The Temporal Knapsack Problem and Its Solution

Mark Bartlett[1], Alan M. Frisch[1], Youssef Hamadi[2], Ian Miguel[3], S. Armagan Tarim[4], and Chris Unsworth[5]

[1] Artificial Intelligence Group, Dept. of Computer Science, Univ. 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] Cork Constraint Computation Centre, Univ. of Cork, Cork, Ireland

[5] Department of Computing Science, University of Glasgow, UK

Abstract. This paper introduces a problem called the temporal knapsack problem, presents several algorithms for solving it, and compares their performance. The temporal knapsack problem is a generalisation of the knapsack problem and specialisation of the multidimensional (or multiconstraint) knapsack problem. It arises naturally in applications such as allocating communication bandwidth or CPUs in a multiprocessor to bids for the resources. The algorithms considered use and combine techniques from constraint programming, artificial intelligence and operations research.

1 Introduction

This paper defines the temporal knapsack problem (TKP), presents some algorithms for solving it and compares the performance of the algorithms on some hard instances. TKP is a natural generalisation of the knapsack problem and a natural specialisation of the multi-dimensional knapsack problem. Nonetheless, it is—as far as we know—a new problem.

In the TKP a resource allocator is given bids for portions of a timeshared resource — such as CPU time or communication bandwidth — or a shared-space resource — such as computer memory, disk space, or equivalent rooms in a hotel that handles block-booking. Each bid specifies the amount of resource needed, the time interval throughout which it is needed, and a price offered for the resource. The resource allocator will, in general, have more demand than capacity, so it has the problem of selecting a subset of the bids that maximises the total price obtained.

We were initially drawn to formulating the TKP from our interest in applying combinatorial optimisation techniques in the context of grid computing.

Applications that use a grid simultaneously require different resources to perform large-scale computations. An advanced-reservation system will be used to guarantee a timed access to resources through some service level agreement [1].

The agreement is reached via negotiation, where end users present reservation bids to resource providers. Each bid specifies the resource category, start time, end time and required quality of service (e.g., bandwidth, number of nodes) [2]. If end users offer a price they are willing to pay for the resource, advanced reservation allocation in a grid infrastructure becomes equivalent to the TKP.

Our algorithms are designed to be used by a resource provider to select efficiently the right subset of customers with respect to resource requirements and the provider's utility. So far advanced reservation is not used in grid infrastructures, which still use specialised *fifo* scheduling policies inherited from high-performance computing. However, the convergence between web services and grid computing, combined with the arrival of the commercial grid, make the efficient use of valuable resources critical [3]. Our algorithms fit well into next-generation grids and represent the first attempts towards efficient grid resource schedulers.

2 The Temporal Knapsack Problem

A formal statement of the TKP is given in Figure 1. Here, and throughout, $bids(t)$ is $\{b \in bids | t \in duration(b)\}$. It is important to notice that TKP is *not* a scheduling problem.

Figure 2(a) illustrates an instance of TKP that has seven bids, $b_1, \ldots, b_7$, and 10 times, $t_1, \ldots, t_{10}$, which are displayed on the x-axis. The instance has a uniform capacity of 10, which is not shown. The optimal solution to this instance is to accept bids b_1, b_4, b_5, and b_6, yielding a total price of 22.

The traditional knapsack problem, as overviewed by Martello and Toth [4], is a special case of TKP in which there is only a single time. Since the knapsack problem, which is NP-hard, is a special case of TKP, TKP is also NP-hard.

Given:	$times$, a finite, non-empty set totally ordered by $\leq$
	for each $t \in times$, $capacity(t)$, a positive integer
	$bids$, a finite set
	for each $b \in bids$,
	$price(b)$, a positive integer
	$demand(b)$, a positive integer
	$duration(b) = [start(b), end(b)]$, a non-empty interval of $times$
Find:	a set $accept \subseteq bids$
Such that:	$\forall t \in times$, $\sum_{b \in (accept \cap bids(t))} demand(b) \leq capacity(t)$
Maximising:	$\sum_{b \in accept} price(b)$

Fig. 1. Definition of the temporal knapsack problem

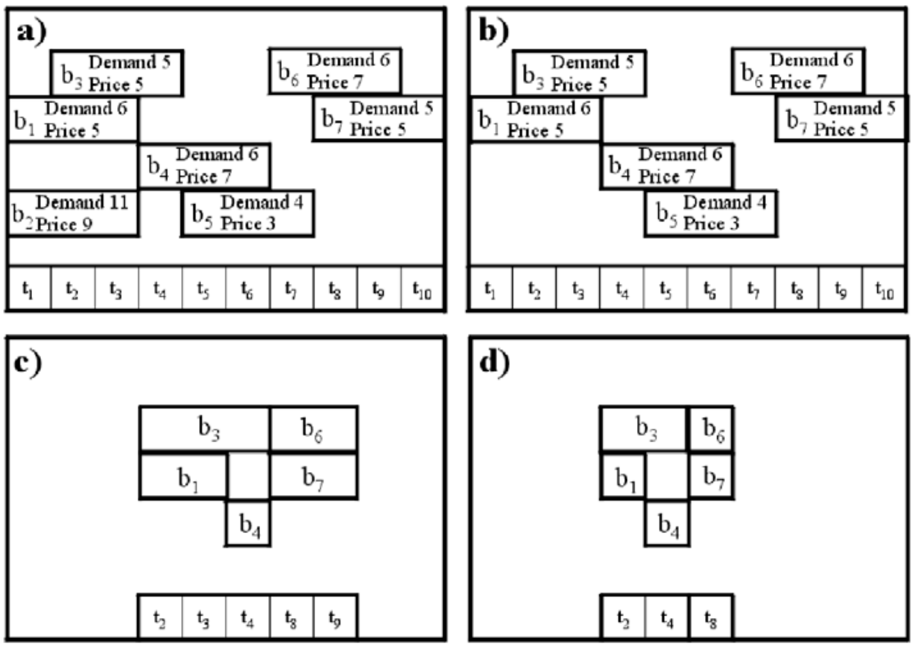

Fig. 2. An instance of the temporal knapsack problem to which the reduce operator is applied

The multidimensional knapsack problem (MKP, also known as the multi-constraint knapsack problem), as overviewed by Fréville [5], is a generalisation of TKP. Each time and bid in the TKP corresponds, respectively, to a dimension and an item in MKP. If $t_1, \ldots, t_n$ are the times in TKP, then every bid b corresponds to an item in MKP whose size is an n-dimensional vector of the form $\langle 0 \cdots 0 \; demand(b) \cdots demand(b) \; 0 \cdots 0 \rangle$ and the MKP capacity is the n-dimensional vector $\langle capacity(t_1) \cdots capacity(t_n) \rangle$. Since TKP polynomially-reduces to MKP, which is NP-easy, TKP is also NP-easy.

TKP readily reduces to integer linear programming. An instance of TKP generates the integer linear program that has a 0/1 variable X_b for each bid b:

$$\begin{aligned} \text{Maximise: } & \sum_{b \in bids} price(b) \cdot X_b \\ \text{Subject to: } & \sum_{b \in bids(t)} demand(b) \cdot X_b \leq capacity(t), \text{ for each } t \in times \\ & X_b \in \{0,1\}, \text{ for each } b \in bids \end{aligned}$$

Each solution to this integer linear program corresponds to an optimal solution to the TKP in which bid b is accepted if and only if X_b is assigned 1.

It shall often be convenient to switch between a TKP formulation, as given in Figure 1, and its linear programming formulation. To do this readily, it is important to notice that each time t produces one linear constraint; we shall refer to this as the constraint at t.

3 The Decomposition Algorithm

A decomposition algorithm for solving the TKP is presented here in three stages of increasing detail. Sec. 3.1 gives the algorithm as a non-deterministic procedure, which implicitly defines a search tree. Sec. 3.2 presents two methods for searching the tree for an optimal solution. Finally, Sec. 3.3 explains how linear programming is used to compute cuts, upper bounds, lower bounds and variable assignments, all of which are used to prune the search space.

3.1 A Non-deterministic Algorithm

Starting with all bids unlabelled, the decomposition algorithm records the solution by labelling each bid either "accept" or "reject." The algorithm employs three basic operations: reduce, which simplifies a problem instance; branch, which generates one branch in which an unlabelled bid is labelled "accept" and another branch in which that bid is labelled "reject"; and split, which decomposes a problem instance into smaller, independent problem instances. Let us begin by exemplifying the three operators.

The reduce operator performs two kinds of simplification. The first kind removes from the problem instance any bid b whose demand exceeds the capacity available at some time t in the duration of the bid. The operator labels all removed bids with "reject." This simplification corresponds to the constraint programming operation of achieving bound consistency on the constraint at t by removing the value 1 from the domain of X_b. For example, in the instance of Figure 2(a), bid b_2 has demand 11 at t_1, t_2 and t_3, yet the capacity at these times is only 10. So, the reduce operator removes b_2 from the instance and labels it "reject," which results in the instance shown in Figure 2(b).

The second kind of simplification removes unnecessary times from the instance. In the instance of Figure 2(b), at times t_1, t_5, t_6, t_7 and t_{10} the total demand does not exceed the capacity. Hence the constraints at these four times are satisfied in all assignments to the variables; in constraint programming terminology, they are entailed. Hence these times can be removed from the instance and from the durations of all bids. The result of this is that the duration of bid b_5 is now the empty set, meaning that b_5 participates in no constraints. Hence, b_5 is accepted in all optimal solutions. (In constraint programming terminology, any feasible solution that rejects b_5 is dominated by another feasible solution that is identical except that it accepts b_5.) Thus, the reduce operator removes b_5 from the instance and labels it "accept," resulting in the instance displayed in Figure 2(c).

There is a second method by which the reduce operator removes times from an instance. It is often the case that two adjacent times have the same bids. In such a case the time with the larger capacity imposes a weaker constraint and therefore can be removed; if the two times have the same capacity, then either time can be removed. In the instance of Fig. 2(c), t_2 and t_3 impose the same constraint as do t_8 and t_9. Thus, t_3 and t_9 can be removed from the instance, resulting in the instance of Figure 2(d).

The split operator decomposes a problem instance into subproblems that can be solved independently. A split can be performed between any two adjacent times, t and t', such that $bids(t) \cap bids(t') = \emptyset$. In the instance of Fig. 2(d), a split can be made in between t_4 and t_8 resulting in two subproblems: one comprising times t_2 and t_4 and bids b_1, b_3 and b_4; and a second comprising time t_8 and bids b_6 and b_7. Splitting is rarely used in constraint programming, though two recent exceptions are the work of Walsh [6] and of Marinescu and Dechter [7].

The branch operator is the familiar one from constraint programming, artificial intelligence and operations research. A bid b is selected, it is then removed from the problem and two branches are generated: one in which b is labelled "reject" and the other in which it is labelled "accept." On the accept branch, $demand(b)$ must be subtracted from the capacity at all times in $duration(b)$.

The decomposition algorithm, which applies the previous operations, is shown as Algorithm 1. This performs some initialisation and then calls the recursive procedure Solve.

Algorithm 1: Decomposition

```
Input: $P$, an instance of TKP;
Reduce($P$);
Split($P$) into set of problems $S$;
for ($s \in S$) do  Solve(s);

Procedure Solve($P$); (where $P$ is an instance of TKP)
if $bids = \emptyset$ then  return;
else
    Select a bid $b$ from $bids$;
    Non-deterministically do one of;
    (1) RejectBid($b$);
        Reduce($P$);
        Split($P$) into set of problems $S$;
        for $s \in S$ do Solve($s$)
    (2) AcceptBid($b$);
        Reduce($P$);
        Split($P$) into set of problems $S$;
        for $s \in S$ do Solve($s$)
```

Let us now turn our attention to the four procedures used in the decomposition algorithm: RejectBid, AcceptBid, Reduce and Split. The algorithms for these support procedures are given in Algorithm 2. In this discussion, and throughout, let $Demand(t)$ be the total demand at time t—that is

Algorithm 2: Support Procedures for the Decomposition Algorithm

```
Procedure RejectBid($b$ : $bids$);
label $b$ reject;
remove $b$ from bids;

Procedure AcceptBid($b$ : $bids$);
label $b$ accept;
remove $b$ from bids;
for $t \in duration(b)$ do subtract $demand(b)$ from $capacity(t)$;

Procedure Reduce(P);
for $b \in bids$ do TestForcedReject($b$);
for $t \in times$ do if $Demand(t) \leq capacity(t)$ then RemoveTime($t$);
if $|times| \geq 2$ then
    set $t_a$ to $min(times)$;
    while $t_a \neq max(times)$ do
        set $t_b$ to $next(t_a)$;
        if $bids(t_a) = bids(t_b)$ then
            if $capacity(t_a) \geq capacity(t_b)$ then RemoveTime($t_a$); set $t_a$ to $t_b$
            else RemoveTime($t_b$)
        else set $t_a$ to $t_b$

Procedure Split($P$);
Let $t_a$ and $t_b$ be two times such that $next(t_a) = t_b$ and $bids(t_a) \cap bids(t_b) = \emptyset$;
if no such times exist then return($P$);
else
    Let $P_1$ be the TKP instance with times $\{t | t \leq t_a\}$ and bids $\{b | end(b) \leq t_a\}$;
    Let $P_2$ be the TKP instance with times $\{t | t \geq t_b\}$ and bids $\{b | start(b) \geq t_b\}$;
    return(Split($P_1$) $\cup$ Split($P_2$))

Procedure RemoveTime($t$ : $times$);
remove t from times;
for $b \in bids(t)$ do
    remove $t$ from $duration(b)$;
    if $duration(b) = \emptyset$ then AcceptBid($b$)

Procedure TestForcedReject($b$:bids);
if for some $t \in duration(b)$, $demand(b) > capacity(t)$ then RejectBid($b$)
```

$\sum_{b \in bids(t)} demand(b)$. Also let $next(t)$ be the smallest time strictly greater than t; $next(t)$ is undefined if t is the largest time.

RejectBid(b) and AcceptBid(b) are simple; both label bid b appropriately and remove it from the problem instance. In addition, AcceptBid(b) must subtract the demand of the bid from the resource capacities available. Recall that reduce performs two kinds of simplification: (1) removing bids that must be rejected (forced rejects) and (2) removing unnecessary times, which may lead to removing bids that must be accepted (forced accepts). This is achieved by performing (1) to completion and then performing (2) to completion. Once this is done, there is no need to perform (1) again; doing so would not force any more rejects. In fact, it can be shown (though space precludes doing so here) that every time Solve is invoked it is given an instance such that

- $\forall t \in times\ \forall b \in bids(t)\ demand(b) \leq capacity(t)$,
- $\forall t \in times\ capacity(t) < Demand(t)$,
- $\forall b\ duration(b) \neq \emptyset$ and
- $\forall t, t' \in times\ t' = next(t) \implies bids(t) \neq bids(t')$.

An instance that has these properties is said to be *reduced* since performing the Reduce operator on the instance would have no effect. For each the two occurrences of Reduce in Solve we have implemented a significant simplification of the operator by considering the context in which it occurs.

As presented, the decomposition algorithm defines an AND/OR search tree.[1] Each node consists of a TKP instance. The root node is an AND node and comprises the initial problem instance. Every leaf node comprises a TKP instance with no bids. The children of an AND node are the (one or more) instances generated by applying the Split operator to the AND node. The set of feasible solutions to an AND node is the cross product of the feasible solutions of its children. Each child of an AND node is an OR node. Each OR node, other than the leaves, has two children generated by the branching in the algorithm—one child in which a selected bid is accepted and one in which that bid is rejected. The set of feasible solutions of an OR node is the union of the feasible solutions of its children.

The Decomposition Algorithm is correct in that the feasible solutions of the AND/OR search tree include all optimal solutions. The feasible solutions of the tree generally contain non-optimal solutions, which is obvious once one notices that the non-deterministic algorithm does not use the price of the bids. The next section considers how to explore the tree to find an optimal solution and how to use bounds on the objective function to prune the tree during the exploration.

3.2 The Search Strategy

This section explains how the standard branch-and-bound framework for OR trees can be adapted to handle AND/OR trees, such as those of the previous subsection. We refer this adapted framework as AOBB. The framework does not specify how the algorithm should choose the next node to be expanded. To do this, we currently employ two strategies: the AO* algorithm described by Nilsson [9] (which is itself based on an algorithm of Martelli and Montanari [10]), and a depth-first algorithm. The AO* search strategy is an extension of the A* algorithm to AND/OR search spaces, and retains two important properties of A*: the first feasible solution found is guaranteed to be an optimal one, and no

[1] The idea that a non-deterministic program implicitly defines an AND/OR tree was used in the very first paper published in the journal *Artificial Intelligence* [8].

algorithm that is guaranteed to find an optimal solution expands fewer nodes than AO* [11]. The drawback of AO* is that it requires a large amount of memory; the number of nodes in memory is $\Omega(2^{|bids|})$. In contrast, depth-first search stores only $\Omega(|bids|)$ nodes. However, in general, depth-first search explores more nodes than necessary to determine an optimal solution.

As with other branch-and-bound algorithms, AOBB stores at each search node an upper and lower bound on the objective function value for the TKP instance at that node. The AOBB algorithm repeatedly (1) selects an unexpanded OR node, (2) expands the OR node and then its children, and (3) propagates new bounds through the tree and uses these bounds to prune the tree. It performs this sequence of three stages until the tree contains no nodes to expand, at which point the result of the pruning is that the tree contains nothing but an optimal solution. Notice that whenever an OR node is expanded its children (which are AND nodes) are immediately expanded, producing OR nodes. Thus, the search tree's new leaves are always OR nodes. The only time an AND node is a leaf is at the start when the tree contains only the root node. Let us now consider the three major stages in more detail.

Stage 1: The node to process next is found. This is where AO* and depth-first differ. AO* selects a leaf node by descending the search tree, starting at the root and taking the child with the highest upper bound at an OR node. AO* allows any child to be taken at an AND node; our implementation takes the child with the largest spread between its upper and lower bounds. It is important to notice that successive descents can take different paths from a node since the node's bounds may change between the descents.

In contrast, when depth-first search expands a node its children are ordered from left to right and this ordering is fixed throughout the execution. Depth-first descends from the root node by always taking the left-most child that has unexpanded descendants. Depth-first search allows the children to be ordered in any manner. Our implementation orders the children of an OR node from left to right so that their upper bounds are non-decreasing and the children of an AND node so that the spread between their upper and lower bounds is non-decreasing.

Stage 2: The node is processed and expanded. The node found by the above stage will be either an OR node or the root node.

In the case of the root node, the resulting problem is reduced and split to form a set of child nodes, which are OR nodes, each containing independent subproblems. For each of these OR nodes, an upper and lower bound on its objective function value is obtained through solving the linear relaxation of the TKP at the node. As explained in the next subsection, at each OR node this linear program is also used to calculate three cuts (implied constraints in constraint programming terminology): a Gomory mixed-integer cut, a reduced costs constraint and a reversed-reduced-cost constraint. The Gomory cut is added to the linear programming form of the problem at the node and at all of its future descendents. This cut reduces the feasible region of the linear program without removing any integer solutions. Bounds consistency is enforced on all three cuts, which might determine the value of certain variables, i.e., whether a bid should be accepted or rejected. The RejectBid and AcceptBid operators are performed as appropriate, followed by the Reduce operator.

In the case of the node to expand being an OR node, an unlabelled bid is chosen to branch on, and two child AND nodes are created, one in which the bid is labelled "accept" and one in which it is labelled "reject." Both of these AND nodes are then processed and expanded in the same way as described for the root node. By expanding an OR node and both its children in a single stage in this way, the algorithm gains efficiency.

Stage 3: The new bound values are propagated and the tree is pruned. Starting at the OR nodes just created and working up the tree to the root, the value of the upper bound (*ub*) and the lower bound (*lb*) are updated for each node as follows.

$$ub(n) = \begin{cases} \max_{n' \in children(n)}(ub(n') + a(n,n')) & \text{if } n \text{ is an OR node} \\ \sum_{n' \in children(n)}(ub(n') + a(n,n')) & \text{if } n \text{ is an AND node} \end{cases}$$

$$lb(n) = \begin{cases} \max_{n' \in children(n)}(lb(n') + a(n,n')) & \text{if } n \text{ is an OR node} \\ \sum_{n' \in children(n)}(lb(n') + a(n,n')) & \text{if } n \text{ is an AND node} \end{cases}$$

where $a(n, n')$ is the sum of the prices of all bids accepted in moving from node n to node n', and $children(n)$ is the set of nodes that are children of n.

As this stage assigns and reassigns bounds, it checks to see if any OR node has one child whose upper bound does not exceed the lower bound of the other child. In such a case the best solution from the first child can be no better than that of the second child, so the first child and all its descendants are removed from the tree.

Having seen how the three stages operate, the last search issue that must be addressed is that of how bids are chosen for branching in Stage 2. We have tried two strategies for this. The *demand* strategy, chooses a bid with the highest demand. The intuition behind this is that labelling a bid with high demand is likely to lead to more propagation than one with low demand. The second strategy, called *force-split* is designed to yield nodes that can be split, preferably near the center. This strategy searches the middle half of the times for a pair of adjacent times, t and t', that minimises the cardinality of $S = bids(t) \cap bids(t')$; ties are broken in favour of the time nearest the center. The algorithm then branches on each bid in S in non-increasing order of their demands. This sequence of branches generates leaf nodes that can each be split between t and t'.

3.3 Generating Cuts and Bounds

This section explains how we use linear programming to generate cuts and bounds on the objective function value. The presentation assumes the reader is familiar with the basic theory underlying linear programming, such as that which is presented by Chvatal [12].

As shown in Section 2, a TKP instance can be represented as an integer linear program. This enables us to generate cuts that reduce the size of the feasible region for TKP without eliminating any potential integer solutions. To enhance the given search strategy, we employ the well-known Gomory mixed-integer cut (GMIC), which is considered one of the most important classes of cutting planes (see [13]).

Using the results from the linear relaxed TKP model ($0 \leq X_b \leq 1, \forall b \in bids$) and an objective function value z of a known feasible integer solution, a valid GMIC [14] for the TKP model can be written as

$$\lfloor z_U \rfloor - z \geq \sum_{\substack{f_i \leq f_0 \\ i \in N_1}} \lfloor -r_i \rfloor X_i + \sum_{\substack{f_j \leq f_0 \\ j \in N_2}} \lfloor r_j \rfloor (1 - X_j) + \sum_{\substack{f_i > f_0 \\ i \in N_1}} \left(\lfloor -r_i \rfloor + \frac{f_i - f_0}{1 - f_0} \right) X_i +$$

$$\sum_{\substack{f_j > f_0 \\ j \in N_2}} \left(\lfloor r_j \rfloor + \frac{f_j - f_0}{1 - f_0} \right) (1 - X_j) + \sum_{\substack{f_k \leq f_0 \\ k \in S}} \lfloor -r_k \rfloor s_k + \sum_{\substack{f_k > f_0 \\ k \in S}} \left(\lfloor -r_i \rfloor + \frac{f_k - f_0}{1 - f_0} \right) s_k$$

where, N_1 (N_2) is the set of indices for non-basic variables at their lower (upper) bounds; S, the set of slack variables s; r, the reduced costs; and z_U, the objective value of the linear relaxation model. It is clear that z_U provides an upper bound for the linear mixed integer TKP. In this notation, $f_0 = z_U - \lfloor z_U \rfloor$; $f_i = -r_i - \lfloor -r_i \rfloor$, $\forall i \in N_1$; and $f_j = r_j - \lfloor r_j \rfloor$, $\forall j \in N_2$.

The generated GMICs are added to the linear relaxed TKP instance and all its descendants in the search tree. Each added GMIC removes some of the non-integer solutions from the relaxed feasible region, but none that are integer. This also helps to improve the upper bound z_U.

We also employ the "reduced costs constraints" (*RCC*) and "reverse-reduced costs constraints" (*R-RCC*) discussed by Oliva et al. [15]. Following their work, the "pseudo-utility criterion" is used to obtain a reasonably good feasible solution. This criterion is computationally cheap, especially once the solution to the linear relaxation is known and the optimal values of the dual variables, λ, are determined. In this criterion all X_b are sorted in non-increasing order of $price(b)/(demand(b) \cdot \sum_{t \in duration(b)} \lambda_t)$ and the demands are satisfied in this order, as long as there is enough capacity. The resulting objective function value, denoted by z_L, yields a lower bound for the optimum z. From

$$z - \sum_{i \in N_1} r_i X_i + \sum_{j \in N_2} r_j (1 - X_j) - \sum_{k \in S} r_k s_k = z_U,$$

one can devise two useful constraints: *RCC* on the right, and *R-RCC* on the left.

$$z_U - z_{U^+} \leq - \sum_{i \in N_1} r_i X_i + \sum_{j \in N_2} r_j (1 - X_j) - \sum_{k \in S} r_k s_k \leq z_U - z_L$$

where z_{U^+} represents an upper bound which is stronger than the one provided by the linear relaxation (z_U). Such an upper bound can be obtained by using a "surrogate relaxation". In our case, this relaxation consists of adding together all of the constraints weighted with their associated dual values.

The aforementioned cuts and constraints are used for propagation purposes. By enforcing bounds consistency, the domains of decision variables including slacks are filtered and in certain cases are reduced to a singleton. Bounds consistency on a *RCC*/*R-RCC* constraint of the form $a \leq \sum_i b_i x_i + \sum_j c_j(1-x_j) \leq d$ gives the following additional bounds on the domains of the variables $x_p \in N_1 \cup S$ and $x_q \in N_2$:

$$\left\lceil \frac{a - \sum_i b_i - \sum_j c_j + b_p}{b_p} \right\rceil \leq x_p \leq \left\lfloor \frac{d}{b_p} \right\rfloor \qquad \text{and}$$

$$1 - \left\lfloor \frac{d}{c_q} \right\rfloor \leq x_q \leq 1 - \left\lceil \frac{a - \sum_i b_i - \sum_j c_j + c_q}{c_q} \right\rceil .$$

The GMIC works in a manner similar to *RCC*.

4 Performance Comparison

This section compares the effectiveness of three algorithms at solving a range of randomly-generated TKP instances. The algorithms considered are:

– the decomposition algorithm using AO* search with the forced-split variable-selection strategy;
– the decomposition algorithm using depth-first search with the forced-split variable-selection strategy; and
– the integer linear program solver provided by CPLEX version 8.1 with the default settings. The solver uses a branch-and-cut algorithm — branch-and-bound augmented by the use of cuts. After an initial "presolve" phase, which removes redundant constraints and attempts to tighten the bounds on the variables, the solver creates a tree, whose root contains the linear relaxation of the problem, and proceeds to expand nodes of this tree until an optimal integer solution has been found. At each node, the linear relaxation of the problem at that node is solved. If this leads to a solution in which some variables have fractional values, a selection of cuts are generated and added to the problem. The problem is then solved again, and if some variables are still non-integer, one is chosen to branch on, producing one child with the chosen variable set to 1 and another with it set to 0.

Preliminary experiments showed that the decomposition algorithm consistently performed better with forced-split selection than with demand selection. This is the case for both AO* search and depth-first search. Consequently, extensive experiments were not performed for the demand strategy.

Our method for randomly generating TKP instances is controlled by six parameters: *ntimes*, *max_length*, *max_demand*, *ucapacity*, *max_rate* and *nbids*. These are all integers, except *max_rate*, which is a floating point number. Given these parameters, an instance is generated that has $times = \{t_1, \ldots t_{ntimes}\}$, ordered in the obvious way, a uniform capacity of *ucapacity*, and *nbids* bids.

Each bid within an instance is generated by randomly choosing its start time, end time, demand and rate from a uniform distribution over the following ranges:

$$\begin{aligned}
&start(b) \in [1, ntimes],\\
&end(b) \in [start(b), \max(ntimes, start(b) + max\_length - 1)],\\
&demand(b) \in [1, max\_demand],\\
&rate(b) \in [1, max\_rate].
\end{aligned}$$

All of these are integers except that $rate(b)$ is a floating point number. From these values, we set $price(b)$ to $round(rate(b) \cdot demand(b) \cdot (end(b) - start(b) + 1))$.

Performance was assessed on a set of randomly-generated instances in which most factors affecting complexity were kept static,

$$\begin{aligned}
&ntimes = 2880,\\
&max\_length = 100,\\
&max\_demand = 50,\\
&ucapacity = 400,
\end{aligned}$$

while varying the values of two parameters: $nbids$ and max_rate. The value 2880 corresponds to the number of 15 minute slots in 30 days. By varying max_rate from 1.0 to 2.0 in increments of .2, and then further increasing its value to 4, 8 and 16, and by varying $nbids$ from 400 to 700 in increments of 50, we generated 63 problem suites, each containing 20 instances. By using these parameter values, we have focussed the majority of our experiments on instances generated with max_rate between 1 and 2, but also consider instances with larger values of max_rate in order to show the effect this parameter has over a greater range.

Figure 3 shows the mean solution time taken by each algorithm for all generated instances in which there are a given number of bids and $max\_rate \leq 2$, The graph reveals that for the problems with the lowest number of bids, both decomposition algorithms outperform the CPLEX solver. However, as the problem size increases, the performance of the decomposition algorithms deteriorates faster than that of CPLEX. The AO* algorithm is unable to solve some instances

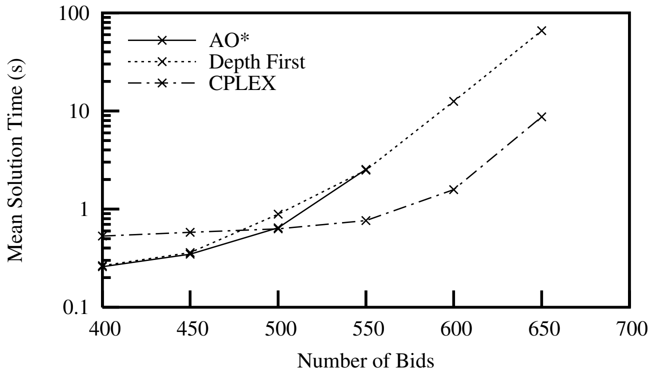

Fig. 3. Mean time taken to solve instances with a given number of bids

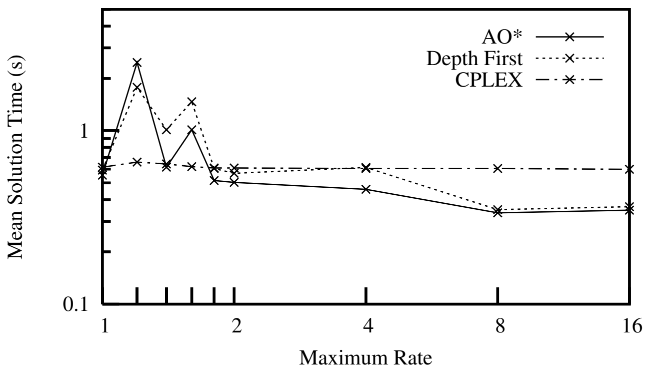

Fig. 4. Mean time taken to solve instances with a given range of rates

with 600 bids as it encounters memory problems and thrashes badly. The depth-first algorithm and CPLEX are both capable of solving all instances with up to and including 650 bids, but encounter problems on some instances with 700 bids; CPLEX through encountering memory-shortage problems and depth-first search through the exceptionally long time required to solve some problem instances.

Figure 4, shows the performance of the three algorithms on all instances in which *max_rate* has a given value and *nbids* $\leq$ 550. The graph shows that for smaller values of *max_rate*, on average the CPLEX solver performs best, followed by the AO* algorithm, with the depth-first exploration proving worse. However for larger values of *max_rate* the AO* and depth-first algorithms clearly outperform the CPLEX solver. It is worth noting that the performance of CPLEX, unlike the other two programs, is barely affected by the value of *max_rate*.

While we have reported mean solution time throughout, it should be mentioned that these are influenced strongly by the extremely long run times required for a few of the instances. For most problem suites, the algorithms solve most instances in very short times; however for a few instances substantially longer is taken, resulting in these instances having a large influence on the mean. Despite this, we report mean times rather than median times (which would not be affected by these extreme values) as the frequency of the very hard instances increases with both increasing numbers of bids and decreasing *max_rate*, and their frequency is a significant component of the difficulty of a particular problem suite.

5 Conclusion and Future work

This paper has defined the temporal knapsack problem and identified it as a formalisation of some problems that naturally arise in making advanced reservations. The TKP specialises the multidimensional knapsack problem by imposing a temporal structure.

We have designed a special-purpose algorithm for solving the TKP. Its novel feature is that it exploits the temporal structure to decompose problem instances into subproblems that can be solved independently. It also uses a branching method that is designed to increase the frequency with which decompositions are made. The decomposition algorithm combines techniques from constraint programming (e.g., bound consistency, entailed constraints), artificial intelligence (e.g., AND/OR search spaces and the AO* and depth-first methods for searching them) and operations research (e.g., linear relaxations, cuts, branch and bound).

The TKP readily reduces to integer linear programming, which can be solved with an off-the-shelf system such as CPLEX.

Our experiments compared the time it takes to solve randomly-generated instances of TKP with three algorithms: CPLEX and the decomposition algorithm with both AO* search and depth-first search. CPLEX and decomposition with AO* search are effective on instances with approximately 650 and 550 bids, respectively, but encounter space problems on larger instances. Decomposition with depth-first search is effective on instances with approximately 650 bids but runs slowly on larger instances, though it does not encounter space problems.

In comparing these solution programs one must consider that the algorithms and implementation of CPLEX have been refined over decades, whereas those of our decomposition algorithm have been refined over months. With this in mind, we speculate that with further development—such as that outlined below—the decomposition algorithm could handle larger instances than CPLEX. We also have come to appreciate that beating CPLEX requires significant effort.

We see many ways in which believe that the decomposition algorithm and its implementation could be improved. The most important improvement would be to employ a search algorithm that takes the middle-ground between time-efficient, space-hungry AO* and time-hungry, space-efficient depth-first search. Such an algorithm could be developed by generalising one of the memory-bounded versions of A*, such as SMA* [16], to operate on AND/OR search trees. It also is likely that the decomposition algorithm would benefit from a better heuristic for choosing where to force splits. We conjecture that a better heuristic could be developed by carefully trading off the advantage of splitting into equal-sized subproblems and the advantage of minimising the amount of branching required to force a split. Finally, the algorithm would surely benefit from further development of its data structures. In particular, it should be possible to efficiently identify a greater number of redundant times.

Acknowledgements

We thank Michael Trick, Michel Vasquez and Lucas Bordeaux for helpful discussions. This project has been partly funded by Microsoft Research. Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Post-doctoral Research Fellowship and S. Armagan Tarim by Science Foundation Ireland under Grant No. 03/CE3/I405 as part of the Centre for Telecommunications Value-Chain-Driven Research (CTVR) and Grant No. 00/PI.1/C075.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: Enabling scalable virtual organization. The International Journal of High Performance Computing Applications **15** (2001) 200–222
2. Roy, A., Sander, V.: Advanced reservation API. GFD-E5, Scheduling Working Group, Global Grid Forum (GGF) (2003)
3. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The physiology of the grid: An open grid services architecture for distributed systems integration (2002)
4. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. Wiley, New York (1990)
5. Fréville, A.: The multidimensional 0-1 knapsack problem: An overview. European Journal of Operational Research **155** (2004) 1–21
6. Walsh, T.: Stochastic constraint programming. In: Proc. of the Fifteenth European Conf. on Artificial Intelligence, IOS Press (2002) 1–5
7. Marinescu, R., Dechter, R.: AND/OR tree search for constraint optimization. In: Proc. of the 6th International Workshop on Preferences and Soft Constraints. (2004)
8. Manna, Z.: The correctness of nondeterministic programs. Artificial Intelligence **1** (1970) 1–26
9. Nilsson, N.J.: Principles of Artificial Intelligence. Tioga (1980)
10. Martelli, A., Montanari, U.: Additive AND/OR graphs. In: Proc. of the Fourth Int. Joint Conf. on Artificial Intelligence. (1975) 345–350
11. Chang, C.L., Slagle, J.: An admissible and optimal algorithm for searching AND/OR graphs. Artificial Intelligence **2** (1971) 117–128
12. Chvatal, V.: Linear Programming. W.H.Freeman, New York (1983)
13. Bixby, R.E., Fenelon, M., Gu, Z., Rothberg, E., Wunderling, R.: MIP: Theory and practice — closing the gap. In Powell, M.J.D., Scholtes, S., eds.: System Modelling and Optimization: Methods, Theory, and Applications. Kluwer Academic Publishers (2000) 19–49
14. Wolsey, L.A.: Integer Programming. John Wiley and Sons, New York (1998)
15. Oliva, C., Michelon, P., Artigues, C.: Constraint and linear programming : Using reduced costs for solving the zero/one multiple knapsack problem. In: Proc. of the Workshop on Cooperative Solvers in Constraint Programming (CoSolv 01), Paphos, Cyprus. (2001) 87–98
16. Russell, S.J.: Efficient memory-bounded search methods. In: Proc. of the Tenth European Conf. on Artificial Intelligence, Vienna, Wiley (1992) 1–5