Efficient Algorithms for Selecting Advanced Reservations

Mark Bartlett[1] Alan M. Frisch[1] Youssef Hamadi[2] Ian Miguel[3] Chris Unsworth[4]

[1] Artificial Intelligence Group, Department of Computer Science, University of York, York, UK
[2] Microsoft Research Ltd., 7 J J Thomson Avenue, Cambridge, UK
[3] School of Computer Science, University of St Andrews, St Andrews, UK
[4] Department of Computing Science, University of Glasgow, UK

December 2004

Technical Report
MSR-TR-2004-132

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

Grid computing aggregates various distributed heterogeneous resources to efficiently solve a variety of large scale parallel applications. This involves the sharing of resources distributed across multiple administrative domains. So far, the majority of Grid research has focused on the problems raised by accessing multiple domains (authentication, performance, etc) (4). This has resulted in important standard definitions[1] and will eventually meet the world of web services (3). However, resource sharing raises the problem of efficient selection and aggregation. The goal of the Gridline project is to study the applicability of constraint programming (1) in Grid resource optimisation. This technology has been very successful in industrial applications (11). This success comes from its high level of expressiveness along with an easy integration with imperative programming. Domains of application include scheduling and allocating both human resources (e.g., crew rostering and nurse scheduling) and material resources (e.g., airport gates and transport fleets). Gridline assumes that Grid resource sharing could greatly benefit from the application of constraint programming. The project employs constraint-based optimisation to produce three demonstrators.

This paper presents the results obtained with the first demonstrator which considers advanced resources reservation (AR), taking the rationale of a Grid resource broker that maximises its utility by choosing the optimal set of customers orders. Advanced reservation will play a major role in Grid systems. This mechanism guarantees the availability of resources to users at some specified future time. Such a contractual mechanism perfectly fits the requirements of complex Grid applications which usually require the combination of various Grid resources (2).

In the following, we first define advanced reservations. Section 2 formalizes the problem. Sections 3 and 4 present our new algorithms. Their theoretical analysis is presented in Section 5. Experimental results are presented in Section 6. Before giving an overall conclusion, Section 7 discusses dynamic requirements for this problem.

2 Advanced Resource Reservation

The design of our AR demonstrator is largely influenced by GGF recommendations (10; 6). In this Grid usage scenario, a broker manages some bounded yet divisible resource, such as network bandwidth or CPU nodes, etc. Since the resources are limited per time unit, the provider may be unable to meet all demands, so the broker must choose which requests are to be accepted. The idea here is to maximise the utility of the broker by selecting an optimal subset of customers' orders.

Figure 1 presents the architecture of this demonstrator. Gridline receives the customers' priced requests via the broker. It also obtains the status of potential resources from the Grid or directly from the broker. Gridline then computes an optimal selection of orders according to prices and penalties. This selection is forwarded to the broker who notifies customers of selection/rejection.

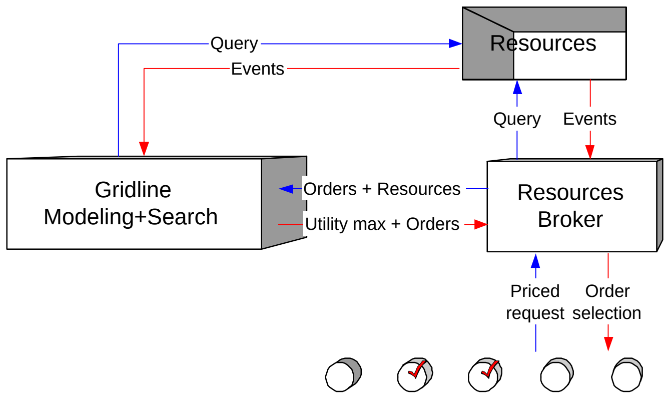

Figure 1. Advanced Resource Reservation

We can remark that there is no direct connection between customers and the Gridline service. This is an important privacy consideration: Gridline can be completely blind to customers' identities and can receive fake but relatively correct (homothetic) information on pricing.

[1] See the Global Grid Forum at www.gridforum.org.

In this figure, Gridline represents some web service used by the broker to optimise its business. However, recent works are using an abstract definition of Grid middleware (8). Such infrastructures could easily embed our demonstrator.

Customer orders use the following reservation pattern:

- Start and end time,
- Requested QoS, i.e., amount of resources per time unit,
- Price proposed for the service, and
- Penalty if QoS is not satisfied.

The previous can be modelled as a temporal knapsack problem, where the hard constraints ensure that QoS is maintained given bounded resources. The optimisation function maximises the gain (price) while minimising the potential loss. It is important to minimise potential loss since the distributed nature of the Grid leads to an increased rate of resource failures. The next section formalizes this problem.

3 The Problem

In the temporal knapsack problem (TKP), a scheduler is given orders for a divisible resource. Each order specifies the amount of quality of service (bandwidth, #CPU, storage, etc.) that is needed and the time interval during which it is needed. It also offers a price for the service. To simplify definition we consider that each price integrates the previously defined potential penalty. The scheduler will, in general, have more demand than capacity, so it has the problem of selecting a subset of the orders that maximise the utility obtained. More formally,

- Given:
  - $times$, a finite set totally ordered by $\leq$
  - $\forall t \in times$, $capacity(t)$, a positive integer
  - $orders$, a finite set s.t. $\forall o \in orders$,
    - $price(o)$, a positive integer
    - $QoS(o)$, a positive integer
    - $duration(o) = [start(o), end(o)]$, an interval of $times$
- Find: a set $accept \subseteq orders$
- Such that: $\forall t \in times$,
  $\sum_{\{o \in accept | t \in duration(o)\}} QoS(o) \leq capacity(t)$
- Maximising: $\sum_{o \in accept} price(o)$

The traditional knapsack problem, as described by Garey and Johnson (5), is a special case of TKP in which there is only a single time. Since the knapsack problem is NP-hard so is TPK.

TPK is a specialisation of the multi-dimensional knapsack problem (also known as the multiconstraint knapsack problem) in which the dimensions (the times) are totally ordered and the items (the bids) have non-zero size on consecutive dimensions.

4 The Decomposition Algorithm

We represent the set $accept$ by labelling each order (or order) with "accept" or "reject"[2]. Our first solver uses branching on accept/reject alternatives combined with a decomposition strategy which breaks the original problem in independent sub-problems. It applies various operations to perform its decomposition.

Figure 2 presents these operations. This example comprises 5 orders competing to access a uniform resource capacity of 10 units. The x-axis displays time slots.

The first operation that we can present is the `RemoveTime`, which is performed each time the capacity exceeds the demand. After this operation, the problem is bounded to time slots t_3, t_4 and t_{16}.

The second operation is the `ForcedAccept` which can force the acceptance of the third order. We apply it each time the required QoS is compatible with the resource capacity. We can easily present `ForcedReject` as the inverse operation, i.e., when the demand is higher than capacity.

[2]Notice that by taking accept to be 1 and reject to be 0, each capacity constraint is a linear pseudo-Boolean inequality and the objective function is a linear pseudo-Boolean expression.

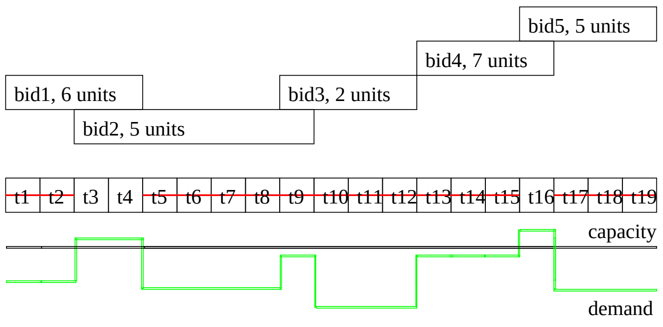

Figure 2. Removal of time slots combined with forced accept and split in independent sub-problems

Finally, we can apply the `Split` operation between orders 3 and 4. This breaks the problem in two independent sub-problems.

The decomposition algorithm which applies the previous operations is shown in Algorithm 2. It starts with the pre-processing defined in Algorithm 1. This pre-processing uses a `Reduce1` operation which is defined in the next section. This operation converts a problem into an equivalent one that contains no `ForcedRejects` or times that can be removed. The `ForcedRejects` operation is defined in the next section. `Reduce1`, `Reduce2` and `Reduce3` are, as we shall see, three methods for achieving this.

Algorithm 1: Pre-processing

```
Input: P, an advanced reservation problem;
begin
    Reduce1(P);
    Split1(P) into set of problems S;
    for (s in S) do
        Solve(s);
end
```

After this initial step, the algorithm successively applies two operations.

The first operation considers the rejection of some order o. After rejection, it applies a `Reduce2` operation on the problem. This reduction yields a simplified problem which may be split into independent sub-problems S. The algorithm is then recursively applied to each sub-problem.

The second operation considers accepting order o. This is followed by a `Reduce3` operation. After this simplification, the problem may be split and the algorithm is applied to each sub-problem.

4.1 Refining the Algorithm

We now refine the decomposition algorithm by defining the reduce and split operations.

In what follows let $demand(t : times)$ be

$$\sum_{\{o:orders \mid t \in duration(o)\}} QoS(o).$$

Also let $next(t)$ be the smallest time strictly greater than t; $next(t)$ is undefined if t is the largest time.

In defining the reduce and split operations we will make use of the following operations:

- RejectOrder(o:orders):
  1. label o reject
  2. remove o from orders

Algorithm 2: The Decomposition Algorithm

```
Input: P, an advanced reservation problem;
begin
    if $orders = \emptyset$ then  return;
    else
        Select an order $o$ from orders;
        Non-deterministically do one of;
        (1) RejectOrder(o);
            Reduce2(P);
            Split(P) into set of problems S;
            for s in S do Solve(s)
        (2) AcceptOrder(o);
            Reduce3(P);
            Split(P) into set of problems S;
            for s in S do Solve(s)
end
```

- AcceptOrder(o:orders):
  1. label o accept
  2. remove o from orders
  3. $\forall t \in$ duration(o) subtract QoS(o) from capacity(t)
- RemoveTime(t:times)
  1. remove t from times
  2. $\forall o \in$ orders do
     - if t in duration(o) then remove t from duration(o)
     - if duration(o) is empty then AcceptOrder(o)
- TestRemoveTime(t:times)
  1. if demand(t) $\leq$ capacity(t) then RemoveTime(t)
- TestForcedReject(o:orders):
  1. if for some t in duration(o), QoS(o) $>$ capacity(t) then RejectOrder(o)

An instance of TKP is said to be reduced if it contains no ForcedRejects and for all times $demand(t) > capacity(t)$.

Reduce1

LEMMA 4.1
For any problem, executing the following results in an equivalent reduced problem:

1. $\forall o \in$ orders TestForcedReject(o)
2. $\forall o \in$ times TestRemoveTime(t)

PROOF 4.1
After step (1), for all orders o, QoS(o) $<$ capacity(t) for all t in duration(o). After step (2), for all t in times demand(t) $>$ capacity(t). Since neither step (1) nor step (2) alter QoS(o') for any remaining order o' nor capacity(t) for any remaining time t, they cannot trigger further forced rejects. Hence, the resulting problem is reduced.

Reduce2

LEMMA 4.2
Let P be any reduced problem and let o be any order in P. Let P' be the problem that results from applying RejectOrder(o) to P'. Executing the following on P' results in an equivalent reduced problem:

- $\forall t \in$ duration(o), TestRemoveTime(t)

PROOF 4.2
Rejecting an order o decreases demand in duration(o) only. Hence, within duration(o), it is possible that demand is below capacity, triggering the removal of times. As per the proof of Lemma 4.1, neither rejecting o nor removing times can trigger further forced rejects. Hence, the resulting problem is reduced.

Reduce3

LEMMA 4.3
Let P be any reduced problem and let o be any order in P. Let P' be the problem that results from applying AcceptOrder(o) to P'. Executing the following on P' results in an equivalent reduced problem:

- $\forall o' \in$ orders such that duration(o') overlaps duration(o) TestForcedReject(o')
- Let R be set of orders rejected in previous step
  - $\forall o' \in R, \forall t \in o'$ do TestRemoveTime(t).

PROOF 4.3
Accepting an order o decreases the capacity in duration(o), which may trigger forced rejects for any order o' whose duration overlaps that of o. As per the proof of Lemma 4.2, rejecting o' may trigger the removal of times in duration(o'), but removing times cannot trigger further forced rejects. Hence, the resulting problem is reduced.

Split Throughout we shall assume that the split operation is performed only on reduced problems. We shall see that this is the case in our algorithm.

Consider the following operation:

- Let t_1 and t_2 be two times such that next(t_1)=t_2 and for no o in orders, duration(o) contains both t_1 and t_2, the problem can be split into two sub-problems :
  1. one containing times $T_1 = \{t | t \leq t_1\}$ and orders $\{o | duration(o) \subseteq T_1\}$
  2. one containing times $T_2 = \{t | t \geq t_2\}$ and orders $\{o | duration(o) \subseteq T_2\}$

LEMMA 4.4
This split partitions the orders. The two resulting sub-problems are independent and, together, are equivalent to the original problem.

PROOF 4.4
Since the two sub-problems have neither common times nor common orders, the optimal solution for one is independent of the optimal solution of the other. Hence, combining their optimal solutions gives the optimal solution to the original problem.

LEMMA 4.5
If a reduced problem is split, then all the newly-created sub-problems are also reduced.

PROOF 4.5
Since the split operation modifies neither the QoS, nor the capacity or demand at any time t, it cannot precipitate a forced reject nor the removal of times. Hence, the sub-problems are reduced.

LEMMA 4.6
If a reduced problem is split, then each of the newly-created sub-problems contains at least two orders.

PROOF 4.6
From Lemma 4.5, each sub-problem is reduced. Therefore, for all times t in a sub-problem demand(t) $>$ capacity(t). Assume that one of the sub-problems has just one order o. Hence, in the times t in duration(o), demand(t) = QoS(o). But since demand(t) $>$ capacity(t), o would be rejected — a contradiction with the fact that the problem is reduced.

4.2 The Search Strategy

The decomposition algorithm is non-deterministic. To make it deterministic, we view the decomposition algorithm as defining a search space and use a deterministic algorithm to search this space. The search space defined by the decomposition algorithm is an AND/OR tree. The original problem is at the root. This is an AND node, which means that it can be decomposed into independent subproblems, each of which is a child of the node. (In the degenerate case, it decomposes into a single subproblem, itself.) Each child of an AND node is an OR node, which means that a branching operation is performed. In the case of the decomposition algorithm, each OR node has exactly two children, one in which a selected order is accepted and one in which that order is rejected. The set of solutions of an OR node is the union of the solutions of its children. Each child of an OR node is an AND node.

A search method is needed to search the AND/OR tree and find the optimal solution within it. For this we use the AO* algorithm described by Nilsson (9), which is itself based on an algorithm of Martelli and Montanari (7). The AO* search strategy has two important properties: the first solution found is guaranteed to be the optimal one; and no algorithm that is guaranteed to find the optimal solution expands fewer nodes than AO*. The drawback of AO* is that requires a large amount of memory.

The AO* search strategy has three major stages.

Stage 1: The node to process next is found. To do this, one recursively descends the search tree, starting at the root and taking the child with the highest upper bound, at an OR node, or the leftmost child which has not been fully expanded, at an AND node.

Stage 2: The node is processed and if appropriate expanded. Processing an OR node consists of choosing an order to branch on (the highest priced order is chosen in the current work) and then the expansion phase involves creating two AND node children (one in which the order is accepted as included and one in which it is labelled as rejected) and estimating their upper bounds using a heuristic described below. Then, the processing stage involves reducing the problem using the `Reduce2` or `Reduce3` operation as applicable. If this reduction manages to label every order as accepted or rejected, the upper bound for the node is set to the value of the sum of the accepted orders values and a flag is set stating the value of this node is known. Otherwise if some variables are left undecided, child OR nodes are created, one for each independent sub-problem that can be formed at the node, and their upper bounds estimated.

Stage 3:The new bound values and flags are propagated back up the path to the root node. Starting at the node just processed and working up the tree to the root, the value of the upper bound (ub) and the known value flag (kv) is updated for each node as follows.

$$ub(n) = \begin{cases} a(n) & \text{if } n \text{ is a leaf node} \\ max(ub(n')) & \text{if } n \text{ is an OR node} \\ a(n) + \sum ub(n') & \text{if } n \text{ is an AND node} \end{cases}$$

$$kv(n) = \begin{cases} true & \text{if } n \text{ is a leaf node} \\ kv(m), & \text{if } n \text{ is an OR node} \\ \quad m \in n' \;\&\; \forall n' & \\ \quad ub(m) \geq ub(n') & \\ \forall n' kv(n') = true & \text{if } n \text{ is an AND node} \end{cases}$$

where $a(n)$ is the value of the assignments made at node n, and n' is the set of children of node n throughout. If the root node is now labelled as having a known value, the algorithm terminates and the upper bound value at the root node is the maximum gain possible (the assignment solution itself can easily be read out from the tree). Otherwise the algorithm loops back to the beginning and conducts a new expansion.

The Upper Bounds To calculate the upper bounds used by the algorithm at a node, the problem at that node is solved with the removal of two constraints which create a new problem that can be solved much faster than the original one and is guaranteed to provide the value of the true problem or an overestimate of it. This problem is created by,

- considering each time point as independent from all others – an order can be allowed to run at some of the times that it requests, without committing the system to run it for the full duration – and,
- linear relaxation of the problems at each time point – an order need not be given either the full QoS requested or no QoS, but any value within this range.

The first modification splits the problem into a set of independent one-dimensional knapsack problems, and the linear relaxation turns each of these into a problem that can be solved more quickly. The objective function for each of the times is the same as the objective function for the original problem, except that the price associated with each order is replaced with the price per unit time. The value of the upper bound is calculated as the sum of the value of the relaxed problem at each time point. Obviously, the result of the less-constrained problem will never underestimate the value of the full problem.

5 The Greedy Algorithm

Our second algorithm performs a greedy selection of the orders. It is presented in Figure 3. This algorithm can use various strategies to select orders. We consider here the following *rate*:

DEFINITION 5.1
If o is an order then $rate(o)$ is $price(o)/(QoS(o) * |duration(o)|)$

Algorithm 3: The Greedy algorithm

```
Input: P, an advanced reservation problem;
begin
    for $t$ = earliest time to latest time do
        while $capacity(t) < demand(t)$ do
            let $o$ be a minimal rate order from $\{o'|t \in duration(o')\}$;
            RejectOrder($o$);
    Let $R$ be the set of all orders labelled reject;
    for $o \in R$, generated from highest to lowest rate do
        if $\forall t \in duration(o), capacity(t) - demand(t) > QoS(t)$ then
            unlabel $o$ and put $o$ in orders;
    Accept all orders;
end
```

The algorithm starts with the processing of conflicting situation, i.e., when the demand exceed the capacity. Each time it finds such a situation, it rejects some minimal rate order. After this initial filtering, the algorithm considers rejected orders. Each time some rejected orders has a QoS requirement compatible with the remaining capacity, the order is unlabeled and put in the initial set of orders. After this, all the unlabeled orders can be accepted.

6 Analysis

6.1 The Decomposition Algorithm

Since the advanced reservation problem is NP-hard, the worst case time complexity is exponential on the size of the input. However, the underlying KNAPSACK problem is well known for its advantageous average time complexity when solved through branch-and-bound (5). We will rely on these classical results to provide an appreciation of the average running time of our algorithm.

THEOREM 6.1
The Decomposition Algorithm only returns correct solutions.

Here we have to consider that each accepted order prunes the related resources through an `AcceptOrder` operation which subtract the corresponding QoS from the resource capacity. This brings correct solutions since subsequent allocation are made with respect to new resource states .

THEOREM 6.2
The Decomposition Algorithm only returns optimal solutions, i.e., where the overall utility is maximized.

Here, we have to prove that our bound propagation mechanism is correct, i.e., that it correctly appreciates the quality of some sub-space. To show that we have to consider that it uses the tree structure to compute the bounds. When it considers an OR node it performs a max operation which correctly reports the highest possible quality with that node. With an AND node it performs a sum which correctly reports the overall quality of the sub-problems.

6.2 The Greedy Algorithm

Time Assuming that demand and capacity information are maintained by arrays indexed by time slots. The initial state of the capacity array is given as an input. Constructing the initial state of the demand array requires iteration over $|orders|$ orders. Now, assuming that we maintain an array, active, of the set of orders active at a particular time. We want to be able to select from each set by rate. If we use a balanced binary search tree, construction takes $O(|times| \times log_2|orders|)$. So, the first loop has worst-case time complexity of: $O(|times| \times |orders| \times log_2|orders|)$

The second loop is over the rejected set, $|R|$ which is bounded by $|orders|$. We can sort this set with $O(|orders| \times log_2|orders|)$ operations. Now, each iteration uses an inner loop that examines capacity and demand for whole duration of the selected order o. Worst case is $|times|$. Calculation is constant time. So, the second loop has worst-case complexity of: $O(|orders| \times log_2|orders|) + O(|orders| \times |times|)$.

So, the overall time complexity is: $O(|times| \times |orders| \times log_2|orders|)$.

Space Demand and capacity are represented through arrays of $|times|$ elements. The active array has size $|times| \times |orders|$. We must also store duration, demand and a state *accepted*, *rejected* or *undecided* for each individual order. So, the space complexity is $O(|times| \times |orders|)$.

Correctness

THEOREM 6.3
The Greedy Algorithm returns only correct solutions, i.e., where demand does not exceed capacity at any time.

By inspection, the first stage of the algorithm produces a set of orders that, if all accepted, do not exceed capacity at any time. Also by inspection, every time an order is added to this set in the second stage, its added demand does not lead to demand exceeding capacity at any time .

7 Experimental Results

The two algorithms were evaluated for speed of execution and, in the case of the greedy algorithm, relative solution quality, which we define as the value of the solution found as a percentage of the optimal solution value.

These properties were measured by assessing performance on a set of randomly-generated instances in which most factors affecting complexity were kept static, while varying the values of two parameters: the number of orders and max_rate. This latter parameter defines an upper bound on the ratio of the highest rate of any order to the lowest rate of any order. Each instance was comprised of a period of 2880 time slots, each of which had capacity 400. The value 2880 corresponds to the number of 15 minute slots in a fortnight.

Each order within an instance was generated by randomly choosing its start time, duration, QoS and rate from a uniform distribution over the following ranges:

$start(o) \in [0, 2880]$,
$duration(o) \in [1, 100]$,
$QoS(o) \in [1, 50]$,
$rate(o) \in [1, max\_rate]$.

From these vales, the price and end time of each order can be derived.

The maximum rate was ranged over the integers from 1 to 10, corresponding to instances in which all orders paid the same rate through to instances in which some orders could offer ten times the rate of others. The number of orders in each instance was set to be either 400, 500 or 600. All results shown in the graphs are averaged over 50 advanced reservation instances for each parameter setting.

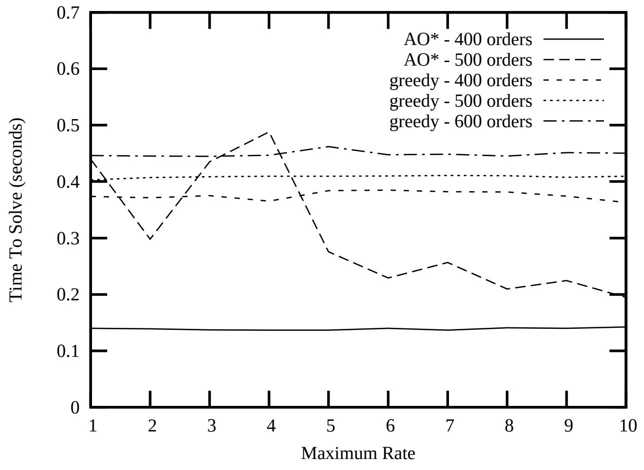

Figure 3. Time taken to solve the advanced reservation problem.

The graph showing times taken to solve the problems (Figure 3) reveal that for the instances with 400 orders, the decomposition algorithm outperforms the greedy algorithm. However, this advantage decreases as the number of orders increases. Indeed the decomposition algorithm often exhausted the available memory space when trying to solve the instances with 600 orders. We therefore do not report results for these instances, though future improvements to the algorithm (such as tighter bounds and explicit pruning) will allow larger instances to be solved. Furthermore, a general trend appears to emerge of instances with smaller ranges of rates being harder to solve for the decomposition algorithm. Again this effect is more pronounced for larger instances.

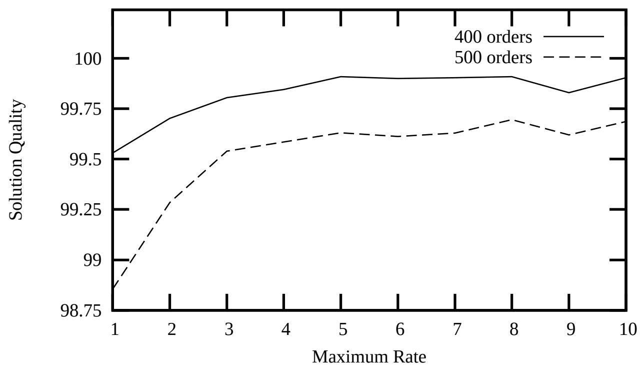

Figure 4. Relative quality of solutions found by the greedy algorithm.

Though the solution time required by the greedy algorithm does not appear to depend on the rate range, the quality of solution found by the algorithm does (Figure 4). The relative qualities of the solutions found by the greedy algorithm are always high for the parameter settings considered, but show a distinct loss of quality for the instances with the lowest range of rates. The relative solution quality also degrades with increasing numbers of orders.

We observe from the results that the instances with the lowest rate ranges are harder to solve than those with a large rate range. This increased complexity is manifested in the decomposition algorithm by an increase in the time to find the optimal solution, whereas the greedy algorithm exhibits a fall in the quality of its solution.

8 Dynamic Settings

This section considers a dynamic scenario for advanced reservation. Indeed, we can imagine that new orders could arrive to the scheduler during the solving process. Similarly, customers could cancel orders during solving. Our scenario considers those possibilities to demonstrate the suitability of our AR algorithms.

We consider a time-line sub-divided into, say, 30-minute steps. Let t denote our current position in this time-line. At time t we have a problem to solve in terms of a number of orders. We assume that the start time of all these orders is greater or equal to $t+1$. Hence, we have 30 minutes to solve this problem, deciding the subset of the orders to accept so as to maximise profit. Having done so, we move to the next time-step $t+1$.

At time $t+1$ we may receive new orders, hence we have a new problem. The form of this problem is determined by our assumed operating conditions:

1. **Decision Deadline:** Assume that each order has a deadline by which we must decide irrevocably whether or not to accept it. If this deadline is 30 minutes from when the order is received, we can never throw out a order to make way for a more profitable one. Hence, at $t+1$ our problem consists of just the orders received at $t+1$, with the available resource adjusted to take account of the decisions made at t. If, however, the deadline is longer, then we can throw out any order that has not yet met its deadline without penalty. So the problem at time $t+1$ considers the new orders and the orders at t that have not met their deadline.

2. **Throw-out Penalty:** We might allow ourselves to throw out an order that we have said yes to by paying a penalty. Now the problem at $t+1$ considers the orders at t that have not started and the new orders at $t+1$.

Finally, at time $t+1$ we may receive order cancellations. If a cancelled order was irrevocably accepted, we must recover its resource and reconsider the allocation of overlapping rejected orders. If that order was waiting for acceptance (cancellation received before decision deadline), we can remove it from the problem.

This is a dynamic environment but, assuming that we can solve each problem within the time allowed by each time-step, we can model it without modification of our current algorithms.

9 Conclusion and Future work

We have presented two dedicated algorithms to tackle the important problem of advanced reservation in Grid infrastructures. Advanced reservation represents an important mechanism which allows applications to request resources for use at a specific time in the future. Our solution takes the rational of a resource broker which maximizes revenue. This maximizing of the utility is possible thanks to the addition of price/penalty information. However, our algorithms are still applicable in non-commercial settings where one can imagine using internal knowledge on resource states to perform job selection. This knowledge could represent some probability for a successful execution which means that the broker maximizes the overall customer satisfaction.

Our algorithms were fully described and analyzed. The decomposition algorithm obtains optimal solutions while for particularly large problems the greedy algorithm can more quickly find solutions that are near optimal. Dynamic settings were rapidly drafted and showed that our methods are applicable with respect to some decision time-step large enough to allow successive solution of modified problems.

Our future work will improve the resistance of our solutions to environmental changes. Those changes can come from the infrastructure (resource failures) or from the customers (addition/removal of orders). One promising way to cope with the previous variations is to improve the robustness and stability of solutions. We believe that this combined with incremental versions of our algorithms will provide a nice answer to the problem of advanced resource reservations in Grid infrastructures.

References

[1] R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.

[2] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda, ‘Mapping abstract complex workflows onto grid environments’, *Journal of Grid Computing*, **1**(1), 25–39, (2003).

[3] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[4] I. Foster, C. Kesselman, and S. Tuecke, 'The anatomy of the Grid: Enabling scalable virtual organization', *The International Journal of High Performance Computing Applications*, **15**(3), 200–222, (Fall 2001).

[5] M. R. Garey and D. S. Johnson, *Computers and Intractibility, a Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., San Francisco, 1979.

[6] J. MacLaren. Advanced reservation: State of the art. GWD-I, Global Grid Forum (GGF), June 2003.

[7] A. Martelli and U. Montanari, 'Additive AND/OR graphs', in *IJCAI-4*, pp. 345–350, (1975).

[8] J. P. Morrison, B. Clayton, D. A. Power, and A. Patil, 'Webcom-G: Grid enabled metacomputing', *The Journal of Neural, Parallel and Scientific Computation*, **12**, 419–438, (2004). Special Issue on Grid Computing.

[9] N. J. Nilsson, *Priciples of Artificial Intelligence*, Tioga, 1980.

[10] A. Roy and V. Sander. Advanced reservation API. GFD-E5, Scheduling Working Group, Global Grid Forum (GGF), May 2003.

[11] M. Wallace, 'Practical applications of constraint programming', *Constraints*, **1**, 139–168, (1996).