Planning to Gather Information

Chung T. Kwok and **Daniel S. Weld**
ctkwok, weld@cs.washington.edu
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Abstract

We describe Occam, a query planning algorithm that determines the best way to integrate data from different sources. As input, Occam takes a library of site descriptions and a user query. As output, Occam automatically generates one or more plans that encode alternative ways to gather the requested information.

Occam has several important features: (1) it integrates both legacy systems and full relational databases with an efficient, domain-independent, query-planning algorithm, (2) it reasons about the capabilities of different information sources, (3) it handles partial goal satisfaction *i.e.*, gathers as much data as possible when it can't gather exactly all that the user requested, (4) it is both sound and complete, (5) it is efficient. We present empirical results demonstrating Occam's performance on a variety of information gathering tasks.

Introduction

The exponential growth of the Internet and World Wide Web has produced a labyrinth of documents, databases and services. Almost any type of information is available *somewhere*, but most users can't find it, and even expert users waste copious time and effort searching for appropriate information sources. Artificial intelligence and database researchers have addressed this problem by constructing integrated information gathering systems that automatically query multiple, relevant information sources to satisfy a user's information request (Etzioni & Weld 1994; Knoblock 1995; Levy, Srivastava, & Kirk 1995). These systems raise the level of the user interface, since they allow the user to specify *what* she is interested in without worrying about *where* it is stored or *how* to access the relevant sources (Etzioni & Weld 1994).

These motivations inspire the Occam[1] planning system which we describe in this paper. Occam automates the process of locating relevant information sources from a repository of source *descriptions* and combining them appropriately to answer users' information requests. Unlike previous implemented systems, Occam integrates relational databases and legacy systems (*i.e.*, those that do not support a comprehensive query interface such as SQL) with an efficient, domain-independent, query-planning algorithm.

A Very Simple Example

Suppose we want to find out the names of all people in an office. If we knew of a relational database containing this information, gathering the information would be easy, but suppose no such database exists. Instead we have only two information sources, namely, the UNIX *finger* command which returns the names of people given their email addresses, and the *userid-room* command which returns email addresses of all the occupants in an office. We can answer the query by first issuing the *userid-room* command and then running *finger* on each of the email address returned. The Occam planner reasons about the capabilities of information sources (*e.g.*, legacy systems such as *finger*, *userid-room* as well as more powerful relational databases) in order to synthesize a sequence of commands that will gather the requested information. Since Occam realizes that information sources may not be exhaustive, when necessary it generates multiple plans in order to gather as much information as possible.

Context

In contrast to previous work from the AI planning community, Occam uses an action language that is designed

[1] William of Occam, 1285–1349, was an English scholastic philosopher best known for "Occam's razor," a rule in science and philosophy which states that the simplest of two or more competing theories is preferable. Occam is quoted as saying "It is vain to do with more what can be done with less." The Occam system embodies this philosophy by seeking the simplest plans that gather all information requested by the user.

to represent information sources; this enables a highly specialized planning algorithm. For example, the only preconditions to Occam operators are *knowledge preconditions* (Moore 1985; Etzioni *et al.* 1992). Furthermore, since the operators executed by Occam are requests to information sources, we need not model causal effects; hence, there are no *sibling-subgoal interactions* such as those characterizing the Sussman anomaly. Occam does not model the world state as do many other AI planners; instead it models the *information state*, which is a description of the information collected by Occam at a particular stage in planning.

In contrast to work on multidatabase systems, Occam provides a single unified world model that is independent from the conceptualization used by the information sources; this greatly simplifies integration of new sources. Moreover, Occam is more expressive than many multidatabase systems, since it is able to model the presence of incomplete information in sources. Unlike most multidatabase systems, Occam is equally adept at extracting information from both legacy systems and full relational databases.

Representing Sites & Queries

Conceptually, Occam allows the user to interact with Internet services through a single, unified, relational database schema called the *world model*. For example, the world model might represent information about a person's email addresses with the *relation schema* `email(F,L,E)`, where `F`, `L`, and `E` represent the *attributes* firstname, lastname and email address respectively. Another relation schema `office(F,L,O)` could be used to record that the person with firstname `F` and lastname `L` has office `O`. Occam also associates a type with each variable and attribute; a variable may only appear as an attribute in a relation if the type matches.

Information-Producing Sites

We represent Internet services and data sources by modeling the type of queries they are capable of handling and by specifying a mapping between their output and relations in the world model. Both purposes are achieved with *operators* which have two parts:

1. A *head* which consists of a predicate symbol denoting the *name* of the operator, and an ordered list of variables called *arguments*. Each variable is possibly annotated with a *binding pattern* (Rajaraman, Sagiv, & Ullman 1995) that indicates that the argument must be *bound* in order for the query to be executed (denoted with the annotation $). Variables with no annotation are *free*.
2. A *body* which is a conjunction of atomic formulae whose predicate symbols denote relations in the world model.[2]

For clarity, we depict operators as expressions with the head on the left, an implication symbol, and the body on the right:[3]

$$op(\mathtt{X}_1,\ldots,\mathtt{X}_n) \Rightarrow \mathtt{r}_1(\ldots,\mathtt{X}_i,\ldots) \wedge \ldots \wedge \mathtt{r}_m(\ldots,\mathtt{X}_j,\ldots)$$

This specification says that when *op* is executed it will return some number of tuples of data, where each tuple may be thought of as an assignment of values to the head's arguments $\mathtt{X}_1,\ldots,\mathtt{X}_n$. The operator specification dictates that for each tuple returned, the logical formula formed by replacing all occurences of the arguments in the body with the corresponding tuple values is satisfiable.

For example, one can model the UNIX finger command with the following operator:

$$finger(\mathtt{F,L,\$E,O,Ph}) \Rightarrow \mathrm{email(F,L,E)} \wedge \mathrm{office(F,L,O)} \wedge \mathrm{phone(O,Ph)}$$

Intuitively, this means that when given an email address (the bound variable `$E`), *finger* produces a set of variable bindings for the free variables `F` `L`, `O` and `Ph`. For example, when `E` is bound to `"sam@cs"`, the following tuples might be returned:

```
⟨"Sam","Smith","sam@cs","501","542-8907"⟩
⟨"Sam","Smith","sam@cs","501","542-8908"⟩
```

The relation `office(F,L,O)` appears in the body of *finger*, hence we can conclude that `office("Sam","Smith","501")` is true, and we know that office `"501"` has at least two phones: `"542-8907"` and `"542-8908"`.

It is important to note that operators are not guarenteed to return *all* tuples that are conceptually part of the world model relations. This is the appropriate semantics for operators, since most data sources are incomplete. The SABRE flight database doesn't record

[2]The body can also contain numerical constraints (*i.e.*, built-in predicates (Ullman 1988 1989, vol.1, p101)), and Occam can generate appropriately constrained plans, but we do not discuss this aspect of Occam in this paper.

[3]A brief comment on notation: we follow Prolog conventions, hence symbols beginning with a capital leter denote variables. All free variables are universally quantified. All variables in the body that don't appear in the arguments are said to be *unbound* and are considered existentially quantified inside the scope of the free variables. In general, we use `typewriter font` for relations in the world model and *italics* for operators and queries. We use the function Args(O) to denote the arguments of the operator O; Body(O) denotes the operator's body, and Name(O) denotes its name.

all flights between two points on a given day, because some airlines are too small to be included. As a result of this inherent database incompleteness, one must often execute multiple operators in order to be sure that one has retrieved as many tuples as possible.[4]

As another concrete example, the operator *userid-room* generates the email addresses E for the occupants in office O.

userid-room($O,E) $\Rightarrow$ office(F,L,O)$\wedge$email(F,L,E)

Note that this operator does not return values for the first and last names associated with each email address E. Nevertheless, variables (*e.g.*, F and L) ranging over these attributes of email and office are necessary to define the query in terms of the relations in the world model; such variables are said to be *unbound*. The interpretation is as follows: if *userid-room* returns a tuple such as ⟨"501","sam@cs"⟩ then

$\exists$F, L s.t. office(F, L, "501")$\wedge$email(F, L, "sam@cs")

Our examples of *finger* and *userid-room* illustrate encodings of legacy systems. For example, UNIX *finger* may be thought of as having access to relational data about names, email addresses, phone numbers and offices, but it does not support arbitrary relational operations. If one wished to know the email address of everyone whose phone number was "555-1212" *finger* would be of little use. Binding patterns are a convenient way to describe legacy information sources, because they indicate the type of queries supported by that site. When a system supports several types of query (but doesn't support full relational operations) it can be described with several operators. Full relational databases are simply described using operators with no bound variables.

Although our syntax for operators looks very different from traditional STRIPS or ADL (Pednault 1989) planning operators, there are many similarities. In particular, while Occam operators have no causal preconditions, the bound arguments in an operator's head represent a form of knowledge precondition (Moore 1985) that is equivalent to the findout goals of UWL (Etzioni *et al.* 1992). There are no causal effects, but the body of an operator is similar to a UWL observe effect. The similarity is not exact, however, because execution of an Occam operator may generate an unbounded number of values.

[4]If one knows that a site *does* contain all tuples, then one could specify this by using $\Leftrightarrow$ to separate the operator head and body. Given such a specification, one could perform local closed world reasoning (Etzioni, Golden, & Weld 1994) to eliminate operators from consideration, but we do not discuss the matter in this paper.

Information Gathering Queries

Queries are very similar to operators: they also have heads and conjunctive bodies, but the direction of implication is reversed. The interpretation is that any tuple satisfying the body (a conjunction of world relations) satisfies the query. For example, if we want to know the first-names of the occupants in an office, we can issue the query

query-for-first-names($O,F) $\Leftarrow$ office(F,L,O)

Note that this query has two arguments, O and F; the binding pattern indicates that O must be bound (*e.g.*, to "429") before the query is executed. Since F has no $ annotatation, the query is requesting a set of values for that variable. For example, if Joe Researcher and Jane Goodhacker are the occupants of office 429, then the tuples ⟨"429","Joe"⟩ and ⟨"429","Jane"⟩ are possible answers for this query.

Plans & Solutions

If the data repository doesn't support relational operations or if the data forming the office relation is distributed across multiple sites, then satisfying queries can be complex.

For example, if one is limited to the operators described above, then the best way to satisfy the example query is to first execute *userid-room*, which returns bindings for the email addresses of the office's occupants. Next one would execute *finger* repeatedly for each binding of E and discard all information returned except for the first-name.

Formally a *plan* has the same representation as an operator whose body is an ordered conjunction of operator *instances*. For example, the previous section's example can be encoded as the two step plan, *p*:

p("429",F) $\Rightarrow$ *userid-room*("429",E)$\wedge$
finger(F,L,E,"429",Ph)

There are two ways of interpreting the body of a plan and both are important. On the one hand, the body can be viewed as a logical conjunction in which case the order is unimportant. On the other hand, the body can be viewed procedurally in which case the order is very important (but the conjunction symbols aren't); in particular, the order lets one determine if operator binding patterns are satisfied.

A plan's head specifies what information is actually returned to the user. For example, although execution of *finger* gathers information about people's last names, the plan shown doesn't return this information to the user.

We say a plan $p(X_1 \ldots X_n) \Rightarrow O_1 \wedge \ldots \wedge O_k$ is a *solution* to the query

$q(\mathtt{Y}_1 \ldots \mathtt{Y}_n) \Leftarrow \mathtt{r}_1(\ldots \mathtt{Y}_i \ldots) \wedge \ldots \wedge \mathtt{r}_m(\ldots \mathtt{Y}_j \ldots)$ if the following two criteria are satisfied:

1. The binding patterns of the plan's operator instances are satisfied. Specifically, if $V is a bound argument of O_b then V must be used as a free argument to some other operator instance O_a where $a < b$ or else a value for V must be a bound argument in the query head (Rajaraman, Sagiv, & Ullman 1995).

2. All tuples satisfying $p(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ must satisfy $query(\mathtt{X}_1, \ldots, \mathtt{X}_n)$. In other words, the following implication must hold:

$$\forall \mathtt{c}_1, \ldots, \mathtt{c}_n \;\; p(\mathtt{c}_1, \ldots, \mathtt{c}_n) \Rightarrow q(\mathtt{c}_1, \ldots, \mathtt{c}_n)$$

where each c_i is a constant.

For example, the example plan, p, shown previously is a solution to *query-for-first-names* because

1. The binding patterns are satisfied: *userid-room*'s has bound variable $0 bound to "429" by the query, and execution of *userid-room* binds E, satisfying *finger*.

2. Every tuple returned by the plan satisfies *query-for-first-names*("429", F). To see that this is true, suppose the tuple of constants $\langle \mathtt{c}_1, \mathtt{c}_2 \rangle$ is returned by the plan.

$$\begin{aligned} p(\mathtt{c}_1, \mathtt{c}_2) &\Rightarrow \textit{userid-room}(\mathtt{c}_1, \mathtt{E}) \wedge \textit{finger}(\mathtt{c}_2, \mathtt{L}, \mathtt{E}, \mathtt{c}_1, \mathtt{Ph}) \\ &\Rightarrow \mathtt{office}(\mathtt{F}_0, \mathtt{L}_0, \mathtt{c}_1) \wedge \mathtt{email}(\mathtt{F}_0, \mathtt{L}_0, \mathtt{E}) \wedge \\ &\quad \mathtt{email}(\mathtt{c}_1, \mathtt{L}, \mathtt{E}) \wedge \mathtt{office}(\mathtt{c}_2, \mathtt{L}, \mathtt{c}_1) \wedge \\ &\quad \mathtt{phone}(\mathtt{0}, \mathtt{Ph}) \\ &\Rightarrow \mathtt{office}(\mathtt{c}_2, \mathtt{L}, \mathtt{c}_1) \\ &\Rightarrow \textit{query-for-first-names}(\mathtt{c}_1, \mathtt{c}_2) \end{aligned}$$

Planning to Gather Information

Figure 1 presents the Occam forward-chaining planner. As input, Occam takes a query and set of operators. As output, Occam produces a set of plans, each of which is guaranteed to be a solution. Starting from the empy sequence, Occam searches the space of totally ordered sequences of operator instances (*i.e.* plan *bodies*). Since there is no bound on the length of useful plans (Kwok & Weld 1996), Occam's search proceeds until all alternatives have been exhausted, or a resource bound is exceeded. At each stage a sequence of operator instances, *Seq*, is removed from *Fringe* and is expanded by postpending an instance of each potential operator. Since operators can be instantiated in several ways (Figure 2), expanding *Seq* will typically cause many new sequences to be added to *Fringe*. FindSolutions determines if any of these sequences can

```
Procedure Occam(Q, O)
  Fringe = {⟨⟩}
  Sol = {}
  Loop until either { Fringe = {}
                      Resource bound reached
    Choose and remove Seq from Fringe
    B ← the set of all variables in Seq ∪
          the values of bound vars in Q
    For each Op ∈ O
      For each Op_i ∈ InstantiateOp(Op, B)
        Seq_i ← Append(Seq, Op_i)
        Fringe ← Fringe ∪ {Seq_i}
        Sol ← Sol ∪ FindSolutions(Seq_i, Q)
  Return Sol
```

Figure 1: A forward-chaining algorithm for generating query plans; input Q is a query, and $\mathcal{O}$ is a set of operators; the output is a set of solutions.

be elaborated into a solution plan; this is akin to evaluating the modal truth criterion (Chapman 1987) as explained below; Occam adds all newly discovered solutions to *Sol*, but in any case *every* sequence is kept on *Fringe* because its children might lead to qualitatively different solutions.

The Example, Revisited

Suppose Occam is called on the *query-for-first-names* example. When the empty sequence is removed from *Fringe*, Occam considers adding instances of operators *finger* and *userid-room*. Since there are no instances in the empty sequence, $\mathcal{B}$ is assigned the value {"429"} because that is the only constant provided as input by the query.

When InstantiateOp is called with *userid-room*, the procedure must create *Val* sets corresponding to *userid-room*'s two arguments, the bound $0 and the free E. Potentially, $0 could be assigned any value (there is only one) in $\mathcal{B}$ that has type which is consistent with offices. Since both "429" and $0 are of type office, $Val(\$0) = \{\text{"429"}\}$; if there had been a type conflict, then $Val(\$0)$ would have been empty and InstantiateOp would have returned no instances. Since E is free, $Val(\mathtt{E})$ is assigned a set containing a newly generated variable, $\{\mathtt{E}_0\}$.

Since both *Val* sets are singletons, there is only one pair in the cross product. Hence, InstantiateOp returns a single instance to Occam: *userid-room*("429", $\mathtt{E}_0$).

In some later iteration of Occam, *Seq* = *userid-room*("429", $\mathtt{E}_0$) will be removed from *Fringe*. $\mathcal{B}$ will now be assigned the value, {"429", $\mathtt{E}_0$}. This is Occam's way of noting that after executing *userid-*

```
Procedure InstantiateOp(Op, B)
  Instances ← {}
  For each variable, V_i, in Args(Op)
    If V_i is bound
    Then Val(V_i) ← {X ∈ B | SameType(X, V_i)}
    Else if V_i is free
      Then Val(V_i) ← {a newly generated var}
  For each tuple ⟨X_1, ..., X_n⟩ in the cross
        product Val(V_1) × ... × Val(V_n)
    Generate a new op instance Op_i such that
      Name(Op_i) ← Name(Op)
      Args(Op_i) ← ⟨X_1, ..., X_n⟩
    Instances ← Instances ∪ {Op_i}
  Return Instances
```

Figure 2: Instantiating an operator; input Op is an operator and $\mathcal{B}$ is the set of bound variables, output *Instances* is a set of operator instances.

room, Occam will have a set of possible values for $\mathtt{E_0}$ and thus can use that variable when instantiating future instances that have bound arguments.

Once again Occam will consider adding instances of *finger* and *userid-room*. When it chooses the former, it must create *Val* sets for *finger*'s arguments: $\mathtt{F}, \mathtt{L}, \mathtt{E}, \mathtt{O}, \mathtt{Ph}$. Since all of these arguments except $\mathtt{E}$ are free, their *Val* sets will contain a single newly generated variable each, *e.g.* $\{\mathtt{F_1}\}, \{\mathtt{L_1}\}, \{\mathtt{O_1}\}, \{\mathtt{Ph_1}\}$. Although there are two members of $\mathcal{B}$, only one has type email address, so $Val(\mathtt{E}) = \{\mathtt{E_0}\}$. Therefore InstantiateOp returns a single instance to Occam: $finger(\mathtt{F_1}, \mathtt{L_1}, \mathtt{E_0}, \mathtt{O_1}, \mathtt{Ph_1})$, and we have a sequence

$$userid\text{-}room(\mathtt{"429"}, \mathtt{E_0}) \wedge finger(\mathtt{F_1}, \mathtt{L_1}, \mathtt{E_0}, \mathtt{O_1}, \mathtt{Ph_1})$$

This sequence is added to *Fringe*, and in addition it is passed to FindSolutions, in order to see if it could be the basis for a solution to the query.

Finding Solutions from Sequences

The previous section described how Occam enumerates the space of totally ordered sequences of operator instances. This section explains how the FindSolutions function tests each sequence to see if it encodes one or more solutions to the query. Note, first, that there is a difference between a *plan* and a *sequence* of operator instances. A plan is represented as an operator, and as such it has both a head and a body; the body determines which actions get executed while the head determines what data gets returned.

When given a sequence, $O_1 \wedge \ldots \wedge O_k$, of operator instances, FindSolutions determines whether there exist any plans of the form $p(\mathtt{X_1} \ldots \mathtt{X_n}) \Rightarrow O_1 \wedge \ldots \wedge O_k$ that are solutions to the query. This test is somewhat akin to the Modal Truth Criterion (Chapman 1987) which tests a partially ordered (hence incomplely specified) plan to see if any solution exists. In the case of Occam, a totally ordered sequence of operators is underspecified because there could be several (or no) heads which render it a solution.

Recall that there are the two requirements for a plan to be a solution to a query. First, the binding patterns of the plan body's operator instances must be satisfied. FindSolutions doesn't need to check this criterion because InstantiateOp is careful in its choice of *Val* sets so that every bound variable is only instantiated with acceptable values. The second condition was that all tuples satisfying $plan(\mathtt{X_1}, \ldots, \mathtt{X_n})$ must satisfy $query(\mathtt{X_1}, \ldots, \mathtt{X_n})$. As shown in Figure 3, the FindSolutions function takes a sequence and generates the set of all plans (having the sequence for their body) whose tuples are guaranteed to satisfy the query. These plans are thus solutions.

```
Procedure FindSolutions(Seq,Q)
Sol ← {}
E ←  ⋀_{Op_i ∈ Seq}  Body(Op_i)
V_E ← the set of all symbols in E
V_Q ← the set of all symbols in Q
For each potential containment map τ : V_Q ↦ V_E
  For each equality mapping ξ : V_E ↦ V_E
    If τ(Body(Q)) ⊆ ξ(E)
    Then P ←  a plan with head p(τ(Args(Q)))
              and body ξ(Seq)
    If P is not redundant then Sol ← Sol ∪ {P}
Return Sol
```

Figure 3: Finding solutions; the input is a query and a sequence of operator instances; the output is a set of plans (the solutions).

Underlying the operation of FindSolutions is the notion of a *containment mapping* between two horn clauses (Ullman 1988 1989, vol.2, p881). A containment mapping from query Q to the formula E is a function τ mapping symbols in Q to symbols in E. If there exists a mapping such that the $\tau(\mathsf{Body}(Q))$ is a subset of the body of E while $\tau(\mathsf{Args}(Q))$ equals the arguments of E, then E logically entails Q.

When FindSolutions is given a sequence of operator instances, *i.e.* a potential *plan body*, it first computes the *expansion* of the sequence by setting E to the conjunction of the *bodies of the operators in the sequence*. $\mathcal{V}_E$ and $\mathcal{V}_Q$ are defined so that FindSolutions can enu-

merate the space of potential containment mappings.[5] If it can find a containment mapping from the query to the expansion E, then this enables the construction of a plan head guaranteeing that all tuples returned by the plan will satisfy the query. FindSolutions also considers possible *equality mappings* which have the effect of requiring that two or more variables in E are constrained to be equal.

The Example, Concluded

Previously we illustrated how Occam generates the promising sequence *userid-room*("429", E_0) $\wedge$ *finger*(F_1, L_1, E_0, O_1, Ph_1). At this point, it will ask FindSolutions to see if any plans could be made (with this conjunction for their body) in order to solve *query-for-first-names*. Given these arguments, FindSolutions expands the sequence, giving E the following value:

$$\texttt{office}(F_0, L_0, \texttt{"429"}) \wedge \texttt{email}(F_0, L_0, E_0) \wedge \texttt{email}(F_1, L_1, E_0) \wedge \texttt{office}(F_1, L_1, O_1) \wedge \texttt{phone}(O_1, Ph_1)$$

$\mathcal{V}_Q$ becomes {"429", F, L} and $\mathcal{V}_E$ becomes $\{F_0, L_0, \texttt{"429"}, E_0, F_1, L_1, O_1, Ph_1\}$. Next, FindSolutions tries different ways to map variables from $\mathcal{V}_Q$ to $\mathcal{V}_E$. Eventually, it considers the following mapping: $\tau(\texttt{"429"}) = O_1$, $\tau(F) = F_1$, and $\tau(L) = L_1$. Suppose ξ is the identity mapping. Applying τ to the query body yields the singleton sequence $\texttt{office}(F_1, L_1, O_1)$, which matches one of the conjuncts in E. Therefore we make a new plan, p:

$$p(O_1, F_1) \Rightarrow \textit{userid-room}(\texttt{"429"}, E_0) \wedge \textit{finger}(F_1, L_1, E_0, O_1, Ph_1)$$

Since p is not redundant (discussed below) it is saved as a solution in *Sol*. In this example, there are no other solution plans with *userid-room*$\wedge$*finger* as body, but in some cases there exist several heads that make a sequence into a solution. When this happens, FindSolutions returns all such plans.

Transformations Based on = Mappings

FindSolutions's inner loop enumerates the space of equality mappings, functions of the form $\xi : \mathcal{V}_E \mapsto \mathcal{V}_E$. By performing this search, FindSolutions considers the possibility of constraining one or more of the variables in the expansion to be equal. Although equality mappings weren't important in the previous example, sometimes they are necessary in order to recognize a solution; see (Kwok & Weld 1996) for an example and further discussion.

[5] In practice, the use of type information and other optimizations allows a much more efficient algorithm than this brute-force enumeration of containment and equality mappings. See the long version of the paper.

Redundant Solutions

We call a solution *redundant* if we can eliminate operator instances from the plan and still obtain a solution. The last line of FindSolutions checks to see if a plan is redundant before adding it to the set of solutions to be returned. To see why this check is essential, note that if a sequence of operator instances corresponds to a solution then *every supersequence* will also generate that solution. Furthermore, recall that Occam keeps all sequences on the *Fringe*, even when they have produced solutions. Thus it is crucial to discard redundant solutions. Space considerations preclude details, but in (Kwok & Weld 1996) we discuss why Occam keeps solution sequences on the *Fringe* and explain how to filter redundant solutions in time which is polynomial in the length of a plan.

Reducing Search

We have implemented several domain-independent, completeness-preserving optimizations. First, the use of type information and constraint satisfaction speeds FindSolutions. Second, *duplicated operator instance pruning* reduces the branching factor by forcing InstantiateOp to check if instances are subsumed by the plan body being extended (Kwok & Weld 1996). Third, *shuffled sequence pruning* achieves the efficiency benefits of a partial-order representation (Barrett & Weld 1994; Minton *et al.* 1992) without the attendant complexity. Note that since Occam generates sequences that are totally ordered, it frequently considers different permutations when the precise order does not matter at all. We avoid the problem by imposing a canonical ordering. We say operator instance O_i is *dependent* on O_j if either 1) O_i has a bound argument that appears as a free variable in O_j, or 2) there exists an instance O_k such that O_i is *dependent* on O_k and O_k is *dependent* on O_j. If two operator instances are *independent* (*i.e.*, neither is dependent on the other), then Occam does not need to consider both ordering permutations. To avoid this redundancy, we assign an (arbitrary) unique number InstanceID(O) to each operator instance O. When creating new sequences by adding operator instances to an existing sequence, Occam prunes the creation if the new instance O is independent of an existing operator instance O_i and InstanceID(O) < InstanceID(O_i).

We demonstrate the performance improvments of these optimizations with 5 problems taken from 4 domains. Although the first three domains are relatively simple (Kwok & Weld 1996), the last two problems are taken from a relatively detailed (*e.g.*, 25 operator) encoding of UNIX commands and Internet information services. In each experiment Occam exhaus-

Query		plain	opt
find-grandparent	explored	46232	413
(depth 7)	time	23	< 1
Patho q(X)	explored	598443	10024
(depth 7)	time	364	2
Car query	explored	97655	2310
(depth 7)	time	975	8
query-for-first-names	explored	62808	8480
(depth 6)	time	346	9
find-email	explored	14249	5257
(depth 4)	time	19	2

Table 1: Performance of Occam; *time* is in CPU seconds (on a Silicon Graphics Indy under Allegro Common Lisp 4.2), *explored* refers to number of sequences visited in the search space, and *depth* refers to the maximum length of sequences considered. In all problems Occam found many different solutions.

tively explores all sequences up to a certain length; the number of sequences explored and the time taken for each experiment is shown in Table 1. Each experiment is run with vanilla Occam (the "plain" column) and also with duplicated operator instance and shuffled sequence pruning engaged (the "opt" column). This preliminary experiment shows that our search control optimizations provide two orders of magnitude speedup.

Related Work

Several researchers in database community are concerned with the integration of heterogenous databases. Prominent projects include the Information Manifold (Levy, Srivastava, & Kirk 1995) and the Tsimmis project (Chawathe *et al.* 1994). From Tsimmis, we adopt the notion of notion of binding templates (Rajaraman, Sagiv, & Ullman 1995). However, for the most part, Tsimmis assumes information integration is done manually, while our work focuses on automating the information-integration process.

The rest of our representation language is based on the encodings described in (Levy, Srivastava, & Kirk 1995), but in contrast to this work we provide implemented algorithms for generating query plans when site descriptions include binding annotations. In addition, we describe several optimizations and demonstrate their effectiveness experimentally. On the other hand, the description language in (Levy, Srivastava, & Kirk 1995) provides a more expressive type hierarchy than that used by Occam.

Several planning systems were designed specifically for information gathering. For example, the XII planner (Golden, Etzioni, & Weld 1994) guides the Internet Softbot, and the Sage planner (Knoblock 1995) controls the SIMS information system (Arens *et al.* 1993). Like Occam, both XII and Sage specify transformations between the information produced by a remote site and an internal world model. But Occam allows a more general class of transformations in several ways, such as representing information sources that generate information which translates into partially specified sentences in the world model. Furthermore, Sage and XII are unable to represent an incomplete source that returns variable number of tuples. However, both Sage and XII can handle goals with negation and disjunction. XII can also perform sophisticated local closed world reasoning. Finally, Sage allows parallel execution in its control of multiple database operations.

Both Sage and XII interleave planning and execution, but another approach is the generation of contingent plans. Most of the planners described above have significant combinatorial explosions and require domain-specific, search control for anything but small problems. For example, XII requires considerable control knowledge in order to handle problems that appear comparable to those in our People domain. A major contribution of our work is the development of a domain-independent, sound and complete algorithm that runs at practical speeds.

Conclusion

We have described a novel planning algorithm, Occam, that is optimized for the problem of gathering and integrating Internet information sources. Since most sites on the Internet do not allow updates, our action language does not support the notion of causal change. Because most information sources are easily accessible, our language does not support traditional preconditions. These restrictions allow a much simpler planning algorithm.

Paring down the action language in some respects allowed us the opportunity to increase it's expressiveness in other ways. While Occam actions don't have traditional preconditions, they may have *knowledge preconditions.* By reasoning about the capabilities of different information sources, Occam can extract data from both legacy systems and full relational databases. Although Occam need not represent changes to the world state, it does reason about changes to the *state of information* during the course of the plan. Unlike previous implemented systems, Occam can reason about the fact that information sources may contain an unbounded amount of information, without assuming that the source contains all possible information. Because of this, Occam handles partial goal satisfaction: when no single plan can gather all information, Occam

generates alternatives that may be executed in parallel to collect as much information as possible. (Kwok & Weld 1996) argues that Occam is both sound and complete. In addition, Occam is efficient as we demonstrated in preliminary empirical tests.

In addition to the planner described here, we have implemented an interesting execution system for Occam; see (Friedman & Weld 1996) for details. Our preliminary experience is that the time spent planning is negligible compare to the time required for execution. Inductive learning techniques can acquire estimates of the speed and extent of each information source; the execution system will use a user-specified utility function to balance the expected time to execute a plan against the number of tuples it is expected to return. Plan quality is the focus of our continuing efforts. In the future we hope to incorporate local closed world information (Etzioni, Golden, & Weld 1994) into our planner so that Occam can reason about situations when it has exhausted all information gathering alternatives.

Acknowledgements

We thank Paul Beame, Oren Etzioni, Marc Friedman, Keith Golden, Steve Hanks, Nick Kushmerick, Alon Levy, and Mike Williamson for helpful discussions. This research was funded in part by Office of Naval Research Grant N00014-94-1-0060, by National Science Foundation Grant IRI-9303461, by ARPA / Rome Labs grant F30602-95-1-0024, and by a gift from Rockwell International Palo Alto Research.

References

Arens, Y., Chee, C. Y., Hsu, C.-N., and Knoblock, C. A. 1993. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems* 2(2):127–158.

Barrett, A., and Weld, D. 1994. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence* 67(1):71–112.

Chapman, D. 1987. Planning for conjunctive goals. *Artificial Intelligence* 32(3):333–377.

Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. 1994. The tsimmis project: Integration of heterogeneous information sources. In *Proceedings of IPSJ Conference.*

Etzioni, O., and Weld, D. 1994. A Softbot-Based Interface to the Internet. *CACM* 37(7):72–76.

Etzioni, O., Hanks, S., Weld, D., Draper, D., Lesh, N., and Williamson, M. 1992. An Approach to Planning with Incomplete Information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning.* San Francisco, CA: Morgan Kaufmann.

Etzioni, O., Golden, K., and Weld, D. 1994. Tractable closed-world reasoning with updates. In *Proc. 4th Int. Conf. on Principles of Knowledge Representation and Reasoning*, 178–189. San Francisco, CA: Morgan Kaufmann.

Friedman, M., and Weld, D. 1996. Decision-theoretic execution of information gathering plans. Technical report, University of Washington, Department of Computer Science and Engineering.

Golden, K., Etzioni, O., and Weld, D. 1994. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. on A.I.*, 1048–1054. Menlo Park, CA: AAAI Press.

Knoblock, C. 1995. Planning, executing, sensing, and replanning for information gathering. In *Proc. 15th Int. Joint Conf. on A.I.*, 1686–1693.

Kwok, C., and Weld, D. 1996. Planning to gather information. Technical Report 96-01-04, University of Washington, Department of Computer Science and Engineering.

Levy, A. Y., Srivastava, D., and Kirk, T. 1995. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems, Special Issue on Networked Information Discovery and Retrieval* 5 (2).

Minton, S., Drummond, M., Bresina, J., and Phillips, A. 1992. Total order vs. partial order planning: Factors influencing performance. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning.*

Moore, R. 1985. A Formal Theory of Knowledge and Action. In Hobbs, J., and Moore, R., eds., *Formal Theories of the Commonsense World.* Norwood, NJ: Ablex.

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, 324–332.

Rajaraman, A., Sagiv, Y., and Ullman, J. 1995. Answering queries using templates with binding patterns. In *Proceedings of the ACM Symposium on Principles of Database Systems.*

Ullman, J. 1988, 1989. Database and knowledge-base systems. In *Database and knowledge-base systems*, volume 1 & 2. Computer Science Press.