Journal of Mathematical Modelling and Algorithms (2005) **4:** 331–353
DOI: 10.1007/s10852-005-0855-4
© Springer 2005

Exact and Heuristic Algorithms for Dynamic Tree Simplification

CARLOS CORREA[1], IVAN MARSIC[1] and XIAODONG SUN[2,⋆]
[1] *Rutgers University, Department of Electrical and Computer Engineering and the CAIP Center, Piscataway, NJ 08854, USA. e-mail: {cdcorrea,marsic}@ece.rutgers.edu*
[2] *Ask Jeeves, Inc., Piscataway, NJ 08854, USA. e-mail: sunxd@math.rutgers.edu*

Abstract. The Tree Knapsack Problem (TKP) is a 0–1 integer programming problem where hierarchy constraints are enforced. If a node is selected for packing into the knapsack, all the ancestor nodes on the path from the root to the selected node are packed as well. One apparent application of this problem is the simplification of computer graphics models. Real applications also use alternative representations of the nodes or whole subtrees, called impostors, to provide simplified trees that are visually acceptable. To account for this simplification, we introduce a generalized TKP, called Exclusive Multiple Choice Tree Knapsack Problem (EMCTKP). We present a dynamic programming algorithm to solve EMCTKP and a heuristic, called Lazy Iterative Arrangement, which reuses previous EMCTKP solutions to solve new instances of the problem. We show that this algorithm and heuristic reduce significantly the computation time of EMCTKP problems when changes in their parameters have spatial and temporal coherence. We also compare our algorithm with commercial integer programming solvers, and show that in our case the computation time grows linearly with the size of the problem tree and the available resources, while for generic IP solvers it is unpredictable and varies over a wide range of values.

Mathematics Subject Classifications (2000): 90-08, 90C10, 90C90.

Key words: structured data simplification, dynamic programming, virtual worlds.

1. Introduction

The tree knapsack problem (TKP) is a generalization of the 0–1 knapsack problem (KP) with additional constraint of partial ordering represented by a rooted tree. If a node is selected, then all the ancestor nodes on the path from the root to this node must be selected as well. In this paper, we consider an extension of TKP called exclusive multiple choice tree knapsack problem (EMCTKP). In EMCTKP, associated with each node there may be several simplifications of the subtree rooted at the node, which we call *impostors*. When an impostor is selected, none of the descendants of this node can be selected. We use EMCTKP to model a class of optimization problems in virtual reality environments. The impostor constraint is quite natural in this kind of problems and we believe EMCTKP may be useful for optimization problems in other domains.

⋆ Most of the work was done while at Rutgers University and Institute for Advanced Study.

Since both EMCTKP and TKP include KP as a special case, they are clearly NP-complete [3]. On the other hand, TKP can be solved using standard dynamic programming (DP) procedure in $O(n \cdot R^2)$, where n is the number of nodes and R is the given capacity or resource. Johnson and Niemi [5] introduced a "left–right" DP algorithm for TKP, which can find the optimal value C^* with a running time of $O(n \cdot C^*)$. Cho and Shaw [1] improved on this approach and gave a better DP algorithm with running time $O(n \cdot R)$. Shaw and Cho [9] also proposed a branch-and-bound (B&B) procedure for the TKP and their experiments showed that it has very good performance.

In this paper, we make two contributions to the computation of EMCTKP: (1) we extend the DP algorithm of Cho and Shaw [1] to EMCTKP while keeping the running time at $O(n \cdot R)$, where n is the number of nodes and impostors in the tree; and, (2) we propose a heuristic called Lazy Iterative Arrangement (LIA) which reduces the running time when the same EMCTKP with varying parameters is computed repeatedly. Since TKP is a special case of EMCTKP, our second contribution also applies to TKP.

Funkhouser and Sequin [2] proposed a greedy algorithm, which is half-optimal but only works for non-hierarchical scenes. For hierarchical structures such as trees, no greedy algorithm can guarantee a near-optimal solution. Mason and Blake [7] proved that half-optimality is feasible when the benefit measure has diminishing returns.

Samphaiboon and Yamada [8] studied the Precedence-Constrained Knapsack Problem (PCKP), which extends TKP by replacing the underlying rooted tree for an acyclic graph. They presented an exact algorithm with some heuristics. Their algorithm for PCKP in the worst case may take time exponential in n, the number of nodes, while our algorithm for EMCTKP takes time linear in n.

The paper is organized as follows. We define EMCTKP in Section 2, then present our DP algorithm for EMCTKP in Section 3. In Section 4, we present the Lazy Iterative Arrangement heuristic. Experimental results are given in Section 5. We conclude the paper in Section 6.

2. Application Background and Definition of EMCTKP

An important application of optimal tree simplification is the transmission and rendering of graphics scenes in environments that are constrained in system resources. The graphics tree is called *scene graph*, where each node represents an individual object or part of the scene. Semantics are embedded in the form of properties of nodes, such as geometric shape, color, position, etc.

In order to describe the problem, it is necessary to define some concepts.

DEFINITION 1. An impostor of a node v_i of a tree is a node that constitutes an alternative representation of the subtree rooted at v_i. The set of impostors of a node v_i is $\text{Impostors}(v_i)$. See Figure 1 for an illustration.

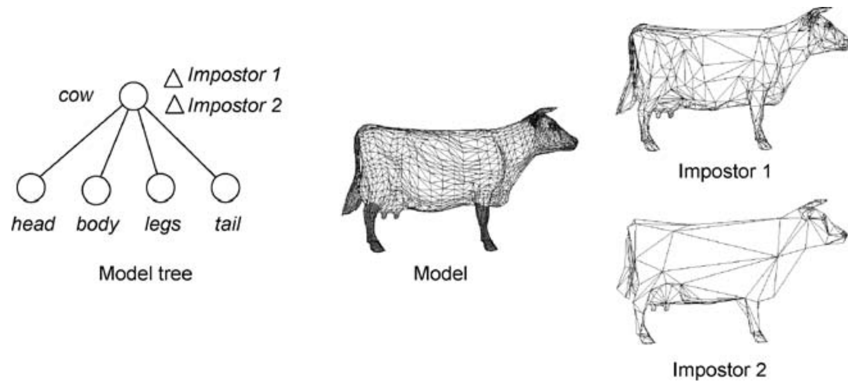

Figure 1. A three-dimensional model and two polygonal impostors.

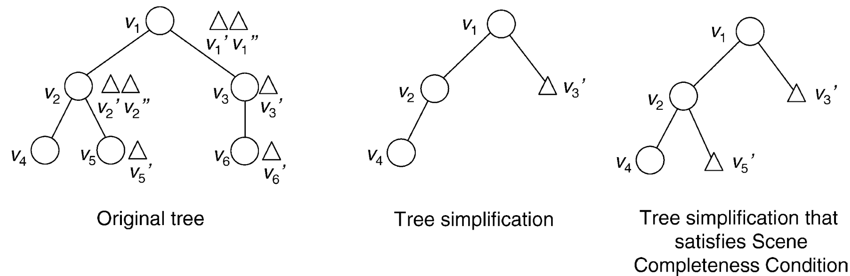

Figure 2. Examples of tree simplification.

DEFINITION 2. A simplified tree $T' = (V', E')$ of a tree $T = (V, E)$ is a tree such that V' is: (1) *closed under the predecessor* [9], i.e., $v_i \in V'$ implies predecessor$(v_i) \in V'$; and, (2) any leaf node $v_i' \in V'$ corresponds to a node $v_i \in V$ or an element in Impostors(v_i). See Figure 2.

DEFINITION 3. A simplified tree $T' = (V', E')$ satisfies *scene completeness condition* (SCC), if for any leaf node v_k whose path from the root is $v_0, v_1, \ldots, v_k$ (v_0 is the root), either $v_0, v_1, \ldots, v_k \in V'$ or there exists a node v_i ancestor of v_k, such that $v_0, v_1, \ldots, v_i' \in V'$ where $v_i' \in$ Impostors(v_i).

Simplified trees are representatives of virtual worlds, where the visible elements, i.e., polygons, are stored in the leaves, while the interior nodes are used to define coordinate frames and represent hierarchy of objects. We see in Figure 2 that the tree in the middle does not satisfy SCC, since the node v_5 has no visual representation. A possible representation for v_5 is either v_5 itself, v_5', or any of the impostors of its ancestors. The tree at the right does satisfy SCC.

The problem of determining the best possible simplification for a given tree can be seen as optimization under constraints, where each node has a set of impostors, and each node or impostor is assigned a benefit and resource cost value. The benefit value b represents the contribution of a given node or impostor to the overall "fi-

delity" of the simplification, whereas the resource cost value r represents the size of data needed to represent the node.

As mentioned above, virtual environments are represented as directed trees. Let parent(i) denote the parent node of node i. Each node is assigned a benefit b and a resource cost r. The simplification problem can be represented as a 0–1 integer-programming problem [4], where the solution is a subtree, rooted in the same root node as the original tree, with maximum total benefit. In case there are no impostors, this leads to the following formulation of the Tree Knapsack Problem (TKP):

$$\begin{aligned}
&\max \sum_i b_i \cdot x_i \\
&\text{Subject to:} \\
&\quad \sum_i r_i \cdot x_i \leqslant R, \\
&\quad x_{\text{parent}(i)} \geqslant x_i, \qquad (1)\\
&\quad x_i = 0 \text{ or } 1,
\end{aligned}$$

where the constraint (1) guarantees that the set of nodes V' of the simplified tree is *closed under predecessor*. The solution of the problem is defined as follows: Node v_i belongs to the simplified tree if and only if $x_i = 1$.

In the case where some nodes contain one or more impostors, additional constraints must be added to the problem. Without loss of generality, we assume that each node has no more than one impostor. An input tree that contains nodes with multiple impostors can be transformed into a tree whose nodes have at most one impostor by adding new interior nodes to the scene graph for each additional impostor. The added nodes have no benefit and no cost, i.e., both are equal to zero. Please note that the number of nodes in the new tree is at most the number of nodes and impostors in the old tree. Other than this the problem is unaltered. This transformation is illustrated in Figure 3. For the sake of simplicity, we will present the formulation of the EMCTKP problem and our algorithms for trees with at most one impostor at each node. However, our algorithms can be easily extended to trees with at most one impostor at each node and, with the transformation, can be extended to trees with arbitrary number of impostors at each node.

Let r_i and b_i be the cost and benefit of node v_i, respectively, and r_i' be the cost of the impostor of node v_i, and b_i' its benefit. Let R be the total available resources. R, b_i, b_i', r_i, and r_i' are all positive integers. Let

$$x_i = \begin{cases} 1 & \text{if node } v_i \in V', \\ 0 & \text{otherwise} \end{cases}$$

and

$$y_i = \begin{cases} 1 & \text{if the impostor of node } v_i \in V', \\ 0 & \text{otherwise.} \end{cases}$$

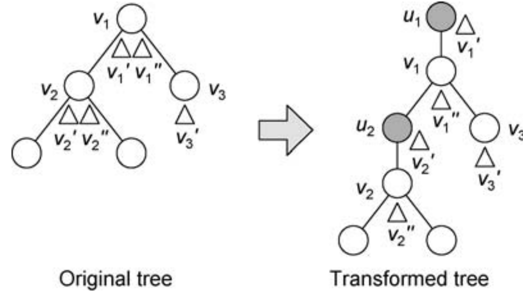

Figure 3. Transformation of tree with multiple impostors per node to tree with exactly one impostor per node. Nodes are represented by circles and impostors by triangles. Nodes u_1 and u_2 are added to represent extra impostors. Since these nodes have no cost, the problem is unaltered.

Then, we have the following optimization problem:

$$\max\left\{\sum b_i \cdot x_i + \sum b_i' \cdot y_i\right\} \tag{2}$$

Subject to:

$$\sum r_i \cdot x_i + \sum r_i' \cdot y_i \leqslant R, \tag{3}$$

$$x_i + y_i \leqslant 1, \tag{4}$$

$$x_{\text{parent}(i)} \geqslant x_i + y_i, \tag{5}$$

$$x_i,\, y_i = 0 \text{ or } 1. \tag{6}$$

This problem is an extension of TKP with two choices at each node, where $x_i = 1$ means that node v_i is selected, and $y_i = 1$ means that the impostor of node v_i is selected. Expression (2) is the benefit function to be maximized, and constraint (3) is the constraint on resources. Constraint (4) is the *exclusivity* constraint, such that the selection of a node and an impostor is mutually exclusive. Constraint (5) is the *closed under predecessor* constraint, which this time also ensures that if either a node or its impostor is selected, then its predecessor must be selected as well. We call this problem *Exclusive Multiple Choice Tree Knapsack Problem* (EMCTKP). Note that if the impostor of a given node is selected, none of its descendants can be selected. This constraint can be defined as follows:

$$x_i + y_{\text{parent}(i)} \leqslant 1. \tag{7}$$

This constraint, however, does not need to be stated explicitly, since it can be derived from the constraints (4), (5) and (6) as follows.

PROPOSITION 1. (4)*,* (5) *and* (6) *imply* (7).

Proof. Assume that $x_i = y_{\text{parent}(i)} = 1$ as opposed to (7). Then, from (4) we have $x_{\text{parent}(i)} = 0$, and from (5) this implies $x_i = 0$, which is a contradiction. □

3. The Dynamic Programming Algorithm

As already mentioned, the above optimization problems are NP-complete. There are practical algorithms that can solve TKP exactly in a reasonable amount of time, for trees of size up to few hundreds of nodes [1, 9]. Greedy algorithms have been considered as practical solutions for this kind of problems, because their computation time is in the order of $O(n \cdot \log n)$. However, for Tree Knapsack Problems, greedy algorithms are not guaranteed to provide near-optimal solutions and the result can be arbitrarily bad.

For this reason, based on the algorithm for TKP in [1], we designed a dynamic programming (DP) algorithm that solves EMCTKP in pseudo-polynomial time. Inherent to DP algorithms is reusing the partial solutions for multiple instances of constraints (in our case resources). We refer to this algorithm as DPR (Dynamic Programming on Resources).

Let tree $T = (V, E)$ be the one in the definition of EMCTKP. We assume that the nodes are labeled in DFS (depth-first search) order starting from 0. For $k \leqslant n$, let $W = W(k) = 0, 1, \ldots, k$ be the subset of V representing nodes labeled (visited) up to the node k and $T_W = (W, E_W)$ be the induced subtree of T. For a given resource r and a given node $v \in W$, we define:

$$P_W(v, r) = \max\left\{ \sum_{0 \leqslant i \leqslant k} b_i \cdot x_i + \sum_{0 \leqslant i \leqslant k} b'_i \cdot y_i \right\}$$

Subject to:

$$x_v + y_v = 1,$$

$$\sum_{0 \leqslant i \leqslant k} r_i \cdot x_i + \sum_{0 \leqslant i \leqslant k} r'_i \cdot y_i \leqslant r,$$

$$\text{for } 0 < i, j \leqslant k \begin{cases} x_i + y_i \leqslant 1, \\ x_{\text{parent}(i)} \geqslant x_i + y_i, \\ x_i, y_i = 0 \text{ or } 1. \end{cases}$$

Then $\max\{P_V(0, R), 0\}$ is the optimal value of EMCTKP.

To find $P_V(0, R)$, we use DFS approach proposed by Johnson and Niemi [5] and Cho and Shaw [1]. The algorithm finds the optimal value $P_V(0, R)$ in $O(n \cdot R)$ time by applying the following recursive procedures (illustrated in Figure 4):

1. (Initialization)

$$P_{W(0)}(0, r) = \begin{cases} b_0, & \text{if } r_0 \leqslant r \leqslant R, \\ 0, & \text{otherwise.} \end{cases}$$

2. (Forward move to expand the set of labeled nodes) For $k \neq 0$ and for each $r = 0, 1, \ldots, R$,

$$P_{W(k)}(k, r) = \begin{cases} P_{W(k-1)}(\text{parent}(k), r - r_k) + b_k, & \text{if } \sum_{j \in \text{path}[0,k]} r_j \leqslant r, \\ 0, & \text{otherwise.} \end{cases}$$

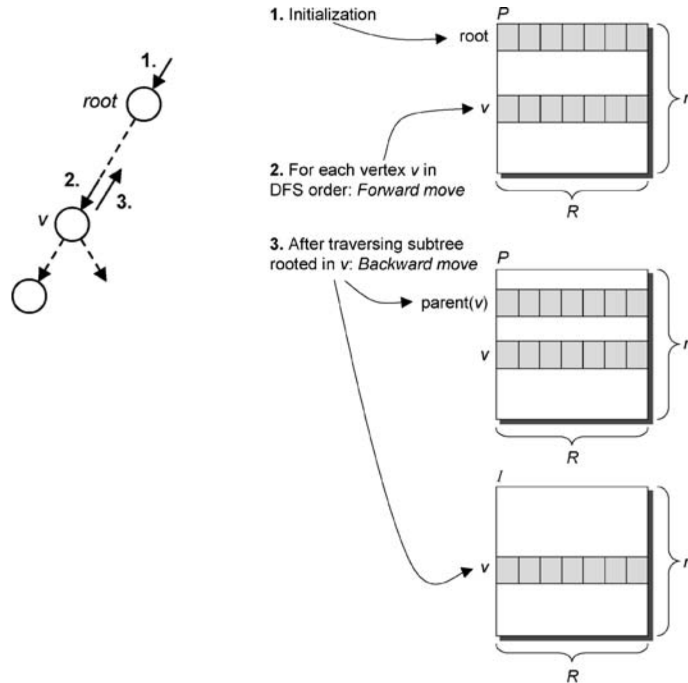

Figure 4. Dynamic programming algorithm for exact EMCTKP.

3. (Backward move to revisit labeled nodes) Let $k \neq 0$ and $W = W(k-1) \cup T(k)$. For each $r = 0, 1, \ldots, R$,

$$P_W(\text{parent}(k), r) \\ = \max\left\{P_{W(k-1)}(\text{parent}(k), r),\ P_{W(k-1)}(\text{parent}(k), r - r'_k) + b'_k,\ P_{W(k)}(k, r)\right\}.$$

For each node k and a given resource r, what is finally stored as $P(k, r)$ will be $P_W(k, r)$, where $W = \{0, 1, \ldots, k-1\} \cup T(k)$ and $T(k)$ is the subtree rooted at k. We use an index table I and assign $I(k, r)$ to indicate its role in the solution that gives $P_W(0, r)$. $I(k, r) = 0$ means $x_k = y_k = 0$; $I(k, r) = 1$ means $x_k = 1$, $y_k = 0$; and $I(k, r) = 2$ means $x_k = 0$, $y_k = 1$, respectively, in the optimal solution that achieves $P_W(k, r)$ for $W = \{0, 1, \ldots, k-1\} \cup T(k)$. The algorithm then finds the values of x_k, $y_k (k = 0, 1, \ldots, n)$ by tracing the index table I in a fashion similar to the algorithm of Cho and Shaw.

3.1. PSEUDO-CODE OF THE DPR ALGORITHM

The following is the pseudo-code for the DPR algorithm. The procedure *Optimal_Value_EMCTKP* builds the P and I tables according to the recursive procedures defined above. It differs from the algorithm of Cho and Shaw in the way the tables are built while performing the *backward_move* procedure, which considers three cases as described above: At a given node v we either include its impostor v',

the node itself, or do not include it at all. Note also the additional check at the end of this procedure: It is possible that the impostor of the root node provides better benefit for a given amount of resources, in which case it is selected.

The procedure *Optimal_Solution_EMCTKP* traverses the table I to determine the optimal solution, in a way similar to the algorithm of Cho and Shaw [1]. Our algorithm differs in that we consider an additional case where $I(k, r) = 2$, which represents the selection of an impostor that contracts the whole subtree below it.

ALGORITHM *Optimal_Value_EMCTKP*

Input: parent(i), b_i, r_i, b'_i, r'_i, R, for $i = 0, 1, 2, \ldots, n$ (nodes are labeled in depth-first-search order)

Output: $P_v(0, r)$ and $I(i, r)$ for all $i \in V$ and for all $r = 0, 1, 2, \ldots, R$

```
begin
    %% comment: Initialization
    r_min := min{r_j | j ∈ V};
    for r := r_min to r_0 − 1 do
        P(0, r) := −∞; I(0, r) := 0;
    for r := r_0 to R do
        P(0, r) := b_0; I(0, r) := 1;
    %% comment: Main Loop
    r_path := r_0;
    for k := 1 to n do
    begin
        Forward_Move(k)
        if (k is a leaf node) then
            v := k;
            do
                Backward_Move(v);
                v := parent(v);
            while ((v has no successor i such that i > k) and (v ≠ 0))
        endif
    end
    %% comment: Multiple choice of the node 0
    if (P(0, r'_0) < b'_0) then P(0, r'_0) = b'_0; I(0, r'_0) = 2;
end

Procedure Forward_Move(u);
begin
    r_path := r_path + r_u;
    for r := r_min to r_path − 1 do P(u, r) := −∞;
    for r := r_path to R do
    begin
        P(u, r) := P(parent(u), r − r_u) + b_u;
    end
end
```

```
Procedure Backward_Move(u);
begin
    r_path := r_path − r_u;
    r'_path := r_path + r'_u;
    for r := r_min to r'_path − 1 do P'(u, r) := −∞;
    for r := r'_path to R do
    begin
        P'(u, r) := P(parent(u), r − r'_u) + b'_u;
    end
    for r := r_min to R do
    begin
        if (P(parent(u), r) ⩾ P(u, r) and P(parent(u), r) ⩾ P'(u, r)) then
            I(u, r) := 0;
        else if (P(u, r) ⩾ P'(u, r)) then
            P(parent(u), r) = P(u, r)
            I(u, r) := 1;
        else
            P(parent(u), r) = P'(u, r)
            I(u, r) := 2;
        endif
    end
end
```

Algorithm Optimal_Solution_EMCTKP

Input: $I(i, r)$ for all $i \in V$ and for all $r = 0, 1, 2, \ldots, R$

Output: $x(i)$ for all $i \in V$ (value 0 means the vertex is not selected, value 1 means the vertex is selected, value 2 means the imposter of the vertex is selected)

```
begin
    for i := 1 to n do x(i) := 0;
    label node 0;
    i := n;
    r := R;
    tail := −1;
    %% comment: Main Loop: keep deleting the "tail"
    while (i > 0 and r > 0) do
    begin
        while (i is unlabeled) do
        begin
            if (I(i, r) = 0) then
                i := i − 1;
                tail := −1;
            else
                if (tail = −1 or I(i, r) = 2) then
                    tail := i;
```

```
                    end if
                    label node i;
                    i := parent(i);
                end if
            end
            if (tail = −1) then
                tail := i;
                x(tail) := 1;
                r := r − r_tail;
            else
                if (I(tail, r) = 1) then
                    x(tail) := 1;
                    r := r − r_tail;
                else
                    x(tail) := 2;
                    r := r − r'_tail;
                endif
            endif
            i := tail − 1;
            tail := −1;
        end
end
```

3.2. ANALYSIS

For a tree with arbitrary number of impostors at each node, let n be the number of nodes and impostors. We first transform the tree to one with at most n nodes and then apply our algorithms to the new tree. The running time for the algorithm *Optimal_Value_EMCTKP* is $O(n \cdot R)$ since it traverses each link of the tree twice (forward and backward moves) and each move on the link takes time $O(R)$. The running time for the algorithm *Optimal_Solution_EMCTKP* is $O(n)$ since it follows the reverse or the DFS order and the time cost at each node is $O(1)$. As for the memory space cost, the tables at each node cost $O(R)$ and the total cost of space is $O(n \cdot R)$.

3.3. HANDLING THE SCENE COMPLETENESS CONDITION

PROPOSITION 2. *A tree simplification problem with SCC can be reduced to a TKP instance.*

Proof. Let us consider the tree T as a problem tree for an EMCTKP problem. Then, each node in T has exactly one impostor. Now, we build the tree T^{scc} as a problem tree for optimal simplification that must satisfy the scene completeness condition. The idea of the transformation is to represent at each node the simplest set of impostors that satisfy such condition, as follows (see Figure 5).

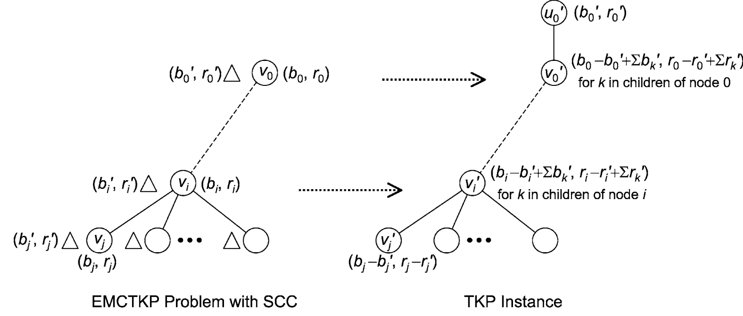

Figure 5. Transformation of an EMCTKP Problem with SCC to a TKP instance.

We begin by creating a root node u'_0 with the benefit and resource values (b'_0, r'_0), which is the simplest impostor that satisfies SCC, since it is the impostor for the entire tree. The next node v'_0 is obtained from the root node v_0 of T, such that it represents the next complete set of impostors. That is, it replaces the impostor of v_0 by the node v_0 and each of the impostors of its children, i.e., v'_0 is defined such that its benefit and resources values are $(b_0 - b'_0 + \sum b'_k, r_0 - r'_0 + \sum r'_k)$, for all children v_k of root v_0, and is added as a child of u'_0.

Then, for each node $v_i \in T$ we do the same, maintaining the hierarchical structure of the original tree, so that the node $v'_i \in T^{\text{scc}}$ is defined as having the benefit and resource values $(b_i - b'_i + \sum b'_\ell, r_i - r'_i + \sum r'_\ell)$, for all children v_ℓ of node v_i. For the case of a leaf node v_j, the benefit and resource values are defined as $(b_j - b'_j, r_j - r'_j)$.

We see that any subtree of T^{scc} satisfies the SCC, since every node has the simplest complete set of impostors for its subtree, and the nodes in any path incrementally exchange the impostor of a lower resolution for a set of impostors of higher resolution, i.e., when adding resource and benefit values, the values for the impostors of internal nodes are cancelled out, and the total resources account for the resources of internal nodes and for the resources of the impostors at the leaves of the subtree.

Since we do not need to specify impostors for any node in T^{scc}, the subtree of T^{scc} with optimal benefit can be obtained by solving TKP. □

We believe that in real applications, some simplifications might require SCC. Our dynamic simplification, as shown next, also applies to such cases.

4. Dynamic Simplification

In virtual reality environments, the benefit or resource metrics of nodes in the tree change due to the changing user interest or dynamic simulation and it is necessary to compute the EMCTKP problem repeatedly with varying parameters. Hence, it would be beneficial if some parts of the old solution could be reused. For our algorithm it is possible to reuse parts of old solutions, based on the following observation.

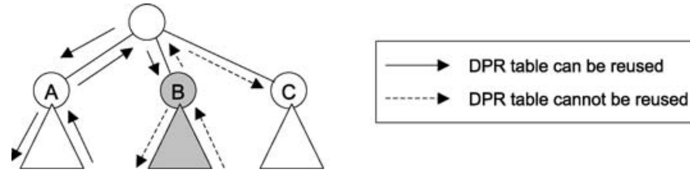

Figure 6. Dynamic simplification, where node B already has been updated. DPR table values of the nodes traversed with the solid line are not affected by the update, whereas those in the dashed line must be re-computed. The algorithm can skip the subtree below node A during the computation of the new solution, and use the results obtained in the previous computation.

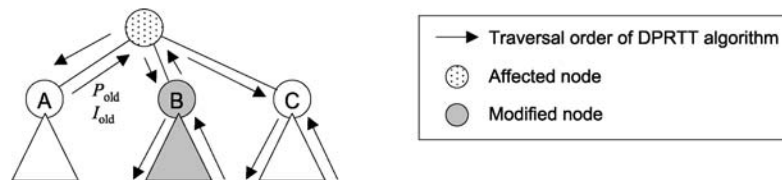

Figure 7. DPRTT traversal. Note that the subtree rooted at node A is skipped during the computation since it is not modified by the update on B. Conversely, the subtree rooted at C needs to be traversed, although is not marked as "modified," since the update on B affects the computation of the solution for the branches to the right.

OBSERVATION 1. Let us assume that the DFS order always takes the left branch first. The main problem with reusing solutions across the tree topology is that an update on a given branch affects the stored solution of the branches to the right, so it cannot be reused. Conversely, the solution to the left of this branch can be reused completely, as illustrated in Figure 6.

In order to adapt to parameter changes in the tree, we extend the DPR algorithm to reuse solutions also along the tree topology. This algorithm is referred to herein as DPRTT (Dynamic Programming on Resources and Tree Topology). For this algorithm, we assume that the values of tables P and I of the last computation of DPR are kept in memory as P_{old} and I_{old}, and nodes that have been modified since the last computation of DPR are marked as "modified." We also assume that nodes in the path from a modified node to the root are marked as "affected," since they need to be traversed in order to reach the modified nodes. Then, the DPRTT algorithm is built by adding functionality in the DPR algorithm as follows. When the algorithm traverses a node which is not marked as "affected" or "modified," it can skip the traversal of the subtree below it and reuse the parts of the previous tables. This is illustrated in Figure 7. The pseudocode for DPRTT is as follows:

ALGORITHM *Optimal_Value_EMCTKP_Dynamic*

Input: parent(i), b_i, r_i, b'_i, r'_i, R, for $i = 0, 1, 2, \ldots, n$ (nodes are labeled in depth-first-search order)

Output: $P_v(0, r)$ and $I(i, r)$ for all $i \in V$ and for all $r = 0, 1, 2, \ldots, R$

begin

%% comment: Initialization

$r_{\min} := \min\{r_j \mid j \in V\}$;

```
for r := r_min to r_0 − 1 do
        P(0, r) := −∞;  I(0, r) := 0;
for r := r_0 to R − 1 do
        P(0, r) := b_0;  I(0, r) := 1;
%% comment: Main Loop
r_path := r_0;
for k := 1 to n do
begin
        if k is marked as modified then
              reuse_flag = false
        endif
        if k is not marked as modified or affected and reuse_flag = true then
              copy row P_old(k) in P(k)
              copy rows I_old(j) in I(j) for all j descendants of k, including k
              mark k as reused
        else
              Forward_Move(k);
        endif
        if (k is a leaf node || k is marked as reused) then
              v := k;
              do
                    Backward_Move(v);
                    v := parent(v);
              while ((v has no successor i such that i > k) and (v ≠ 0))
              if k is marked reused then
                    k = k+ Descendants(k)
              endif
        endif
end
%% comment: Multiple choice of the node 0
if (P(0, r_0() < b_0() then P(0, r'_0) = b'_0;  I(0, r'_0) = 2;
end
```

where **Descendants**(u) is a procedure that returns the number of nodes in the subtree rooted at node u. This is used in the algorithm to avoid traversing those subtrees whose solutions can be reused.

A key problem with this approach is that if updates occur repeatedly in the leftmost branch of the tree, it is not possible to reuse any part of the tree. We can overcome this issue by invoking the following observation.

OBSERVATION 2. The solutions obtained by DPR and DPRTT do not depend on the DFS order in which the tree is traversed to find those solutions.

Based on this observation, the algorithm can be improved as follows: After computing the simplified tree, it is possible to rearrange the DFS order of the tree for the next computation such that modified nodes appear on the rightmost branches. In the following section we present an approach to perform such arrangement, which we call Lazy Iterative Arrangement.

Note that the DPRTT algorithm and the arrangement heuristic have a larger impact when updates have spatial and temporal coherence. Updates in benefit or resources metric are said to exhibit *spatial coherence* if the updated nodes are relatively close to each other, i.e., changes are likely to be localized in a subtree rather than dispersed across the entire tree. Similarly, updates are said to have *temporal coherence* if for a node that has been updated, it is likely that the same node will to be updated soon again.

LAZY ITERATIVE ARRANGEMENT

In order to reduce the running time of our DPRTT algorithm we propose an algorithm, called Iterative Arrangement, which rearranges the depth-first-search (DFS) order of the tree. The goal of the algorithm is to move the modified branches of the tree to the right-hand side of the tree. This way, the DPRTT algorithm can reuse a large part of the previous solutions, since further updates are more likely to be performed on those branches.

We use a flag "modified" to mark the nodes whose parameters were modified since the last iteration of the algorithm, and a flag "affected" to mark the nodes on the path between a "modified" node and the root. Since our algorithm can reuse the result of the old solution on the unmarked nodes before hitting a node marked "modified" in the DFS order, we use the following iterative algorithm to find a DFS order that maximizes the reuse which reduces the running time.

ALGORITHM *Iterative_Arrangement*

Input: A marked tree T

Output: A DFS order of the tree and the degree of reuse (the number of unmarked nodes with DFS order lower than any node marked "modified")

begin

v := RootOf(T);
For w, an unmarked child of v, assign the lowest DFS order available;
For w, a child of v marked "modified," assign the highest DFS order available;
For w, a child of v marked "affected," run Iterative Arrangement algorithm on Tree(w);
Order all "affected" children of v in a descendent order of their degree of reuse and assign lower DFS order to the child with a higher degree of reuse.

return sum of the sizes of subtrees rooted at unmarked children plus the degree of reuse of the "affected" child of the lowest rank

end

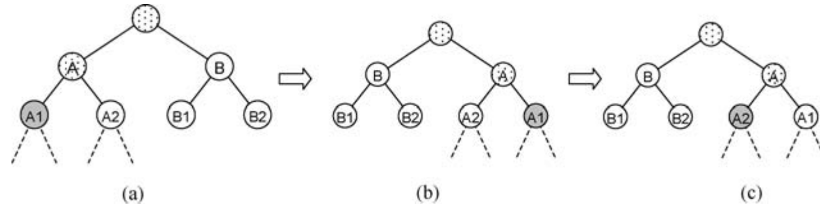

Figure 8. Example of an iterative arrangement: (a) node A1 is marked as "modified" and nodes A and the root are marked as "affected," while other nodes remain unmarked; (b) the resulting tree after the iterative arrangement. Further updates on nodes below A1 will require a low optimization cost; (c) here node A2 is marked as "modified." In this case, the degree of reuse of A2 may be less than the specified threshold, so we do nothing. This is the lazy iterative arrangement.

Let n be the size of the tree. The cost of the algorithm is mainly due to the sorting of degrees of reuse for the children of the internal nodes. So, running time of the algorithm is $O(\sum_v n_v \cdot \log n_v)$ where n_v is the number of children of v. Since $\sum_v n_v = n - 1$, the running time of the algorithm is at most $O(\sum_v n_v \cdot \log n) = O(n \cdot \log n)$.

In the last step of the iteration, we may assign the lowest DFS order to the "affected" child that saves the largest computation time (highest degree of reuse) and not care about the order of other "affected" children. The running time of the algorithm is then reduced to $O(n)$.

Different criteria may be used to set the "modified" flag on a node, such as the recent history of modifications of the node. Since the profile of a past modification may not always correctly predict future modifications, we suggest lazy arrangements, i.e., we set a threshold of reuse and only perform *Iterative_Arrangement* if the degree of reuse is greater than the threshold. An example of iterative arrangement and lazy iterative arrangement is shown in Figure 8.

5. Experimental Results

We apply the above algorithms for simplification of complex 3D virtual scenes. In this context, the benefit metric of a particular node corresponds to the fidelity of the simplification, according to a set of (perceptual) criteria defined by the user. Examples of such criteria are: *size*, in terms of dimensions in the virtual world; *accuracy* of the representation, which measures how similar impostors are compared to the original object; *focus-of-attention*, which assigns greater importance to objects near the user's view, and semantics, which accounts for the intrinsic importance of some object types. The resource metric corresponds to the cost of representing a particular scene node. In out test scenario, we measure resources as the number of polygons.

Virtual reality scenes have an inherent spatial and temporal coherence. This is because the tree topology is used to represent composition relationships, thus

the elements in the same subtree are usually closer to each other than to elements in a different subtree. In addition, the tree topology is often used to represent a spatial subdivision of a 3D scene. This type of virtual worlds has a greater spatial consistency.

We classified the virtual scenes according to their spatial organization into three categories:

Quadtrees these are the scenes organized in a recursive spatial subdivision, such that the virtual world is partitioned into quadrangular regions (*quads*) of the same size, each of which is partitioned into smaller quads, and so on, down to some predefined depth. The topology of this type of scenes is a tree of degree 4 and depth d.

Grids these are the scenes whose elements are arranged in a grid of size $m \times n$.

Arbitrary these are the scenes whose elements are positioned arbitrarily in a virtual region.

It can be seen that scenes in these categories range from uniformly high coherence (quadtrees) to occasionally high coherence (arbitrary). This gives us both optimistic and pessimistic performance bounds of the algorithm, respectively.

For testing our algorithm, we created a number of scenes from each of these categories, varying the main parameter in each case: The depth for *quadtrees*, the dimensions for grids, and the number of elements in the case of *arbitrary* scenes. The leaf nodes of the virtual scenes are the nodes that contain the graphical properties information, while the internal nodes are used to represent hierarchical relationships between the different objects. In our test scenes, leaf nodes were populated randomly from a collection of predefined virtual objects, mostly buildings.

Table I shows the summary information for the test datasets. Note that corresponding scenes in each category have a similar distribution of resources and benefit. This is because they represent to some extent the same distribution of objects in the 3D environment, although arranged differently in hierarchical structure. Note also the differences in benefit values: For instance, benefit values in *city4_3* are in the range [0, 808] while most of the other scenes have values in the range [0, 3000]. This occurs because of the *focus-of-attention* parameter in the computation of benefit values. According to this parameter, objects which are completely within the view extents are given a maximum importance (3000 in our case), while objects partially in the view have a lower importance proportional to the distance between the object's center and the view's center. In the case of *city4_3*, it happens that no object is entirely covered by the user's view. Benefit and resource values b_i, b'_i, r_i, r'_i are independent. Although for a set of impostors of the same node it is true that increasing resources results in increasing benefit, this does not necessarily extend to the entire tree. We see that the importance heuristic may give more benefit to an object with few polygons in the middle of the area of interest, while other objects not directly of interest might have a larger amount

Table I. Statistic information for input data from the test scenes

Scene		Max	Min	Mean	StdDev
	r_i	5576	0	347.67	1061.56
quadtree_2	b_i	3000	0	82.02	295.38
(depth = 2)	r_i'	1117	0	75.21	180.42
	b_i'	2887	24	168.49	365.89
	r_i	5576	0	340.6746	1122.3762
quadtree_3	b_i	808	0	40.059525	77.6373
(depth = 3)	r_i'	1117	2	77.44265	197.38295
	b_i'	686	24	85.575	73.766235
	r_i	5576	0	328.35883	1088.3132
quadtree_4	b_i	3000	0	37.838596	86.366486
(depth = 4)	r_i'	1117	2	74.00621	189.54845
	b_i'	3000	24	81.03906	98.53059
Scene		Max	Min	Mean	StdDev
	r_i	5576	0	448.6526	1342.831
grid 4 × 4	b_i	989	0	69.43221	169.8539
	r_i'	1117	2	99.81761	238.6829
	b_i'	885	24	139.6101	179.5135
	r_i	5576	0	396.5945	1229.072
grid 8 × 8	b_i	3000	0	48.83614	159.4594
	r_i'	1117	2	85.91821	212.0276
	b_i'	2887	24	101.7423	193.3379
	r_i	5576	0	353.2278	1165.436
grid 16 × 16	b_i	3000	0	39.76489	90.61723
	r_i'	1117	2	78.65595	202.9349
	b_i'	2887	24	83.97665	100.9245
Scene		Max	Min	Mean	StdDev
	r_i	5576	0	287.07144	925.2833
arbitrary_16	b_i	100	0	99.20635	8.873285
	r_i'	1035	2	62.7193	152.72362
	b_i'	100	24	76.62573	25.870586
	r_i	5576	0	422.0894	1297.554
arbitrary_64	b_i	3000	0	54.81702	209.4886
	r_i'	1117	2	94.37715	230.6186
	b_i'	3000	24	113.1017	264.5567
	r_i	5576	0	350.44617	1129.7971
arbitrary_256	b_i	3000	0	41.30176	112.21889
	r_i'	1117	2	78.31136	197.21097
	b_i'	3000	24	85.87992	125.77382

Table II. Size of the test datasets

Scene	n	R_{TOTAL}
quadtree_2	127	44421
quadtree_3	503	171374
quadtree_4	2025	663944
grid_4 × 4	117	52866
grid_8 × 8	475	188477
grid_16 × 16	1913	674865
arbitrary_16	123	32604
arbitrary_64	469	198083
arbitrary_256	1913	675840

Table III. Percentage of modified nodes during navigation

Scene	% modified nodes (mean)	Scene	% modified nodes (mean)	Scene	% modified nodes (mean)
quadtree_2	10.16051	*grid_4×4*	9.892528	*arbitrary_16*	11.6144
quadtree_3	3.976143	*grid_8×8*	3.789474	*arbitrary_64*	3.933534
quadtree_4	0.844047	*grid_16×16*	0.852016	*arbitrary_256*	0.894085

of resources. The benefit indicates also the accuracy of a given impostor. For a regularly shaped object, an impostor of considerably lower resources might still very much resemble the original object, i.e., the accuracy of the representation is high. Conversely, a reduction of an object of irregular shape might result in an impostor of low quality, i.e., the benefit value drops dramatically.

Table II shows the relative sizes of the test datasets, in terms of number of nodes n and the total amount of resources, i.e., the sum of r_i for all nodes i in the problem trees.

Changes in the benefit values are obtained by modifying the *focus-of-attention* parameter described above. This parameter is defined as a point (x, y, z), which is moved in a predefined path within the boundaries of the virtual scene. This movement is intended to represent a typical user navigation around the virtual scene. The mean number of modified nodes for the different test scenes is shown in Table III as a percentage of the total number of nodes for each tree. Given that the dynamic changes simulate typical user navigation, the number of modified nodes is practically the same for all data sets, which can be seen from Tables II and III, i.e., the simulation shows the result of a user navigating at roughly the same speed for smaller and larger environments. Having the same navigation conditions helps us to measure the impact of the dynamic algorithm with respect to the tree size.

Table IV. Computation time (msec) for DPRTT with lazy iterative arrangement vs. DPR algorithms, for Quadtrees

d	n	R	DPRTT/LIA	DPR
2	127	1000	4.6	8.028571
		5000	21.86667	39.60952
		10000	42.4	79.02857
		30000	112.93	242.97
3	503	1000	13.06132	31.23585
		5000	59.69811	159.5047
		10000	115.566	305.6321
		50000	621.6132	1605.392
4	2025	1000	35.58028	124.7041
		5000	166.9289	625.7959
		10000	329.039	1234.33
		50000	—*	—*

*The problem could not be solved due to insufficient computer memory.

Table V. Computation time (msec) for DPRTT with lazy iterative arrangement vs. DPR algorithms, for Grids

Dimensions	n	R	DPRTT/LIA	DPR
4×4	117	1000	4.598039	7.960784
		5000	21.16667	35.19608
		10000	42.7451	71.40196
		30000	117.73	223.31
8×8	475	1000	15.42534	29.76471
		5000	66.12669	146.3439
		10000	136.1041	297.9321
		50000	794.6652	1537.484
16×16	1913	1000	46.21678	123.8438
		5000	274.4802	587.0769
		10000	529.5851	1180.142
		50000	—	—

The algorithm was initially coded in the Java language and ran on a 2.2 GHz Intel XEON P4, with 1 GB RAM. Table IV shows the results of the experiments for *Quadtrees* with respect to the depth of the tree d. Table V shows the results of the experiments for *Grids* with respect to the dimensions of the grid, and Table VI shows the results of the experiments for *Arbitrary* scenes with respect to the

Table VI. Computation time (msec) for DPRTT with lazy iterative arrangement vs. DPR, for arbitrary scenes

Number of subscenes	n	R	DPRTT/LIA	DPR
16	123	1000	5.320755	7.518868
		5000	24.32076	37.88679
		10000	48.79245	76.33962
		30000	135.42	242.91
64	469	1000	17.48039	29.41667
		5000	75.51471	146.2206
		10000	152.9216	288.0784
		50000	786.2206	1474.103
256	1931	1000	43.97436	116.9821
		5000	258.5949	705.1564
		10000	527.0256	1175.956
		50000	—	—

number of subscenes (random virtual objects). We also show the resulting size of the tree and different settings for R (1000, 5000, 10000 and 50000). Note that for the smaller data sets, the maximum R is 30000, given that the limit of $R = 50000$ exceeds the total amount of resources required for the entire tree. The resulting size of the tree is not that of a perfectly balanced quadtree or grid. This is because the virtual objects used to populate the leaf nodes of the tree are themselves subtrees of different complexity.

We also compared our algorithm with a generic integer-programming solver, CPLEX 8.1.0 (`http://www.cplex.com`), on the same datasets as above. In order to obtain comparable results, we re-wrote the algorithm in C++, given that we found differences in performance that were unfavorable for the Java version. This is presumably due to the memory management of the Java virtual machine and the lack of optimizations that makes the C++ version faster. Table VII shows the results for our dynamic algorithm, compared to CPLEX 8.1.0. In many cases, CPLEX took very long time to compute the optimal solution, so a time limit of 5 seconds was introduced. (Observe, for instance, that the worst case is reported as approximately 5 secs for some of the datasets; in reality, the algorithm takes longer to compute the exact solution.) The results were obtained in a 2.1 GHz Pentium IV with 512 MB RAM. The missing entries (—) show the cases where the problem could not be solved due to insufficient computer memory.

From these tables, we make the following observations:

(i) DPRTT/LIA is consistently faster than DPR, even on arbitrary scene instances. This is due to the inherent coherence of virtual worlds and user navigation.

Table VII. Comparison of our algorithm vs. CPLEX 8.1.0

Scene	R	DPRTT/LIA			CPLEX		
		Best	Average	Worst	Best	Average	Worst
Quadtree_2	1000	0	3.1555555	16	0	35.2	484
	5000	0	17.333334	47	0	114.0889	1016
	10000	15	37.488888	109	0	20.84444	47
	20000	31	77.8	265	0	14.73333	47
Quadtree_3	1000	0	9.166667	32	16	1775.044	5031
	5000	15	44.58889	188	31	165.7889	5031
	10000	31	92.41111	438	31	1207.933	5047
	20000	62	195.83333	1016	31	1367.189	5047
Quadtree_4	1000	0	25.51111	156	266	3731.244	5250
	5000	46	113.69444	813	250	904.4334	2250*
	10000	109	237.70555	1859	266	1546.35	5313
	20000	—	—	—	282	556.5222	4219
Grid 4 × 4	1000	0	2.7777777	16	0	216.5111	1844
	5000	0	17.355556	47	0	21.88889	63
	10000	15	33.377777	94	15	22.2	47
	20000	31	72.888885	218	0	22.91111	78
Grid 8 × 8	1000	0	10.8	32	15	763.6667	5047
	5000	15	49.8	187	31	143.6333	3969
	10000	46	103.12222	391	31	630.2889	5047
	20000	125	238.7111	890	47	1100.156	5047
Grid 16 × 16	1000	15	23.36111	125	141	1239.072	5438
	5000	78	150.43889	656	250	519.4722	2250*
	10000	187	318.56668	1593	266	329.6833	547
	20000	—	—	—	281	334.6889	516
Arbitrary 16	1000	0	3.7575758	16	0	54	828
	5000	0	16.575758	47	15	21.57576	47
	10000	15	36.484848	110	0	148.8485	1469
	20000	46	77.15151	235	0	15.30303	31
Arbitrary 64	1000	0	11.534091	32	31	160.0227	5047
	5000	15	49.875	172	31	510	5062
	10000	46	110.75	406	31	2006.523	5047
	20000	109	247.01137	860	47	648.6023	5047
Arbitrary 64	1000	15	24.660606	125	250	846.2727	2266*
	5000	93	171.67273	672	266	561.5819	2266*
	10000	172	371.81213	1657	281	504.0303	5282
	20000	—	—	—	297	354.9826	438

*For these values, the time limit was set to 2 instead of 5 seconds.

(ii) DPRTT/LIA is faster on quadtrees than on grid scenes, as expected. This is due to the quadtree topology, which provides a higher coherence than in grids. In addition, as the depth of the quadtree grows, DPRTT/LIA improvement over DPR grows faster compared to the case where the dimensions of the grid increase. This favors the use of quadtrees or similar structures (such as octrees) for building virtual worlds.

(iii) Memory requirements are roughly the same. DPRTT/LIA only adds the requirement of storing the tables of the previous computation, which are of size $n \cdot R$.

(iv) For larger scenes it can become prohibitive to run DPRTT/LIA or DPR, due to insufficient computer memory. In practical applications such as virtual world simplification, this is usually done on the visible part of a scene, which is typically a small part of the entire scene. The sizes and topologies of the test scenes are entirely representative of real-world applications.

(v) CPU time for the DPR/LIA algorithm appears to grow linearly with R. This is an indication of the result of applying iterative arrangement. The algorithm reuses as much as it can from the previous solution, so on the average it behaves as if the DPR algorithm was run on a tree of smaller size. Given that DPR is linear with respect to R (while n is constant), then DPR/LIA is also expected to be linear, but with a smaller slope.

(vi) For small amount of available resources, our algorithm consistently outperforms generic integer-programming solvers. We also see that this difference is noticeable for the larger datasets. Indeed, the key difference is the consistency of the results of our algorithm, which increases linearly with R and n, compared to CPLEX, where the variation in time is considerably larger, and in some cases, it was not possible to obtain a provable optimal solution in the allotted time. Considering this, we can say that our algorithm performs better than generic integer-programming solvers. This is expected, given that specialized solutions are likely to perform better than generic solutions. However, we must consider the drawbacks of our algorithm when dealing with larger datasets and limit in resources. First, it appears from the results that while computation time grows almost linearly with R for our algorithm, CPLEX is less sensitive to such changes, i.e., at some point, CPLEX might outperform our algorithm in the *best* case. Second, our algorithm might require a considerable amount of memory, and computations times might be large due to virtual memory paging for the dynamic programming tables.

6. Conclusions

We have presented the EMCTKP problem as a generalization of the Tree Knapsack Problem and a dynamic programming algorithm to solve it exactly, that runs in $O(n \cdot R)$ time. We also showed that in some dynamic scenarios, spatial and temporal coherence could be exploited to improve the computation time. We have developed an algorithm and a heuristic, called Lazy Iterative Arrangement, which improves on the dynamic programming algorithm by rearranging the order in which the solution

is found in order to reuse past solutions. We have shown that the algorithm can be used for trees with up to a few thousand nodes in real-time, in regular computing environments. The test problems and the results are available online at:

http://www.caip.rutgers.edu/disciple/emctkp/

As part of the future work, we plan to explore other approaches, such as branch-and-bound algorithms, in order to solve larger-size problems.

Acknowledgements

The authors gratefully acknowledge an anonymous reviewer's suggestion for inclusion of Proposition 1 as well as its proof.

The research was supported by the NSF grant ANI-01-23910, US Army CECOM Contract No. DAAB07-02-C-P301, and by the Rutgers Center for Advanced Information Processing (CAIP) and its corporate affiliates. Xiaodong Sun was also supported under the NSF grant CCR-99-87845 while at School of Mathematics, Institute for Advanced Study, Princeton, NJ 08540.

References

1. Cho, G. and Shaw, D. X.: A depth-first dynamic programming algorithm for the tree knapsack problem, *INFORMS J. Computing* **9**(4) (1997), 431–438.
2. Funkhouser, T. and Sequin, C. H.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments, in *Proceedings of the SIGGRAPH Computer Graphics Annual Conference*, ACM Press, New York, pp. 99–108.
3. Garey, M. R. and Johnson, D. S.: *Computers and Intractability. A Guide to the Theory of NP-Completeness*, Freeman and Company, San Francisco, CA, 1979.
4. Ignizio, J. P. and Cavalier, T. M.: *Linear Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1994.
5. Johnson, D. S. and Niemi, K. A.: On knapsacks, partitions, and a new dynamic programming technique for trees, *Math. Oper. Res.* **8**(1) (1983), 1–14.
6. Martello, S. and Toth, P.: *Knapsack Problems: Algorithms and Computer Implementations*, Wiley, New York, 1990.
7. Mason, A. E. W. and Blake, E. H.: A graphical representation of the state spaces of hierarchical level-of-detail scene descriptions, *IEEE Trans. Visual. Computer Graph.* **7**(1) (2001), 70–75.
8. Samphaiboon, N. and Yamada, T.: Heuristic and exact algorithms for the precedence-constrained knapsack problem, *J. Optim. Theory Appl.* **105**(3) (2000), 659–676.
9. Shaw, D. X. and Cho, G.: The critical-item, upper bounds, and a branch-and-bound algorithm for the tree knapsack problem, *Networks* **31**(4) (1998), 205–216.