A Theory of Plan Merging

David E. Foulser *   Ming Li †   Qiang Yang ‡

Key words: Task planning, plan merging, longest common subsequence, shortest common supersequence, dynamic programming.

Abstract

Merging operators in a plan is an important way for plan optimaization. Few attempts were made in formalizing operator merging in a mathematically rigorous way. This paper provides a computational theory with which optimal and approximate algorithms are available for plan optimization. The optimal solution extends dynamic programming method to handle nonlinear plans in a novel way. Furthermore, with worst and average case complexity analysis and empirical tests, we demonstrate that efficient and well-behaved approximation algorithms are applicable for optimizing plans with large sizes.

1 Introduction

The value of helpful or positive interactions among the different parts of plans has been recognized early in AI planning research [?, ?, ?, ?]. A helpful goal interaction occurs in a plan, or among several plans, if certain portions of the plan can be modified to improve its quality. An important type of helpful goal interaction occurs when certain operators in a plan can be grouped, or *merged*, together in such a way as to make the resulting plan more efficient to execute. This happens often

*Department of Computer Science, Yale University, 2158 Yale Station, New Haven, CT 06520, USA. Supported in part by Office of Naval Research grant N00014-86-J-1906 and by NIH Grant R01 LM05044 from the National Library of Medicine.

†Department of Computer Science, University of Waterloo. Waterloo, Ontario, Canada, N2L 3G1. Supported in part by NSERC operating grants OGP0036747 and OGP0046506.

‡Department of Computer Science, University of Waterloo. Same address as above. Supported in part by NSERC operating grant OGP0089686.

in domains where redundant setup and restore operations can be eliminated in the execution of consecutive tasks and where redundant journeys can be eliminated by fetching multiple objects at once.

Few attempts have been made in developing a computational theory on formalizing operator merging in planning. In this paper, we provide a formalization for plan merging. In particular, we review the kinds of plan merging that have been considered in the previous systems, and discuss the complexity of the problem in general. Based on the formalization, we present a dynamic programming algorithm for determining the optimal solution by reducing the problem to the shortest common subsequence problem, a variant of the longest common subsequence problem, and apply several known results from that area. We also extend the dynamic programming method to handling partially ordered plans in a novel way. One drawback of the dynamic programming method is that it becomes computationally infeasible for problems of larger sizes. While we are able to phrase an optimal algorithm for general purpose, domain-independent plan merging, its time may be prohibitive for inputs of practical sizes. To make the planning problem more tractable, most existing systems that consider helpful interactions employ certain kinds of greedy algorithms for plan merging [?, ?, ?, ?]. Thus, we also describe four approximation algorithms for merging plans, and analyze their average case merging complexity. The approximation algorithms all have low-order polynomial time complexity.

The remainder of the paper has the following organization. Section 2 reviews the problem domains for planning methods and surveys some recent work in this area. Section 3 gives a formal description of plan merging. Section 4 presents the shortest common supersequence (SCS) as the optimal merging of several linear plans, develops the computational algorithm for the SCS, and indicates extensions to weighted plan mergings, to mergings of partially ordered plans, and to the generation of partially (rather than linearly) ordered plans as output of the merging method. Section 5 develops four approximation plan merging algorithms and analyzes their behavior. Section 6 presents experimental results for the four heuristic methods. Conclusions are stated in section 7.

2 Background

We consider a plan as a partially ordered set of operators, where each operator is defined through its preconditions, effects, and a cost value. An agent executing a plan may also have certain knowledge about the domain that enables it to make various inferences about the plan. Using knowledge concerning the positive relationships among parts of the plan, the agent can combine operators to reduce the plan's cost. Given this knowledge, there may be many ways of combining a plan's operators with a different cost for each combination. Substantial computational complications may arise when various possible helpful interactions occur simultaneously. We now describe several problem domains where this type of inference about plans plays an important role. We then review previous attempts at solving this planning inference problem.

2.1 Problem Domains

Plan optimization via operator merging is important in many domains. In human problem solving, people constantly take advantage of positive goal relationships to allow reduction in plan costs. Many examples are given by Wilensky [?]. Operator merging is equally important in areas of robot task planning and scheduling. For example, consider a blocks world domain where blocks are of different sizes. To pick up one block of a certain size, the robot arm has to mount a gripper of appropriate size. Suppose that only one robot arm exists, and in order to grab a block of a different size, the robot has to unmount the current gripper and mount the gripper with the new size. In this case, it is more efficient to group block stacking operations that use the same type of grippers. Consider the example shown in Figure 1, where blocks are of three different sizes. Suppose that the robot can use 3 different grippers of type A, B and C to pick up the corresponding type of block. Then a minimal cost way for achieving the goal would be to move $A1$ and $A2$ to the table together, move B to the table, then move C to the top of $A1$ and $A2$, and finally, move B to the top of C. By moving $A1$ and $A2$ together, the robot is able to merge a gripper unmount/mount pair into a null action, saving one gripper change operation.

The blocks world problem is illustrative of the role of operator merging in robotic assembly domains. Identical plan merging issues also arise in the domain of automated manufacturing where process plans for metal-cutting [?], set-up operations [?] and tool-approach directions [?] need to be optimized. Similarly, in the area of query optimization in database systems [?], as well as domains having multiple agents [?, ?], operator merging in multiple plans seems inevitable.

The general framework common to these problem domains is that operators in one plan or several plans can often be merged to improve the quality or reduce the cost of the overall plan.

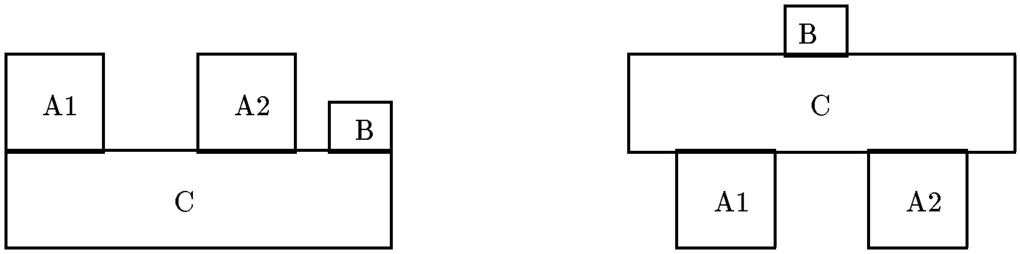

Figure 1: Blocks world where blocks are of different sizes.

For large plans with many expensive operations, or for repetitively executed plans, it may be very profitable to determine the optimal merged plan. However, when many opportunities exist for operator merging, it may be expensive to find the optimal overall plan. We now review the previous planning systems that aim at taking advantage of operator merging.

2.2 Previous Work

NOAH [?] is one of the first nonlinear planners that explicitly seek opportunities for plan merging. It relies on its set of Critics to handle possible interactions among the different parts a plan. Three critics are introduced for purpose of improving the quality of plans, including "Eliminate redundant preconditions," "use existing objects" and "optimize disjuncts." For example, the "eliminate redundant preconditions" critic can be considered as merging two or more operators that are used to achieve the same preconditions, as long as no precedence relation in the plan is violated. SIPE [?] and NONLIN [?] are other planning systems that have more advanced capabilities in operator merging. For example, SIPE is able to recognize that a goal is achievable by another operator in the current plan. In such situations, it will impose constraints on orderings and variable bindings so that the operator is used to achieve it. This process is known as *phantomization* of a goal, and is used for the purpose of improving plan quality.

Hayes [?] proposed a domain-dependent approach for operator merging in machining domains. In her Machinist system, a greedy algorithm is implemented, and compared with human machinists in terms of the quality of the plans produced. It was shown that it can out-perform an experienced human process planner.

As an intermediate approach that falls between domain-independent and domain-dependent extremes, Yang, Nau and Hendler [?] proposed a limited-interaction approach to planning, taking into consideration operator merging as one type of interactions allowed. In recognizing the complexity of the general problem, they imposed certain restrictions that work well for a number of domains. An interesting feature of their system is that they considered the case where there is more than one alternate plan for each goal, and proposed a branch and bound algorithm for selecting the plans that can be merged optimally from a set of alternative choices.

Wilensky [?] considers *positive goal relationships*, in the context of cognitive modeling of human problem solving. Two goals have positive relationship if the achievement of one helps the achievement of the other, and thus it is more economical to achieve both together. Since it was intended for analyzing the types of goal relationships occurring at a meta level, Wilensky's analysis did not provide a detailed computational theory for plan merging.

The majority of planning research so far has been aimed at dealing with negative interactions in plans such as conflicts and resource competitions [?, ?, ?, ?]. Although the systems described above also attempt to handle positive interactions, what seems lacking is a theory that rigorously formalizes operator merging in a mathematically rigorous way, and efficiently computes either optimal plans or heuristically determined high quality plans. We now formalize the problem of merging partially ordered plans.

3 Formal Description of the Problem

3.1 Formal Definition of Operator Merging

Given a set of goals to be achieved, a plan Π is a partially ordered set of operators, where each operator α is represented by preconditions P_α and effects E_α, which for the sake of simplicity are assumed to be sets of literals (positive or negative atomic sentences). A plan Π may be represented as a graph $\Pi = (O, B)$ where the vertices O are the set of operators in Π and the edges B are the set of precedence relations in Π. In a least-commitment plan, there is usually a set of constraints on the binding of variables, known as codesignation and noncodesignation constraints [?], which we omit here for simplicity.

In addition to preconditions and effects, we also assume that each operator α has an associated cost $\mathrm{cost}(\alpha)$, and that the cost for a plan Π, denoted $\mathrm{cost}(\Pi)$, is the *sum* of the costs of the operators in Π.

In a partially ordered plan, operator α necessarily precedes β if α precedes β under every consistent total ordering of the plan. We will use $\Box$ and $\Diamond$ to denote necessarily and possibly, respectively, and $\prec$ to denote precedes.

For a given plan, there may be some operators in the plan that can be grouped together, and replaced by a less costly operator that achieves all the useful effects of the grouped operators. In such a case, we say that the operators are *mergeable*. We formalize this notion below.

We start by defining what operators in a plan can be grouped together. A set of operators Σ in a plan $\Pi = (O, B)$ induces a subplan (Σ, B_Σ) within Π, where B_Σ is a maximal subset of B that are relations on Σ. Operators in Σ can be *grouped together* if and only if no other operator outside Σ is necessarily between any pair in Σ. More precisely,

$$\forall \alpha, \beta \in \Sigma, \neg\exists\gamma \in O(\Pi) - \Sigma, \text{ such that } \Box(\alpha \prec \gamma) \textit{ and } \Box(\gamma \prec \beta).$$

Of particular interest are operators that are unordered with respect to one another, although we shall see in a later section that dependent operators can be effectively pipelined. Depending on the applicable cost function, arbitrary plans can be merged by first merging all unordered sets of operators and then pipelining the merged results.

We assume that the plan Π is justified, in that every operator in Π is useful in establishing some preconditions of other operators, or the goals themselves. Operator α necessarily establishes a precondition p for an operator β, denoted by $\text{Establishes}(\alpha, \beta, p)$ [?], if and only if p is a precondition of β and effect of α, and no operator necessarily between α and β has p as an effect.

Let Π_Σ be a subplan of $\Pi = (O, B)$ induced by Σ, where Σ is a set of operators that can be grouped together. Consider the collective behaviour of Π_Σ in Π. Some effects of the operators in Σ are *useful* in Π, because they establish the preconditions of some *other* operators in Π that are outside Σ, or the goals directly. Other effects are either side-effects of the operators, or are used to achieve the preconditions of the operators within Σ. We use $\text{NetEffects}(\Sigma, \Pi)$ to denote the set of all useful effects of the operators in Σ. Likewise, we use $\text{NetPrecond}(\Sigma, \Pi)$ to denote the set of all preconditions of the operators in Σ not achieved by any operators in Σ. More formally,

$$\text{NetEffects}(\Sigma, \Pi) = \{e \in E_\alpha \mid \alpha \in \Sigma, \exists\beta \in (O - \Sigma) \textit{ such that } \text{Establishes}(\alpha, \beta, e)\}.$$

$$\text{NetPrecond}(\Sigma, \Pi) = \{p \in P_\beta \mid \beta \in \Sigma, \exists\alpha \in (O - \Sigma) \textit{ such that } \text{Establishes}(\alpha, \beta, p)\}.$$

We now give a more precise definition for plan merging. A set Σ of operators is *mergeable* in a plan $\Pi = (O, B)$ if and only if $\exists B', \mu$, where B' is a set of precedence relations on Σ , and μ is an operator, such that

1. Σ can be grouped together in Π,
2. $B' \bigcup B$ is consistent, and in plan $\Pi' = (O, B \bigcup B')$

   $$P_\mu \subseteq \text{NetPrecond}(\Sigma, \Pi'),$$

   and

   $$\text{NetEffects}(\Sigma, \Pi') \subseteq E_\mu.$$

   That is, the operator μ can be used to achieve all the useful effects of the operators in Σ while requiring only a subset of their preconditions, after precedence constraints B' is imposed on plan Π.
3. $\text{cost}(\mu) < \text{cost}(\Sigma)$.

The operator μ is called a *merged* operator of Σ in the plan Π, and denoted by $\mu = m_\Pi(\Sigma)$ (or simply $\text{m}(\Sigma)$ if it is clear about the plan Π).

The definition for operator merging clearly covers the examples given in the previous section. As an example in the common sense domain, consider a plan for achieving two goals, going to school S from home H and back, and going from home to the grocery store G to get X and back. This plan consists of two subplans,

$$goto(H, S) \prec goto(S, H),$$

$$goto(H, G) \prec get(X, G) \prec goto(G, H).$$

The two subplans can be merged by replacing $goto(S, H)$ and $goto(H, G)$ by $goto(S, G)$. If G is located between home H and school S, then the resultant plan

$$goto(H, S) \prec goto(S, G) \prec get(X, G) \prec goto(G, H)$$

costs less than the original one.

In the above example, the set B' of precedence relations is $\{goto(S,H) \prec goto(H,G)\}$. Let Π' be the plan with B' imposed on the original plan, then

$$\text{NetPrecond}(\Sigma, \Pi') = \{at(S)\}, \ \text{NetEffects}(\Sigma, \Pi') = \{at(G)\}$$

where $\Sigma = \{goto(S,H), goto(H,G)\}$. Thus, Σ can be merged into the operator $m(\Sigma) = goto(S,G)$. Notice that there is another way of merging the operators in this plan, i.e., $\{goto(G,H), goto(H,S)\}$ can be merged into $goto(G,S)$, but one has to decide which way to merge since it is impossible to merge both sets of operators. The ability optimally to choose between several inconsistent possible mergings is an important feature of our dynamic programming algorithm.

Let Π be a plan and Σ be a set of operators in Π that are mergeable. Then after merging Σ in Π, every precedence relation in Π that involves an operator $\alpha \in \Sigma$ and an operator $\beta \notin \Sigma$ is replaced by the same relation, except every occurrence of α is replaced by μ. Let the new set of precedence relations be *NewB*. Formally, let α, β be operators in $O(\Pi)$ such that $(\alpha \prec \beta) \in B$, then

$$NewB = \begin{cases} \alpha \prec \beta \text{ if } \alpha, \beta \notin \Sigma \\ \mu \prec \beta \text{ if } \alpha \in \Sigma, \beta \notin \Sigma \\ \alpha \prec \mu \text{ if } \alpha \notin \Sigma, \beta \in \Sigma \end{cases}$$

The plan Π with the operators in Σ merged into μ is defined as a plan Π', in which

1. $O(\Pi') = (O(\Pi) - \Sigma) \bigcup \mu$, and
2. $B(\Pi')$ is *NewB*.

The merged plan is denoted as $\text{Merge}(\Pi, \Sigma, \mu)$.

Operators can be merged in both correct or temporally incorrect plans. A plan $\Pi = (O, B)$ is *correct* if B is a partially ordered relation, every precondition of every operator in O is necessarily established, and all goals are necessarily achieved. Operator merging, as defined here, is not a means of making an incorrect plan correct, but rather to make a plan more efficient. Therefore, in general, we do not require that an incorrect plan be made correct after merging. However, we would like to ensure that after merging, the correctness of a plan is preserved. This is guaranteed by the following theorem:

Theorem 3.1 *Let Π be a correct plan and Σ be a set of operators in Π that are mergeable. Let μ be a merged operator of Σ. Suppose that $E_\mu \subseteq$ NetEffects(Σ, Π). Then* Merge(Π, Σ, μ) *is also correct.*

Proof: ¿From the condition that $E_\mu \subseteq$ NetEffects(Σ, Π) and the definition for merged operator μ, we have $E_\mu =$ NetEffects(Σ, Π). Therefore, replacing Σ by μ does not add any additional side-effects while preserving all the useful effects of operators in Σ. Also from the definition of μ, the set of preconditions of μ is a subset of NetPrecond(Σ, Π). Furthermore, the additional orderings imposed on Merge(Π, Σ, μ) are consistent with that in Π. Therefore, all preconditions and goals are necessarily established in Merge(Π, Σ, μ). The latter is correct as a result. $\square$

It can be shown by induction that the correctness of a plan is preserved by merging operators in a plan any number of times, as long as the condition in the theorem is satisfied.

3.2 Finding Mergeable Operators

There are two important issues in optimizing Π. The first one is to find the set Σ of mergeable operators and, for each set of mergeable operators, find one or more merged operators μ. The second issue is computing the optimal way to merge the plan, given that a set of merged operators are found.

The problem of finding operators that can be merged in a plan Π is very difficult if no additional domain knowledge is given. To make the process more efficient, various domain knowledge can be employed. For example, one way for the operators in Σ to be merged is when they contain various sub-operators which cancel each other out, in which case the merged operator μ would correspond to the set of operators in Σ with these sub-operators removed. This case corresponds to what Wilensky [?] calls "partial-plan merging" [?]. Another is when all the operators in Σ share a common schema, and in this case, the goals of these operators can be achieved by executing the schema only once. This case corresponds to what Wilensky calls "common-schema merging."

In this paper, we assume that knowledge is available about what operators can be merged, and for each set of these operators, what the merged operators are. In particular, we assume that each operator α in Π is associated with one or more "types", Types(α), so that a group of operators in Σ are mergeable if and only if they all share the same type. For example, in the robot task planning domain, two tool-changing operators have the same type if and only if the tools involved have identical features.

In the remaining of the paper, we concentrate on the second issue, that of finding and analyzing methods for computing the optimal plan. We will call this problem the *plan merging problem.* We start by discussing its complexities in the next section.

3.3 Complexity

Several complications exist that make the plan merging problem computationally expensive in general. First of all, for a given set Σ of operators to be merged, there may be several alternative merged operators, $\{\mu_1, \ldots, \mu_i\}$, to choose from, each with a different set of preconditions , effects and cost value. Second, an operator may lie in the intersection of several non-identical groups of operators, but not all operators in these groups may be merged, even though all the operators in question are unordered in a plan. For example, in the blocks world domain, there may be a gripper capable of picking up blocks of sizes A and B, and another gripper capable of picking up blocks of sizes A and C, but no gripper can pick up a block of type B and C. Then a gripper-changing operator for picking up a block of type A may be merged that for either B or C, but not all three can be merged together. An optimization algorithm has to make a choice in such a situation. A third complication occurs because the partial order on Π may render inconsistent some pairs of mergings. For example, consider the two plans

$$a_1 \prec c_1 \prec b_1 \text{ and } a_1 \prec b_1 \prec c_1,$$

where operators can have three types, a, b or c. Thus, after operator merging, one merged plan is

$$m(\{a_1, a_2\}) \prec b_2 \prec m(\{c_1, c_2\}) \prec b_1,$$

another merged plan is

$$m(\{a_1, a_2\}) \prec c_1 \prec m(\{b_1, b_2\}) \prec c_2.$$

However, it is impossible to merge both pairs b_1, b_2 and c_1, c_2.

To remove the first complication, we assume that for each set Σ of mergeable operators, there is a unique merged operator μ. In addition, we assume that operator merging is an atomic action, so that the merged operator μ may not be combined with any other operators.

The second complication is easily handled by imposing a distance metric on the operators. The distance metric induces a cost function for unmerged and merged sets of operators. The problem is then to find the merged plan with the minimal cost. Section 4.4 below indicates a few useful cost functions. However, for the sake of simplicity in describing the algorithms, we will assume that, as in [?], operators are partitioned into equivalence classes, such that a set of operators in Σ are mergeable only if they belong to the same equivalence class.

We now consider the computational complexity as a result of the third complication. The problem is to decide which set of mergeable operators to merge, if temporal orderings prevent all of them to be merged together. To see the complexity involved, consider a simplified plan Π, consisting of a set of linear sequences of operators. Assume that the cost of a plan is the total number of operators in the plan, and that a set of operators are mergeable if and only if they belong to the same equivalence class and are unordered with respect to each other. Then the plan merging problem is equivalent to the problem of finding the Shortest Common Supersequence, that is, finding the shortest sequence S such that every sequence in Π is a subsequence of S. We describe the problem in more detail below.

4 Optimal Plan Merging via Shortest Common Supersequences

Let there be k input plans S^1, S^2, ..., S^k, each of which is a linear sequence of operators drawn from a finite assortment (or alphabet) $\mathcal{A} = \{1, \ldots, m\}$ of operator types. Let the $|S^i|$ denote the length of S^i. We write a sequence $S^i = s^i_1 s^i_2 \ldots s^i_{|S^i|}$.

A supersequence $U = u_1 \ldots u_P$ of $S = s_1 \ldots s_M$ has the property that there exist $i_1 < \cdots < i_M$ such that $u_{i_1} = s_1$, ..., $u_{i_M} = s_M$. In other words, the operators of S can be found in order in the sequence U. We say that U is a common supersequence of S and T if it is separately a supersequence of S and of T. Elements of S and T may overlap in a common supersequence U, so that $|U| \leq |S| + |T|$. For any set of input sequences, there exists at least one SCS.

Initially, suppose an input plan is composed of linear task sequences and we desire a linear merged plan. Let a task sequence of length N have cost N. In this case, the optimal task schedule for an input plan is the shortest common supersequence (SCS) of the linear input task sequences. It is known that the SCS can be computed analogously to the longest common subsequence (LCS) [?], which has been extensively analyzed in the literature [?].

The problem of finding the SCS is NP-complete [?]. We show that with a fixed number of input sequences, the SCS may be calculated in polynomial time.

As described in Section 2, many planning systems adopted approximation algorithms for operator merging, partly because of the perceived intractability of the SCS computation. We analyze the performance of a spectrum of approximation algorithms as an alternative to the potentially large polynomial cost of generating the optimal plan. But to begin with, we first consider the problem of finding the optimal solution using dynamic programming methods.

For the simplicity of exposition, we will describe our algorithms and their analysis with a simplifying assumption: that the plan Π consists of linear sequences of operators, and that operators can be merged if and only if they belong to separate sequences and of the same type. After introducing the methods, we consider how to relax these assumptions in sections 4.6 and 4.7.

4.1 Relationship of SCS and LCS

The key observation in relating the SCS to LCS is that each overlap of p identical tasks from p input sequences produces a $p-1$ savings in overall cost in the supersequence. In fact, each subsequence of length n common to p input sequences produces a cost savings of $n(p-1)$ in the common supersequence. In other words, each reduction in the SCS length is attributable to a corresponding increase in the LCS length. Therefore, the SCS can be determined by finding the LCS weighted to account for multiplicity of overlaps in the input sequences. It is important to note that the LCS is typically calculated with unit weights independent of degree of overlap, which is not sufficient to form the SCS.

We now display the relationship of the LCS and SCS lengths. Let $L = l_1 \ldots l_n$ be an operator sequence, each of whose operators is common to at least two of the input sequences $S^1, \ldots, S^k$. Let the cost of L, $C_L = \sum_{i=1}^{n} c(l_i)$, be the sum of the costs of its elements, where the cost $c(l_i)$ of an element l_i is one less than its repeat multiplicity in the inputs. For instance, if l has multiplicity three, its cost is $c(l) = 2$. Then C_L represents the degree of overlapping of $S^1, \ldots, S^k$ encompassed in L. The choice of $|\mathrm{LCS}| = \max_L C_L$ is a multi-dimensional extension of the standard LCS measure on two input sequences.

Theorem 4.1 *Let* $|\mathrm{LCS}| = \max_L C_L$ *and let* $|\mathrm{SCS}|$ *be the number of elements in the SCS of* k

input sequences S^i, $1 \le i \le k$. *Then*

$$|\mathrm{LCS}| + |\mathrm{SCS}| = \sum_{i=1}^{k} |S^i|.$$

The theorem generalizes to multiple input sequences the known result for a pair of inputs [?], allowing us to find the SCS by finding the weighted LCS of the input sequences. We omit the proof of the theorem (cf. Manber [?]) and instead present an algorithm for determining the SCS.

4.2 Computational Domain for SCS Calculations

Consider a k-dimensional computational array M, each of whose elements is an ordered m-tuple indicating the multiplicities of each of the m letters of the alphabet $1, 2, \ldots, m$. To dimension i of M assign the sequence S^i, giving a size $|S^1| \times |S^2| \times \ldots \times |S^k|$. Define the m-tuple contents of an element of M to be

$$(p_1, \ldots, p_m) = M(j_1, j_2, \ldots, j_k),$$

where p_l is the multiplicity of letter l in $s^1_{j_1}, \ldots, s^k_{j_k}$, for $1 \le l \le m$. Note that $p_l \ge 0$ and that $\sum_{l=1}^{m} p_l = k$. Define also the identically-sized array A, which will be used to represent the answers of the sequence computation.

We now show how the m-tuples of M may be used to calculate subsequence and supersequence lengths in A. Any monotone path on lattice elements from $A(0, \ldots, 0)$ to $A(|S^1|, \ldots, |S^k|)$, which proceeds by making unit increments to one or more of its coordinate indices as indicated below, describes a supersequence of S^1, ..., S^k.

Let T trace a monotone path from $A(0, \ldots, 0)$ to $A(|S^1|, \ldots, |S^k|)$, whose segments join elements from A of the form

$$A(i_1, i_2, \ldots, i_k) \rightarrow A(i_1 + \delta_1, i_2 + \delta_2, \ldots, i_k + \delta_k),$$

where each $0 \le \delta_1, \ldots, \delta_k \le 1$ and where there are identical sequence letters $s^{j_1}_{i_1+1} = s^{j_2}_{i_2+1} = \ldots = s^{j_r}_{i_r+1}$ wherever $\delta_{i_1} = \delta_{i_2} = \ldots = \delta_{i_r} = 1$. In terms of the input sequences, the path segments identify common letters in a monotone order. Thus the monotone path that T traces on M and A has a unique correspondence to a supersequence of the sequences S^1, ..., S^N.

The cost of a path T is simply the number of segments from $A(0, \ldots, 0)$ to $A(|S^1|, \ldots, |S^k|)$. The cost lies in the range $\max(|S^1|, \ldots, |S^k|) \le |T| \le \sum_{i=1}^{k} |S^i|$. The minimal length path is easily computed by reference to the m-tuples of M using the following recurrence relation (let the partial path lengths be stored in the elements of A):

$$A(i_1, \ldots, i_k) = \min_{1 \leq j \leq m} 1 + A(i_1 - \delta_1, \ldots, i_k - \delta_k),$$

where each δ_i is either 0 or 1 and all incremented indices refer to identical letters in the input sequences. The basis for the recurrence relation is that a minimal path through $A(i_1, \ldots, i_k)$ must contain the minimal path from $A(0, \ldots, 0)$ to $A(i_1, \ldots, i_k)$. Let $A(i_1, \ldots, i_k) = \infty$ if any $i_j < 0$.

4.3 Dynamic Programming Algorithm for the SCS

Let M represent the array of sequence letter multiplicities. Let A represent the shortest common supersequence cost, with the understanding that $A(i_1, \ldots, i_k) = 0$ if any $i_j \leq 0$ is out of bounds. Let $L(i_1, \ldots, i_k; l)$ represent the set of sequence indices j for which sequence elements $s^j_{i_j} = l$. Let R represent the reverse links necessary to reconstruct the actual SCS on completion of the algorithm. We now present the initialization and computation phases of the algorithm.

Initialization of $M(i_1, \ldots, i_k)$ and $L(i_1, \ldots, i_k; l)$:
for $i_1 = 1$ **to** $|S^1|$ **do**
    for $i_2 = 1$ **to** $|S^2|$ **do**
        $\vdots$
        for $i_k = 1$ **to** $|S^k|$ **do**
            for $l = 1$ **to** m **do**
                $L(i_1, \ldots, i_k; l) = \{$ set of sequence indices for which $s^j_{i_j} = l\}$
                $p_l = |L(i_1, \ldots, i_k; l)|$
            endfor
            $M(i_1, \ldots, i_k) = (p_1, \ldots, p_m)$
        endfor
        $\vdots$
    endfor
endfor

Compute the cost $A(i_1, \ldots, i_k)$ and the reverse links $R(i_1, \ldots, i_k)$:
for $i_1 = 0$ **to** $|S^1|$ **do**

```
    ⋮
    for i_k = 0 to |S^k| do
        minval = ∞
        for l = 1 to m do
            for all non-empty subsets σ ⊂ L(i_1, ..., i_k; l) do
                for j = 1 to k do
                    δ_j = { 1  if j ∈ σ
                          { 0  otherwise
                endfor
                if minval > 1 + A(i_1 − δ_1, ..., i_k − δ_k) then
                    minval = 1 + A(i_1 − δ_1, ..., i_k − δ_k)
                    R(i_1, ..., i_k) = σ
                endif
            endfor
        endfor
        A(i_1, ..., i_k) = minval
    endfor
    ⋮
endfor
```

The key steps are the location of the path segment yielding the minimal path and the contributing sequence indices σ, updating the cost to reflect the minimal path, and setting the reverse link to that path segment. The reverse links R are indicated by storing the vector of sequence indices whose letters match and are to be merged. In order to reconstruct the SCS from R, one simply follows path segments starting from $R(|S^1|, \ldots, |S^k|)$ to $R(0, \ldots, 0)$, which yields the reverse of T.

The cost of the method is $O(\prod_{i=1}^{k} |S^i|)$, assuming a fixed number k of input sequences and a fixed alphabet size m.

4.4 Extensions for Linear Inputs

One may consider a variety of weightings applied to the basic SCS problem. Each type of weighting will reflect the effects of combining tasks in various ways.

4.4.1 Task Weightings

First consider the weighted SCS in which each task l has an arbitrary non-negative cost $w(l)$, for $1 \leq l \leq m$. The task weighted SCS is now computed by our algorithm with the modification that minval is updated according to

$$\text{minval} = w(l) + A(i_1 - \delta_1, \ldots, i_k - \delta_k).$$

In this setting, those tasks with large weights are to be preferentially overlapped, even if their number of occurrences is small.

4.4.2 Overlap Weightings

The model problem assumes that two identical tasks can be overlapped to yield only the cost of a single task. One might define an adjusted cost function $w(j)$ for $1 \leq j \leq k$ to give the cost of overlapping j identical elements. One then uses the weighted SCS iteration described above. It is worth noting that if $w(j)$ is chosen so that

$$w(j_1 + j_2) < w(j_1) + w(j_2),$$

then the weighted method will produce a weighted SCS with an overlap of multiplicity $j_1 + j_2$, whereas the opposite ordering on w allows an overlap of multiplicity j_1 followed immediately by an overlap of multiplicity j_2, which may be less helpful in plan merging.

It is straightforward to combine the features of task weightings and overlap weightings to yield the weight function $w(l, j)$ based on the letter l and the degree j of overlap.

4.4.3 Non-identical Overlaps

In other problem domains to which the LCS methods are applied, it often makes sense to talk of non-identity overlaps, or inexact mergings. One might therefore consider assigning a fixed matching weight to each combination of two or more tasks, on the assumption that a set of j non-identical tasks could be combined to effect some savings in cost.

In this case, one defines the weighting function $w(k_1, \ldots, k_m)$ based on k_i occurrences of letter i, for $1 \leq i \leq m$. The inner loop of the SCS computation is then taken over all 2^k subsets of the tasks $s^1_{i_1}$, ..., $s^k_{i_k}$.

4.4.4 Pipelined Overlaps

All of the previous models have considered overlapping individual tasks from two or more sequences. However, it is quite possible that a succession of identical overlaps from the same sequences could have lower cost than the sum of the individual overlaps. For instance, if

$$S^1 = S^2 = (1,1,1,2,2,2,3,3,3),$$

one could create three overlapped pairs of ones, then a triplet of twos, then a triplet of threes. In the pipelined regime, the successive overlapping ones could be successively processed to achieve a cost savings over three separate pairs of ones. The necessary weight function $w(i,j,p)$ is based on j occurrences of letter i in separate sequences pipelined to a depth of p, where $1 \leq j \leq k$. For pipelined weights such that

$$w(i,j,p+p') < w(i,j,p) + w(i,j,p'),$$

long pipelined occurrences are encouraged. Furthermore, under this circumstance, only a count of the current pipeline depth p need be kept in order to maintain an $O(\prod_{i=1}^{k} |S^i|)$ algorithm. In some cases where the subadditivity condition is violated, one obtains an algorithm with running time $O(\max_i |S^i| \prod_{i=1}^{k} |S^i|)$, again assuming fixed m and k.

4.5 Efficient Implementations

Several implementation efficiencies can be realized with our SCS algorithm. For instance, the sparse LCS computation of Hirschberg [?] can be modified to quickly calculate the SCS when very few common tasks are shared between sequences. The methods of Myers [?] and Ukkonen [?] for computing the LCS of nearly identical inputs could be modified for an SCS computation. The SCS problem on multiple input sequences could benefit from the theoretical work of Altschul and Lipman [?], who devised a way to use all pairwise comparisons to constrain the search space for the optimal LCS.

There are also a number of heuristic methods embodied in computer software for comparing long DNA and amino acid (protein) sequences, some of which could be modified for use in the SCS problem. A few authors whose LCS methods could be used to advantage in the SCS problem are mentioned in the references [?, ?, ?, ?, ?, ?, ?].

4.6 Dynamic Programming Methods for Partially Ordered Inputs

Just as the dynamic programming method can be used to compute an optimal plan from two or more linear input sequences, a similar computation determines the optimal merged plan from a partially ordered input plan. To fix ideas, we consider as input a single plan P, partially ordered by Π, which is the union of one or more disjoint, connected, partially ordered plans. The method we present creates the optimal merged plan from P.

The basis of the computation is a multi-dimensional grid created as follows. Let ω represent a set of operators from P, such that all operators in ω are unordered by Π with respect to one another. We say that ω is maximally unordered if every operator in $P - \omega$ is ordered by Π with respect to ω.

Maximally unordered elements ω have a partial ordering induced by the partial order on the operators in ω. Thus for ω_1 and ω_2, $\omega_1 \prec \omega_2$ iff $\nu_1 \prec \nu_2$ for all $\nu_1 \in \omega_1$ and $\nu_2 \in \omega_2$. Then let M be the set of all maximally unordered ω derived from P. M is the computational domain for the generalized dynamic programming algorithm.

As an example, consider P to be the union of two linear plans $S = s_1 s_2 \ldots s_m$ and $T = t_1 t_2 \ldots t_m$. Each maximally unordered ω is a pair of operators $\omega = (s_i, t_j) \in S \times T$. Then M is just the set of all ordered pairs $M = \{(s_i, t_j)\}_{i=1,j=1}^{m,n}$, which, under the orderings induced by S and T, gives the $m \times n$ computational grid of the standard dynamic programming algorithm.

The computation of an optimal plan proceeds as in the linearly ordered case, starting from the subset of M containing elements with no preconditions and proceeding through their descendants to the elements with no net-effects. As with the case of linear inputs, the optimal plan is constructed as a monotone sequence of elements in M, where the transition from ω_1 to an adjacent ω_2 indicates the merging of two or more plan operators in P, or the inclusion of a single unmerged operator in the resulting optimal plan.

The computational work of constructing the optimal linear plan is proportional to $|M|$, the number of maximally independent sets of operators taken from the partial order graph of the input plan P. The computational complexity of the method is strongly dependent on the degree to which the plan P is ordered or not. For $P = \cup_i S^i$, a union of linear input sequences, the cost is thus the product of sequence lengths $\prod |S^i|$, which may be computationally intractable for even a small number of moderate-length sequences. For a binary tree containing N nodes, there seem to be exponentially many maximally independent sets of operators, giving exponential complexity. However, for a plan in which each of N tasks is unordered with respect to at most a constant number c of other operators, the maximum number of unordered sets is linear in N. Therefore one might expect to find practical instances in which the optimal plan determination is computationally feasible.

4.7 Retaining Partial Order Information

As normally conceived, the dynamic programming algorithm produces a linearly ordered output plan. However, it is possible to modify the dynamic programming methods to retain a fair degree of the partial ordering of the input plans. The basic concept is as follows: between two task mergings in the optimal supersequence there may exist several task insertions without any merging. These insertions may be done in any order consistent with the partial order Π. While the dynamic programming method typically orders these insertions, it may be reformulated to leave them partially ordered for later processing. For instance, the two plans

$$\text{AAABCDEEE and AAAFGHEEE}$$

can be merged to form

$$\text{AAA}\begin{Bmatrix}\text{BCD}\\ \text{FGH}\end{Bmatrix}\text{EEE}$$

or any consistent linear plan; the partially ordered output may be preferable for planning purposes, particularly in the case of parallel process execution by multiple agents.

5 Approximation Algorithms

For problems of large sizes, the complexity of the dynamic programming methods may be too high to be useful. An alternative is to use approximation algorithms that operate in low-order polynomial time but output a suboptimal supersequence. An important issue is to determine the qualities of these algorithms in worst and average cases. In this section, we consider a spectrum of algorithms and their analysis, and in the next section, we show the experimental results. For each algorithm below, we assume the following input and output:

Input: A set Π of plans, which are assumed to be a set of k linear sequences of operators, each sequence has n operators.

Output: A merged superplan S for Π, which is a supersequence of the plans in Π.

It is also assumed that the operators are drawn from equivalence classes $\alpha_1, \alpha_2, \ldots, \alpha_m$, where m is assumed to be a constant in this section for simplicity.

As stated above, our key motivation is to develop and analyze approximation algorithms that perform in low-order polynomial time. Thus, the main concern here is not the time complexity of the algorithms, but the quality of the superplan produced by each algorithm. Therefore, the term "complexity" below refers to the total number of operators in the merged superplans.

Given a plan $\Pi = (O, B)$, one can find a set of operators lacking preconditions Start(Π). Formally,

$$\text{Start}(\Pi) = \{\alpha \in O \mid \neg\exists\beta \in O,\ \beta \prec \alpha.\}$$

Let Σ be a set of operators in a plan $\Pi = (O, B)$. The updated plan remove(Σ, Π) is the plan with all operators in Σ removed, and all the precedence relations relevant to operators in Σ removed.

All algorithms below basically operate by sweeping through the input plans in a left to right manner. In the sweeping process, operators that are mergeable are merged and collected into a superplan. Although the description of the algorithms and subsequent analysis assume the left to right way of merging, all the results apply equally to merging in the opposite direction, *i.e.*, in a right to left way. Such a process does not violate any existing precedence constraint in Π.

Below we discuss each algorithm in turn, and consider their worst case and average case complexities in terms of the total number of operators in the merged plan. The algorithms are discussed in increasing order of sophistication.

5.1 Algorithm M1

Our first algorithm, M1, is the most greedy one. It looks for as many merges as possible in each iteration. In particular, it takes an operator on the left side of the remaining plan Π, and looks for nearest merges by searching through each of the next plans from left to right for operators that can be merged with α. The operators that are mergeable with α can be considered as forming a "thread" that partitions the plan Π into three subplans, where Π_{11} is on the left of the thread, Π_{12}

is on the right of the thread, and Π_2 is the set of those sequences not touched by the thread. Below is a recursive version of the algorithm.

Algorithm M1(Π).

1. If $\Pi = \emptyset$ then return $\emptyset$. Otherwise, arbitrarily find $\alpha \in \text{Start}(\Pi)$. Let Σ be a leftmost maximal set of operators in Π mergeable with α. Let μ be the merged operator of Σ.

2. Partition Π into two sets of sequences, Π_1 and Π_2, such that each sequence in Π_1 contains an operator in Σ, and no operator in any sequence of Π_2 is a member of Σ.

3. For each operator sequence S in Π_1, let α' be the operator in Σ. Split S at α' into S_1 and S_2, so that $S = S_1 || \alpha' || S_2$. Let Π_{11} be the set of all S_1, and Π_{12} be the set of all S_2.

4. Return $\text{M1}(\Pi_{11} \bigcup \Pi_2) || \mu || \text{M1}(\Pi_{12})$.

Theorem 5.1 *Algorithm M1 has worst case complexity $O(n^2)$.*

Proof: We will consider plans with just two operators α, β. We will construct n plans each being a sequence of n operators from $\{\alpha, \beta\}$, arranged as rows in an $n \times n$ matrix M. We will show that M1 will construct a supersequence of length $\Omega(n^2)$ for some M. We now construct M. The first row of M contains entirely α. Row i of M contains $n - i$ initial tasks α, followed by $i - 1$ tasks β, and then a terminal α. At step i of the algorithm, M1 removes column i from the first $n - i$ rows, and the entirety of row $n + 1 - i$. M1 is then applied to an equivalent submatrix of size $n - i \times n - i$. Each merge requires $n + 1 - i$ elements in the merged plan, for a total cost of $n(n + 1)/2$. Concatenation of the input plan gives n^2 tasks, so the worst case behavior is indeed $\Theta(n^2)$. □

5.2 M2

Algorithm M2 is less greedy, and is the most straightforward algorithm. It merges the operators in a plan Π in a left to right scanning process. In each iteration, it merges all of the leftmost operators into the m types of merged operators in the supersequence, removes the leftmost operators from the remaining plan and continues until no operators are left in the original plan.

Algorithm M2.

1. $S := \emptyset$,

2. let $\Sigma :=$ Start(Π). Partition Σ into m classes, such that each class Σ_i contains operators that are mergeable. Let μ_i be the merged operator, for each class i.

3. $\Pi :=$ remove(Σ, Π),

4. For $i = 1, 2, \ldots, m$, $S :=$ insert(μ_i, S).

5. If Π is empty, then return S, otherwise, goto 2.

Theorem 5.2 *The worst case and average case complexity for M2 are both mn.*

The proof for the theorem is trivial, and is thus omitted here.

5.3 Algorithm M3

The next algorithm, algorithm M3, is slightly more sophisticated than M2 in that during each iteration, it only merges the operators in the partitioned subclass Σ_i with the greatest cardinality.

Algorithm M3.

1. $S := \emptyset$,

2. let $\Sigma :=$ Start(Π), Partition Σ into m classes, such that each class Σ_i contains operators that are mergeable. Let Σ_l be the subclass with the largest cardinality, and let μ be the merged operator for Σ_l.

3. $\Pi :=$ remove(Σ_l, Π),

4. $S := insert(\mu, S)$,

5. If Π is empty, then return S, otherwise, goto 2.

We now analyze the worst case and average case complexity of M3. M3 appeals to our intuition as a more aggressive algorithm. However, as the following theorem shows, it actually performs worse than the trivial algorithm M2 in the worst case. This is of course counterintuitive since we expect M3 performs better in general. Such intuition is captured in our average case analysis: for a random instance, M3 does perform provably better than M2. We first give the worst case analysis and then give the average case analysis (under uniform distribution).

Theorem 5.3 *Given a set of n plans each of length n, the worst case complexity of M3 is $\Theta(n \log n)$.*

Proof: We first give a set of n plans each containing n operators in $\{\alpha, \beta\}$, from which algorithm M3 will produce a superplan of size $\Omega(n \log n)$. We conveniently arrange the plans into an n by n matrix M such that each row of M corresponds to a plan, one operator per entry. We recursively construct M: Its first $\lfloor n/2 \rfloor + 1$ rows are full of α. The first column of the rest of the rows contains β. Therefore, according to its description, algorithm M3 always chooses the first column of the first $\lfloor n/2 \rfloor + 1$ rows to merge at the first n steps. At step $n+1$, M3 chooses the first column of β from the rest of rows. After the execution of the first $n+1$ steps, we are left with a matrix of roughly $\lfloor n/2 \rfloor - 1$ by $n-1$. If we recursively apply above construction to the remaining matrix, it is apparent that the merge process will go on for $\Omega(n \log n)$ steps.

We now show that M3 always produces a superplan no more than size $O(n \log n)$ on plans drawn from a binary task alphabet (for alphabet size $k > 2$ the proof is similar). The minimum number of operators possibly merged by M3 in successive steps is a monotonically decreasing function $f(k)$ of step number k. The worst case behavior of M3 maximizes the step number at which $f(k)$ vanishes. (We approximate $f(k)$ by its continuous analog $f(x)$.) All realizations of $f(x)$ have $\int_0^\infty f(x)dx = n^2$, that is, M3 merges all n^2 tasks into a superplan. As well, $f(x) \geq (2n)^{-1} \int_x^\infty f(y)dy$, for M3 is able to merge at least half of the number of plans not yet exhausted by step k. The worst case behavior of M3 is obtained by setting $f(x)$ to its minimum value at all times. In this case, $f(0) = n/2$ and $f(x) = (2n)^{-1} \int_x^\infty f(y)dy$, with solution $f(x) = (n/2)e^{-x/2n}$. For large n, this functions first vanishes when $x = cn \log n$ for some $c > 0$.

□

We now examine the average case complexity of M3. We employ a new tool, Kolmogorov complexity, in the proof of the following theorem. An equivalent proof of somewhat greater generality is obtainable using probability theory, but the Kolmogorov argument is simpler. The Kolmogorov complexity $K(x)$ of a string x, over alphabet $\{\alpha_1, \ldots, \alpha_m\}$, is the length of shortest PASCAL program (encoded over the same alphabet) that prints x with empty input. We call x Kolmogorov random (or, simply, random) if $K(x) \geq |x| - \log |x|$. Easy counting shows that there are at least $(1-1/n)2^n$ random strings of length n. We refer the reader to Li and Vitanyi [?] for an introduction to Kolmogorov complexity.

Theorem 5.4 *Given n random plans each containing at most n operators from the set* $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, *and for any small positive constant* ϵ, *the average case complexity of M3 is no greater than* $n(m+1)/2 + O(n^{1/2+\epsilon} \log n)$.

Proof: We first give an informal description of the ideas. We will consider a fixed set of Kolmogorov random plans, which includes nearly all plans of a given length. The advantage of considering such set is that if we can show that on this *particular* set of plans M3 has certain complexity then M3 has the same complexity on average over all plans. Arrange a single Kolmogorov random string of length n^2 into a matrix row by row, giving n plans. Each of these plans is also Kolmogorov random. Then the claim is that, in the first column, there are almost exactly n/m operators of each type α_i. And if we merge tasks of type α_j, then the operators ordered after the merged α_j must be again almost evenly divided into m groups with n/m^2 tasks for each α_i; in the Kolmogorov case there are no realizations which deviate significantly from the mean. This nice property continues to hold until some row is completely removed, but by then, the rest of rows are all pretty short (about $O(n^{1/2+\epsilon})$). Further we show any Kolmogorov random string (over some m alphabet) of length n is a subsequence of $\pi^{n(m+1)/(2m)+o(n)}$ for any permutation π of the m letter alphabet.

Without loss of generality, we assume that we merge n plans, each containing n operators from $\{\alpha_1, \ldots, \alpha_m\}$[1]. Take a Kolmogorov random string x of length n^2 such that $K(x) \geq |x| - \log |x|$ and partition it into an $n \times n$ plan array M, whose rows are random plans.

It is now sufficient to show that, for this random M, M3 constructs a supersequence of length at most $n(m+1)/2 + O(n^{1/2+\epsilon} \log n).Thisisbecausemostmatrices$,1- $1_{\overline{n}}$ portion of them, are Kolmogorov random within a logarithmic additive factor (*i.e.*, $K(x) \geq |x| - \log |x|$). We will show that M3 constructs merged plans of length $n(m+1)/2 + o(n)$ for fraction $1 - \frac{1}{n}$ of the inputs, and of

[1] In case some plan contains less than n operators, we can always add to it a (uniform) random sequence (over our alphabet) so that it has length n. Then in the superplan, we remove these added operators. We consider plans of same length only in order to provide a clean analysis.

length $n \log n$ in the worst case for the other $\frac{1}{n}$ of inputs. On average, our bound follows. We need the following fact about Kolmogorov complexity.

Lemma 5.5 *Let x be a Kolmogorov random string over an m letter alphabet with $|x| = n$. Define an ij-block to be adjacent tasks of types α_i followed by α_j. Fix any $\epsilon > 0$. Let $K(x) \geq n - O(\log n)$. Then for large n,*

(1) For each $i \in \{0, \ldots, m-1\}$, i appears in x at most $\frac{n}{m} + n^{1/2+\epsilon}$ times, and at least $\frac{n}{m} - n^{1/2+\epsilon}$ times.

(2) For each $i, j \in \{1, 2, \ldots, m\}$, x contains at most $\frac{n}{m^2} + n^{1/2+\epsilon}$, and at least $\frac{n}{m^2} - n^{1/2+\epsilon}$, ij-blocks.

(3) Let s be a subsequence of x. We write $x - s$ to denote the new string formed by deleting s from its corresponding places from x. We say s is an enumerable subsequence of x if there is a program, p, such that $K(p|x - s) \leq O(\log n)$, that outputs s together with the locations of each character of s in x. For any enumerable subsequence s of x of length $n' = n^\delta$, for $\delta > 0$, (1) and (2) are true with x and n replaced by s and n' respectively.

Remark. Intuitively, this lemma simply states that every letter appears equally likely in a random sequence and in an "easily enumerable" subsequence of the random sequence (since this subsequence also has to be random). In particular, the fisrt row of M is of course "enumerable"; Also the sequence of tasks (say, in top down order) that appeared behind the group merged by M3 at some step is an enumerable order. This important fact will be used later, and will be simply referred to as Lemma 5.5 (1)(3).

Proof: We prove the second statement in (1) by contradiction. Suppose letter $t \in \{1, \ldots, m\}$ appears in x at most $d = \frac{n}{m} - n^{1/2+\epsilon}$ times. Then there are at most $\binom{n}{d}(m-1)^{n-d}$ strings of length n with d occurrences of t. By standard an estimation[2], $\log_m \binom{n}{d}(m-1)^{n-d} \leq n - \delta n^\epsilon$ for some $\delta > 0$. Hence x can be coded in $n - \delta n^\epsilon$ digits, implying that x is not random and a contradiction. A similar argument proves the first half of (1).

In order to prove (2), one only needs to observe that by pairing the letters two ways: x_{2i-1} with x_{2i} for $i = 1, 2, \ldots, n/2$, and x_{2i} with x_{2i+1} for $i = 1, 2, \ldots, n/2 - 1$, we reduce the problem to two half size problems in form of (1) with alphabet size m^2. Then (2) follows immediately.

[2] An estimation can be found in M. Li and P. Vitanyi, introduction to Kolmogorov complexity and its applications, Addison-Wesley, To appear.

(3) automatically follows from (1), (2), and the definition of a $\log n$-enumerable subsequence of x.

$\square$

Lemma 5.6 *For Kolmogorov random inputs x, M3 will keep outputting π^*, where π is a fixed permutation of $\alpha_1, \alpha_2, \ldots, \alpha_m$, until some row of M is completely deleted (*i.e.*, some plan is totally merged).*

Proof: We prove by induction, on steps (loops) of M3, with the following induction hypothesis from which the lemma follows: Before some row is completely removed,

(a) For each operator α_i, Σ always contains at least $n/m^2 - n^{1/2+\epsilon}$ tasks α_i.

(b) If at step s an operator α_i is put into the supersequence, then from the end of step s (after deleting the α_i from Σ), for each operator α_j there will be always at least $n/m^2 - mn^{1/2+\epsilon}$ more α_j than α_i in Σ until α_j is chosen to be in the supersequence (*i.e.*, α_j in Σ are deleted).

At the first step, Item (a) is true by Lemma 5.5 (1)(3). Item (b) is trivially true since nothing is deleted yet. Assume (a) and (b) are true until the end of step s and operator α_i is put into the supersequence at step s. To prove (a) we note that at step s we must have deleted at least n/m α_i by algorithm M3. It is easy to see that the subsequence appearing behind the deleted letters is enumerable. By Lemma 5.5 (1)(3), each operator appears at least $n/m^2 - n^{1/2+\epsilon}$ times in this subsequence of length at least n/m, from which (a) follows. In order to prove (b), notice that the number of each element is increased by $d/m \pm n^{1/2+\epsilon}$ uniformly, where $d \geq n/m$ is the number of α_i deleted from Σ in step s. But only the operator α_i starts from 0, and all others had at least $O(n/m^2)$ elements in Σ already. At each of the next m rounds, all operators are increased by almost equal amount with a variance of up to $n^{1/2+\epsilon}$, which will never make up the discrepancy. (Note the difference here between Kolmogorov random strings, from which we have excluded the set of strings having variance that could reverse the order of some (i, j) pair, and probabilistically random strings, which contain such inputs with small probability.) So operator α_i will only become the most frequent letter after every other operator has appeared in the supersequence exactly once. We thus have proved (b).

By (a) and (b), it is clear that M3 will continue to output $(\pi)^*$ until some row of M (some plan) is completely merged, where π is a fix permutation of $\alpha_1, \alpha_2, \ldots, \alpha_m$. $\square$

Lemma 5.7 *Let $K(x) \geq n - O(\log n)$ where $x \in \{\alpha_1, \ldots, \alpha_m\}^*$ and $n = |x|$. Let π be a permutation of $\alpha_1 \ldots \alpha_m$. Let $h(n, \epsilon) = n^{1/2+\epsilon}$ for notational convenience. Then*

(1) x is a subsequence of $\pi^{n(m+1)/(2m)+h(n,\epsilon)}$ for any $\epsilon > 0$.

(2) x is not a subsequence of $\pi^{n(m+1)/(2m)-h(n,\epsilon')}$ for any fixed ϵ'.

Proof: By Lemma 5.5 (2), in x there are at least $(\frac{1}{2} - \frac{1}{2m})n - O(n^{1/2+\epsilon})$ (consecutive) pairs $x_i x_{i+1}$ in x such that x_i appears in π before x_{i+1}. For such a pair, x_{i+1} is fitted into π for "free". The other $n(\frac{1}{2} + \frac{1}{2m}) + o(n)$ pairs require one copy of π per task, so therefore x can fit into a sequence $\pi^{cn+o(n)}$ for $c = \frac{1}{2} + \frac{1}{2m}$, which proves (1). But there are also at least same amount of (consecutive) pairs $x_j x_{j+1}$ such that x_j appears in π after x_{j+1}. For such a pair x_{j+1} needs to be put into a next π block. By letting ϵ in Lemma 5.5 (2) be less than the above ϵ', we have proved (2). □

We complete the proof of main theorem. By Lemma 5.6, M3 continues to output a prefix of π^*, where π is a permutation of $\alpha_1 \ldots \alpha_m$, until some row r of M is completely deleted. However, by Lemma 5.7, every row of M is a subsequence of $\pi^{n(m+1)/(2n)+h(n,\epsilon)}$ for any $\epsilon > 0$, but not a subsequence of $(\pi)^{n(m+1)/(2m)-h(n,\epsilon')}$ for any fixed ϵ'. So when row r of M is completely deleted, all the rest of rows are of length at most $O(n^{1/2+\epsilon})$. From this point on, the worst case analysis (Theorem 5.3) of M3 guarantees that M3 will finish within $O(n^{1/2+\epsilon} \log n)$ steps. When r is totally deleted, the partial supersequence has length at most $n(m+1)/2 + O(n^{1/2+\epsilon})$, by Lemma 5.7 (1). Therefore the total length of the supersequence constructed for M is $n(m+1)/2 + n^{1/2+\epsilon} \log n$ for any small $\epsilon > 0$.

Then since M is formed from a Kolmogorov random string, this is also the average length, for the remaining (non-Kolmogorov random) matrices contribute to the total average at most an $n \log n$ term times their average $1/n$ occurrence. □

Remark. We provide another point of view. Since the Kolmogorov random inputs have the behavior that M3 outputs a repeating pattern (Lemma 5.6), we can model its merging behavior by a simple Markov chain on m states. Let state σ_i refer to the task group Σ_l with i^{th} greatest cardinality. When tasks are merged, all tasks in state σ_1 are randomly distributed to other states, while tasks in σ_2, ..., σ_m move to states σ_1, ..., σ_{m-1}, respectively. The probability transition matrix for the process is just

$$\begin{pmatrix} 1/m & 1/m & 1/m & \cdots & 1/m \\ 1 & 0 & & & \\ & 1 & 0 & & \\ & & \ddots & \ddots & \\ & & & 1 & 0 \end{pmatrix}$$

and its stationary vector is $2/(m(m+1))(m, m-1, \ldots, 1)$. At each step of M3, there is thus a compression ratio of $2n/(m+1)$, so that the n^2 tasks can be merged to form a superplan of expected length $n(m+1)/2$.

5.4 Algorithm M4

Algorithm M4 combines the advantages of both M2 and M3. As M3, it looks for the largest set of mergeable operators to merge from the left frontier of the remaining plan. However, it is careful to avoid the worst case behavior of M3. Also similar to M2, it also collects all the other operators on the left frontier as well, and merges them before looking for new operators to merge in the next iteration.

Algorithm M4.

1. $S := \emptyset$,

2. let $\Sigma := \mathrm{Start}(\Pi)$, and $T := \{\mathrm{Types}(\alpha) \mid \alpha \in \Sigma\}$.

3. Until T is empty, do

- **a.** Partition Σ into m classes, such that each class Σ_i contains operators that are mergeable. Let Σ_l be the subclass with the largest cardinality, and let μ be the merged operator for Σ_l.
- **b.** $\Pi := \mathrm{remove}(\Sigma_l, \Pi)$, $S := \mathrm{insert}(\mu, S)$, $T := T - \mathrm{Types}(\mu)$.
- **c.** $\Sigma := \{\alpha \mid \alpha \in \mathrm{Start}(\Pi) \textit{ and } \mathrm{Types}(\alpha) \in T\}$.

5. If Π is empty, then return S, otherwise, goto 2.

Theorem 5.8 *The worst case complexity of M4 is mn and the average case complexity of M4 is the same as M3.*

As is expected, the algorithms which merge tasks using the least amount of information (M1 and M2) perform the worst, while the algorithms using more global information (M3 and M4) perform better on the average. Our analyses are not compared to the behavior of the optimal plan merging, as its average case behavior is an open problem. A related open problem is the average case behavior of the LCS of two or more strings drawn from a finite alphabet.

As in the case for the optimal dynamic programming method, it is also possible to extend our four algorithms to handling as well as outputing partially ordered plans. The basic idea is to update the plan after each merging step by leaving as many as possible unordered operators.

To verify our theoretical analysis, we have conducted a series of experiments, which are discussed in the next section.

6 Experimental Results

In this section, we compare the empirical behavior of the algorithms over several sets of randomly generated test cases. Each random test case is a set of linearly ordered sequences of operators with equal lengths. Each sequence is generated by assuming a uniform distribution of operator types. Test cases are distinguished by three parameters: the size of the operator alphabet, the length of each input sequence and the number of input sequences. Test programs were written in Kyoto Common Lisp.

The tests are grouped into two classes. The first class aims at comparing each approximation algorithm with the optimal solution generated using the dynamic programming method. Each test datum obtained in this class corresponds to the average result over five inputs. Figure 2 shows the length of the supersequences generated by the approximation and optimal algorithms as a function of the length of each input sequences. Figure 3 shows the results as a function of alphabet size. It is clear from these tests that algorithm M4 performs the best on the average among all four algorithms, while M1 performs the worst. As the length of each input sequence increases (Figure 2), algorithms M1 and M2 perform increasingly worse when compared with the optimal, while M3 and M4 stay fairly close to the optimal solutions. In Figure 3, M3 and M4 perform much better than M1 and M2 with small alphabet sizes. But as the size of the alphabet increases, all four approximation algoirthms deviate from the optimal. Since the dynamic programming algorithm has a higher time and space complexity, no tests were done with the changing number of input sequences.

The second group consists of tests comparing the performance of the approximation algorithms with large input sizes. Each test in this group corresponds to the average over 10 randomly generated data. For those input sequences, the optimal dynamic programming algorithm becomes infeasible to execute. Figures 4, 5 and 6 show the performance of each algorithm as a function of the length of each input sequence, the size of the alphabet, and the number of input sequences, respectively. It is again clear that algorithms M3 and M4 perform increasingly better than M2, which in turn performs increasingly better than M1 with the length of input sequence and the size of alphabet. Also note that no algorithm is affected by the number of input sequences (Figure 6), as might be inferred from our average complexity theorems. Further, it is worth noting that as the length and number of input sequences gets larger, the empirical behavior of the algorithms converges closer to our theorectical average case analysis in the previous section.

In Section 5 we stated that algorithm M1 not only has the highest worst case complexity, it also has the worst average case behavior. It is our conjecture that M1 has a average case complexity of $O(n \log n)$, where n is the length of input sequences, taking plans with binary alphabet as input, and taking the length and the number of input sequences being equal. In supporting our conjecture, we conducted experiment of M1 . The result is shown in Firure 7, which shows that the average case behavior of M1 is indeed worse than $O(2n \log n)$.

In summary, we conclude that algorithm M4 has the best empirical performance among all approximation algorithms when the input size gets large. However, with small enough input sizes, the difference between them is not significant.

7 Conclusion

In this paper, we have presented a representational formalism as well as a computational theory for optimal and approximate plan merging. With plans of relatively small sizes, our dynamic programming method can be used to compute the optimal solution. Various extensions of the algorithm are considered that extends the results in supersequence calculations in a novel way. For plans with large sizes, the optimal algorithm is no longer feasible. In such cases, approximation

Figure 2: Tests with fixed number of sequences ($k = 4$) and alphabet size ($m = 2$). Each datum is an average over 5 random inputs.

Figure 3: Tests with fixed number of sequences ($k = 4$) and sequence length ($n = 10$). Each datum is an average over 5 random inputs.

Figure 4: Tests with fixed number of sequences ($k = 4$) and alphabet size ($m = 2$). Each datum is an average over 10 tests.

Figure 5: Tests with fixed number of sequences ($k = 4$) and sequence length ($n = 10$). Each datum is an average over 10 tests.

Figure 6: Tests with fixed alphabet size ($m = 2$) and sequence length ($n = 10$). Each datum is an average over 10 tests.

Figure 7: Tests with fixed alphabet size ($m = 2$). Each datum is an average over 10 tests.

algorithms can be utilized that computes plans with high quality. We have presented approximation algorithms with increasing degrees of greediness, and shown, through theoretical and empirical analysis, that the algorithm M4 achieves the best worst and average case performance.