 150$)	11.061	1.1783	27.976

Table 3: Average running times in seconds for different instances with $D = 900$

In Table 3 is shown that running time decreases as cost gets higher which is supported by the fact that a smaller number of Pareto points is present at high costs just as in Figure 12c. Again we obtain the shortest running times from data instances containing agents with medium reliability.

To explain the cause of the huge difference in running times we look at Figure 13 which contains the Pareto fronts of one data instance with medium reliability. The running times are denoted in the legend to indicate the effect on the frontier.

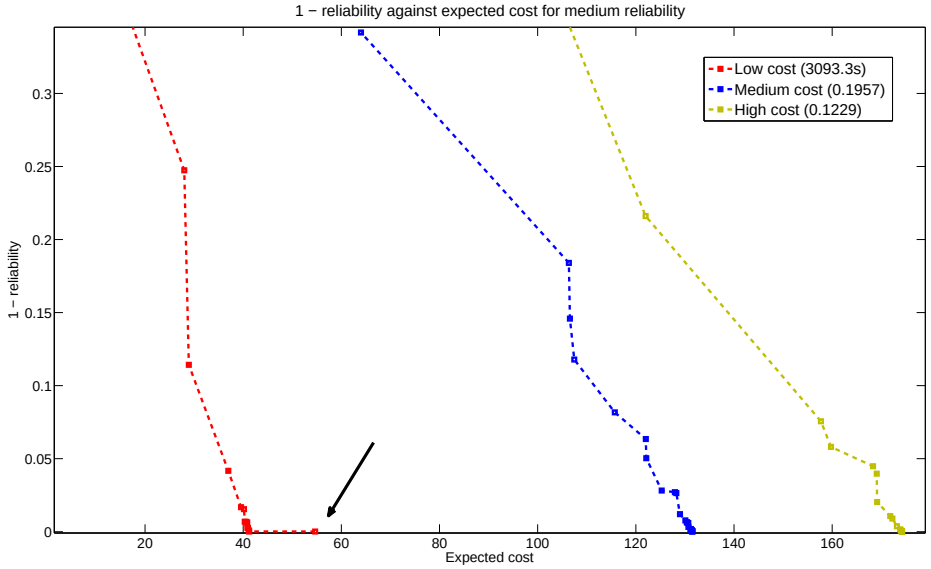


Figure 13: Three Pareto fronts for low, medium and high cost

The legend in Figure 13 shows the cost region and running times for three different Pareto fronts. The running time of the Pareto front with low cost is particularly large. This is caused by the relatively large upper bound (denoted by the arrow) compared to the other Pareto points on that front.

Such extraordinary running times are a defficiency of the modified branch and bound algorithm since that upper bound is not updated but used globally. An extreme example of this behaviour is depicted in the following graph.

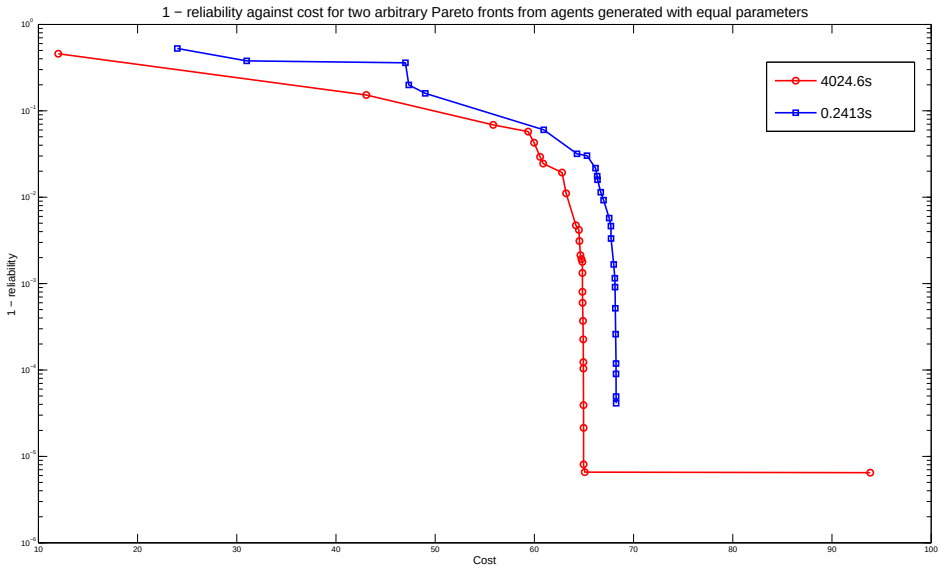


Figure 14: 2 Pareto fronts with their running times for agents generated with equal parameters

As can be seen in Figure 14, Pareto fronts for similar sets of agents can differ significantly. To have a clear view at the high reliability region, the y-axis has a logarithmic scale. The Pareto point on the red frontier with maximum reliability lies far away from the closest optimal point. Since the branch and bound algorithm explores each possible solution (satisfying the deadline constraint) with cost lower than this point, the running time for this Pareto front is very large. Moreover, the difference in reliability between the two points on the red frontier with highest reliability is only 0.001%. We will discuss a heuristic to overcome the problem of large running times in Section 8.2.

In the next section we will give a discussion on our results.

8 Trade-offs

8.1 Trade-off between cost and reliability

In Section 7 we have shown the results from our experiments. These test cases are only based on theoretical data. Unfortunately, we do not have acces to real data, but we can give qualitative arguments which apply to real cases. For real world applications, it is interesting to know what the trade-off in costs is for having a higher reliability. Table 4 gives the set of Pareto points. Figure 15 gives a graphical representation of the frontier.

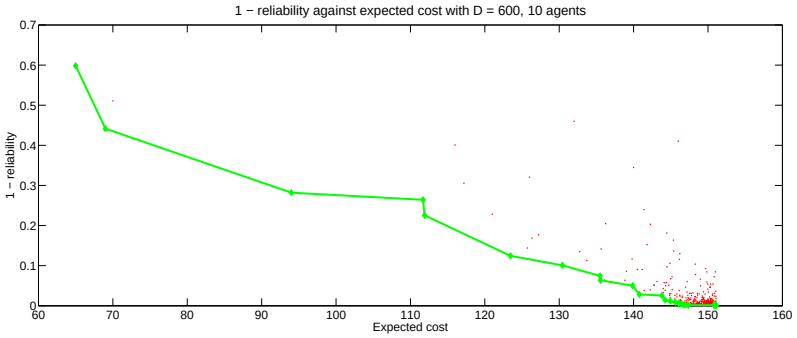


Figure 15: Pareto front for a data set with 10 agents and $D = 600$

% increase in cost	Reliability	% increase in cost	Reliability
0	0.4017	122.92	0.9886
6.15	0.5585	124.00	0.9913
44.62	0.7181	124.77	0.9932
71.85	0.7359	124.77	0.9942
72.15	0.7745	125.54	0.9954
90.00	0.8755	125.85	0.9965
100.62	0.8993	126.00	0.9972
108.46	0.9255	126.62	0.9981
108.62	0.9364	126.62	0.9983
115.23	0.9501	132.00	0.9985
116.62	0.9716	132.15	0.9988
121.23	0.9845	132.15	0.9990
121.85	0.9855	132.46	0.9992

Table 4: Pareto points of Figure 15

From Table 4 we can see that a cost increase of 6.15% already yields a reliability increase of 15 percentage points.. To increase the reliability to 95% the costs are more than doubled compared to the lowest cost. To determine whether this increase in cost is worth the higher reliability, a cost-benefit analysis

should be performed. When an auctioneer is able to tell what the penalty costs are for delaying a project, he can choose the Pareto solution that fits him best.

These results are based on just one data instance of ten agents. To get real insight in this matter, we should test for a lot of different data instances, just as in Section 7. Since the Pareto fronts are not continuous, it is actually impossible to compare different sets on reliability; there might not be Pareto efficient solutions with the exact same reliability which makes them incomparable.

However, it is possible to compare value which lie close to each other. We chose to determine the percentage cost increase from the cheapest possibility to the cost for 95%, 98% and 99% reliability. To make the reliabilities comparable, we searched within a range defined below.

Reliability to find	95%	98%	99%
Search range	93-97	97-99	>99

The following results are an average for 1000 different instances containing 10 agents each with $D = 500$.

Reliability range	93-97%	97-99%	>99%
Cost increase compared to cheapest agent	98.13%	107.2%	114.6%

A cost increase of about 100% for a reliability of 95% or more can be expected compared to hiring the cheapest agent. Again, it is up to the auctioneer to decide which factors are most important to him.

8.2 Trade-off between running time and accuracy

As shown in Figure 14 there can be huge differences in running time for sets of agents that seem very similar on the eye. To avoid extremely long running times we propose a heuristic that significantly shortens the running time.

In practice an auctioneer may want to have an indication of the cost for reliabilities up to 99%. Since major part of the running time is spended on finding Pareto points in the region >99%, the running time can be small when these points are not taken into account. Technically, this is achieved by pruning nodes in the branch and bound tree with reliability higher than the maximum value. To examine this theory we used three different maximum reliability values, namely 99%, 99.9% and 100% to see how the running times differ. We used a data set containing 1000 instances of 10 agents each with $D = 500$.

Maximum reliability	99%	99.9%	100%
Average running time in seconds	0.2672	2.332	4.894

Table 5: Average running times of 1000 instances for different maximum reliabilities, each instance contains 10 agents and $D = 500$

In Table 5 the average running times for different values of the desired maximum reliability are given. From these results it is clear that the running time

can be decreased almost 20-fold if a concession is made. However, such a trade-off contradicts the goal of finding the whole Pareto front and is not further explored for that reason.

In section 9 we keynote our final thoughts on the problem and give our conclusion.

9 Conclusion and further research

We developed two methods to find optimal solutions for the procurement problem. Both the adaptive weighted sum method and the modified branch and bound method are able to generate the Pareto front. Since the adaptive weighted sum method has to solve the problem in Section 5.1 numerous times, it performs significantly less well than our new branch and bound algorithm in terms of running time. However, the adaptive weighted sum method might add value to the new branch and bound algorithm since only a certain region is searched. Searching the whole region causes the new branch and bound method to be slow sometimes. Therefore, the two different methods of bounding combined might yield even better results in terms of running time.

We have seen in Figure 9 that a deadline constraint might result in higher costs for the same reliability. Sometimes it can be advantageous to increase the deadline constraint a bit to save costs. To achieve this, the model should be extended to a multi-objective optimization problem. Then, we want to optimize not only reliability and cost but also deadline.

In earlier research by Zhang and Verwer (2012) a method for receiving trustworthy information from each agent was proposed. The combination of obtaining reliable information about reliability and deadline and bi-objective optimization results in a useful tool for our economy. The right implementation of such a tool in several companies could yield higher profits, less delays and thus higher efficiency.

Acknowledgements:
Many thanks to Adriana Gabor and Yingqian Zhang for their extensive support.

References

[1] P.J. Bentley and J.P. Wakefield. Finding acceptable pareto-optimal solutions using multiobjective genetic algorithms. *University College London*.

[2] Gendreau Berube and Potvin. An exact epsilon-constraint method for bi-objective combinatorial optimization pforblems: Application to the traveling salesman problem with profits. *European Journal of Operations Research*, 194:39–50, 2009.

[3] Tobias Buer and Herbert Kopfer. A pareto-metaheuristic for a bi-objective winner determination problem in a combinatorial reverse auction. *Munich Personal RePEc Archive*, 36062, 2012.

[4] Indraneel Das and John Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *NASA Contracter report 201616*, ICASE Report No. 96-62:1–38, 1996.

[5] Matthias Ehrgott. Multiobjective optimization. *Association for the ADvancement of Artificial Intelligence*, 04:11, 2008.

[6] I.Y. Kim and O.L. de Weck. Adaptive weighted-sum method for bi-objective optimization: Pareto front generation*. *Struct Multidisc Optim*, 10.1007:149–158, 2005.

[7] Lothar Thiele Marco Laumanns and Eckart Zitzler. An adaptive scheme to generate the pareto front based on the epsilon-constraint method. *ETH Zurich, Institute for Operations Research*.

[8] S Marglin. *Public Investment Criteria*. MIT Press, 1967.

[9] Iyad Rahwan, Wayne Wobcke, Sandip Sen, and Toshiharu Sugawara, editors. *PRIMA 2012: Principles and Practice of Multi-Agent Systems - 15th International Conference, Kuching, Sarawak, Malaysia, September 3-7, 2012. Proceedings*, volume 7455 of *Lecture Notes in Computer Science*, 2012. Springer. ISBN 978-3-642-32728-5.

[10] Pradyumn Kamur Shukla and Kalyanmoy Deb. On finding multiple pareto-optimal solutions using classical and evolutionary generating methods. *KanGAL Report Number*, 2005006:1–38, 2005.

[11] Francis Sourd and Olivier Spanjaard. Multi-objective branch-and-bound. application to the bi-objective spanning tree problem. *Laboratoire d'Informatique de Paris*.

[12] Yingqian Zhang and Sicco Verwer. Mechanism for robust procurements. In Rahwan et al. [9], pages 77–91. ISBN 978-3-642-32728-5.

Appendix

Algorithm 1 Knapsack dynamic programming principle

The principle of this knapsack algorithm is based on dynamic programming. First we initialize the variable $V[0, w] = 0$ with $0 \leq w \leq D$. We can define $V[i, w]$ recursively as follows for all w:

1. If the new item is more than the current weight limit, that is $d_i > w$, than $V[i, w] = V[i-1, w]$.
2. If $d_i \leq w$, than $V[i, w] = max(V\{i-1, w], v_i + V[i-1, w-d_i])\ \ for\ \ 1 \leq i \leq m,\ \ 0 \leq w \leq D$.
3. Then, the optimal solution is equal to $V[m, D]$ with m as the total number of agents.

Algorithm 2 Knapsack pseudocode

```
1: input: values, volumes, Capacity
2: n ←size of values
3: A ←zero matrix with size (n+1 x Capacity +1)
4: % Comment % A(j+1, Y+1) means the value of the best knapsack with
   capacity Y using the first j items
5: for j ← 1 : n do
6:    for Y ← 1 : Capacity do
7:       if (volumes(j) > Y) then
8:          A(j+1, Y+1) ← A(j, Y+1)
9:       else
10:         A(j+1, Y+1) ← max( A(j, Y+1), values(j)+A(j, Y−volumes(j)+
            1) )
11:      end if
12:   end for
13: end for
14: best ← A(n+1, Capacity+1)
15: a ← best
16: j ←size of values
17: Y ← Capacity
18: while a > 0 do
19:   while A(j+1, Y+1) = a do
20:      j ← j − 1
21:   end while
22:   j ← j + 1
23:   add j to set Opt_S
24:   Y ← Y − volumes(j)
25:   j ← j − 1
26:   a ← A(j+1, Y+1)
27: end while
28: output: Opt_S
```

Algorithm 3 Adaptive weighted sum method

input: Paretopoints (contains Pareto points found. Initial: two x and y coordinates for points with lowest cost and highest reliability within D)

input: info (the information of all agents including cost, backup cost, reliability and deadline)

input: D (D is the deadline constraint)

Segmentlengths contains the euclidian distance between two consecutive Pareto points

$S_{f1,0}$ = average reliability of two extreme Pareto points

$S_{f2,0}$ = average cost of two extreme Pareto points

epsilon,δ_j, C % Constants which determine minimum distance between Pareto points and the number of refinements

$m \leftarrow 1$, $np_1 \leftarrow 1$, $np_2 \leftarrow 0$. np_1 and np_2 are respectively the number of Pareto points before and after solving the branch and bound

while $(m > 0$ & $np_1 \neq np_2)$ **do**

  Paretopoints $\leftarrow$ sort Paretopoints by ascending cost

  $i \leftarrow 1$

  while $(i <$ size of *Paretopoints*$)$ **do**

    Segmentlengths(i)$\leftarrow$ euclidian distance between Paretopoints

    if (Segmentlengths(i) $<$ *epsilon*) **then**

      Paretopoints(i,:) = {} delete Pareto points

      Segmentlengths(i,:) = {} delete segment

    else

      $i = i + 1$

    end if

  end while

  if $(np_1 \neq np_2)$ if new Pareto points are found in last iteration **then**

    $np_1 \leftarrow$ size of *Paretopoints*

    $m \leftarrow$ size of *Segmentlengths*

    $l_{ave} \leftarrow$ average length of *Segmentlengths*

    for $i \leftarrow 1 : m$ **do**

      $n(i) = round(C \cdot \frac{Segment(i)}{l_{ave}})$ % nr. of refinements in region i

      if $(n(i) > 1)$ **then**

        $\Delta\lambda = \frac{1}{n(i)}$ % calculate step size for weights

        p_x^2 =x-coordinate of the i-th Pareto point

        p_x^1 =x-coordinate of the i+1-th Pareto point

        p_y^1 =y-coordinate of the i+1-th Pareto point

        p_y^2 =y-coordinate of the i-th Pareto point

        $\theta = tan^{-1}(-\frac{p_y^1 - p_y^2}{p_x^1 - p_x^2})$ % establish lower and upper bounds

        $LB_x = p_x^2 + \delta_J cos(\theta)$

        $UB_x = p_x^1 - \delta_J cos(\theta)$

        $LB_y = p_y^1 + \delta_J sin(\theta)$

        $UB_y = p_y^2 - \delta_J sin(\theta)$

        for all $\lambda : 0$ with steps $\Delta\lambda$ till 1 (% see Algorithm 4) **do**

          [Paretopoints] $\leftarrow$ **BB**($LB_x, UB_x, LB_y, UB_y, \lambda, S_{f1,0}, S_{f2,0}$,Paretopoints,info,D)

        end for

      end if

    end for

  end if

  $np_2 \leftarrow$ size of *Paretopoints*

end while

output: *Paretopoints*

Algorithm 4 Original branch and bound algorithm for AWS

input: $LB_x, UB_x, LB_y, UB_y, \lambda$,Paretopoints,info,D
$n \leftarrow$ number of agents
$LB_g \leftarrow \frac{\lambda}{S_{f1,0}} \cdot LB_y + \frac{1-\lambda}{S_{f2,0}} \cdot LB_x$
$UB_g \leftarrow \frac{\lambda}{S_{f1,0}} \cdot UB_y + \frac{1-\lambda}{S_{f2,0}} \cdot UB_x$
$i \leftarrow 1$, $j \leftarrow 1$, $m_{old} \leftarrow 1$, $S \leftarrow$ all individual agents
while $(i <= n+1)$ **do**
  for $a \leftarrow m_{old}$: number of agents in S **do**
    $solutions(a,1)$ =**Lowerbound**(info, S, λ, D, S_f) (see Algorithm 5)
    $solutions(a,2) = \frac{\lambda}{S_{f1,0}} \cdot$**RL**(info, S (a)) $+\frac{(1-\lambda)}{S_{f2,0}} \cdot$ **expcost** (info, S (a))
    $solutions(a,3) =$ **deadline** (info, S (a))
  end for
  % Refinement %
  $x \leftarrow 1$
  while ($x = 1$ & number of solutions $\neq 0$ & $m_{old} \neq 0$) **do**
    $[UB_{min}\ z]$ =location and value of minimum of newly found solutions
    if (deadline of sequence z from solutions $> D$) **then**
      delete solution z
      delete sequence z
    else
      $x = 0$
    end if
  end while
  $m \leftarrow$size of $solutions$
  $k \leftarrow min(m_{old}, m)$
  while ($k <=$size of $solutions$ & $m \neq 0$ & $m_{old} \neq 0$) **do**
    if (LB of solution $k > UB_{min}$ or deadline of solution $k > D$) **then**
      delete solution k
      delete sequence k
    else
      $k = k + 1$
    end if
  end while
  % Add new sequences %
  if $(i <= n)$ **then**
    $m \leftarrow$number of sequences
    for $k \leftarrow m_{old} : m$ **do**
      for $l \leftarrow 1 : n$ **do**
        if (l is not yet in sequence S) **then**
          $S \leftarrow [S\ \ l]$ append agent l to sequence S
        end if
      end for
    end for
    $m_{old} \leftarrow m + 1$
  end if
  $j \leftarrow j + 1$
  $i \leftarrow i + 1$
end while

Continuing Algorithm 4:

t←number of points in Paretopoints
$z \leftarrow 1$
while z=1 **do**
  [*top seqnum*] =minimum lower bound from all nodes
  if $(top < LB_g)$ **then**
    delete value *top* from solutions
  else
    $z = 0$
  end if
end while
seq =sequence with minimum lower bound
$Paretopoints(t+1, 1)$ =**expcost**(info, *seq*)
$Paretopoints(t+1, 2)$ =**RL**(info, *seq*)
output: *Paretopoints*

Algorithm 5 Lowerbound

input: info, S, λ, D, S_f
info_temp←info, $n = size(info)$, $z \leftarrow 1$, $S_1 \leftarrow S$
lc =agent with lowest cost not in S1
$S1$ = [$S1$ lc] append agent with lowest cost to $S1$
T =constant with high value to give agents in sequence S high profit
for all agents in S **do**
  $r_i = r_i \cdot T$ % assures that each agent in S will be in sequence S_{rel}
end for
S_{rel} =**knapsack**(d_i, r_i, D)
% S_{rel} is a sequence with maximum possible reliability containing all agents from S
output: LB= $\frac{\lambda}{S_{f1,0}} \cdot$**RL**(info, S_t) $+\frac{(1-\lambda)}{S_{f2,0}} \cdot$ **expcost** (info, S1)

Algorithm 6 Modified branch and bound

input: $info, D$
n←number of agents
$i \leftarrow 1$, $j \leftarrow 1$, $m_{old} \leftarrow 1$, $S \leftarrow$ all n individual agents
$UB_{min} \leftarrow$ cost of Pareto point with highest reliability (by **knapsack**)
while $(i <= n+1)$ **do**
  for $a \leftarrow m_{old}$: number of agents in S **do**
    $solutions(a, 1)$ =**expcost** (info, S (a))
    $solutions(a, 2)$ = **deadline** (info, S (a))
  end for
  % Refinement %
  $m \leftarrow$size of $solutions$
  $k \leftarrow min(m_{old}, m)$
  while $(k <=$size of $solutions$ & $m \neq 0$ & $m_{old} \neq 0)$ **do**
    if (LB of solution $k > UB_{min}$ or deadline of solution $k > D$) **then**
      delete solution k
      delete sequence k
    else
      $k = k + 1$
    end if
  end while
  % Add new sequences %
  if $(i <= n)$ **then**
    $m \leftarrow$number of sequences
    % for all sequences at current depth level
    for $k \leftarrow m_{old} : m$ **do**
      % iterate over all agents
      for $l \leftarrow 1 : n$ **do**
        % if agent l is not in S
        if (l is not yet in sequence S) **then**
          $S \leftarrow [S \ \ l]$ append agent l to sequence S
        end if
      end for
    end for
    $m_{old} \leftarrow m + 1$
  end if
  $j \leftarrow j + 1$
  $i \leftarrow i + 1$
end while
$CostRelD \leftarrow$compute cost, reliability and deadline for all sequences in S
if $(CostRelD \neq \emptyset)$ **then**
  $[Paretopoints, S_{opt}]$ = **GetBestValues** $(CostRelD, S, D)$ filter dominated solutions described in step 2.
end if
output: $Paretopoints$

Algorithm 7 Set data for agents

The function $UnifDiscRand(x)$ draws a discrete random number between 0 and x

$UnifContRand(y, z)$ draws a continuous random number between y and z.

$c_i = UnifDiscRand(100) + 50$
$d_i = UnifDiscRand(100) + 50$
$b_i = UnifDiscRand(10) + 5$
if $(c_i + d_i >= 260)$ **then**
    $r_i = UnifContRand(0, 0.4) + 0.6$
end if
if $(220 < c_i + d_i < 260)$ **then**
    $r_i = UnifContRand(0, 0.4) + 0.5$
end if
if $(180 < c_i + d_i < 220)$ **then**
    $r_i = UnifContRand(0, 0.4) + 0.4$
end if
if $(140 < c_i + d_i < 180)$ **then**
    $r_i = UnifContRand(0, 0.4) + 0.3$
end if
if $(c_i + d_i < 140)$ **then**
    $r_i = UnifContRand(0, 0.4) + 0.2$
end if
output: c_i, d_i, b_i, r_i