

Partitioning Activities for Agents^{*†}

Fatma Özcan and V.S. Subrahmanian

Department of Computer Science
University of Maryland, College Park
{fatma,vs}@cs.umd.edu

Abstract

There are now numerous agent applications that track interests of thousands of users in situations where changes occur continuously. [Shim *et al.*, 1994] suggested that such agents can be made efficient by merging commonalities in their activities. However, past algorithms cannot merge more than 10 or 20 concurrent activities. We develop techniques so that a large number of concurrent activities (typically over 1000) can be partitioned into components (groups of activities) of small size (e.g. 10 to 50) so that each component's activities can be merged using previously developed algorithms (e.g. [Shim *et al.*, 1994]). We first formalize the problem and show that finding optimal partitions is NP-hard. We then develop three algorithms - *Greedy*, *A*-based* and *BAB* (branch and bound). *A*-based* and *BAB* are both guaranteed to compute optimal solutions. *Greedy* on the other hand uses heuristics and typically finds suboptimal solutions. We implemented all three algorithms. We experimentally show that the greedy algorithm finds partitions whose costs are at most 14% worse than that found by *A*-based* and *BAB* — however, *Greedy* is able to handle over thousand concurrent requests *very fast* while the other two methods are much slower and able to handle only 10-20 requests. Hence, *Greedy* appears to be the best.

1 Introduction

The number of software agents deployed on the Internet has grown dramatically over the last few years. Major corporations have agents that continuously track events — some corporations use such agents to identify and exclude IP addresses that “scrape” data from their sites. Other corporations use agents that track some phenomenon (e.g. stock quotes) and take different actions for different clients who have registered parameters to be tracked and actions to be taken when certain

conditions are satisfied. Yet other corporations have agents that dynamically adjust plans (e.g. airlines) when events occur that cause those plans to go awry (e.g. a snowstorm).

In this paper, we focus on agents that have a high volume of activity. *Our work is not going to help (very much) agents that have a relatively low amount of activity.* If the agent can leverage common aspects across its diverse activities so as to reduce its load, its performance will improve greatly. Much work has been done on merging activities to minimize load — such work spans databases [Shim *et al.*, 1994] and planning [Yang *et al.*, 1992] — however the problem of merging such activities in an optimal way is known to be NP-complete, and hence these algorithms tend to perform well when 10 to 20 activities or fewer are being merged. However, agents tracking user interests for major news organizations like CNN have a huge number of users with registered interests, and an enormous number of changes occurring continuously. The same is true in the case of stock market agents.

In this paper, we develop *partitioning* techniques. Such techniques work as follows: when a large set S of activities (e.g. 1000) are waiting to be performed, then our methods partition S . Such a partition breaks S into disjoint components $S_1, \dots, S_n$ where $S = S_1 \cup \dots \cup S_n$. The activities in a single component may then be merged by methods for merging activities, whether they are queries or plans using methods such as those of [Shim *et al.*, 1994; Yang *et al.*, 1992]. The cost of executing a component is just the cost of executing the activities in the component *after merging* activities. The cost of a partition is the sum of the costs of the components. The cost (after merging) of the activities in a partition may be estimated using techniques such as those in [Shim *et al.*, 1994; Yang *et al.*, 1992]. We would like to find a partition with minimal (estimated) cost.

Section 2 formalizes this problem and shows that finding a partition with minimal expected cost is NP-hard. Section 3 develops three algorithms to address this problem. The first algorithm is based on the A^* algorithm — we introduce a heuristic function and show that it is admissible [Nilsson, 1986]. Hence, this algorithm is guaranteed to find a partition with minimal cost. The second algorithm is a greedy algorithm which is not guaranteed to find a minimal cost partition. The third algorithm is a branch and bound algorithm which is also guaranteed to find an optimal cost solution. We have implemented all the algorithms. Section 4 describes the results

^{*}Parts of this research were supported by Army Research Lab contract DAAL0197K0135, Army Research Office grant DAAD190010484 and DARPA/Rome Labs grant F306029910552

[†]Some proofs are omitted due to space limitations

of detailed experimentation with these algorithms. The experiments show that though the greedy algorithm finds suboptimal solutions (typically 14% worse savings are realized when compared to the optimal savings realizable), it is able to handle 1800 concurrent activities in about 16 seconds, while the other two algorithms are unable to handle a large number of activities.

2 Problem Definition

An agent may engage in some space of *activities*. For example, the space of activities associated with a database agent may be all SQL queries. The space of activities associated with a planning agent may be the set of all possible planning problems. The space of activities associated with a CNN-style news agent may be the set of all interests that could possibly be associated with users. In the sequel, we use $\mathcal{A}$ to denote some set of activities drawn from such an activity space. $\mathcal{A}$ denotes activities that the agent has to perform, but has not yet performed.

Definition 1 (Partition) A partition $\mathcal{P}$ of a set $\mathcal{A}$ of activities is a set $\{A_1, A_2, \dots, A_n\}$ where each A_i is a non-empty subset of $\mathcal{A}$ and

1. $\mathcal{A} = A_1 \cup A_2 \cup \dots \cup A_n$
2. $i \neq j \rightarrow A_i \cap A_j = \emptyset$.

Each A_i is called a **component** of the partition $\mathcal{P}$. When the first condition is replaced by $\mathcal{A} \supseteq A_1 \cup A_2 \cup \dots \cup A_n$, $\mathcal{P}$ is called a subpartition.

A partition $\mathcal{P}$ of $\mathcal{A}$ splits a set of activities (to be done) into components. The activities in each component A_i can be separately merged, perhaps using methods for query merging [Shim *et al.*, 1994] or plan merging [Yang *et al.*, 1992].

Definition 2 (Cost estimation function) A cost estimation function c takes a set of activities as input and returns a real number as output. c is required to satisfy at least the following axioms: (I) $c(\emptyset) = 0$, (II) $A_1 \subseteq A_2 \rightarrow c(A_1) \leq c(A_2)$.

Intuitively, $c(A_i)$ denotes the estimated cost of executing the activities in A_i after merging. When $A_i = \{a_i\}$ is a singleton set, we will abuse notation and write $c(a_i)$ instead of $c(\{a_i\})$. We will also assume that the computation of c takes polynomial time. This is consistent with algorithms to estimate merged costs of sets of queries such as those of [Shim *et al.*, 1994] as well as task merging methods [Yang *et al.*, 1992]. In this paper, capitalized A 's denote sets of activities. Lower case a 's denote individual activities.

Problem 1 (Activity Partitioning Problem (APP)) A set $\mathcal{A}$ of activities, a cost estimation function c , and a partition $\mathcal{P}$ of $\mathcal{A}$. Is it the case that there is no other partition $\mathcal{P}'$ of $\mathcal{A}$ such that $\sum_{A_i \in \mathcal{P}'} c(A_i) < \sum_{A_i \in \mathcal{P}} c(A_i)$?

We will prove below that APP is NP-hard by using the **zero-one multiple knapsack problem (MKP)** [Martello and Toth, 1987; Kellerer, 1999]. MKP can be stated as: Given a set N of n items, a set M of m knapsacks, profit and weight vectors p_j and w_j ($1 \leq j \leq n$), and the capacity vector c_i ($1 \leq i \leq m$),

$$\begin{aligned} & \text{subject to } \max \sum_{i \in M} \sum_{j \in N} p_j x_{i,j} \\ & \sum_{j \in N} w_j x_{i,j} \leq c_i \quad \forall i \in M \\ & \sum_{i \in M} x_{i,j} \leq 1 \quad \forall j \in N \\ & x_{i,j} \in \{0, 1\} \quad \forall i \in M, \forall j \in N \end{aligned}$$

The decision problem version of MKP takes in addition to the above parameters, an assignment of items to knapsacks. It returns true if the assignment is an optimal solution to the search problem and false otherwise.

Theorem 1 The activity partitioning problem is NP-hard.

Proof Sketch: We will transform the MKP decision problem to APP. Let m be the number of components in $\mathcal{P}$. Create three tables with the following database schemas: R_m (knapsackid, capacity), R_n (itemid, profit, weight) and R_p (knapsackid, itemid). Insert an entry for each item into R_n , and an entry for each knapsack into R_m . Create an entry in R_p for each item, knapsack pair present in the assignment α . Then, for each item i , we create an SQL query q_i given by:

```
select * from R_n, R_m, R_p
where R_p.itemid = i and R_n.itemid = R_p.itemid
and R_m.knapsackid = R_p.knapsackid and
sum(R_n.weight) < R_m.capacity
```

Let the cost, $c(q_i) = -\sum_{j \in q_i} p_j$.

It is easy to see that the tables and queries can be constructed in polynomial time. Therefore, when we solve this instance of APP, we will minimize the total cost of the queries. As a result, the sum of the profits will be maximized in the solution. Hence, APP is NP-hard. $\square$

3 Activity Partitioning Algorithms

Although, we can transform the MKP problem into APP, we cannot immediately use the available approximation algorithms [Martello and Toth, 1987; Kellerer, 1999] for the MKP problem. This is because in MKP the profits of items are constants, that is they do not vary with different knapsacks. However, in our case, the cost of a set of activities not only depends on which component it belongs to, but also depends on which other activities are present in the component. Hence, we will describe new algorithms based on different heuristics to solve the query partitioning problem. We will start with an A^* -based algorithm, which is guaranteed to find an optimal solution.

3.1 A^* -based Algorithm

To adapt the A^* algorithm [Nilsson, 1986], we first need to define the state space, the cost function g and the heuristic function h .

Definition 3 (State) A state s is any subpartition of $\mathcal{A}$. State s is a goal state if it is a partition of $\mathcal{A}$. Finally, the start state $s_0 = \emptyset$.

Definition 4 (Functions $g(n)$, $h(n)$ and $f(n)$) Suppose node n has state $s = \{A_1, A_2, \dots, A_m\}$, and let

$incr(a, s) = \min\{\min\{c(a \cup A_i) - c(A_i) \mid 1 \leq i \leq m\}, c(a)\}$. Then, $g(n)$ and $h(n)$ are defined as follows:

$$\begin{aligned} g(n) &= \sum_{i=1}^m c(A_i) \\ h(n) &= \min\{incr(a, s) \mid a \in \mathcal{A} - (\bigcup_{i=1}^m A_i)\} \\ f(n) &= g(n) + h(n) \end{aligned}$$

where c is the cost estimation function.

Intuitively, $g(n)$ says the cost of node n is the sum of the costs of the individual components. $h(n)$ underestimates the cost to a goal state. This is done as follows. Consider each activity a that is in $\mathcal{A}$ but that is not in the subpartition associated with node n . Such an activity can either be placed in one of the components of node n or may be in a new component. Evaluate the cost of each of these alternatives and choose the minimal increase $incr_a$ in cost for adding a to node n . To obtain a partition, every activity that is not in the current subpartition must be added to node n eventually, so the cost of a solution must be at least greater than the current cost by $incr_a$ for all such activities a . This provides the rationale for $h(n)$.

Our A^* -based algorithm is given in Figure 1. The algorithm makes use of a function, **pick**($s, \mathcal{A}$), that chooses an activity from $\mathcal{A}$ which has not been assigned to any of the components in state s .

If the while loop of the algorithm terminates, this implies that no goal state was reached and hence there is no solution. Note that if the heuristic function h used by the A^* algorithm is admissible, then the algorithm is guaranteed to find the optimum solution [Nilsson, 1986]. The heuristic h is admissible iff $h(n) \leq h^*(n)$ where $h^*(n)$ is the lowest actual cost of getting to a goal node from node n . The following result shows that our heuristic is admissible.

Theorem 2 (Admissibility of h) For all nodes n , $h(n) \leq h^*(n)$. Hence, A^* -based algorithm finds an optimal partition of $\mathcal{A}$.

Theorem 3 The heuristic function h satisfies the monotone restriction.

The monotone restriction requires that for all n , $h(n) \leq h(n') + (g(n') - g(n))$. Since, the h function satisfies the monotone restriction, the first goal state found by the A^* -based algorithm is guaranteed to be the optimal solution.

3.2 Greedy Algorithm

Our greedy algorithm uses the notion of a cluster graph to solve the activity partitioning problem. Given a set $\mathcal{A}$ of activities, a cluster graph is a weighted graph whose vertices are *disjoint sets* of activities. Given any set $\mathcal{A}$ of activities, we may associate with it, a *canonical cluster graph*.

Definition 5 (Canonical Cluster Graph ($\mathcal{CCG}$)) A cluster graph for a set $\mathcal{A}$ of activities is an undirected weighted graph where:

1. $V = \{\{a_i\} \mid a_i \in \mathcal{A}\}$,
2. $E = \{(\{a_i\}, \{a_j\}) \mid a_i, a_j \in \mathcal{A} \text{ and } c(a_i) + c(a_j) - c(\{a_i, a_j\}) > 0\}$,
3. $w(\{a_i\}, \{a_j\}) = c(a_i) + c(a_j) - c(\{a_i, a_j\})$

Intuitively, when we have an edge between activities a_i, a_j then this means that there is some savings to be derived by

```

 $A^*$ -based( $\mathcal{A}$ )
/* Input:  $\mathcal{A}$  (a set of activities) */
/* Output:  $\mathcal{P}$  if one exists, NIL otherwise */

OPEN :=  $\emptyset$ 
forall  $a_i \in \mathcal{A}$  do
   $n_i.state := \{\{a_i\}\}$ 
  compute  $g(n_i)$  and  $h(n_i)$ 
  insert  $n_i$  into OPEN
sort OPEN in increasing order of  $f(n_i)$ 
while (OPEN  $\neq \emptyset$ ) do
   $n := \text{OPEN.head}$ 
  delete  $n$  from OPEN
   $s := n.state$ 
  if  $s$  is a goal state then Return( $s$ )
  else /* expand and generate children */
     $a_j := \text{pick}(s, \mathcal{A})$ 
    forall  $A_i \in s$  do
      /* insert  $a_j$  into  $A_i$  */
       $s' := \{A_1, \dots, A_{i-1}, A_i \cup \{a_j\}, \dots, A_m\}$ 
      create a new node  $n'$ 
       $n'.state = s'$ 
      compute  $g(n')$  and  $h(n')$ 
      insert  $n'$  into OPEN
    /* also create a new component with  $a_j$  */
    create a new node  $n'$ 
     $n'.state := \{A_1, \dots, A_m, \{a_j\}\}$ 
    compute  $g(n')$  and  $h(n')$ 
    insert  $n'$  into OPEN
  end(while)
Return (NIL)
End-Algorithm

```

Figure 1: The A^* -based Algorithm

merging the two activities together. The edge label, $w(a_i, a_j)$ denotes that saving. (Again, we abuse notation and write $e(a_i, a_j)$ instead of $e(\{a_i\}, \{a_j\})$ and $w(a_i, a_j)$ instead of $w(\{a_i\}, \{a_j\})$.)

The greedy algorithm is shown in Figure 2. Intuitively, **Greedy** tries to build a partition iteratively. In each iteration, it finds an edge with highest weight (savings) in the cluster graph and deletes it. If more than one such edge exists, one is arbitrarily picked. It examines the two activities associated with that edge and checks to see if either of those activities already occur in a partition. There are four cases to check depending on whether one, both or neither activities occur in an existing partition. If both a_i, a_j occur in existing different components, then it may be possible to move one of them from one component into the other. This should be done only if it yields savings. If one of a_i, a_j is in an existing component but not the other, then the other can be placed in the same component. It is easy to see that the number of iterations of the loop of this algorithm is $O(\text{card}(\mathcal{A})^2)$. Each execution of the loop may take $O(\text{card}(\mathcal{P}))$ time in the worst case to check if a_i, a_j occur in a component. It is therefore easy to see that **Greedy** is certainly polynomial in the size of the input $\mathcal{A}$ as long as

the cost estimation function c runs in polynomial time.

```

Greedy( $\mathcal{CCG}$ )
/* Input:  $\mathcal{CCG}$  (canonical cluster graph of  $\mathcal{A}$ ) */
/* Output:  $\mathcal{P}$  */

 $\mathcal{P} := \emptyset$ 
 $n := 0$ 
while ( $\exists e \in \mathcal{CCG}$ ) do
   $e := (a_i, a_j)$ , s.t.  $w(e)$  is maximum in  $\mathcal{CCG}$ 
  delete  $e$  from  $\mathcal{CCG}$ 
  if ( $\exists A_l \in \mathcal{P}$  s.t.  $a_i \in A_l$ ) then
    if ( $\exists A_k \neq A_l \in \mathcal{P}$  s.t.  $a_j \in A_k$ ) then
      if ( $c(A_l \cup \{a_j\}) + c(A_k - \{a_j\}) < c(A_l) + c(A_k)$ )
        then delete  $a_j$  from  $A_k$ 
        insert  $a_j$  into  $A_l$ 
      if ( $c(A_k \cup \{a_i\}) + c(A_l - \{a_i\}) < c(A_l) + c(A_k)$ )
        then delete  $a_i$  from  $A_l$ 
        insert  $a_i$  into  $A_k$ 
    else /*  $\nexists A_k$  s.t.  $a_j \in A_k$  */
      insert  $a_j$  into  $A_l$ 
    else /*  $\nexists A_l$  s.t.  $a_i \in A_l$  */
      if ( $\exists A_k \in \mathcal{P}$  s.t.  $a_j \in A_k$ ) then
        insert  $a_i$  into  $A_k$ 
      else
         $n := n + 1$ 
         $A_n := \{a_i, a_j\}$ 
         $\mathcal{P} := \mathcal{P} \cup \{A_n\}$ 
  end(while)
Return ( $\mathcal{P}$ )
End-Algorithm

```

Figure 2: The Greedy Algorithm

3.3 Branch and Bound Algorithm

In this section, we describe a branch and bound algorithm, which is similar to the A^* -based algorithm, but uses a different search strategy.

The BAB algorithm maintains *nodes* n which having the following fields: *state* (as defined in Def. 3), *gval* and *uval* defined below. *gval* specifies the value $g(n)$ (cf. Def. 4), while *uval* contains $u(n)$ which is defined below.

Definition 6 Let node n have state $s = \{A_1, A_2, \dots, A_m\}$, then $u(n) = \sum_{a_i \in \mathcal{A}'} c(a_i)$, where $\mathcal{A}' = \mathcal{A} - \bigcup_{i=1}^m A_i$, and c is the cost estimation function.

Intuitively, $g(n) + u(n)$ overestimates the cost of the minimal cost solution reachable from node n .¹ **BAB** (Figure 3) also starts out with an empty partition and uses the same expansion strategy as the A^* -based algorithm. The OPEN list which is organized in ascending order of $g(n) + u(n)$ and the **BAB** algorithm always chooses the first node in OPEN for expansion. If the g value of this first node exceeds that of the best solution

¹Contrast this with $h(n)$ presented earlier that represents an underestimate.

```

Branch&Bound( $\mathcal{A}$ )
/* Input:  $\mathcal{A}$  (set of activities) */
/* Output:  $\mathcal{P}$  */

create a new node  $n$ 
 $n.state := \{\}$ 
 $n.uval := \sum_{a_i \in \mathcal{A}} c(a_i)$ 
 $n.gval := 0$ 
insert  $n$  into OPEN
 $bestValue := n.gval + n.uval$ 
 $bestPartition := n.state$ 
while (OPEN  $\neq \emptyset$ ) do
   $n := \text{OPEN.head}$ 
  delete  $n$  from OPEN
  if ( $n.gval < bestValue$ ) then
    if ( $n.gval + n.uval < bestValue$ ) then
       $bestValue := n.gval + n.uval$ 
       $bestPartition := n.state$ 
  if ( $n$  is not a goal state) then
    /* expand and generate children */
     $a_j := \text{pick}(s, \mathcal{A})$ 
    forall  $A_i \in n.state$  do
      /* insert  $a_j$  into  $A_i$  */
       $s' := \{A_1, \dots, A_{i-1}, A_i \cup \{a_j\}, \dots, A_m\}$ 
      create a new node  $n'$ 
       $n'.state = s'$ 
      compute  $n'.gval$  and  $n'.uval$ 
      insert  $n'$  into OPEN
    /* also create a new component with  $a_j$  */
    create a new node  $n'$ 
     $n'.state := \{A_1, \dots, A_m, \{a_j\}\}$ 
    compute  $n'.gval$  and  $n'.uval$ 
    insert  $n'$  into OPEN
  end(while)
Return ( $bestPartition$ )
End-Algorithm

```

Figure 3: The Branch and Bound Algorithm

found so far, then we do not need to expand this node. If however, the g -value is less than that of the best solution found so far, then there is a chance that this node leads to a better solution and hence we expand it (unless it is a solution in which case we discard it).

Theorem 4 **BAB**($\mathcal{A}$) finds an optimal partition of $\mathcal{A}$.

4 Implementation/Experimental Results

We have implemented all three algorithms described in this paper in C++ on a Sun Ultra1 machine with 320 MB memory running Solaris 2.6. There is a total of 1600 lines of code for our implementation and experiments. We ran two sets of experiments — one to determine which algorithm was fastest, and how many concurrent activities could be handled, and another to determine how the algorithms performed in terms of savings generated by them.

Experiment 1: We first fixed an *overlap probability*. Given a pair of activities from $\mathcal{A}$, the overlap probability gives the

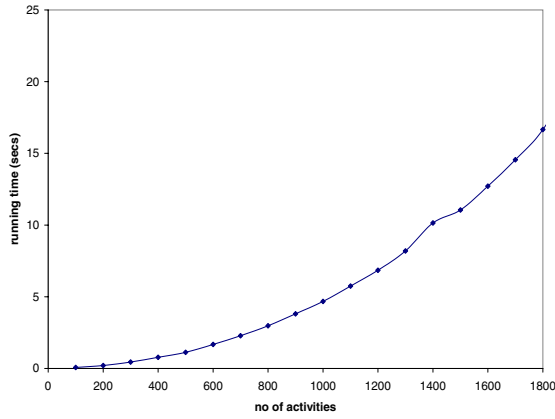


Figure 4: Scalability of the **Greedy Algorithm**

probability that the pair will overlap. We then fixed an **overlap degree** which measures the savings obtained when two activities overlap. Suppose we know activities a_1, a_2 overlap. The **overlap degree** in this case is $\frac{c(a_1) + c(a_2) - c(\{a_1, a_2\})}{\min(c(a_1), c(a_2))}$. Intuitively, the larger the overlap degree, the greater the savings obtained by merging two activities.

Once an overlap degree and overlap probability are fixed, we automatically generated sets of activities that have that overlap degree (on the average) and that overlap probability. We then ran all three algorithms on the data. For each number of activities, we took an average over 10 sets of data. Table 1 shows the results of experiments when overlap probability varies between 0.4 and 0.6 and when the overlap degree is 0.6 (1).

No. of activities	<i>Greedy</i>	<i>BAB</i>	<i>A*-based</i>
5	2.2	81.1	68.3
6	2.75	417.1	328.5
7	2.8	1570.5	1420.7
8	3.3	6752	9941.2
9	4.7	5243.4	18281.9
10	6	10109	44782.6

Table 1: Execution Times (milliseconds) (Overlap_Degree=0.6)

It turns out that both *A*-based* and *BAB* ran out of memory when the number of activities exceeded about 10. The reason for this is that the OPEN list quickly grows overwhelmingly large. Hence, the above table has only 10 activities. In contrast, *Greedy* did not run out of space. In order to see how well *Greedy* could do, we continued to run experiments by increasing the number of activities. Figure 4 shows that *Greedy* scales well and can handle 1800 concurrent activities in about 16 seconds.

Experiment 2: As both *A*-based* and *BAB* compute optimal solutions, but *Greedy* does not always do so, we wanted to see how much “worse” the solution produced by *Greedy* was compared to the optimal solution. When we have a set $\mathcal{A}$ of activities, and a partition $\mathcal{P}$ of $\mathcal{A}$, the savings realized by $\mathcal{P}$ is given by $(\sum_{a \in \mathcal{A}} c(a)) - (\sum_{A_i \in \mathcal{P}} c(A_i))$.

Figure 5 shows that in our experiments, optimal solution produced by *A*-based* (whose savings equals that produced

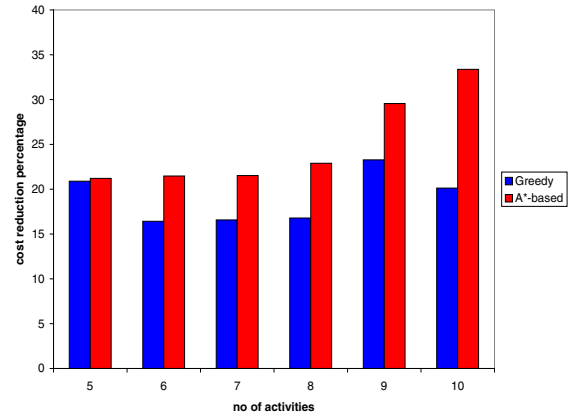


Figure 5: Cost Reduction Percentage Comparison

by *BAB*) yielded a saving of 21–34%, while *Greedy* yields savings of 17–23%.

Although we could not get the optimal solution after 10 activities, we still ran the *Greedy* algorithm up to 1800 activities to see the quality of partitions it produced. Figure 6 shows the results. As seen from the graph, the performance of the *Greedy* algorithm did not degrade, and stayed about the same as we increased the number of activities in the input sets.

Both *A*-based* and *BAB* are unable to perform *at all* when there are more than 10–20 concurrent activities. On larger sets of activities, they both quickly run out of memory — in contrast, *Greedy* scales up very effectively when many activities occur. In a few seconds, it can partition over a thousand concurrent activities, and produce a solution that is at most 40% worse than the optimal.

5 Related Work and Conclusions

Over the last few years, there has been tremendous interest in collaborative information agents. Techniques from this fertile field have found ample applications in online news sources that monitor user interests and changing news events and create personalized news reports. They have also been used extensively to make trades in financial markets for different users as stock conditions change. They have been used to dynamically bid for a variety of clients in online auctions [Schwartz and Kraus, 1997]. There are many impressive agent systems — these include *Internet Softbot* [Etzioni and Weld, 1994], *Retzina* [Decker *et al.*, 1997], *Infosleuth* [Bayerdo *et al.*, 1997], *SIMS* [Arens *et al.*, 1993] and many others.

There have been important efforts to optimize the performance of such agents. For instance, [Adali *et al.*, 1996; Ashish *et al.*, 1999] show how to use intelligent caching methods to improve the performance of systems that access different data sources.

In both planning and databases, researchers have noticed that when a server receives numerous requests, then it may make sense to leverage commonalities across those requests instead of serially processing the requests. This is accomplished by merging (planning requests or query requests). Important advances in this field were made by [Yang *et al.*, 1992] and [Shim *et al.*, 1994]. However, in both cases, the problem of merging plans and merging queries to minimize expected

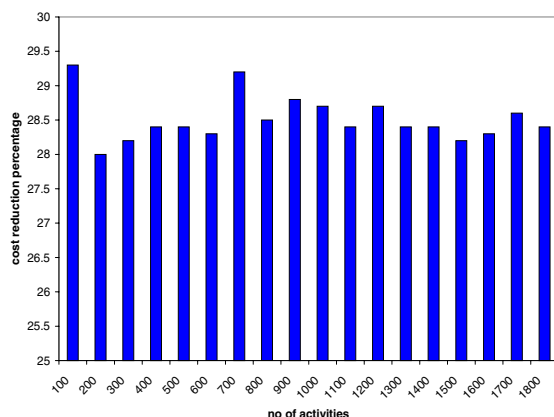


Figure 6: Cost Reduction Percentage of the **Greedy Algorithm**

cost of execution is NP-hard and hence, these methods worked well when the numbers of activities (planning or querying) were limited. [Shim *et al.*, 1994] conducted experiments with 10 concurrent queries, while [Yang *et al.*, 1992] handled up to 18 concurrent planning problems.

In this paper, we build on these important advances. Specifically, we start by assuming the existence of some way of merging k activities or fewer. Thus, if the activities are database queries, then k may be set to 10 and the [Shim *et al.*, 1994] algorithm may be used. Alternatively, if the activities are planning problems, the perhaps one could set k to 18 and use the methods of [Yang *et al.*, 1992] to merge plans. Now consider the situation where such a database or planning agent has hundreds or thousands of activities it is currently engaged in. Such a set of activities may be partitioned into components of size k (or less) so that each component's activities can be merged using an aforementioned merging algorithm such as [Yang *et al.*, 1992; Shim *et al.*, 1994]. If the sum of the costs of the individual components of the partition is minimized, then we would have an optimal way of executing such a large set of activities.

Unfortunately, this problem — which we call the activity partitioning problem (APP) in this paper — is shown to be NP-hard. As a consequence, there is no *exact* algorithm to solve this problem in polynomial time (unless $P = NP$ which is widely believed to not be the case). We propose two exact algorithms, *A*-based* and *BAB*, to solve this problem. As these algorithms compute exact solutions, they are easily seen to take exponential time. As expected from earlier results of [Yang *et al.*, 1992; Shim *et al.*, 1994], these algorithms stop performing well when over 10–20 activities are being merged. To alleviate this problem, we propose a *polynomial algorithm* called **Greedy** which exhibits the following nice properties. First, it can (in a matter of seconds) partition a set of thousands of activities which reflects an order of magnitude improvement over *A*-based* and *BAB*. Second, the partitions it computes turn out to have at most 14% worse savings than those computed by *A*-based* and *BAB*. As a consequence, we believe that **Greedy** is superior to both *A*-based* and *BAB* for real world problems.

References

- [Adali *et al.*, 1996] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.
- [Arens *et al.*, 1993] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent Cooperative Information Systems*, 2(2):127–158, 1993.
- [Ashish *et al.*, 1999] Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Selectively materializing data in mediators by analyzing user queries. In *Proceedings of the 4th IFICIS International Conference on Cooperative Information Systems, (CoopIS)*, pages 256–266, Edinburgh, Scotland, September 1999.
- [Bayardo *et al.*, 1997] R. Bayardo *et al.* Infosleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments. In J. Peckham, editor, *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 195–206, Tucson, Arizona, 1997.
- [Decker *et al.*, 1997] K. Decker, K. Sycara, and M. Williamson. Middle Agents for the Internet. In *IJCAI*, pages 578–583, Nagoya, Japan, 1997.
- [Etzioni and Weld, 1994] O. Etzioni and D. Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, 1994.
- [Kellerer, 1999] H. Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In *Proceedings of APPROX'99*, volume 1671 of *Lecture Notes in Computer Science*, pages 51–62, Berkeley, CA, 1999.
- [Martello and Toth, 1987] S. Martello and P. Toth. Algorithms for knapsack problems. In S. Martello, G. Laporte, M. Minoux, and C. Ribeiro, editors, *Annals of Discrete Mathematics 31: Surveys in Combinatorial Optimization*. North-Holland, 1987.
- [Nilsson, 1986] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, 1986.
- [Schwartz and Kraus, 1997] R. Schwartz and S. Kraus. Bidding Mechanisms for Data Allocation in Multi-Agent Environments. In *Intl. Workshop on Agent Theories, Architectures, and Languages*, pages 56–70, Providence, RI, 1997.
- [Shim *et al.*, 1994] K. Shim, T. K. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data and Knowledge Engineering*, 12(2):197–222, 1994.
- [Yang *et al.*, 1992] Q. Yang, D. Nau, and J. Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(2):648–676, 1992.