In Proceedings, IJCAI-97.

Efficiently Executing Information-Gathering Plans

Marc Friedman **Daniel S. Weld**
Department of Computer Science and Engineering
University of Washington, Box 352350 Seattle WA 98195–2350 USA
{friedman, weld}@cs.washington.edu

Abstract

We describe Razor, a planning-based information-gathering agent that assists users by automatically determining which Internet information sites are relevant to their query, accessing those sites in parallel, and integrating the results. Razor uses a disjunctive graph-based plan representation. It then uses a novel and powerful form of local completeness reasoning in order to transform those plans into contingent plans of high quality. These contingent plans can be efficiently executed, obtaining more answers at less cost than the original plans. We focus in this paper on the algorithms underlying the plan transformation process.

1 Introduction

Thanks to the Internet, thousands of structured information sources are available for querying, and the number and variety of these sites is growing rapidly. While a wide range of questions can be answered via the Internet, the morass of sources means that users cannot easily get the information they need. Humans face three problems when trying to gather information. First, they must determine which of the myriad sites has information relevant to their question. Second, they must learn to navigate the sites' idiosyncratic interfaces. Third, for many queries they must integrate the data returned by several different sites.

By automating this process, a software agent can greatly simplify the task of gathering information. For example, a user could ask for reviews of all movies starring Marlon Brando playing in Boston. To gather the desired information, the agent must reason about the contents and capabilities of different information sources. In this case, no single information source can answer the query, and there are several choices of how to do it. The agent might first go to the Internet Movie Database to get a list of movies starring Marlon Brando, then go to MovieLink to see which of these movies is showing in Boston, finally to Ebert to get reviews of each of the relevant movies. Because most information sources are incomplete, it is often necessary to execute more than one such plan. For example, since Ebert contains only a fraction of the movie reviews on the web, the agent can return more information by also going to Cinemachine.

1.1 Execution Policies

Often we find two sites with overlapping information content. In the simplest case, this occurs when mirror sites are created for a popular (and hence heavily loaded) resource. For example, the Internet Movie Database has three mirror sites (US, UK, and Italy). This overlap creates a dilemma for agent designers — on the one hand we wish to tell the agent about the existence of all three sites (what if the US and UK sites are down?), but on the other hand we don't wish the agent to visit all sites in parallel since this wastes resources and undermines the very purpose of creating mirror sites.

Furthermore, mirror sites are just the simplest case of overlap. Some sites aggregate information that may be found from other sources. For example, the SABRE airline system provides information about all United flights, but doesn't report on the flights of Southwest Airlines. We say that SABRE *subsumes* United (defined in Section 3), but not Southwest. Given that some sequence of actions A subsumes some other sequence B, there are at least three execution policies:

- *Brute force* – Just ignore subsumption, and execute everything greedily.
- *Aggressive* – Execute both alternatives in parallel, but cancels all communication with B once A has successfully returned.
- *Frugal* – Initially, run only A. When A fails (perhaps because SABRE is overloaded), begin B. If subsumption is reflexive (as is the case with mirror sites) then the source with fastest expected response is executed at first.

While an aggressive policy is more thrifty with valuable resources than a brute force one, ultimately only something like a frugal policy will scale. There are good reasons to use one of these resource-conscious strategies rather than a brute force one. Unnecessary accesses to the internet waste memory for network connections and buffering. They waste time copying data, parsing text, and doing database operations. In the future, information sources may charge a micropayment for each access as well. If enough rapacious information agents populate the internet, then servers and the internet itself will degrade. Worst of all, on a slow network (*e.g.* modem or wireless link), excess accesses may flood the connection and slow the agent.

1.2 Contributions

This paper describes Razor, an information-gathering agent that automatically plans how to gather information that answers a query, reasons about the value of accessing the different sources, and then executes the actions in a manner that reduces wasteful accesses to overlapping sites. Specifically, we make the following contributions:

- We define an expressive form of *local completeness* statements (*e.g.*, conditions under which an information source will return *all* relevant information, or strictly more than another source) and provide algorithms for reasoning with them. Our methods improve on previous formulations [Etzioni *et al.*, 1994; Levy, 1996] by allowing a richer set of constraints scoping these statements.
- We present an algorithm that reasons about local completeness to determine when one portion of an information-gathering plan subsumes another, in the sense of returning at least as much information. We compare portions of the graph that return sets of intermediate values, and look at how those sets overlap.
- We describe a labeling algorithm that uses subplan subsumption to generate a contingent plan which implements the aggressive (or frugal) policy.

In Section 2 we review the basic information-gathering problem. In Section 3 we look at what happens when we introduce local completeness. Section 4 describes how to use local completeness to determine subsumption relationships within a plan. Section 5 explains how to use subsumption to implement a policy, such as aggressive or frugal, in a contingent plan. The paper concludes with a discussion of Razor's implementation status, related work, and conclusions

2 Background: Sites, Queries, & Plans

Razor represents knowledge about the Internet using a logical theory consisting of two types of relations: world relations (written in `courier font`), which are a part of the user's *world ontology*, and site relations (written in SMALL CAPS). World relations express relationships of interest to a user: site relations merely label the collections of data accessible on the internet.

For example, the world ontology might represent information about movies with four relations: `actor-in`$(Movie, Role, Actor)$, `review-of`$(Movie, Review)$, `year-of`$(Movie, Year)$, and `shows-in`$(Movie, City, Theater)$. When posing queries, the user need only mention these world relations; Razor shields the user from the details of the available information sources by reasoning about site descriptions.

2.1 Site Descriptions

We model an information source with a *site description*. The site description defines a new site predicate in terms of a conjunction of relations in the world ontology. For example, given the world ontology shown above, the source description in Equation 1 says that the Internet Movie Database takes one input (the name of a movie) and returns tuples of the form $\langle M,A \rangle$ such that the actor A plays some role $Role$ in movie M. The inputs are returned implicitly.[1]

$$\textsc{ImdbCast}(\$M, A) \Rightarrow \texttt{actor-in}(M, Role, A) \quad (1)$$

The description's *head* (*i.e.*, its left hand side) shows the type signature of the tuples returned, while its *body* (RHS) lists the world relations satisfied by the tuple elements. Site descriptions can encode the fact that an information source supports only a subset of full relational algebra. Specifically, by annotating head arguments with \$, we specify that the site *requires* values for those variables as input [Rajaraman *et al.*, 1995]. Thus a call to ImdbCast can only return the data in the table for specified movie M.

Formally, site descriptions have a simple semantics; each one is syntactic sugar for a first-order sentence forming part of the agent's logical theory Δ. For example, the ImdbCast definition translates into:

$$\forall\ \mathrm{M, A}\ \ \textsc{ImdbCast}(M, A) \Rightarrow \exists\ \mathrm{Role}\ \mid \texttt{actor-in}(M, Role, A) \quad (2)$$

This makes it clear that ImdbCast doesn't report *what* role the actor plays, just that *there exists* a role. Furthmore, the implication sign $\Rightarrow$ in Equation 2, rather than $\Leftrightarrow$, indicates that most sites contain *correct* but possibly *incomplete* information. This means that querying ImdbCast with M bound to the movie `The Godfather` returns some subset of the tuples that satisfy the world relation `actor-in(The Godfather,`$Role, A$`)`; the site is *not* guaranteed to return all such actors.

Figure 1 defines the sites used throughout the paper.

2.2 Queries

A user queries Razor with a conjunctive, function-free (*i.e.* datalog) query. For instance, we can encode the information-gathering problem from this paper's introduction as:[2]

$$\texttt{query1}(M, Review) \Leftarrow \texttt{actor-in}(M, Role, \texttt{Brando}) \wedge \texttt{shows-in}(M, \texttt{Boston}, Thtr) \wedge \texttt{review-of}(M, Review).$$

The query head indicates what information is desired (tuples of $\langle M,Review \rangle$). The query body is a conjunction of equality constraints that define the inputs to the query (the start state), and world relations to be satisfied by all the tuples returned (the goal). In planning terms, the inputs map to the start state and the world relations to be satisfied are the goal. The agent satisfies the query by returning as many tuples of $\langle M,Review \rangle$ as it can find that satisfy this start state and goal.

The query can be viewed as syntactic sugar for a logical sentence defining a new relation `query1`:

[1] Each site also has a corresponding wrapper. The *wrapper* defines interface code to connect to the information source, feed it some inputs, and parse the response into tuples. The wrapper allows Razor to treat each source as a database table.

[2] The symbol $\Leftarrow$ can be read as 'if'.

$\textsc{ImdbActor}(\$Actor, Movie)$ $\Rightarrow$ `actor-in`$(Movie, Role, Actor)$
$\textsc{Ebert}(\$Movie, Review)$ $\Rightarrow$ `review-of`$(Movie, Review)$
$\textsc{Cinemachine}(\$Movie, Review)$ $\Rightarrow$ `review-of`$(Movie, Review)$
$\textsc{MovieLink}(\$City, Movie, Theater)$ $\Rightarrow$ `shows-in`$(Movie, City, Theater)$
$\textsc{MetroCinema}(\$City, Movie, Theater)$ $\Rightarrow$ `shows-in`$(Movie, City, Theater)$
$\textsc{ImdbCast}(\$Movie, Actor)$ $\Rightarrow$ `actor-in`$(Movie, Role, Actor)$
$\textsc{EbertActor}(\$Actor, Movie, Year)$ $\Rightarrow$ `actor-in`$(Movie, Role, Actor) \wedge$ `year-of`$(Movie, Year)$

Figure 1: Site descriptions of a few movie sites; dollar signs denote arguments that must be bound as input.

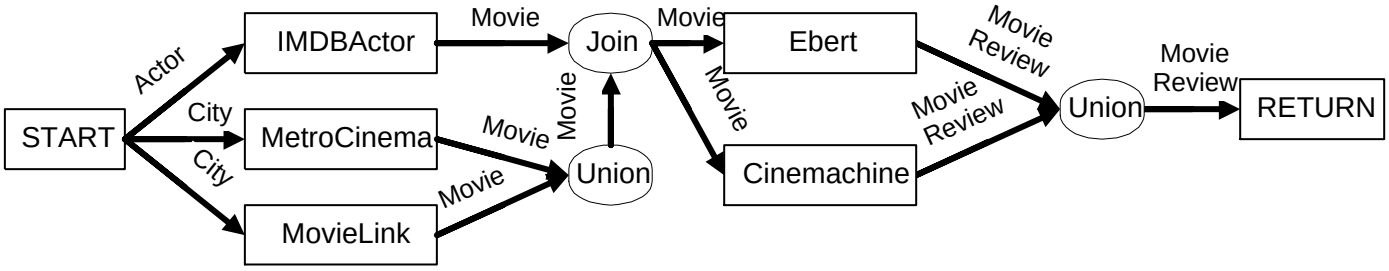


Figure 2: A solution plan to `query1` using the first five sites.

$$\forall M, Review \quad \texttt{query1}(M, Review) \Leftrightarrow \quad (3)$$
$$\exists\, Thtr, Role \mid \texttt{actor-in}(M, Role, \texttt{Brando}) \wedge \texttt{shows-in}(M, \texttt{Boston}, Thtr) \wedge \texttt{review-of}(M, Review)$$

2.3 Plans

A plan is a directed graph of information-gathering actions representing data flow dependencies. Specifically, a plan has three types of nodes: information-gathering operators (i.e. instances of site descriptions), database operators (Join, Select, and Union), or the special nodes Start and Return. The Start node provides the input assignments, while Return node collects the answers. Edges of the plan graph are labelled with the attributes whose values propagate; tuples of those attributes flow along each directed edge, and each edge potentially performs a database projection operation on its inputs. Figure 2 shows a plan for `query1`.

Plans have both operational and declarative semantics. In operation, the plan is a data flow graph of queries to network information sources. Multiple tuples of data flow along edges. Whatever tuples flow into the Return node are collected and returned to the user as answers. In this example, MetroCinema and MovieLink perform the same function, returning movies playing in a city. Since any movie returned by either is correct, we take the union. Because the same movie M needs to play in the city *and* star the actor, we join the results of the Union and ImdbActor, equating the movie columns. We then use the resulting values of M as inputs to Ebert and Cinemachine. Declaratively, this graph is equivalent to the following datalog program defining predicate `plan1`:

$$\texttt{plan1}(M, Review) \Leftarrow \textsc{ImdbActor}(\texttt{Brando}, M) \wedge \texttt{tempm}(\texttt{Boston}, M) \wedge \texttt{tempr}(M, Review).$$

$$\texttt{tempm}(City, M) \Leftarrow \textsc{MovieLink}(City, M, Thtr).$$
$$\texttt{tempm}(City, M) \Leftarrow \textsc{MetroCinema}(City, M, Thtr).$$
$$\texttt{tempr}(M, Review) \Leftarrow \textsc{Ebert}(M, Review).$$
$$\texttt{tempr}(M, Review) \Leftarrow \textsc{Cinemachine}(M, Review).$$

A datalog program captures the graph's logical content, so we will freely convert between the two forms. Note that cycles are indeed possible in the plan; this corresponds to recursion in the datalog form. A *solution plan* is just one that is executable, and semantically correct. A plan is executable if the input variables required by each step are connected to variables of the correct type along each in-edge, and those variables are available from the predecessor node. A graph is semantically correct if it returns only answers to the query (see also [Kwok and Weld, 1996; Levy, 1996]).

There are several ways to construct plans. Our first implementation used a partial-order version of the Occam planner [Kwok and Weld, 1996] to produce a (complete yet potentially infinite) stream of conjunctive plans; this stream was truncated and merged into a graph. The Razor system currently under development uses the recursive plan compilation procedure described in [Duschka and Levy, 1997] augmented with additional type inference and modified to produce acyclic graphs.

3 Representing Local Completeness

A single plan may contain many alternative ways to get an answer. Since most information sources are incomplete, Razor has to execute *all* of these alternatives to guarantee the maximal set of answers. However, in some cases a source may be complete, and by reasoning about this completeness, Razor can avoid redundant work. For instance, the SABRE online flight reservation system lists all United and American Airlines flights. If the user wanted a United flight and had already contacted SABRE, then there is no reason to go to

United's web page. But if the user were flexible with respect to airlines, she might try Southwest as well.

We represent these relationships *between* sites with *local completeness* axioms. This section describes our local completeness representation, and explains how they simplify and generalize those of [Etzioni *et al.*, 1994; 1997] and [Levy, 1996].

Intuitively, a *local completeness* (LC) *rule* states the conditions under which a site contains exhaustive information, *i.e.* when the site contains *all* information in the corresponding part of the world ontology. At the syntactic level, an LC rule takes the form $head \Leftarrow body$ where the head specifies a site relation and the body is a conjunctive constraint of world relations, site relations, and built-in relations like inequality. Moveover, the head is an information site, and the body must be subsumed by the body of the site description, using only the theory Δ of site descriptions (more on subsumption in section 4). To make this concrete, we present some examples of local completeness statements, in order of increasing complexity.

Suppose MovieLink lists all the movies playing anywhere, and their theaters. We encode this with LC as follows:

$$\text{MovieLink}(City, Movie, Theater) \Leftarrow \texttt{shows-in}(Movie, City, Theater). \quad (4)$$

This is the simplest kind of LC statement, equivalent to the local closed world declarations of [Etzioni *et al.*, 1994] — a site is complete over the entire relation in the body of its site description. Assuming the site is available, one would never need to go anywhere else but to MovieLink to find movies playing in some locale.

As a second example, suppose that it takes some time for a movie's statistics to be entered in ImdbCast, but we know it is complete up until last year. We write this as:

$$\text{ImdbCast}(M, A) \Leftarrow \texttt{actor-in}(M, R, A) \wedge \texttt{year-of}(M, Y) \wedge (Y \leq 1996). \quad (5)$$

ImdbCast's completeness is local, because it is limited to the subspace constrained by $\texttt{year-of}(M, Y) \wedge (Y \leq 1996)$. Note that this case can be expressed in the formulation of [Levy, 1996], but is not solvable by Levy's polynomial time algorithm because there is a world relation (`year-of`) in the constraint.

As a third example, note that the content of some information sources is highly *correlated*. [Levy, 1996] does not allow the expression of relationships between two sites, but we encode them as follows:

$$\text{MovieLink}(City, Movie, Theater) \Leftarrow \text{MetroCinema}(City, Movie, Theater). \quad (6)$$

Here we said that MovieLink lists all of MetroCinema's reviews. Correlations are especially common because Razor treats each kind of information request as a separate information source, yet some requests rely on the same underlying database. For a non-movie example, note that the SABRE airline reservation system contains all flights shown on the American Airlines web page, but SABRE does not contain *all* flights, so an agent should consider Southwest and other information sources.

A more interesting case occurs when the sites include different types of data. For instance, whenever Ebert reviews a movie, he also lists the movie's cast:

$$\text{EbertActor}(A, M, Y) \Leftarrow \text{Ebert}(M, Review) \wedge \texttt{actor-in}(M, Role, A) \wedge \texttt{year-of}(M, Y). \quad (7)$$

In our final example, ImdbCast has local completeness over a part of itself. We are confident that if a movie title is listed there, then all of its cast is listed, since casts of movies do not change. However, we cannot say whether a movie will be listed there at all, other than by actually checking. It is easy to encode this dependency:

$$\text{ImdbCast}(M, A) \Leftarrow \texttt{actor-in}(M, R, A) \wedge \text{ImdbCast}(M, A_2). \quad (8)$$

We add to our theory Δ all the local completeness axioms.

4 Subsumption

Consider the entire set of data that flows along each edge into a Union node throughout plan execution. Ideally, we would like to infer set cover relationships among the edges leading into each Union. Then we would know what actions we could defer without jeopardizing completeness.

The reasoning tool at our disposal is *subsumption*, which is essentially set inclusion. Razor considers the edges into a Union node pairwise, and tries to discover pairwise set cover relationships (*i.e.*, subsumption relationships) deducible from the local completeness statements in Δ. The rest of this section elaborates this idea, which is given as an algorithm in Figure 3.

4.1 Alternatives

Razor tests for subsumption on the portions of the graph leading into the same Union node. More specifically, they should produce the same type of outputs from the same inputs. We call such subplans *alternatives*, and we characterize their graph properties as follows. Consider Union node U, with at least two predecessor nodes called M and N. Subgraphs G_M and G_N are alternatives whenever:

- M is the sink node of G_M and N is the sink node of G_N.
- All nodes in G_M are connected to M by a path entirely within G_M. Same for G_N.
- $\text{Predecessors}(G_M) = \text{Predecessors}(G_N)$, where the predecessors of G are the predecessor nodes of the nodes in G which are not themselves in G.

These graph properties roughly realize the natural notion of alternativeness. Function FindSubsuming-Alternatives looks at all pairs of disjoint alternatives in $O(|P|^3)$ operations and $O(|P|^2)$ calls to Subsumes*.

4.2 Subsumption over Alternatives

Function Subsumes* is a subgraph subsumption reasoning algorithm based on local completeness.[3] We use the

[3] An infinite depth limit makes it complete in the absence of built-in functions.

```
FindSubsumingAlternatives(acyclic plan P)
  Each node has one marker per possible sink node
  Reset all markers on all nodes
  Foreach union node U
    Foreach immediate predecessor node R of U
      Traverse the links backwards from R
      marking all nodes reached with tag R.
      (All other nodes have implicit tag ¬R)
    Foreach pair ⟨M,N⟩ of predecessors of U
      A_M = { nodes marked (M ∧ ¬ N)}
      A_N = { nodes marked (N ∧ ¬ M)}
      If Subsumes*(A_M,A_N) Then Found(A_M,A_N,U)
      If Subsumes*(A_N,A_M) Then Found(A_N,A_M,U)
Subsumes*(subgraph G, subgraph G')
  If G contains a union node
    Construct G_1 ... G_n |
      G_i has the ith alternative in place of the union
    Return ⋁_i Subsumes*(G_i,G')
  Else If G' contains a union node
    Construct G'_1 ... G'_n |
      G'_i has the ith alternative in place of the union
    Return ⋀_i Subsumes*(G,G'_i)
  Else Return Subsumes(G,G')
Subsumes(conjunctive subgraphs G, G')
  Return Contains*(G,G'∧ Expansion(G'))
Contains*(conjunction C, conjunction C')
  If Contains(C,C') then return true
  Else Choose a site relation T in C
    Choose an LC-rule R with head matching T and mgu σ
    Let E = RemoveConjunct(C,T) ∧ Body(R)|_σ
    If Contains*(E,C') return true
```

Figure 3: Pseudocode for subsumption detection. Expansion takes a conjunction of actions and replaces each action with the instantiated rule body from its site description. Contains searches for a containment mapping from C to C'[Ullman, 1989, p 881]; if found this proves that the extention of C contains that of C'. Choose marks a nondeterministic choice; backtracking must be used to consider all possibilities for completeness.

fact here that acyclic graphs are equivalent to logical recursive rules with database operations replaced by conjunction and disjunction. We say that subgraph G *subsumes* subgraph G' if from the site definitions and LC statements, Δ, one can deduce that G' is *contained* in G. At the heart of the algorithm is a containment mapping routine for conjunctions [Ullman, 1989, p 881]. If G contains G', executing G will gather at least the tuples gathered by G'.[4]

Looking again at our running example, we would compare the two subgraphs (call them G and G') consisting of MovieLink and MetroCinema respectively. The two graphs have the same input $City$, and their other variables are renamed away:

$$G = \text{MovieLink}(City, M, Theater) \quad (9)$$
$$G' = \text{MetroCinema}(City, M', Theater') \quad (10)$$

[4] In terms of formal logic, Subsumes* is just searching for a proof that for some substitution σ, $\Delta \models G'|_\sigma \Rightarrow G|_\sigma$. Contains* does backward-chaining search through Δ. Contains does unification.

Since MovieLink contains all possible movies and theaters (Equation 4), Subsumes* is able to determine that G subsumes G'. Let's walk through the process. Since neither graph contains a union node, Subsumes* quickly calls Contains*(G, G'). This causes a call to Contains with C set to the conjunction of G' and the expansion of its single conjunct.

$$C' = \text{MetroCinema}(City, M', Theater') \wedge \texttt{shows-in}(M', City, Theater') \quad (11)$$

C is just G. There is no containment mapping from C to C' so Contains returns false. At this point Contains chooses a conjunct and an LC-rule from Δ that will let it prove the implication. Let T perforce be MovieLink$(City, M, Theater)$, and suppose then that R is Equation 4. E becomes

$$E = \texttt{shows-in}(M', City, Theater') \quad (12)$$

The next (recursive) call to Contains* has C' unchanged (Equation 11), but C is now set to the previous value of E (Equation 12). Now there *is* a containment mapping: setting $M = M'$ and $Theater = Theater'$ makes C match the last conjunct of C'. Contains* returns true.

This algorithm works because containment is transitive. We build a chain connecting G to G'. C' is $G'\wedge$ Expansion(G'), which contains G' by construction. Similarly, C contains E on each recursive call. So whenever E contains C', the chain is complete.

In the worst case Subsumes* takes time exponential in three quantities:

- the number of Union nodes in series in G and G' (breaking loops),
- the depth limit on recursion, and
- the number of repeated predicates in the containment mapping.

However in practice, these quantities are all very small integers. Furthermore, preliminary experiments suggest that subsumption reasoning actually takes much less time than actually execution of the subsumed information-gathering actions, even given a fast network connection.

5 Contingent Planning

Razor uses the subsumption relationships it finds to turn the original plan into a more resource-conscious one.

Instead of executing all possible alternative subgraphs in a plan, Razor determines which ones can be skipped without losing tuples. However, we cannot simply prune away subsumed subgraphs of the plan, since that would eliminate the possibility of recovering from unavailable source failures. For example, if the SABRE site is down, Razor should contact United airlines, but this would be impossible if we have removed all alternatives to SABRE from the graph. Our approach is to generate a *contingent plan* by annotating the graph with *guards*. This allows Razor to execute either an *aggressive* or *frugal* policy.

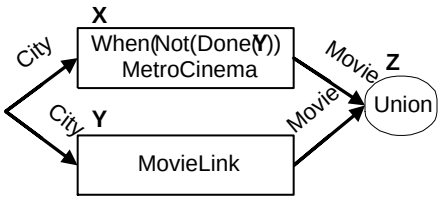
The contingent plan differs from the original plan in that a node may fire only when its guard is true. Guards are logical formulae that refer to the *status variables* of

other nodes. There is one status variable per node, with the possible values SLEEPING, RUNNING, FAILED, and DONE. Guards make the execution of a node contingent on the execution status of other parts of the graph. Whatever policy is chosen, the executor's task is then simply to:

- fire RUNNING nodes when their inputs are available,
- complete RUNNING nodes when their data arrives,
- update the status variables accordingly, and
- awaken or put to sleep nodes whose guards changed value.

5.1 Generating the Guards

In our example, using the aggressive policy, the only change between the original plan and the contingent plan is one guard on the METROCINEMA node, causing it to sleep if the MOVIELINK call is DONE. Here is a close-up of that part of the contingent plan:



In fact, the aggressive execution policy only tests when sink nodes are DONE. No guards start out false; only when a subsuming alternative becomes DONE do some computations get put to sleep—those completely subsumed by alternatives that are DONE. This is conservative, in the sense that it will not lose any answers that the brute force method will find, and its use of internet resources is at least as good. The algorithm in Figure 4 implements the aggressive policy. It labels the nodes in subsumed alternatives A_N, from the list of subsumption relationships found by FindSubsumingAlternatives. Note that the guard of a node P is always *true* unless P is subsumed by some DONE alternative for every UNION node that uses P. Otherwise, the guard is the negated conjunction of the conditions on each UNION making P redundant. The guard says, in effect, that if at least one of the subsuming alternatives on every path from P is DONE, then P is unnecessary.

The frugal execution policy (not detailed here) has a similar structure, with some extra machinery. The guard on node X will be simply `failed(`Y`)` rather than `not(done(`Y`))`. In addition the frugal policy needs an extra module to prevent deadlock. This module has to detect and break cycles in the subsumption relation between nodes, by using estimates of utility (*e.g.*, expected speed or micropayment cost) to prevent alternatives of higher utility from waiting for alternatives of lower utility.

6 Status & Future Work

We have fully implemented one information-gathering agent and are now constructing its successor Razor. The first, "Razor96" is written in Common Lisp and uses the Occam planner [Kwok and Weld, 1996] as its planning

GenerateGuards()
  Declare clause[all nodes,union nodes]
  Initialize all clauses to *false*
  Foreach $\langle A_M, A_N, U \rangle$ we found
    Foreach node P in A_N
      Add `status(`M, DONE`)` to clause[P,U]
  Foreach node P
    S_P = { union nodes reachable from P without passing through any other union nodes }
    P.guard = $\neg \bigwedge_{U \in S_P}$ (clause[P,U])

Figure 4: Pseudocode for aggressive guard generation.

module. Since Occam produces a stream of conjunctive plans, the subsumption module just compares pairs of conjunctive plans. Subsumes* extends and improves this algorithm. Furthermore, Razor96 does not use guards; rather, it prunes any subsumed conjunctive plans. It is therefore not insured against information source failures. This pruning algorithm prunes over 99% of the plans Occam produces on various movie-related queries. We found that most plans were pruned on their own (lack of) merit; either because they contain unnecessary actions (nonminimality), or are found redundant without using the recursive clause of Contains*. Using equation 4 further reduced the number of plans by a factor of 2.67. The entire subsumption and pruning process increased planning time only 53%.

The frailty of our original Lisp implementation, combined with an improved undertanding of the ideal system architecture, led us to design the Razor system described in this paper. Our new C++ implementation uses recursive plans [Duschka and Levy, 1997] to construct the plan graph. Recursive plans are already far more compact than Occam plans, so we anticipate that the improvement in graph *size* from labelling will be far less drastic than in our previous implementation, however we still expect dramatic improvement over brute force execution. We are currently porting the subsumption and execution systems, and coding guard-generation. Razor will learn utility estimates for sites, and implement the frugal policy. Further down the road we will look at *optimal* strategies for maximizing the answers returned by a deadline.

7 Related Work

Razor draws together work in databases and AI planning. Several implemented systems (*e.g.*, [Chawathe *et al.*, 1994; Adali *et al.*, 1996; Daruwala *et al.*, 1995]) integrate multiple database systems using hand-tailored query plans. They focus on orthogonal problems, such as expressing ontologies and converting between them. The Internet Softbot [Etzioni and Weld, 1994] applies AI planning techniques to the problem of accessing online information as well as update. The SIMS database integration system [Arens *et al.*, 1996] applies AI planning to the problem of relational database integration with a rich hierarchical type system. The Information Manifold (IM) [Levy *et al.*, 1996a] is similar to SIMS and Razor. Like SIMS, IM handles object type hierarchies. Like Razor, IM is geared toward users of the web. This

incarnation of the IM system is incapable of handling information sources other than full relational databases. [Levy *et al.*, 1996b] describes an apparently later version of the IM which allows bound variables in site descriptions. This version uses an incomplete algorithm, however, for generating bounded-length information-gathering plans. For example, the IM can't find plans that require more steps than there are goal conjuncts; see [Rajaraman *et al.*, 1995] for examples.

[Etzioni *et al.*, 1994] introduced the notion of local closed world and presented fast algorithms for inference and update. [Levy, 1996] generalizes Etzioni *et al.*'s local completeness framework and presents a reduction of the answer-completeness problem to the query-independence problem. We further extend Levy's representation and solve the extended problem directly. Although there is a progression in expressiveness from the Internet Softbot to the IM and Razor only the Internet Softbot handles the case of update since its actions can affect the information sources.

Several planners in the AI literature represent contingencies within the plan [Peot and Smith, 1992; Draper *et al.*, 1994; Pryor and Collins, 1996]. However, the absence of causal (state-changing) actions in an information-gathering context leads to significant differences. In a sense, Razor's non-contingent plan corresponds to a *universal plan* [Schoppers, 1987] and we introduce contingencies only for efficiency's sake.

8 Conclusions

Razor is an information-gathering agent that plans how to answer user queries, reasons about the redundancy of accessing the different sources, and then executes the actions in a manner that attempts to get the maximum information to the user without useless work. Razor uses a library of expressive *local completeness* declarations in order to determine which alternative subplans obtain similar information. Razor then constructs a *contingent plan* that encodes an efficient execution policy which is resilient in the face of information-source failure.

9 Acknowledgements

This research would not have been possible without the efforts of Chung Kwok, whose previous research laid the foundations for ours. He also contributed his time and code. We also thank Bob Doorenbos, Oren Etzioni, Keith Golden, Nick Kushmerick, Marc Langheinrich, Neal Lesh, Alon Levy, Kelly Murray, Dan Shiovitz, and David Smith, each of whom helped in some invaluable way. This research was funded 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, by a National Science Foundation Graduate Fellowship, and by a gift from Rockwell International Palo Alto Research Lab.

References

[Adali *et al.*, 1996] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Procceedings of SIGMOD-96*, 1996.

[Arens *et al.*, 1996] Y. Arens, C. Knoblock, C. Chee, and C. Hsu. SIMS: Single interface to multiple sources. TR RL-TR-96-118, USC Rome Labs, 1996.

[Chawathe *et al.*, 1994] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. IPSJ Conf*, 1994.

[Daruwala *et al.*, 1995] A. Daruwala, C. H. Goh, S. Hofmeister, K. Hussein, S. Madnick, and M. Siegel. The context interchange network. In *IFIP WG2.6 Sixth Working Conference on Database Semantics*, 1995.

[Draper *et al.*, 1994] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Intl. Conf. on AI Planning Systems*, June 1994.

[Duschka and Levy, 1997] O. Duschka and A. Levy. Recursive plans for information gathering. In *Proc. 15th Int. Joint Conf. on AI*, 1997.

[Etzioni and Weld, 1994] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.

[Etzioni *et al.*, 1994] Oren Etzioni, Keith Golden, and Dan Weld. Tractable closed-world reasoning with updates. In *Proc. 4th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 178–189, 1994.

[Etzioni *et al.*, 1997] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89(1–2):113–148, January 1997.

[Kwok and Weld, 1996] C. Kwok and D. Weld. Planning to gather information. In *Proc. 13th Nat. Conf. on AI*, 1996.

[Levy *et al.*, 1996a] A. Levy, A. Rajaraman, and J. Ordille. Query-answering algorithms for information agents. In *Proc. 13th Nat. Conf. on AI*, 1996.

[Levy *et al.*, 1996b] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference*, 1996.

[Levy, 1996] A. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22nd VLDB Conference*, 1996.

[Peot and Smith, 1992] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Intl. Conf. on AI Planning Systems*, pages 189–197, June 1992.

[Pryor and Collins, 1996] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *J. Artificial Intelligence Research*, 1996.

[Rajaraman *et al.*, 1995] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. ACM Symp on Principles of Database Systems*, 1995.

[Schoppers, 1987] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proc. 10th Int. Joint Conf. on AI*, pages 1039–1046, August 1987.

[Ullman, 1989] J. Ullman. Database and knowledge-base systems. In *Database and knowledge-base systems*. Computer Science Press, 1989.