CS-TR-4416, UMIACS-TR-2002-96

Improving Performance of Agents by Activity Partitioning[*]

Fatma Özcan[†]
IBM Almaden Research Center
650 Harry Road
San Jose, California
fozcan@almaden.ibm.com

V.S. Subrahmanian
Department of Computer Science
University of Maryland
College Park, Maryland
vs@cs.umd.edu

October 29, 2002

Abstract

There is growing interest in software agents [11, 13, 10, 26, 6] that provide a variety of services to humans, other agents, and third party software applications. Some of these agents are engaged in hundreds of activities at any given time point. In such cases, agents may try to examine a set S of activities and leverage commonalities between them in order to reduce their load. We call this *activity merging*. Unfortunately, in most application domains, activity merging turns out to be NP-complete. Thus, for each application domain, there is an integer k (which varies from domain to domain) such that activity merging can merge up to k activities while satisfying the application's performance expectations. In this paper, we consider the problem of what to do when the set of activities exceeds k. Our approach partitions $\mathcal{A}$ into disjoint sets $A_1 \cup A_2 \cup \ldots \cup A_n$ such that each A_i contains at most k activities in it (thus the activities in each A_i can be merged using a merging algorithm). When creating such partitions, we would like to ensure that the activities inside each A_i share a lot of commonality, so that merging yields a lot of savings. In this paper, we propose two optimal algorithms (based on the A^* algorithm and the branch and bound paradigm), as well as numerous greedy algorithms to solve the problem. We have implemented these algorithms and conducted detailed experiments. The results point out which algorithms are most appropriate for scaling agent performance.

Index Terms: software agents, activity merging, activity partitioning, multiple activity optimization.

[*]Parts of this research were supported by Army Research Lab contract DAAL0197K0135, Army Research Office grant DAAD190010484, DARPA/Rome Labs grant F306029910552, and by the ARL CTA on advanced decision architectures.

[†]The work reported here was done while the author was at the University of Maryland, College Park

1 Introduction

The number of deployed software agent applications in the world has increased dramatically. For example, software agents are being used for tasking a group of sensors (cf. Brooks[7]) and for monitoring the results of the sensing activities. In the commercial world, there is growing interest in using agents for activities such as creation of multimedia presentations [23], for supply chain management related activities [30], for financial portfolio management activities[8] and health care applications[31].

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*. Some agent researchers argue that it is preferable to work with simple agents which can then be combined to form intelligent coalitions to attack semantically and computationally hard problems. Though such agents might perform only one specialized task, they may receive a large volume of requests. If the agent can leverage common aspects across its diverse activities so as to reduce its load, its performance will improve greatly. The idea of merging commonalities amongst diverse activities that an entity is performing has a long history in computer science. Disk servers [24] schedule reads not of one job at a time, but of a *set* of jobs precisely for this reason — when traversing the tracks and sectors on a disk, they might as well perform reads and writes for pending jobs (without having to revisit those sectors and tracks). The same is true of multimedia storage servers — when shipping data across the network, techniques such as adaptive piggy-backing[14, 3] attempt to merge commonalities between requests so as to ship media objects once rather than twice. Database servers do the same — rather than process one query at a time, it often makes sense to process multiple queries [27] and avoid doing any repetitive work. The same is true of AI planning systems as well where multiple planning requests may be merged to create a lower cost plan[32].

Despite the great interest in merging techniques, almost all the above merging problems are NP-hard. What this means is that merging is effective when the number of activities being merged is "not too large." Thus, for any given application domain a, there is some integer k_{a} which describes how many tasks can be successfully merged while still keeping within the performance expectations of the application. *In this paper, we address the problem of what do when the number of activities a* **single** *agent needs to engage in exceeds k_{a}, assuming that we have an existing method for merging.*

Simply put, suppose we are given a set $\mathcal{A}$ of activities that an agent a is involved in where $|\mathcal{A}| > k_{\mathsf{a}}$. Furthermore, suppose there exists a known method M_{a} for merging the activities engaged in by agent a. In our work, we would like to split the set $\mathcal{A}$ of activities into a partition $\mathcal{A} = A_1 \cup \ldots A_n$ such that activities that have common computations are grouped together in each A_i. Then, we can merge the activities in A_i using the merge algorithm M_{a}.

Clearly, there are many possible ways of finding such a partition. We would like to find a partition that is "optimal" in the sense that the total cost of performing the activities in $\mathcal{A}$ is minimized. Different partitions will have different costs.

In this paper, we start out by formalizing the activity partitioning problem in Section 2. Then, in Section 3, we develop a set of algorithms to solve the problem. The algorithms fall into two categories — optimal algorithms (which find an optimal solution) and heuristic algorithms (which find a suboptimal solution but usually run much faster than the optimal algorithms). Two optimal algorithms, and several heuristic algorithms are proposed. In Section 4, we perform a detailed experimental analysis of the algorithms. We report on the results of the experiments and determine which of the algorithms is most appropriate. Related work is presented in the concluding section (Sec. 5).

2 Problem Definition

An agent may engage in some space of *activities*. For example, an agent that tasks sensors may receive high level tasking requests from a GUI client (which in turn receives the tasks from a human user) and convert these into tasks that are compatible with the capabilities of an array of sensors. Likewise, 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 $\{A_1, A_2, \ldots, A_n\}$ where each A_i is a non-empty subset of $\mathcal{A}$ and*

1. $\mathcal{A} = A_1 \cup A_2 \cup \ldots \cup A_n$
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 $\mathcal{A} \supseteq A_1 \cup A_2 \cup \ldots \cup A_n$, $\mathcal{P}$ is called a* sub-partition.

A partition $\mathcal{P}$ of $\mathcal{A}$ splits a set of activities (to be done) into components. In our work, we would like to partition $\mathcal{A}$ into a set $A_1 \cup A_2 \cup \ldots \cup A_n$ such that each component A_i contains activities that can be merged with "significant" cost savings. For example, if $\mathcal{A}$ consists of database activities, then the queries inside a component A_i may be merged using query merging methods (e.g. [27]). Likewise, if $\mathcal{A}$ consists of AI planning activities, then the planning tasks inside a component A_i may be merged using plan merging methods [32]. To determine what partitions are better than others, we need a cost estimation mechanism.

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) $c(a_i) \geq 0$ (II) $c(\emptyset) = 0$, (III) $A_1 \subseteq A_2 \rightarrow c(A_1) \leq c(A_2)$.*

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 abuse notation and write $c(a_i)$ instead of $c(\{a_i\})$. We 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 [27] as well as task merging methods [32]. In this paper, capitalized A's denote sets of activities. Lower case a's denote individual activities.

Problem 1 (Activity Partitioning Problem(APP)) *Given 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 $\sum_{A_i \in P'} c(A_i) < \sum_{A_i \in P} c(A_i)$?*

We prove below that APP is NP-hard by using the **zero-one multiple knapsack problem (MKP)** [19, 17]. 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 ($1 \leq i \leq m$),

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

The above formulation is as a search problem. MKP may be stated as a decision problem by adding an extra input item α, an assignment of items to knapsacks. The decision problem version of MKP returns true if the assignment α is an optimal solution to the search problem and false otherwise. The following theorem shows that the activity partitioning problem is NP-complete.

Theorem 1 *The activity partitioning problem is NP-complete.*

Proof: As membership in NP is immediate, we prove NP-hardness. We 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[1]: 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:

[1]Informally, schemas list the attributes of a relational table.

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, $c(q_i) = -\sum_{j \in q_i} p_j$. [2]

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 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 available approximation algorithms [19, 17] for the **MKP** problem. This is because in **MKP** the profits of items are constants, that is they do not vary with different knapsacks. Similarly, although APP looks very similar to graph or set partitioning [9] at first, we also cannot use the approximation algorithms developed for graph/set partitioning. In the classical graph partitioning problem [12, 16, 22], the graph has a fixed structure and the goal is to partition the vertices such that the sum of edge weights between partition components is minimized. Similarly, in the case of set partitioning [12] the sizes (or weights) associated with each element in the set remains a constant.

However, in our case, the cost of a set of activities is not equal to the sum of such costs. It could be equal to the sum or larger or smaller. Thus, if we try to encode APP via graph/set partitioning, we would quickly find that neither the savings nor the weights are additive, making it unclear how to use (or if it is even possible to use) graph/set partitioning algorithms for APP. The cost estimation function can be *any* function satisfying the axioms described above. The cost of a set of activities takes into account, commonalities amongst the tasks, as well as the fact that performing some tasks in the set may make it easier (or harder) to perform others. This feature distinguishes APP

[2] Query q_i guarantees that assignment of items to knapsacks does not exceed knapsack capacities, and minimizing the cost guarantees that the profits are maximized.

from MKP, set partitioning and graph partitioning problems. It is not difficult to see, for example, that set partitioning can be encoded into APP. However, to use set partitioning algorithms to solve APP, we would need to show that all instances of APP can also be encoded via set partitioning. The same applies to graph partitioning and MKP. We describe *new* algorithms based on different heuristics to solve APP.

3.1 Running Example

In this section, we present a small running example which will be used to illustrate the algorithms developed later on in the paper.

Let $\mathcal{A}$ be an activity set containing five activities a_1, a_2, a_3, a_4 and a_5. The individual costs of these activities are as follows:

$$c(a_1) = 10,\ c(a_2) = 8,\ c(a_3) = 7,\ c(a_4) = 5,\ and\, c(a_5) = 9$$

Further, let the pairwise costs of these activity sets given as follows:

$$\begin{aligned}
&c(\{a_1, a_2\}) = 12,\ \ c(\{a_1, a_3\}) = 17,\ \ c(\{a_1, a_4\}) = 16,\ \ c(\{a_1, a_5\}) = 19,\\
&c(\{a_2, a_3\}) = 11,\ \ c(\{a_2, a_4\}) = 13,\ \ c(\{a_2, a_5\}) = 19,\\
&c(\{a_3, a_4\}) = 7,\ \ c(\{a_3, a_5\}) = 16,\ \ c(\{a_4, a_5\}) = 11
\end{aligned}$$

In the following sections, we will use this activity set to demonstrate our partitioning algorithms. Costs of other subsets of $\mathcal{A}$ will be defined as needed.

3.2 A*-based Algorithm

We start with an A*-based algorithm, which is guaranteed to find an optimal solution. To adapt the A^* algorithm [20], 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* sub-partition *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 c is a cost estimation function and node n has state $s = \{A_1, A_2, \ldots, A_m\}$, 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:*

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

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 sub-partition 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 sub-partition 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 $h(n)$. The following example shows how $g(n)$ and $h(n)$ are calculated.

Example 3.1 *Consider the activity set $\mathcal{A}$ given in Section 3.1. Let a node n have state $s = \{\{a_1, a_2\}\}$. Suppose $c(\{a_1, a_2, a_3\}) = 20$, $c(\{a_1, a_2, a_4\}) = 17$ and $c(\{a_1, a_2, a_5\}) = 25$. Then, $g(n)$ and $h(n)$ are computed as follows:*

$$\begin{aligned} g(n) &= c(\{a_1, a_2\}) = c(a_1) + c(a_2) = 18 \\ h(n) &= min\{incr(a_3, s), incr(a_4, s), incr(a_5, s)\} \\ incr(a_3, s) &= min\{c(\{a_1, a_2, a_3\}) - c(\{a_1, a_2\}), c(a_3)\} = min\{20 - 12, 7\} = 7 \\ incr(a_4, s) &= min\{c(\{a_1, a_2, a_4\}) - c(\{a_1, a_2\}), c(a_4)\} = min\{17 - 12, 5\} = 5 \\ incr(a_5, s) &= min\{c(\{a_1, a_2, a_5\}) - c(\{a_1, a_2\}), c(a_5)\} = min\{25 - 12, 9\} = 9 \\ h(n) &= 5 \end{aligned}$$

Our A^****-based*** algorithm is given in Figure 1. The A^****-based*** algorithm expands nodes by picking an unexpanded node n from the **OPEN** list such that $g(n) + h(n)$ is minimal. The **OPEN** list keeps activities in the increasing order of their $g(n) + h(n)$ values. Once the algorithm chooses a node to expand, it picks an activity a_i by using a ***pick*** function. The ***pick*** function ,***pick(***$s, \mathcal{A}$***)***, orders activities and chooses from this list, the first activity which has not been assigned to any of the components in state s. In this paper, ***pick*** orders activities based on their index order. The A^*-***based*** algorithm terminates when it finds the first goal state. The following example demonstrates how nodes are expanded.

Example 3.2 *Consider the activity set $\mathcal{A}$ in Section 3.1. Suppose the A^***-based** algorithm picks a node n with state $\{\{a_1, a_2\}, \{a_3\}\}$ to expand next, and the **pick** function returns a_4. The algorithm generates the following three next states: $\{\{a_1, a_2, a_4\}, \{a_3\}\}$, $\{\{a_1, a_2\}, \{a_3, a_4\}\}$, and $\{\{a_1, a_2\}, \{a_3\}, \{a_4\}\}$.*

If the algorithm returns NIL, 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 an optimal solution [20]. The heuristic function h is admissible if and only if $h(n) \leq h^*(n)$ 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 $\mathcal{A}$.*

Proof: Suppose n is a node generated during the execution of the A^***-based** algorithm. Let s be the state of n, and s^* be the best goal state reachable from n. Suppose s is of the form $\{A_1, \ldots, A_m\}$ and s^* is of the form $\{A_1^*, \ldots, A_n^*\}$, where $n \geq m$. Then, $h(n) = min\{incr(a, s) \mid a \in \mathcal{A} - (\bigcup_{i=1}^{m} A_i)\}$ and $h^*(n) = \sum_{i=1}^{n} c(A_i^*) - \sum_{i=1}^{m} c(A_i)$. Thus, $h(n) \leq h^*(n)$, for all n, because if $s \neq s^*$, then s^* contains at least one more activity a', and a' incurs at least $min\{incr(a', s)\}$, i.e., minimum of the minimum cost increment of those activities which are not in s, because

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

OPEN := $\emptyset$
$a_i$ := pick($\emptyset$, $\mathcal{A}$);
$n_i.state$ := $\{\{a_i\}\}$ ;  compute $g(n_i)$ and $h(n_i)$;  insert $n_i$ into OPEN
while (OPEN $\neq \emptyset$) do
   $n$ := OPEN.head;  delete $n$ from OPEN; $s$ := n.state
   if $s$ is a goal state then Return(s)
   else /* expand and generate children */
      $a_j$ := pick(s, $\mathcal{A}$)
      forall $A_i \in s$ do
         /* insert $a_j$ into $A_i$ */
         $s' := \{A_1, \ldots, A_{i-1}, A_i \cup \{a_j\}, \ldots, A_m\}$
         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 := $\{A_1, \ldots, A_m, \{a_j\}\}$
      compute $g(n')$ and $h(n')$; insert $n'$ into OPEN;
end(while)
Return (NIL)
End-Algorithm
```

Figure 1: A^****-based*** Algorithm

$A_1 \subseteq A_2 \rightarrow c(A_1) \leq c(A_2)$ and $c(a') > 0$ (axioms (I) and (III) of the cost estimation function). Therefore, $h(n) \leq h^*(n)$. $\square$.

The A^****-based***algorithm has the property that if the heuristic h satisfies the so called *monotone restriction* [20], then the first goal state encountered during the search is guaranteed to be the optimal solution. The monotone restriction requires that for all n, $h(n) \leq h(n') + (g(n') - g(n))$.

Theorem 3 *The heuristic function h satisfies the monotone restriction.*

Proof: Let n be a node generated during the execution of the A^***-based** algorithm and let n' be one of the nodes generated by expanding n. Further let s be the state of n and s' be the state of n'. Suppose $pick(s, \mathcal{A}) = a$. Then, $incr(a, s) \leq g(n') - g(n)$, because the only difference between s and s' is the addition of a in s', and a incurs at least $incr(a,s)$ of additional cost. As, $h(n) = min\{incr(a,s) \mid a \in \mathcal{A} - (\bigcup_{i=1}^{m} A_i)\}$, $h(n) \leq incr(a,s)$, because $h(n)$ is the minimum of all cost increments. Thus, $h(n) \leq g(n') - g(n)$. As, $h(n') \geq 0$ (due to definitions of cost estimation function and h), we can conclude that $h(n) \leq g(n') - g(n) + h(n')$; i.e. h satisfies the monotone restriction. $\Box$.

As our heuristic function is admissible and it satisfies the monotone restriction, the first goal state encountered in the A^****-based*** algorithm is guarenteed to be the optimal solution.

3.3 Branch and Bound (BAB) Algorithm

In this section, we define a branch and bound (BAB) procedure which is also guaranteed to find an optimal solution - however, the search strategy used is somewhat different. The BAB algorithm maintains *nodes* n which have the following fields: ***state*** (as defined in Definition 3), ***gval*** and ***uval***. ***gval*** specifies the value $g(n)$ (cf. Definition 4), while ***uval*** contains $u(n)$ which is defined below. Intuitively, $g(n) + u(n)$ *overestimates* the cost of the minimal cost solution reachable from node n.[3]

Definition 5 *Let node n have state $s = \{A_1, A_2, \ldots, A_m\}$. 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 a cost estimation function.*

Example 3.3 *Consider the activity set provided in Section 3.1. Let n be a node with state $s = \{\{a_1, a_2\}, \{a_5\}\}$. Then,*

$$\begin{aligned} g(n) &= c(\{a_1, a_2\}) + c(a_5) = 12 + 9 = 21 \\ u(n) &= c(a_3) + c(a_4) = 7 + 5 = 12 \end{aligned}$$

The ***BAB*** algorithm is provided in Figure 2, and uses the following definition of an "expansion."

Definition 6 *A partition P of $\mathcal{A}$ is reachable from a sub-partition P' if and only if there is a sequence $P_1, \ldots, P_n$ of sub-partitions such that*

[3]Contrast this with $h(n)$ presented earlier that represents an *underestimate*.

1. $P_1 = P'$ and $P_n = P$, and

2. $(\forall i)(\exists a_j \in \mathcal{A} - \bigcup_{A_k \in P_i} A_k)$ such that P_{i+1} is an expansion of P_i by a_j.

P_{i+1} is an expansion of P_i by a_j if and only if either

1. $\exists A_k \in P_i$ such that $A_k \cup \{a_j\} \in P_{i+1}$, or

2. $P_{i+1} = P_i \cup \{\{a_j\}\}$.

The algorithm is first called with the parameters $\langle \mathcal{A}, \emptyset, Max\_Cost \rangle$, where Max_Cost = $\sum_{a_i \in \mathcal{A}} c(a_i)$. The algorithm carries $bestValue$ across recursive calls to prune nodes that would not lead to cheaper solutions. In each recursive invocation, the algorithm prunes the sub-partitions in $partList$ whose $gval$ exceeds that of the best solution found so far, and expands the other sub-partitions with an activity in $\mathcal{A}$. The algorithm recursively calls itself with one less activity in $\mathcal{A}$ and the resulting set of sub-partitions obtained in this invocation. To further facilitate pruning, the OPEN list is organized in ascending order of $g(n) + u(n)$ and the ***BAB*** algorithm always chooses the first node in OPEN for expansion. The recursion terminates when there are no more activities in $\mathcal{A}$. The following example demonstrates how the ***BAB*** algorithm works for the first three recursive calls for the activity set given in Section 3.1.

Example 3.4 *The **BAB** algorithm is first invoked with $\langle \{a_1, a_2, a_3, a_4, a_5\}, \emptyset, \sum_{a_i \in \mathcal{A}} c(a_i) = 39 \rangle$. As $partList$ is empty in the first invocation, the algorithm executes lines 6-8, and creates a new node n with $state = \{\}$, $uval = 39$, and $gval = 0$, and inserts it into OPEN. The algorithm executes lines 13-20 next, as $n.gval = 39 \leq bestValue = 39$. It picks a_1 (as ordered by their index) and creates a new sub-partition $\{\{a_1\}\}$ and inserts it into $partitions$. As there are no more nodes in OPEN, and $\mathcal{A} \neq \emptyset$, the algorithm recursively calls itself with the values $\langle \{a_2, a_3, a_4, a_5\}, \{\{a_1\}\}, 39 \rangle$ (line 28).*

In the second invocation, the algorithm executes lines 1-4, creating a node n with $state = \{\{a_1\}\}$, $n.uval = 29$, and $n.gval = 10$. This new node is now picked for expansion as it is *the only node in OPEN. The* ***BAB*** *algorithm chooses to insert a_2 next. It creates two new sub-partitions $\{\{a_1, a_2\}\}$ and $\{\{a_1\}, \{a_2\}\}$, and inserts them into $partitions$. The algorithm is now invoked with values $\langle\{a_3, a_4, a_5\}, (\{\{a_1, a_2\}\}, \{\{a_1\}, \{a_2\}\}), 39\rangle$ (line 28).*

In the third recursive call of the algorithm, lines 1-4 are executed again. This time two new nodes are created as follows:

$$\begin{aligned} n_1.state = &\ \{\{a_1, a_2\}\},\ n_1.uval = &\ 21,\ n_1.gval = &\ 12 \\ n_2.state = &\ \{\{a_1\}, \{a_2\}\},\ n_2.uval = &\ 21,\ n_2.gval = &\ 18 \end{aligned}$$

The algorithm picks a_3 to insert next. Since, $n_1.gval + n_1.uval < n_2.gval + n_2.uval$, n_1 is expanded first. In this case, $n_1.gval + n_1.uval = 21 + 12 = 33 < bestValue = 39$, hence $bestValue$ is updated to be 33. The algorithm creates the sub-partitions $\{\{a_1, a_2, a_3\}\}$ and $\{\{a_1, a_2\}, \{a_3\}\}$ and inserts these two sub-partitions into $partitions$ (lines 16-20). Then, n_2 is deleted from OPEN and considered for expansion (lines 11 and 12). The algorithm creates the following three sub-partitions and inserts them into $partitions$: $\{\{a_1, a_3\}, \{a_2\}\}$, $\{\{a_1\}, \{a_2, a_3\}\}$ and $\{\{a_1\}, \{a_2\}, \{a_3\}\}$. The ***BAB*** *algorithm is once again invoked recursively with values $\langle\{a_4, a_5\}, (\{\{a_1, a_2, a_3\}\}, \{\{a_1, a_2\}, \{a_3\}\}, \{\{a_1, a_3\}, \{a_2\}\}, \{\{a_1\}, \{a_2, a_3\}\}, \{\{a_1\}, \{a_2\}, \{a_3\}\}), 33\rangle$.*

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

Proof: We proceed by induction on the size of $\mathcal{A}$.

Base Step: $|\mathcal{A}| = 2$, i.e., $\mathcal{A} = \{a_1, a_2\}$. There are two possible partitions for $\mathcal{A}$; $P_1 = \{\{a_1, a_2\}\}$ and $P_2 = \{\{a_1\}, \{a_2\}\}$. The algorithm is first called with $\langle\{a_1, a_2\}, \emptyset, c(a_1) + c(a_2)\rangle$. Without loss of generality, let us assume that the ***BAB*** algorithm choose a_1 to insert first. It then creates the sub-partition $\{\{a_1\}\}$, and invokes itself recursively with the parameters $\langle\{a_2\}, \{\{\{a_1\}\}\}, c(a_1) + c(a_2)\rangle$.

The ***BAB*** algorithm expands the sub-partition $\{\{a_1\}\}$ with a_2 to create nodes n_1 and n_2 such that $n_1.state = \{\{a_1, a_2\}\} = P_1$ and $n_2.state = \{\{a_1\}, \{a_2\}\} = P_2$. As there are no more

[4]When we say $a_i \in P$, we mean that $\exists A_j \in P$ such that $a_i \in A_j$.

activities left in $\mathcal{A}$, $n_1.uval = n_2.uval = 0$. Moreover, $n_1.gval = c(P_1)$ and $n_2.gval = c(P_2)$. If $c(P_1) < c(P_2)$, then the algorithm sets $bestValue$ to $c(P_1)$ and returns P_1, which is the optimal solution. Note that at this step $bestValue = c(P_2)$. Otherwise, i.e., $c(P_1) > c(P_2)$ then node n_1 is

```
Branch&Bound($\mathcal{A}$, $partList$, $bestValue$)

/* Input:  $\mathcal{A}$: set of activities */
/*         $partList$ (set of sub-partitions) and $bestValue$ */

/* Output: $\mathcal{P}$ */

(1)   forall $P \in partList$ do
(2)     create a new node $n$;   $n$.state := $P$;
(3)     $n.uval := \sum_{a_i \in \mathcal{A}, a_i \notin P} c(a_i)$ ; [4]  $n.gval := \sum_{A_i \in P} c(A_i)$;
(4)     insert $n$ into OPEN;
(5)   if ($partList = \emptyset$) then
(6)     create a new node $n$;   $n$.state := $\{\}$;
(7)     $n.uval := \sum_{a_i \in \mathcal{A}} c(a_i)$;   $n.gval := 0$
(8)     insert $n$ into OPEN;
(9)   $partitions := \emptyset$;
(10)  $a_j$ := pick(n.state, $\mathcal{A}$);
(11)  while (OPEN $\neq \emptyset$) do
(12)    $n$ := OPEN.head;  delete $n$ from OPEN;
(13)    if ($n.gval \leq bestValue$) then
(14)      if ($n.gval + n.uval \leq bestValue$) then
(15)        $bestValue := n.gval + n.uval$;
(16)      forall $A_i \in n.state$ do
(17)        $P' := \{A_1, \ldots, A_{i-1}, A_i \cup \{a_j\}, \ldots, A_m\}$;
(18)        insert $P'$ into $partitions$;
(19)      $P' := \{A_1, \ldots, A_m, \{a_j\}\}$;
(20)      insert $P'$ into $partitions$;
(21)  end(while)
(22)  if ($\mathcal{A} = \emptyset$) then
(23)    $minCost := c(partitions.head)$;   $minPart := partitions.head$;
(24)    forall $P \in partitions$ do
(25)      if ($c(P) < minCost$) then
(26)        $minCost := c(P)$;   $minPart := P$;
(27)    Return ($minPart$);
(28)  else BAB ($\mathcal{A} - \{a_j\}, partitions, bestValue$);
End-Algorithm
```

Figure 2: Branch and Bound Algorithm

pruned, because $n_1.gval = c(P_1) > c(P_2) = bestValue$, hence the algorithm returns P_2 which is the optimal solution. If $c(P_1) = c(P_2)$, then the algorithm returns one of them and the other is in $partitions$.

Inductive Step: Let us state the inductive hypothesis as: $\forall i$, *if $\mathcal{A}$ has i elements then every optimal partition of $\mathcal{A}$ is reachable from the set "$partitions$."*

Let $|\mathcal{A}| = k + 1$. Let us pick an arbitrary activity $a_i \in \mathcal{A}$ and remove it from $\mathcal{A}$ to get $\mathcal{A}' = \mathcal{A} - \{a_i\}$. By the inductive hypothesis, the ***BAB*** algorithm finds an optimal partition for $\mathcal{A}'$. Let us consider the set $partitions$ which is computed in the last recursive invocation of the algorithm and also contains the optimal solution. We expand all sub-partitions $P \in\ partitions$ with a_i. At this stage, $bestValue = min_{P \in partitions}\{c(P)\}$. The ***BAB*** algorithm inserts a_i into each component $A_i \in P$, for all $P \in\ partitions$, and also creates partitions $P \cup \{\{a_i\}\}$. Among these, the ***BAB*** algorithm chooses the partition with the least cost, because it would update $bestValue$ only if it finds a node whose state has a lower cost (i.e. $gval$). Note that at this stage all $uval$'s are 0, as there are no activities left. As the ***BAB*** finds the least cost way of including a_i into the nodes in sol and the cost of a set of activities does not decrease when we add more activities (i.e. $A_1 \subseteq A_2 \to c(A_1) \leq c(A_2)$, axiom III of the cost estimation function), the partition found by the ***BAB*** algorithm for $\mathcal{A}$ is the optimal partition. $\square$

3.4 Greedy Algorithms

The algorithms described in sections 3.2 and 3.3 are guaranteed to find an optimal partition of a set of activities. As the activity partitioning problem is NP-hard, these algorithms take exponential time. In this section, we develop greedy algorithms which run fast, but may find a suboptimal solution. These algorithms use a concept called a *cluster graph*, which is a heuristic representation to capture savings between two activity sets. 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 7 (Canonical Cluster Graph ($\mathcal{CCG}$)) *A* canonical 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. $E = \{(\{a_i\}, \{a_j\}) \mid a_i, a_j \in \mathcal{A}$ *and* $c(a_i) + c(a_j) - c(\{a_i, a_j\}) > 0\}$,
3. $w(\{a_i\}, \{a_j\}) = c(a_i) + c(a_j) - c(\{a_i, a_j\})$

Example 3.5 *The canonical cluster graph associated with the activity set $\mathcal{A}$ in our running example is provided in Figure 3. Edge weights are computed as follows:*

$$\begin{aligned}
w(\{a_1\},\{a_2\}) &= c(a_1) + c(a_2) - c(\{a_1, a_2\}) = 10 + 8 - 12 = 6\\
w(\{a_1\},\{a_3\}) &= c(a_1) + c(a_3) - c(\{a_1, a_3\}) = 10 + 7 - 17 = 0\\
w(\{a_1\},\{a_4\}) &= c(a_1) + c(a_4) - c(\{a_1, a_4\}) = 10 + 5 - 16 = -1\\
w(\{a_1\},\{a_5\}) &= c(a_1) + c(a_5) - c(\{a_1, a_5\}) = 10 + 9 - 19 = 0\\
w(\{a_2\},\{a_3\}) &= c(a_2) + c(a_3) - c(\{a_2, a_3\}) = 8 + 7 - 11 = 4\\
w(\{a_2\},\{a_4\}) &= c(a_2) + c(a_4) - c(\{a_2, a_4\}) = 8 + 5 - 13 = 0\\
w(\{a_2\},\{a_5\}) &= c(a_2) + c(a_5) - c(\{a_2, a_5\}) = 8 + 9 - 19 = -2\\
w(\{a_3\},\{a_4\}) &= c(a_3) + c(a_4) - c(\{a_3, a_4\}) = 7 + 5 - 7 = 5\\
w(\{a_3\},\{a_5\}) &= c(a_3) + c(a_5) - c(\{a_3, a_5\}) = 7 + 9 - 16 = 0\\
w(\{a_4\},\{a_5\}) &= c(a_4) + c(a_5) - c(\{a_4, a_5\}) = 5 + 9 - 11 = 3
\end{aligned}$$

As only $w(\{a_1\},\{a_2\}), w(\{a_2\},\{a_3\}), w(\{a_3\},\{a_4\})$ and $w(\{a_4\},\{a_5\})$ have positive values, the only edges in the canonical cluster graph associated with $\mathcal{A}$ are shown in Figure 3.

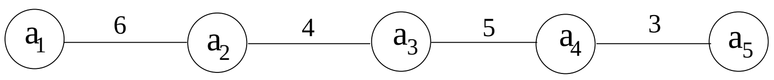

Figure 3: Example Cluster Graph

The basic greedy algorithm is shown in Figure 4. The ***Greedy-Basic*** uses canonical cluster graphs, which capture pair-wise savings between two activities. The ***Greedy-Basic*** algorithm tries to build a partition iteratively. In each iteration, it finds an edge with the 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 component of the current partition. There are four cases to check depending on whether one, both or neither activities occur in an existing component.

Case 1: If both a_i and a_j are in the same component, then we do not have to do anything.

Case 2: If one of a_i or a_j is in an existing component but not the other, then the other can be placed in the same component.

Case 3: If neither one is in any of the components, then we create a new component $\{a_i, a_j\}$.

Case 4: If both a_i and a_j occur in existing but 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.

It is easy to see that the loop of ***Greedy-Basic*** is executed $O(|E|)$ times, where $|E|$ is the number of edges in the canonical cluster graph. Moreover, each execution of the loop may take $O(|V|)$ time (where $|V|$ is the number of vertices in the canonical cluster graph) to check if a_i, a_j occur in a component. Therefore, the **Greedy-Basic** algorithm is polynomial in the size of the input $\mathcal{A}$ *as long as* the cost estimation function c runs in polynomial time.

The following example illustrates how the ***Greedy-Basic*** algorithm works.

Example 3.6 *Consider the canonical cluster graph in Figure 3. Initially, $\mathcal{P} = \emptyset$. In the first iteration, the maximum weight edge (a_1, a_2) is picked, and deleted from the graph. As both a_1 and a_2 are not in $\mathcal{P}$, the **Greedy-Basic** algorithm creates a new component A_1 and inserts a_1 and a_2 into A_1. In the next iteration, the algorithm picks the edge (a_3, a_4), which has weight 5. Again, neither a_3 nor a_4 is in the current solution. Hence a new component A_2 is created and the sub-partition $\mathcal{P} = \{\{a_1, a_2\}, \{a_3, a_4\}\}$ is obtained. The next edge to be picked is (a_2, a_3).*

```
Greedy-Basic($\mathcal{CCG}$)
/* 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 4: Basic Greedy Algorithm

Here, both a_2 and a_3 are in the solution, but are in two different components of P. The algorithm considers the following three alternatives: $P_1 = \{\{a_1, a_2\}, \{a_3, a_4\}\}$, $P_2 = \{\{a_1, a_2, a_3\}, \{a_4\}\}$, and $P_3 = \{\{a_1\}, \{a_2, a_3, a_4\}\}$. Recall from Example 3.1 that $c(\{a_1, a_2, a_3\}) = 20$ and suppose that $c(\{a_2, a_3, a_4\}) = 15$. Our algorithm computes the cost of these sub-partitions as: $c(P_1) = 12 + 7 = 19$, $c(P_2) = 20 + 5 = 25$ and $c(P_3) = 10 + 15 = 25$ and chooses P_1. In the last iteration, the algorithm picks the edge (a_4, a_5). As a_4 is already in the current solution, the algorithm inserts a_5 into the same component as a_4. The algorithm terminates when it exhausts all the edges in the graph, and returns the partition $\mathcal{P} = \{\{a_1, a_2\}, \{a_3, a_4, a_5\}\}$.

One problem with the ***Greedy-Basic*** algorithm is the following: Once we start inserting activities into components, the edge labels in the graph should be updated. For instance in Example 3.6, when the algorithm creates the component $\{a_1, a_2\}$, the savings obtained by optimizing a_2 and a_3 together is no longer 4. The reason for this is that when a_1 and a_2 are optimized together, the plan to execute a_2 might change, e.g. to reuse results of a_1. As a result, the savings obtained when optimizing a_2 and a_3 together is different from the savings obtained when all three are optimized together. One obvious way to solve this problem is to recompute the edge weights for those edges that are adjacent to the deleted edges after each iteration. To update edge weights and properly merge vertices after each iteration, we define a routine, called ***Reduce***. ***Reduce*** takes a cluster graph and an edge (A_i, A_j) in this cluster graph as input and merges A_i, A_j together into one vertex $A_i \cup A_j$ such that all edges between other vertices A_k and either A_i or A_j are now between that vertex and the merged vertex $A_i \cup A_j$. The weights of these edges are recomputed to reflect the savings between those vertices and the newly merged vertex $A_i \cup A_j$.

Definition 8 (Cluster Graph Reduction; Reduce) *The function $Reduce$ takes a cluster graph $CG = (V, E, w)$, and an edge $e = (A_i, A_j)$ in this graph as input, and produces another cluster graph $CG' = (V', E', w')$ as output such that,*

1. $V' = V - \{A_i, A_j\} \cup \{A_1 \cup A_j\}$.
2. $E' = E - (\{(A_i, A_k) \mid A_k \in CCG\} \cup \{(A_j, A_k) \mid A_k \in CCG\} \cup \{(A_k, A_i) \mid A_k \in CCG\} \cup \{(A_k, A_j) \mid A_k \in CCG\})$.
3. $$w'(A_k, A_l) = \begin{cases} w(A_k, A_l) & \textit{if } A_k \notin \{A_i, A_j\} \\ c(A_i \cup A_j) + c(A_k) - & \\ \; c(A_i \cup A_j \cup A_k) & \textit{otherwise} \end{cases}$$

Reduce is easily implementable in time linear in the number of edges in the cluster graph. The next greedy algorithm, ***Greedy-WU***, which is provided in Figure 5, uses ***Reduce*** to adjust edge weights correctly. Whenever it picks the maximum weight edge from the graph, it checks if the activity sets associated with that edge are already in some component. If so, it simply deletes that edge, and continues. If one of the activity sets is a subset of an existing component, then ***Greedy-WU*** (Greedy with Weight Update) algorithm unions that activity set with the existing component, and applies ***Reduce*** to create a new vertex and update the weights of adjacent edges. If neither activity set associated with the edge is in a component, then it creates a new component which is the union of the two activity sets, and invokes ***Reduce*** to create a new vertex and recompute the weights of adjacent edges.

The running time of the ***Greedy-WU*** algorithm is $O(|E| \times (|E| + |V|^2))$: The algorithm makes $O(|E|)$ iterations and each iteration takes at most $|V|^2 + |E|$ time to execute: $O(|V|^2)$ time to check subsets, and $O(|E|)$ to execute the ***Reduce*** routine. The following example illustrates how the ***Greedy-WU*** algorithm works.

```
Greedy-WU($\mathcal{CCG}$)
/* Input: $\mathcal{CCG}$ (canonical cluster graph of $\mathcal{A}$) */
/* Output: $\mathcal{P}$ */
$\mathcal{P} := \emptyset$;  $n := 0$;
while ($\exists e \in \mathcal{CG}$) do
  $e := (A_i, A_j)$, s.t. $w(e)$ is maximum in $\mathcal{CG}$
  delete $e$ from $\mathcal{CG}$
  if ($\exists A_l \in \mathcal{P}\ s.t.\ A_i \subseteq A_l$) then
    if ($\exists A_k \neq A_l \in \mathcal{P}\ s.t.\ A_j \subseteq A_k$) then
      delete $(A_i, A_j)$ from $CG$;
    else /* $\nexists A_k\ s.t.\ A_j \subseteq A_k$ */
      $A_l := A_l \cup A_j$;
      $CG :=$ Reduce$(CG, A_l, A_j)$;
  else /* $\nexists A_l\ s.t.\ A_i \subseteq A_l$ */
    if ($\exists A_k \in \mathcal{P}\ s.t.\ A_j \subseteq A_k$) then
      $A_k := A_k \cup A_i$;
      $CG :=$ Reduce$(CG, A_k, A_i)$;
    else
      $n := n + 1$;  $A_n := A_i \cup A_j$;
      $CG :=$ Reduce$(CG, A_i, A_i)$;
      $\mathcal{P} := \mathcal{P} \cup \{A_n\}$;
end(while)
Return ($\mathcal{P}$)
End-Algorithm
```

```
Greedy-NMA($\mathcal{CCG}$)
/* 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 ($\nexists A_l \in \mathcal{P}\ s.t.\ a_i \in A_l$) then
    if ($\nexists A_k \neq A_l \in \mathcal{P}\ s.t.\ a_j \in A_k$) then
      $n := n + 1$;  $A_n := \{a_i, a_j\}$;
       $\mathcal{P} := \mathcal{P} \cup \{A_n\}$;
    else /*$\exists A_k \in \mathcal{P}\ s.t.\ a_j \in A_k$ */
      insert $a_i$ into $A_k$;
  else /* $\exists A_l \in \mathcal{P}\ s.t.\ a_i \in A_l$ */
    if ($\nexists A_k \neq A_l \in \mathcal{P}\ s.t.\ a_j \in A_k$) then
      insert $a_j$ into $A_l$;
end(while)
Return ($\mathcal{P}$)
End-Algorithm
```

Figure 5: Greedy with Weight Update and Greedy with No Move Around Algorithms

Example 3.7 *In the first iteration, the **Greedy-WU** algorithm picks the edge (a_1, a_2), which has the maximum weight. It creates a new component $A_1 = \{a_1, a_2\}$ and inserts A_1 into $\mathcal{P}$. It then invokes **Reduce(**$\{a_1\}, \{a_2\}$**)**, which merges the vertices containing a_1 and a_2, and recomputes the weight $w(\{a_1, a_2\}, \{a_3\})$. Recall from Example 3.1 that $c(\{a_1, a_2, a_3\}) = 20$. Then:*

$$w(\{a_1, a_2\}, \{a_3\}) = c(\{a_1, a_2\}) + c(a_3) - c(\{a_1, a_2, a_3\}) = 12 + 7 - 20 = -1$$

*As the weight is negative, no edge is created between $\{a_1, a_2\}$ and a_3, resulting in the cluster graph given in Figure 6. In the next iteration, the maximum weight edge (a_3, a_4) is picked. As neither a_3 nor a_4 is in the solution, a new component $A_2 = \{a_3, a_4\}$ is created and inserted into $\mathcal{P}$. **Reduce(**$\{a_3\}, \{a_4\}$**)** is then invoked, and the weights are updated. Suppose $c(\{a_2, a_3, a_4\}) = 15$. Then,*

$$w(\{a_3, a_4\}, \{a_5\}) = c(\{a_3, a_4\}) + c(a_5) - c(\{a_3, a_4, a_5\}) = 7 + 9 - 15 = 1$$

The cluster graph after this iteration is shown in Figure 7. Finally, the only edge left, i.e., $(\{a_3, a_4\}, \{a_5\})$, is picked. Hence, a_5 is not in the current solution, it is inserted into A_2 and the algorithm returns $\mathcal{P} = \{\{a_1, a_2\}, \{a_3, a_4, a_5\}\}$.

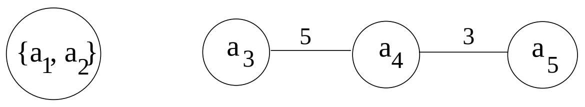

Figure 6: Cluster Graph After the 1st Iteration

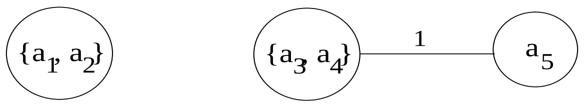

Figure 7: Cluster Graph After the 2nd Iteration

When we create the canonical cluster graph associated with a set $\mathcal{A}$ of activities, we only need to invoke the cost estimation function on sets containing two elements. However, the ***Greedy-Basic*** algorithm may need to invoke the cost estimation function on activity sets involving many activities. We can improve the running time of the ***Greedy-Basic*** algorithm by restricting the number of choices it examines at each iteration. More specifically, if we do not move activities across components once they are inserted into some component, the greedy algorithm does not have to consider costs of alternatives when two activities are in two different components in the current solution. This give rise to the ***Greedy-NMA*** (Greedy with No Move Around) algorithm, which does not modify the current solution when two activities are in the solution, but in two different components. This algorithm is provided in Figure 5. The ***Greedy-NMA*** algorithm also runs in $O(|E| \times |V|)$ time. The following example illustrates the ***Greedy-NMA*** algorithm.

Example 3.8 *As in the case of the* **Greedy-Basic** *algorithm (Example 3.6), the* **Greedy-NMA** *algorithm picks the edge* (a_1, a_2) *and* (a_3, a_4) *in the first and second iterations, respectively, and creates the partial solution* $\mathcal{P} = \{\{a_1, a_2\}, \{a_3, a_4\}\}$*.* **Greedy-NMA** *differs from* **Greedy-Basic** *in the next iteration, when it picks the edge* (a_2, a_3)*. Instead of considering alternatives as the* **Greedy-Basic** *algorithm does, it does not change the current solution. In the final iteration, the algorithm picks* (a_4, a_5) *and inserts* a_5 *in the same component as* a_4*, resulting in the final partition* $\{\{a_1, a_2\}, \{a_3, a_4, a_5\}\}$.

3.4.1 Improving the Quality of the Partitions Produced By the Greedy Algorithms

In this section, we study how we can increase the quality of the partitions found. The lower the cost associated with a partition, the better the quality. To achieve this, we introduce a procedure called ***ImproveQuality*** which takes a partition produced by one of the greedy algorithms, and tries to decrease the cost by moving some of the activities across components of the input partition. This procedure is described in Figure 8.

The ***ImproveQuality*** procedure maintains a data structure, $list$, which is a sorted array of activities in decreasing order of their individual costs. It then considers the first max_no number of activities from $list$, i.e., the first max_no number of activities with the highest individual costs. For each activity a_j the procedure examines, it deletes a_j from its current component, and tries to insert a_j into every other component in the current solution. If the cost of the resulting (sub)partition is less than that of the current solution, the procedure replaces the current solution with this new (sub)partition with the least cost.

```
***ImproveQuality***($\mathcal{A}$, $\mathcal{P}$, $max\_no$)

/* **Input**:  $\mathcal{A}$: a set of activities */
/*           $\mathcal{P} = \{A_1, \ldots, A_m\}$ : a partition of $\mathcal{A}$ */
/*           $max\_no$ : maximum number of activities to be considered */

/* **Output**:  $\mathcal{P}$ */

$list$ := sort activities in decreasing order of their individual costs
$currentP$ := $\mathcal{P}$;
forall $list[i]$, s.t. $i < max\_no$ do
  find $A_j$, s.t. $list[i] \in A_j$
  remove $list[i]$ from $A_j$
  forall $A_k \neq A_j \in currentP$ do
    $newP$ := $\{A_1, \ldots, A_k \cup \{list[i]\}, \ldots, A_m\}$
    if ($c(newP) < c(currentP)$) then
      $currentP := newP$;
Return ($currentP$);
```

Figure 8: Procedure to Improve the Quality of Output Partitions

3.5 Hill Climbing Algorithm

The last algorithm that we consider uses a steepest descent hill climbing strategy. The ***HillClimb*** algorithm, provided in Figure 9, creates partitions by inserting one activity into the current subpartition at a time. It starts with an empty partition and inserts the activity into the component that incurs the least *additional* cost. The ***HillClimb*** algorithm performs $\mathcal{A}$ iterations, one iteration per activity, and each iteration runs in $O(|n_i|)$ time, where n_i is the number of components in iteration i. This is because of the fact that in each iteration, it either inserts an activity into one of the existing n_i components, or creates a singleton component. In the worst case, there are i components in iteration i, and the ***HillClimb*** algorithm creates one more component in each iteration, i.e., inserts activities as singletons. In that case, the total number of cases the algorithm has to consider is $\frac{|A|\times(|A|+1)}{2}$, hence the algorithm runs in $O(|A|^2)$ time. The following example demonstrates how the ***HillClimb*** algorithm works.

```
**HillClimb**($\mathcal{A}$, $no\_of\_activities$)
/* **Input**: $\mathcal{A}$ (a set of activities), $no\_of\_activities$ (no of activities) */
/* **Output**: $\mathcal{P}$ */

$P$ := $\{\{a_1\}\}$;  count := 1;
while ( count < no_of_activities) do
   $P$ := $P \cup \{\{a_{count+1}\}\}$;
   count := count + 1;
   $minCost$ := $\sum_{A_i \in P} c(A_i)$;
   $minPart$ := $P$;
   forall $A_i \in P$ do
     $newP$ := $\{A_1, \ldots, A_{i-1}, A_i \cup \{a_{count+1}\} \ldots, A_m\}$
     $newCost$ = $\sum_{A_i \in newP} c(A_i)$;
     if ($newCost < minCost$) then
       $minCost$ := $newCost$;
       $minPart$ := $newP$;
   $P$ := $newP$;
end(while)
Return ($minPart$);
End-Algorithm
```

Figure 9: ***Hill Climb*** Algorithm

Example 3.9 *Consider the activity set $\mathcal{A}$ described in Section 3.1. Suppose the activities are inserted in the order they are given. First, $P_1 = \{\{a_1\}\}$. When the algorithm inserts a_2, it can either insert a_2 in the same component as a_1 or it can create a new component that contains only a_2. $c(\{\{a_1, a_2\}\}) - c(P_1) = 12 - 10 = 2$, and $c(\{a_1\}) + c(\{a_2\}) - c(P_1) = (10 + 8) - 10 = 8$. As the first sub-partition causes a smaller cost increase, $P_2 = \{\{a_1, a_2\}\}$ and $c(P_2) = 12$. a_3 is inserted next. Again there are two possibilities. This time, $c(\{\{a_1, a_2, a_3\}\}) - c(P_2) = 20 - 12 = 8$, and $c(\{\{a_1, a_2\}, \{a_3\}\}) - c(P_2) = (12 + 7) - 12 = 7$. As the second option yields a smaller cost increase, $P_3 = \{\{a_1, a_2\}, \{a_3\}\}$ and $c(P_3) = 12 + 7 = 19$. In the next iteration, the **HillClimb** algorithm inserts a_4. Now, there are three possibilities: a_4 can be inserted into either one of the existing components, or a new component that contains only a_4 can be created. Recall from Example 3.1 that $c(\{a_1, a_2, a_4\}) = 17$. Then, $c(\{\{a_1, a_2, a_4\}, \{a_3\}\}) - c(P_3) = (17 + 7) - 19 = 5$, $c(\{\{a_1, a_2\}, \{a_3, a_4\}\}) - c(P_3) = (12 + 7) - 19 = 0$, and $c(\{\{a_1, a_2\}, \{a_3\}, \{a_4\}\}) -$*

$c(P_3) = (12 + 7 + 5) - 19 = 5$. *As the second alternative involves a lower cost increase,* $P_4 = \{\{a_1, a_2\}, \{a_3, a_4\}\}$*, and* $c(P_4) = 19$. *Finally,* a_5

current partition or by creating a new partition *without* violating the component size constraint. The definition of $h(n)$ is now given in terms of the new $incr(-,-)$ function. It is easy to see that h defined thus is still an underestimate of the cost to a solution from the current node.

4 Experimental Results

We have implemented all algorithms described here (versions of the ***BAB*** and ***HillClimb*** algorithms with component constraints were not implemented as they don't scale well). We wrote a program to simulate the cost estimation function. This program takes *number of activities, number of clusters, overlap probability, overlap degree*, and *maximum cost* as input. It uniformly assigns a cost from the [0,maximum cost] interval to individual activities. It then computes costs of activity sets as follows. As each activity is likely to overlap with only some (not all) other activities, there will be clusters of activities that can potentially be merged. To simulate this behavior, we first assign each activity to one of the clusters. Only activities in the same cluster overlap. An *overlap probability* determines activity pairs in a cluster that share computations. Given a pair of activities from $\mathcal{A}$, the overlap probability gives the probability that the pair will overlap. We then use an *overlap degree* to measure the savings obtained when two activities *do* 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.

In the following sections, we describe several experiments to evaluate the performance of the algorithms in terms of both their running times, and the quality of the solutions they produce. Moreover, we evaluated the algorithms presented in Section 3 when there were no component constraints, as well as when there were such constraints. In all experiments reported here, we ran the algorithms multiple times on a Sun Ultra1 machine with 320 MB memory running Solaris 2.6 and used the averages in the graphs. In the following, we will report $|CI|/average \times 100$, (CI = Confidence Interval) for 95% confidence intervals in the graphs where we plot running times. The reason for reporting a normalized number is due to the high deviation between the running times of the algorithms.

4.1 Performance of Algorithms without Component Constraints

In this section, we report on experiments that evaluate the performance of activity partitioning algorithms without component size constraints.

4.1.1 Scalability of the Algorithms

The first experiment compares the running time of our algorithms. Table 1 show the results. It turns out that A^****-based*** ran out of memory when the number of activities exceeded 10, and ***BAB*** ran out of memory when the number of activities exceeded 11. The reason for this is that the OPEN list quickly grows overwhelmingly large. Hence, the tables only show results for 10 activities. Although the A^****-based*** algorithm is faster than ***BAB***, it runs out of memory faster. Both algorithms have much longer running times compared to the heuristic-based algorithms. This is expected as the problem is NP-hard, and both A^****-based*** and ***BAB*** algorithms find optimal solutions. $|CI|/average \times 100 < 15$ for the tibastar algorithm; it is less than 20 for the ***BAB*** algorithm and it is less than 4 for the greedy algorithms.

Scalability of the *Greedy* Algorithms

As the heuristic algorithms were very fast, we increased the number of activities to be partitioned. We first fixed the overlap degree and overlap probability constant at 0.2, and ran two experiments. In the first experiment, we kept the number of activities within a cluster constant at 25, and in the second, the number of activities within a cluster was 50. Figures 10 and 11 show the results.

In this experiment, the ***HillClimb*** algorithm ran out of memory around 400 activities and did not scale up very well. Recall that the ***HillClimb*** algorithm uses the same expansion function as the A^****-based*** algorithm, but keeps the best sub-partition at each node. When the number of activities in the input set exceed 350-400 activities the algorithm generates too many children nodes and runs

No of Activities	A^**-based*	***BAB***	***Greedy-Basic***	***Greedy-NMA***	***Greedy-WU***	***Hill Climb***
5	17.24	19.84	5.28	5.04	7.92	6.44
6	53.4	58.26	6.46	6.2	10.72	8.54
7	142.72	158.44	8.2	7.58	14.84	11.02
8	479.02	503.5	10.02	9.4	19.26	13.8
9	1431.7	1657.5	11.94	11.56	25.14	17.16
10	4189.44	3432.38	15.68	13.8	31.4	20.92
11		4149.56	18.7	16.14	38.12	24.96

Table 1: Running times of the Algorithms (in millisecs) for (o-prob=0.8, o-degree=0.6, no-of-clusters=1, max-cost=100)

out of memory. Therefore, we only show the scalability results for the greedy algorithms. This result suggests that *the cluster graph representation is a very effective heuristic*.

When the number of activities within a cluster is 25 (Figure 10), the running times of the algorithms are very close, and they handle 1000 activities in about 140secs. On the other hand, when the number of activities within a cluster is increased to 50, then the execution time of the ***Greedy-WU*** algorithm is much longer than the other two greedy algorithms. This is expected because the ***Greedy-WU*** recomputes edge weights after each iteration, and the larger the number of activities, the larger the number of edges in the cluster graph. Moreover, all algorithms run much slower in the second experiment.

In this experiment we measured the effect of changing the overlap probability on the running time of the greedy algorithms. Recall that greedy algorithms use the cluster graph representation and their running time depends on the number of edges in the cluster graph. Moreover, the overlap probability affects the number of edges in the cluster graph. For this experiment, we kept the overlap degree constant at 0.4 and ran experiments with overlap probabilities 0.2, and 0.4. The number of activities within a cluster was also kept constant at 25. The results are shown in Figures

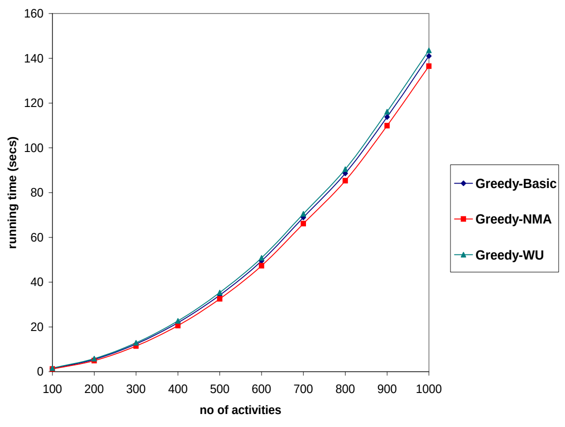

Figure 10: Running time for (o-prob=0.2, o-degree=0.2, no-of-activities/cluster=25, max-cost=100)

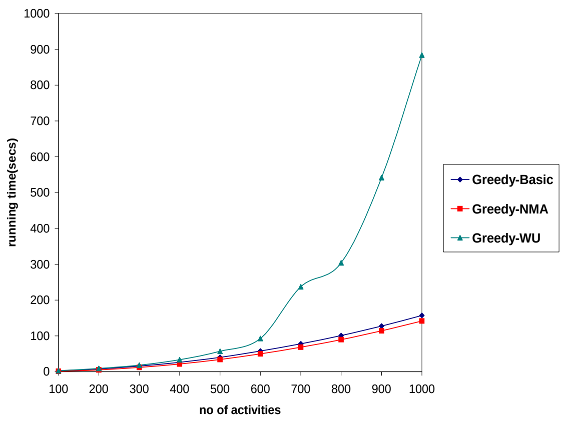

Figure 11: Running time for (o-prob=0.2, o-degree=0.2, no-of-activities/cluster=50, max-cost=100)

12 and 13. $|CI|/average \times 100 < 1.4$ in Figure 12 for 95% confidence interval, and it is less than 1.6 in Figure 13. As expected, the algorithms have a slightly longer running time and the relative difference between the algorithms is more explicit when the overlap probability is higher (i.e 0..4).

4.1.2 Quality of the Output Partitions

As the greedy algorithms may compute suboptimal solutions, we evaluated the quality of solutions produced against the optimal solution. We used the following metric for this purpose. Given a set $\mathcal{A}$ of activities, and a partition $\mathcal{P}$ of $\mathcal{A}$, the cost reduction percentage realized by $\mathcal{P}$ is

$$\frac{\sum_{a\in\mathcal{A}} c(a) - \sum_{A_i\in\mathcal{P}} c(A_i)}{\sum_{a\in\mathcal{A}} c(a)}.$$

We use percentage savings instead of absolute numbers because we use a simulated cost estimation function. As savings percentage is already a normalized number, we directly report confidence interval in the graphs where we plot savings percentages.

We first studied how savings are affected by changes in the amount of overlap between activities. We kept the overlap probability constant at 0.4, and ran experiments for overlap degrees 0.2, 0.4,

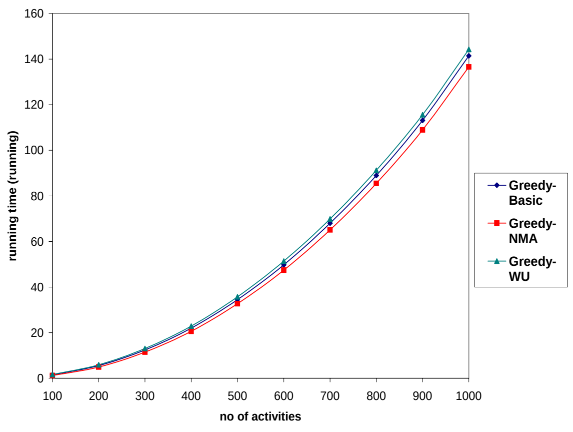

Figure 12: Running time for (o-prob=0.2, o-degree=0.4, no-of-activities/cluster=25, max-cost=100)

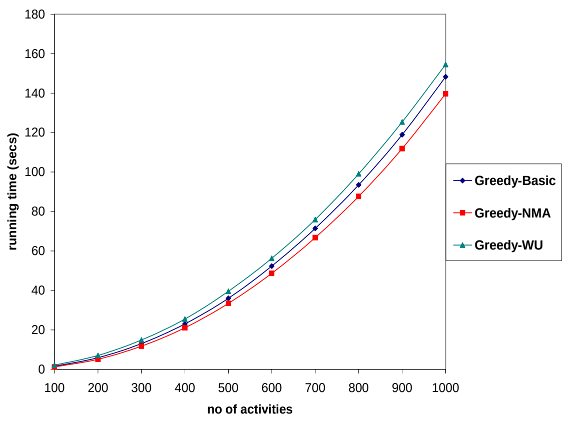

Figure 13: Running time for (o-prob=0.4, o-degree=0.4, no-of-activities/cluster=25, max-cost=100)

0.6 and 0.8. Figures 14, 15, 16 and 17 show the results. We only plotted savings generated by the ***BAB*** algorithm as both ***BAB*** and A^****-based*** find optimal solutions and hence produce the same result. In Figure 14 $|CI| < 1$, in Figure 15 $|CI| < 1.8$, in Figure 16 $|CI| < 2.3$ and finally in Figure 17 $|CI| < 3.5$ for 95% confidence intervals.

In general, the solutions produced by the ***Greedy-Basic*** algorithm are about 0.8–13% worse than the optimal solutions. The solutions produced by the ***Greedy-NMA*** algorithm are 0.8–20% worse, and the ones produced by the ***Greedy-WU*** algorithm are 0.06–9.4% worse than the optimal solutions. The quality of the solutions produced by the ***HillClimb*** algorithm is not as good as the greedy solutions, and they are about 2.4–23% worse than the optimal solutions. As we increase the overlap degree in the input activity set, the difference between the savings generated by the heuristic algorithms and the ones computed by the optimal algorithms also increases.

We also studied how savings changed with changes in overlap probability. We kept the overlap between the activities constant at 0.6, and ran experiments for overlap probabilities 0.4, 0.6 and 0.8. Figures 16, 18, and 19 show the results of these experiments. In Figure 16 $|CI| < 2.3$, in Figure 18 $|CI| < 2.6$ and in Figure 19 $|CI| < 2.7$ for 95% confidence intervals. As these and

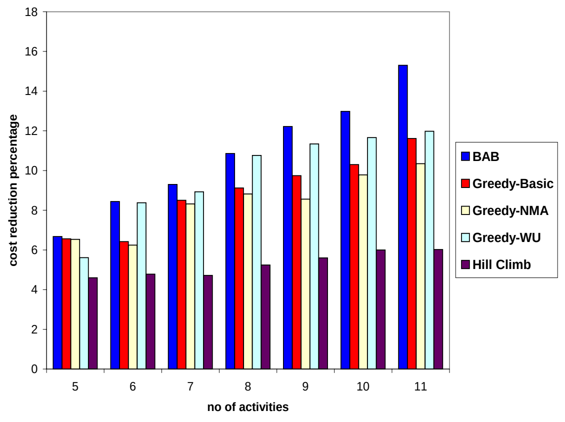

Figure 14: Savings for (o-prob=0.4, o-degree=0.2, no-of-clusters=1, max-cost=100)

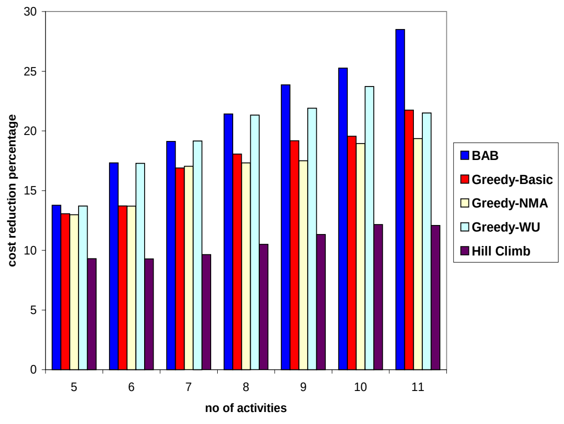

Figure 15: Savings for (o-prob=0.4, o-degree=0.4, no-of-clusters=1, max-cost=100)

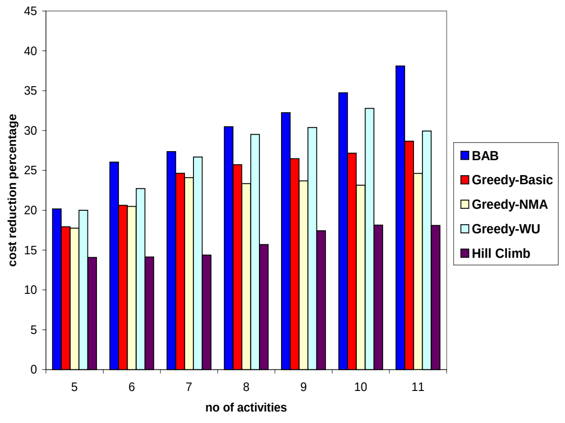

Figure 16: Savings for (o-prob=0.4, o-degree=0.6, no-of-clusters=1, max-cost=100)

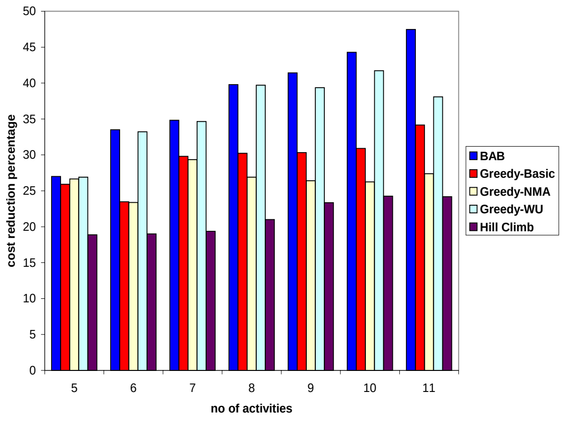

Figure 17: Savings for (o-prob=0.4, o-degree=0.8, no-of-clusters=1, max-cost=100)

previous results show that confidence intervals change more relative to the overlap degree than to the overlap probability, we conclude that the greedy algorithms are more sensitive to the overlap degree.

As the overlap probability increases, the savings obtained by partitioning also increases. The relative performance of the greedy algorithms and the ***HillClimb*** algorithm stays the same. However, the quality of the partitions produced by the heuristic algorithms seem to degrade as the overlap

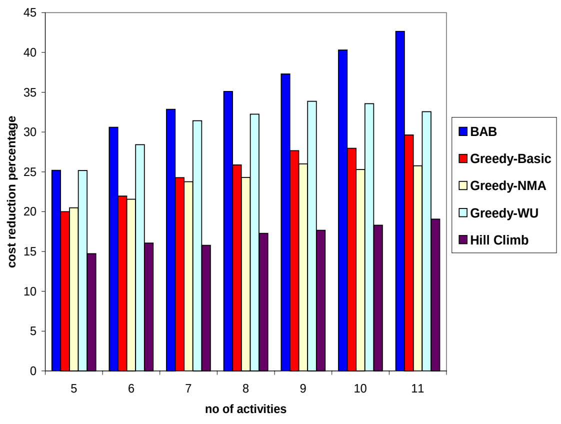

Figure 18: Savings for (o-prob=0.6, o-degree=0.6, no-of-clusters=1, max-cost=100)

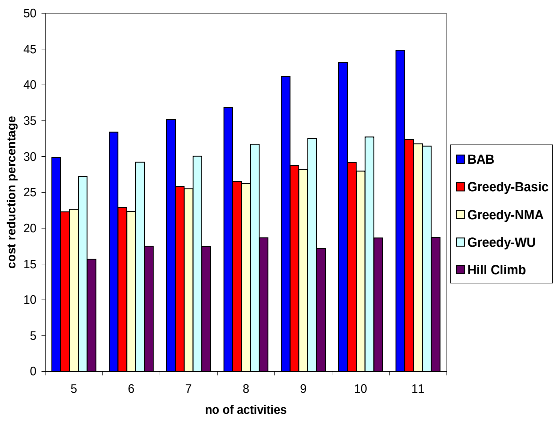

Figure 19: Savings for (o-prob=0.8, o-degree=0.6, no-of-clusters=1, max-cost=100)

probability increases. The reason for this is that when the overlap probability is higher, more activities share computations. As the heuristic algorithms consider relatively few alternatives, their performance degrades. Nevertheless, we do not expect real applications to have 60% to 80% overlap.

Although we could not get the optimal solution after 11 activities, we still ran the greedy algorithms up to 1000 activities to see the quality of partitions they produced. We kept the overlap probability constant at 0.2, and overlap degree constant at 0.4, and ran two experiments. In the first one, the number of activities within a cluster was 25, and in the second one it was 50. Figures 20 and 21 show the results. For 95% confidence intervals, $|CI| < 0.6$ in Figure 20, and $|CI| < 0.9$ for ***Greedy-Basic*** and ***Greedy-NMA***, and $|CI| < 4$ for ***Greedy-WU*** in Figure 21. The ***Greedy-WU*** algorithms is more sensitive as the number of edges in the cluster graph increases as it reshapes the graph after each iteration. Moreover, as seen from the graphs, the performance of the greedy algorithms does not degrade, and stays about the same as we increase the number of activities in the input sets. In both graphs, the ***Greedy-NMA*** algorithm produces the worst results as it explores the smallest number of alternatives when generating partitions. The quality of the partitions produced by the ***Greedy-Basic*** and the ***Greedy-WU*** algorithms are comparable. When the number of activities within a cluster is small, and hence there are fewer edges in the cluster graph, the ***Greedy-Basic*** algorithm does better than the ***Greedy-WU***. However, when the number of activities within a cluster increases the ***Greedy-WU*** algorithm catches up with the ***Greedy-Basic*** algorithm.

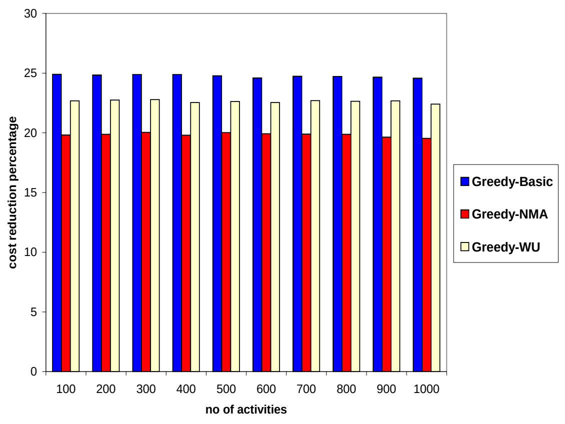

Figure 20: Savings for (o-prob=0.2, o-degree=0.4, no-of-activities/cluster=25, max-cost=100)

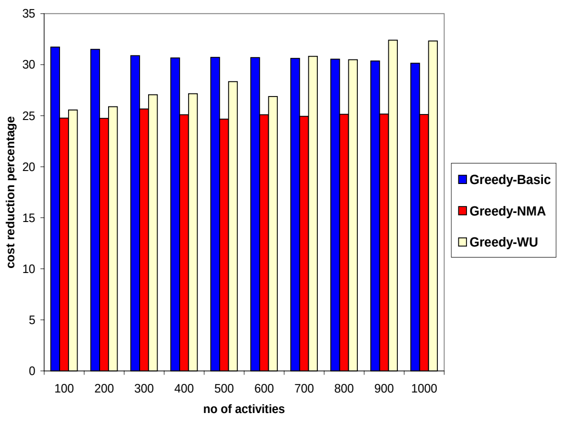

Figure 21: Savings for (o-prob=0.2, o-degree=0.4, no-of-activities/cluster=50, max-cost=100)

These results show that although the ***Greedy-WU*** algorithm uses more accurate cost saving estimates, it is still outperformed by the ***Greedy-Basic*** algorithm which tries to adapt its savings estimates as the computation of the output partition proceeds by moving activities across components. It turns out that this heuristic is more effective in capturing the real savings than updating weights. Moreover, ***Greedy-Basic*** is faster than ***Greedy-WU***, and it is less sensitive to the number of edges in the cluster graph. As a result, we believe that ***Greedy-Basic*** is the better choice.

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, the greedy algorithms scale up very effectively when many activities occur. Furthermore, the ***Hill-Climb*** algorithm also does not scale up very well, and produces worse quality results than the greedy algorithms. *The results suggest that the cluster graph representation is a very effective heuristic, and hence pairwise savings are a good first approximation.*

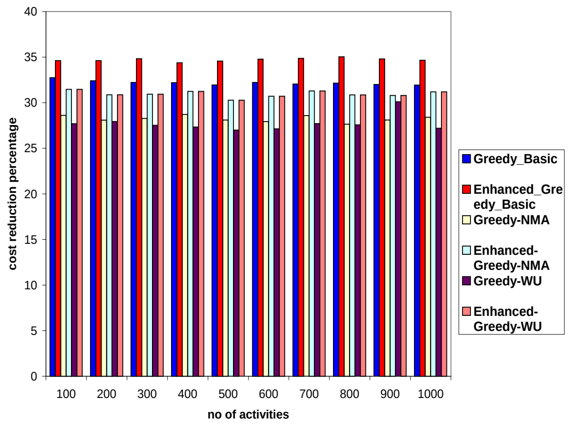

Figure 22: Savings for (o-prob=0.4, o-degree=0.4, no-of-activities/cluster=25, max-cost=100, max-no = 0.2 * no-of-activities)

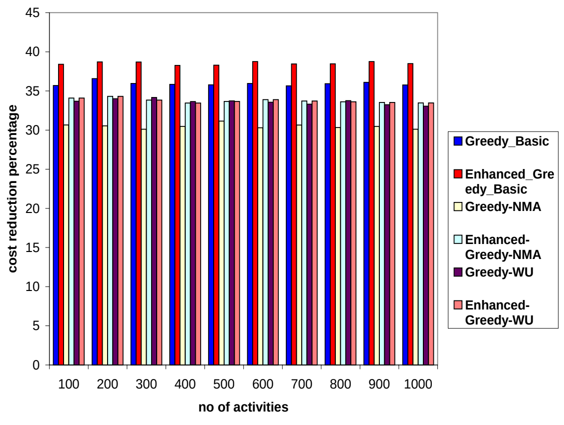

Figure 23: Savings for (o-prob=0.4, o-degree=0.6, no-of-activities/cluster=25, max-cost=100, max-no = 0.2 * no-of-activities)

4.1.3 Performance of the Enhanced Greedy Algorithms

We wanted to explore how the ***ImproveQuality*** procedure can improve the cost of the partitions produced by the greedy algorithms. We have created three new versions of the greedy algorithms: The ***Enhanced-Greedy-Basic***, the ***Enhanced-Greedy-NMA*** and the ***Enhanced-Greedy-WU*** algorithms are created by invoking the ***ImproveQuality*** procedure with the output partitions produced by the ***Greedy-Basic***, the ***Greedy-NMA*** and the ***Greedy-WU*** algorithms, respectively.

Figures 22 and 23 show the cost reduction percentages obtained when the overlap probability is 0.4, the number of activities considered (i.e. max_no) is 20% of the number of activities, and the overlap degree is 0.4 and 0.6. In Figure 22 $|CI| < 0.8$ and in Figure 23 $|CI| < 0.9$ for 95% confidence intervals. As seen from the Figures, the ***ImproveQuality*** procedure slightly increases the cost reduction percentages obtained by each greedy algorithm by 2-4%. On the other hand, the running times of the algorithms increase considerably. Figure 24 shows the running times of the greedy algorithms, and their enhanced versions. In Figure 24 $|CI|/average \times 100 < 4$ for 95% confidence interval. We see that the running times of the enhanced versions of the algorithms are double or more the running times of their vanilla counterparts. These results suggest that the greedy algorithms achieve substantial savings, and the quality of the partitions obtained by the ***ImproveQuality*** procedure does not offset its cost.

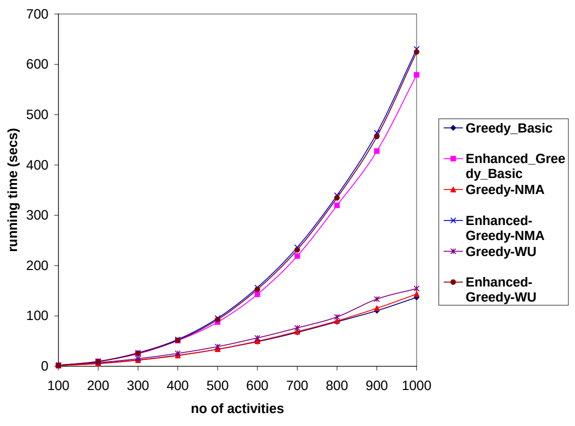

Figure 24: Running time of the Greedy Algorithms (o-prob=0.4, o-degree=0.4, no-of-activities/cluster=25, max-cost=100, max-no = 0.2 * no-of-activities)

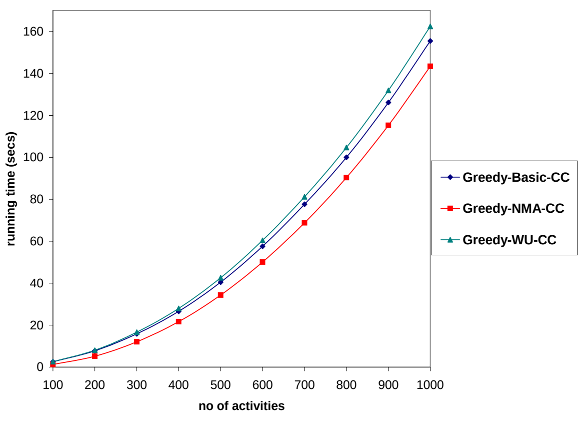

Figure 25: Running time when component size $\leq$ 25 (o-prob=0.2, o-degree=0.4, no-of-activities/cluster=50, max-cost=100)

4.2 Performance of the Algorithms with Component Constraints

To evaluate the performance of the algorithms with component size constraints, we ran two experiments. In both experiments, the overlap probability was 0.2, the overlap degree was 0.4 and the number of activities within each cluster was 50. In the first experiment, we examined the running times of the algorithm when k = 25. Figure 25 shows the running times of the algorithms. In this graph, $|CI|/average \times 100 < 2.2$ for 95% confidence interval. As can be seen from the figure, the execution time of the algorithms did not change much when component size constraints were present.

We also studied how savings change when we have component size constraints. Figures 26 and 27 show the cost reduction percentages when k = 25, and k =10, respectively. In this experiment, number of activities per cluster is 50. In Figure 26 $|CI| < 0.8$ and in Figure 27 $|CI| < 0.6$ for

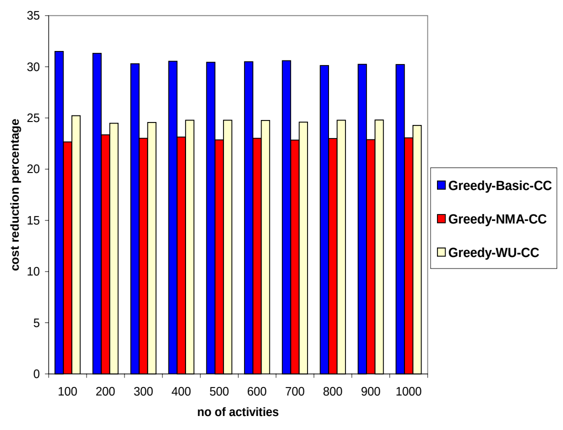

Figure 26: Savings when component size $\leq$ 25 (o-prob=0.2, o-degree=0.4, no-of-activities/cluster=50, max-cost=100)

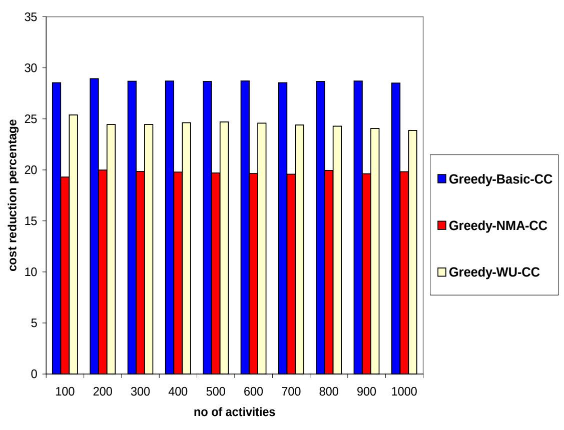

Figure 27: Savings when component size $\leq$ 10 (o-prob=0.2, o-degree=0.4, no-of-activities/cluster=50, max-cost=100)

95% confidence interval. As expected, the cost reduction percentages are slightly smaller (1%–4%) when the components are required to be less than or equal to 10. When we compare these cost reduction percentages with the case when there were no component size limits (Figure 21), hWU-n 0.Ñ.2(with)-30mponen Sar(WU-nby321.2(with)-1.7(WUze)]TJ 16. 4513 0 0 40014 T-.8(-)1 Tm ( )Tj 2 for multiple users interested in obtaining different sensory information about a given location or region. An agent that provides access to real-time stock data may participate in both numerous multi-agent applications (e.g. risk analysis applications, predictive applications, portfolio management applications) but also provide direct services to individual and institutional investors.

At any given point in time, agents of the kind mentioned above may have numerous activities to perform. Improving the performance of such agents poses a major challenge. Past work on this topic has focused on the following approaches to the problem:

1. **Caching strategies:** Adali et. al. [2, 1] and Ashish et. al. [4, 5] have developed strategies where the agents precompute certain critical components of their task and cache the results. This cache may then be used to improve the performance of the agent.

2. **Merging strategies:** In many domains, researchers have noticed that *merging* tasks is often preferable to servicing tasks in a queue fashion. This work goes all the way to the design of disk controllers. It is often preferable to schedule a *set* of disk reads and writes rather than execute them one at a time. The reason is that to read a sector on the disk, some rotational and radial movements may be needed to bring the disk head over the disk track and sector to be read[24]. Merging strategies have also been used by multimedia researchers [3, 14], database researchers [27] to merge queries and planning researchers[32] to merge the construction of multiple plans. In agents research, the only merging strategies we have seen is the work of Ozcan et. al. [21] who show how merging and caching methods can be used to scale agent performance.

Our work leverages from the two types of methods described above. It hinges on the observation that in most domains d, merging works well only for some number k_d of activities. The number varies from domain to domain. The reason is that as the number of tasks increases, the time taken to merge increases substantially — [27] proves that merging is NP-hard for databases, and Ozcan et. al. [21] prove that merging is NP-hard in most domains.

In this paper, we study the problem of what to do when an agent has a set of activities far exceeding that agent's k value. We propose a *splitting* strategy. In this strategy, we find a partition of the agent's activities. The partition consists of a set of components such that within each component are activities that "are similar" and that show promise of yielding substantial savings if they are merged together. We define the problem of finding an optimal partition and show the problem is NP-hard. We then define a set of algorithms to find partitions — some of these algorithms will find an optimal partition, while the ***Greedy*** algorithms will find suboptimal partitions. We have implemented these algorithms and conducted detailed experiments in order to determine which ones are the best.

There has been substantial work in the literature on graph partitioning algorithms [16, 22], which are somewhat similar to our greedy algorithms. Our greedy algorithms are based on the heuristic that modeling pair-wise savings between activities provides a good approximation. Since cluster graphs are only an approximation to the original problem, the optimal partitioning of the cluster graph does not correspond to the optimal solution to the activity partitioning problem. Hence, graph partitioning algorithms are not directly applicable.

Probably, the closest work on graph partitioning is by Pinar and Hendrickson [22], where the authors try to partition and distribute a set of tasks to a set of processors by also factoring in interactions among the partitions incurred due to inter-process communications. In their solution, they use one of the classical methods to come up with an initial partition, and then improve the cost by reassigning some of the tasks. Although their approach might be exploited for APP, coming up with an initial solution is not a trivial task. Besides, our goal is not merely to distribute the activities across agents, and balance the load, but our objective is to group together those activities that may share computations for a *single* agent.

It is important to note that the algorithms in this paper are not tied to a specific agent framework - as such, the principles outlined here are applicable to different agent environments such as [11, 13, 10, 26, 6]. Our work is also reminiscent of work on creating coalitions [28, 18, 25]. In [28], Sen and Dutta propose a stochastic search algorithm (called OBGA) to compute optimal coalition structures. Given a set of agents, they partition the agents into coalitions. A coalition is similar to a component in our framework. They focus on finding partitions, taking into account the fact that value of a coalition depends on the values of other coalitions (they argue that other coalition formations approaches like [18, 25] do not take this into account). We do not do this for components. Instead, we argue that the value of a component is an arbitrary function (satisfying the cost estimation axioms) of the contents of the component. They do not do this. Likewise, Shehory and Kraus [29] develop methods for allocating tasks between agents but do not address the problem of how a single agent buckets tasks so as to reduce its own load. Hannebauer[15] develops algorithms for distributing tasks between multiple constraint solvers. Given a set of tasks, they find a partition with high quality. The idea is that each component is a set of tasks that can be given to a specific agent. They measure quality of a partition by internal problem complexity and communication costs. We do not consider the latter (for our problem, it is irrelevant). But in addition, they use a specific quality function (given at the end of section 3.1 of [15]). In contrast, our work applies to *any* cost estimation function that satisfies the axioms applicable to cost estimation functions.

References

[1] 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.

[2] S. Adali and V.S. Subrahmanian. Intelligent caching in hybrid knowledge bases. In N. Mars, editor, *Proc. 1995 Intl. Conf. on Very Large Knowledge Bases*, pages 247–256, Twente, The Netherlands, May 1195. IOS Press.

[3] C. Aggarwal, J. Wolf, and P. S. Yu. On optimal piggyback merging policies for video-on-demand systems. In *Proceedings of SIGMETRICS*, pages 200–209, Philadelphia, PA, 1996.

[4] Naveen Ashish. Optimizing information agents by selectively materializing data. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference*, page 1168, Madison, Wisconsin, USA, July 1998. Doctoral Consortium Abstract.

[5] Naveen Ashish, Craig A. Knoblock, and Cyrus Shahabi. Selectively materializing data in mediators by analyzing user queries. In *Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems, (CoopIS)*, pages 256–266, Edinburgh, Scotland, September 1999.

[6] 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.

[7] R. R. Brooks and S. S. Iyengar. *Multi-Sensor Fusion: Fundamentals and Applications with Software*. Prentice Hall, 1998.

[8] K. Decker, K. Sycara, and D. Zeng. Designing a multiagent portfolio management system. In *Proceedings of AAAI Workshop on Intelligent Adaptive Agents*, 1996.

[9] M. Dell'Amico, F. Maffioli, and S. Martello (eds). *Annotated Bibliographies in Combinatorial Optimization*. J. Wiley & Sons, 1997.

[10] Thomas Eiter, V. S. Subrahmanian, and George Pick. Heterogeneous Active Agents, I: Semantics. *Artifical Intelligence*, 108(1-2):179–255, 1999.

[11] O. Etzioni and D. Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, 1994.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[13] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):49–53, 1994.

[14] L. Golubchik, J. C.-S. Lui, and R. R. Muntz. Adaptive piggybacking: a novel technique for data sharing in video-on-demand storage servers. *ACM Multimedia Systems Journal*, 4(3):140–155, 1996.

[15] M. Hannebauer. A formalization of autonomous dynamic reconfiguration in distributed constraint satisfaction. *Fundamenta Informaticae*, 43(1-4):129–151, 2000.

[16] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16:452–469, 1995.

[17] 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.

[18] K. Larson and T.W. Sandholm. Anytime coalition structure generation: An average case study. In *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 40–47. ACM Press, 1999.

[19] 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.

[20] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, 1986.

[21] F. Ozcan, V.S. Subrahmanian, and J. Dix. Improving performance of heterogeneous agents. Technical Report CS-TR-4204, UMIACS-TR-2000-77, Dept. of Computer Science, Univ. of Maryland, College Park, MD, USA, December 2000.

[22] Ali Pinar and Bruce Hendrickson. Partitioning for complex objectives. In *Proceedings of Irregular'01*, 2001.

[23] Thomas Rist, Elisabeth Andre, and Jochen Muller. Adding animated presentation agents to the interface. In *Intelligent User Interfaces*, pages 79–86, 1997. citeseer.nj.nec.com/rist97adding.html.

[24] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[25] T.W. Sandholm, K. Larson, M. Anderson, O. Shehory, and F. Tohme. Coalition structure generation with worsti case guarantees. *Artificial Intelligence*, 111(1-2):209–238, 1999.

[26] 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.

[27] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, June 1990.

[28] S. Sen and P.S. Dutta. Searching for optimal coalition structures. In *Proceedings of the 4th International Conference on Multiagent Systems*, pages 286–292, Boston, MA, July 2000.

[29] O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1-2):165–200, 1998.

[30] J.M. Swaminathan, S.F. Smith, and N.M. Sadeh. Modeling supply chain dynamics: A multi-agent approach. *Decision Sciences*, 29(3), 1998.

[31] P. Szolovits, J. Doyle, W.J. Long, I. Kohane, and S.G. Pauker. Guardian angel: Patient-centered health information systems. Technical Report TR-604, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, May 1994.

[32] Q. Yang, D. Nau, and J. Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(2):648–676, 1992.