Utilizing Volatile External Information During Planning

Tsz-Chiu Au and **Dana Nau** and **V.S. Subrahmanian**[1]

Abstract. There are many practical planning situations in which planners may need information from external sources during the planning process. We describe the following:

- Wrappers that may be placed around conventional (isolated) planners. The wrapper replaces some of the planner's memory accesses with queries to external information sources. When appropriate, the wrapper will automatically backtrack the planner to a previous point in its operation.
- Query-management strategies for wrappers. These dictate when to issue queries, and when/how to backtrack the planner.
- Mathematical analysis and experimental tests. Our results show conditions under which different query management strategies are preferable, and demonstrate that certain kinds of planning paradigms are more suited than others for planning with volatile information.

1 INTRODUCTION

A fundamental assumption of most AI-planning research is that the planner is *isolated*: it is given a problem specification when it is invoked, and receives no further input while it is running. In many practical planning situations, this assumption is unrealistic. Planners may need query information sources such as database systems [3], CAD systems [24], human users [20], and web services [27].

One problem with such queries is *lag time*. Dix *et al.* [4] found that when the planner queried other agents for information rather than having it available internally, the planning time increased by more than an order of magnitude.

Another problem with the queries is *information volatility during planning*. In practical planning situations, the planning activity may occur over a period of hours, days, or weeks [18, 11]: a period much longer than the actual execution time of the plan. This means that some of the relevant information is likely to change during the planning process: if we lock a database or reserve a ticket, the lock or reservation may lapse; if we obtain a sensor reading, we may be unable to guarantee its accuracy for more than a short period of time. If the information changes, the plan may need to be revised.

The focus of the paper is how to manage planning systems, queries, and responses in order to plan with volatile information. Our contributions are as follows:

- We describe *wrappers* that may be placed around isolated planners, to enable them to issue queries for volatile external information and make appropriate use of the answers. The wrapper replaces some of the planner's memory accesses with queries that the wrapper will direct to either an internal cache or an external information source. When appropriate, the wrapper will automatically backtrack the planner to a previous point in its operation.
- We describe several different query-management strategies for wrappers to use. These strategies dictate when to issue queries, and when and how to backtrack the planner.
- We describe experimental tests of several query management strategies on wrapped versions of three well-known planners, SNLP [17], Graphplan [2], and SHOP2 [21]. Our experimental results show conditions under which different kinds of query management strategies are preferable, and conditions under which certain planners are preferable.

[1] Department of Computer Science, University of Maryland, College Park, Maryland, U.S.A. emails:{chiu,nau,vs}@cs.umd.edu

2 OUR MODEL

In this section we describe how to map a conventional planning procedure A into a *wrapped procedure* $\hat{A}$, which is also called a *wrapper*, that gets information from external sources during planning. We do not care what kind of planning procedure A is, except that A is invoked on a problem description P written in some language L, A gets no additional information after it has been invoked, and if A terminates it either returns failure or returns a plan or policy π that is a solution for P.[2]

The language L will usually have many different types of syntactic expressions. We now define a language $\hat{L}$ that includes all of the expressions of L, and also includes additional symbols called *unknowns*. Each unknown u has one of L's expression types assigned to it, and is only allowed to denote expressions of the same type. Expressions in $\hat{L}$ are like the ones in L, except that in $\hat{L}$ they may contain (or be) unknowns of the appropriate types. An expression e is *u-unground* if it is an unknown or contains one or more unknowns; otherwise it is *u-ground*. An expression e' is a *u-instance* of e if e' can be obtained from e by substituting expressions for unknowns.

If a planning problem P is u-ground, then its *solutions* in $\hat{L}$ are the same as its solutions in L. If P is u-unground, then π is a solution for P iff π is a solution for every u-ground instance of P.

We now describe the *wrapper* $\hat{A}$. For each unknown u in P, there will be an information source $\sigma(u)$ which $\hat{A}$ may query for the value for u. $\hat{A}$ will have a cache for holding answers to queries; we assume that the size of the cache is unlimited. Each time A needs to know the value of some unknown u in P, $\hat{A}$ may either retrieve a value from the cache or send a query to $\sigma(u)$.

While $\hat{A}$ is running on P, it will issue some sequence of queries $q_1, q_2, \ldots$, which may either be synchronous (i.e., $\hat{A}$ pauses until an answer is received) or asynchronous ($\hat{A}$ continues to operate, and may issue additional queries). For each query q, we let $u(q)$ be the

[2] In the worst case, we can wrap any planning algorithm by making a copy of the planning algorithm's execution state whenever it makes a query. Depending on the planner, it may be sufficient to store much less information.

unknown whose value is requested, and $t_l(q)$ be the *lag time* (the elapsed time between issuing q and getting a response).

Associated with q is an *expiration time*, the amount of time that the response to q is guaranteed to remain valid. After the expiration time has passed, the value of the unknown $u(q)$ may change.

3 QUERY MANAGEMENT STRATEGIES

In the previous section we described one of the query management strategies that the wrapped procedure $\hat{A}$ might use. We will call that query strategy the *eager update strategy*, since the wrapper re-issues a query immediately after its value expires.

The eager update strategy is not always best. When a query q has expired, the value of the unknown $u(q)$ will not necessarily change—and if it does change, it might later return to the previous value. Thus instead of reissuing the query immediately (which will incur a lag time and will increase the load on the network), it may be better to use a *lazy update strategy*: continue planning *as if* the value of $u(q)$ were unchanged—and once we find what seems to be a solution, reissue the query to make sure whether the solution is correct.

It is not hard to show that both strategies are sound, and we have derived conditions under which they are complete. We have also developed a *periodic* update strategy that is intermediate between the eager and lazy strategies. We omit these results due to lack of space.

The implementation of the wrapped procedure needs to maintain a data structure called a *query tree*. Suppose P is u-unground, and let $\mathcal{P}$ be the set of all u-ground instances of P. Suppose we run $\hat{A}$ on P, using the following *query-management strategy*: (1) if A needs a value for an unknown u that is not in the cache, issue a query for u and wait for a response; (2) if a query q expires, immediately *re-issue* it (i.e., issue a query q' with $u(q') = u(q)$), and wait for a response; and (3) if $v(q') \neq v(q)$ then backtrack A to the point where q was issued, and proceed using $v(q')$ as the value for $u(q)$. We now consider two cases:

Case 1: no query ever expires. Then $\hat{A}$ will never backtrack over A's execution, so $\hat{A}$'s execution trace on P is basically the same as A's execution trace on some instance of P. There is one possible execution trace for each combination of responses to the queries. These execution traces form a tree Y in which each internal node corresponds a query, each edge corresponds to a portion of an execution trace, and each terminal node corresponds to the termination of an execution trace. We will call this tree $\hat{A}$'s *query tree* for P. We let $\tau_1, \tau_2, \ldots, \tau_n$ be the terminal nodes, and parent(ν) be the parent of each node ν.

Case 2: some query q expires. Then $\hat{A}$ will backtrack A to the point in Y where q was issued, and reissue q. If the answer to the query differs from before, then A will proceed down a different branch from before.

As an example, consider a simple transportation-planning problem in which Jim wants to travel from one city to another, he may go by either train or airplane, and he does not know the price of the train ticket, the price of the airline ticket, and the amount of money in his bank account.

We can model this as a u-unground planning problem P in which there are three numerical unknowns: train_price, airline_price, and bank_balance. Due to lack of space, we will not give the details of the initial state and the operators—but the basic idea is that in order to decide whether Jim can afford to fly, we will need to retrieve airline_price and bank_balance, and in order to decide whether he can go by train, we will need to retrieve train_price and bank_balance.

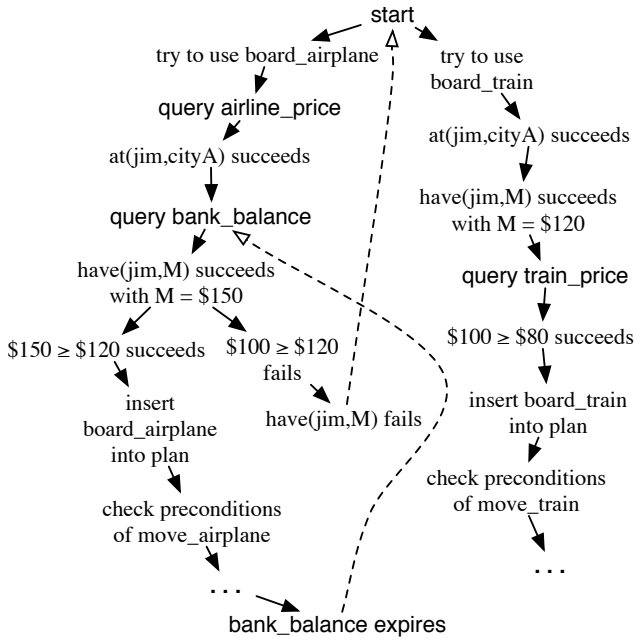

Figure 1. Query tree for the example.

Let A be a planner that uses a depth-first forward search, and $\hat{A}$ be the wrapped planner. First, A starts checking whether Jim can afford to fly. First, A tries to access the value of airline_price, so $\hat{A}$ queries a travel agent (query q1 in Figure 1). After 10 minutes, the answer is $120. Next, A tries to access bank-balance, so $\hat{A}$ queries the bank (query q2 in the figure). After 5 minutes, the answer is $150. Jim can afford to fly to city B, so A starts putting the appropriate steps into the plan for his flight. However, after 5 more minutes the bank notifies $\hat{A}$ that the bank balance has changed. This may invalidates the current plan, so $\hat{A}$ backtracks A to q2 and reissues it (query q3 in the figure). The new value of bank_price is 100. So Jim can no longer afford to fly, and then A starts checking whether he can afford to go by train. To do this, it needs to know train_price, so $\hat{A}$ queries to the train station. After 15 minutes, the answer is $80. Thus Jim can afford to go by train.

4 ANALYSIS OF RUNNING TIMES

We now analyze the query management strategies' running times.

As $\hat{A}$ progresses, it will explore the nodes and edges of the query tree. We assume that $\hat{A}$ will represent the query tree in its cache as follows. For each node ν, $\hat{A}$ will store the unknown u currently being queried and the current execution state for A; and for each edge emanating from ν, $\hat{A}$ will store the corresponding value of u. By storing this information, $\hat{A}$ can backtrack to ν quickly if u's value expires. Furthermore, if the re-issued query produces a combination of values that $\hat{A}$ has seen before, $\hat{A}$ will be able to resume A's execution at the node corresponding to this set of values, without having to re-execute all of A's computations leading to that node.

We let d be the average number of query nodes on each path in the query tree, t_l be the average lag time for any query, t_q be the average time between any two consecutive queries, and t_e be the average time between any two consecutive expirations of queries. For $i = 1, \ldots, n$, we let T_i be the amount of CPU time A spends on the edge between parent(τ_i) and τ_i.

For our analysis, we will consider the case where the maximum expiration time is small enough for at least one expiration to occur on each path in the query tree. Each time an expiration occurs, $\hat{A}$ using eager update strategy will cache A's execution state, indexed by the associated set of values for the unknowns, then backtrack to the query node whose query expired and reissue the query. When the new answer to the query arrives, $\hat{A}$ will either jump back to the place it just left, or will jump to some other path in the query tree, depending on whether or not the new value for $u(q)$ is the same as the old one. $\hat{A}$ using lazy update strategy, however, will not cache A''s execution state and backtrack immediately when an expiration occurs; it will do so only when $\hat{A}$ reaches a terminal node, at which it will re-issue all pending expirations at once. We let m^{eager} and m^{lazy} be the total number of jumps $\hat{A}$ with eager update strategy and $\hat{A}$ using lazy update strategy makes before it terminates, respectively.

Lazy update strategy. In the lazy update strategy, $\hat{A}$ only checks for expirations when A reaches a terminal node. This happens when A either finds a solution or exits with failure. Let $\tau_1, \ldots, \tau_k$ be all of the terminal nodes that $\hat{A}$ visits. If $\hat{A}$'s queries ever again produce the same set of answers that led to some τ_i, $\hat{A}$ will go immediately to the cached value for τ_i, at which point it can immediately exit. Thus, for each τ_i, the computational work done by A to get to τ_i will only need to be done once, and k is equal to m^{lazy}. Let T_{avg} be the average value of $\{T_i : \hat{A} \text{ visits } \tau_i\}$, and T_{GP}^{lazy} be the sum of the CPU times for all edges of the query tree that are above $\tau_1, \ldots, \tau_k$ but not adjacent to $\tau_1, \ldots, \tau_k$. For each i, we let t_i and t_i' be the times when $\hat{A}$ jumps to a path from the root of the query tree to the τ_i and away from the path, respectively. For each i, let r_i be the maximum lag time for all queries issued at time t_i' (i.e., $r_i = t_{i+1} - t_i'$). It follows that the total running time of the lazy update strategy is

$$T_{lazy} = T_{GP}^{lazy} + m^{lazy} T_{avg} + \sum_{i=1}^{m^{lazy}} r_i.$$

Eager update strategy. In this strategy, $\hat{A}$ immediately backtracks to the query node for the unknown u_i when the value of u_i expires, the execution trace for A at τ_i is usually left unfinished. Each time $\hat{A}$ visits the path to to τ_i, it spends an average of t_e time extending A's execution trace before it moves to another path. $\hat{A}$ will exit as soon as it reaches a terminal node τ_i. Let $p = T_{min}/t_e$ be the average number of visits for each path toward each terminal node. Let $\tau_1, \ldots, \tau_j$ be all of the terminal nodes that $\hat{A}$ visits. When $\hat{A}$ terminates, each path from the root of the query tree to any τ_i has already been visited p times on average, and $\hat{A}$ has spent T_{min} amount of CPU time to work on each path on average. There are a total of $m^{eager} \times p$ transitions from one path to another, and since there is only one query at each transition, each transitions takes an average t_l amount of time, where t_l is the average lag time of each query. T_{GP}^{eager} be the sum of the CPU times for all edges of the query tree that are above $\tau_1, \ldots, \tau_j$ but not adjacent to $\tau_1, \ldots, \tau_j$, where j is equal to $m^{eager} \times p$. Thus, the total running time of the eager update strategy is

$$T_{eager} = T_{GP}^{eager} + m^{eager} T_{min} + m^{eager} p t_l.$$

Comparison between T_{lazy} and T_{eager}. It is difficult to say which of T_{GP}^{lazy} and T_{GP}^{eager} is larger: on one hand, it is likely that $m^{lazy} < m^{eager}$, but on the other hand, the eager strategy will spend less CPU time on each path toward each terminal nodes. The total CPU time can be much larger for the lazy strategy than the eager one ($m^{lazy} T_{avg} \geq m^{eager} T_{min}$) when T_{avg} is much larger than T_{min}, but the total lag time can be much smaller for the lazy strategy than the eager one ($\sum_{i=1}^{m^{lazy}} r_i \leq m^{eager} p t_l$) when p is much larger than 1. Thus, either strategy can have a larger running time than the other. If expirations are frequent (t_e is small), then p is large, so the third term of the equation for T_{eager} will be much larger than for T_{lazy}. Furthermore, the lazy update strategy can issue several queries simultaneously, while the eager update strategy cannot. Thus in this case, usually $T_{lazy} < T_{eager}$. However, if T_{min} is very small and expirations occur less frequently, then potentially $T_{eager} < T_{lazy}$.

5 EXPERIMENTS

Since the analysis required several simplifying assumptions, an important question is whether these hypotheses are true even when the assumptions are not satisfied. We now investigate this experimentally. Our experimental hypotheses were that (1) the lazy update strategy would usually issue fewer queries than the eager update strategy, and (2) the lazy update strategy would usually produce a smaller total running time (including both CPU time and lag time) than the eager update strategy.

Experimental Setup: For our experimental tests, we used wrapped versions of three planners: SNLP, Graphplan, and SHOP2.[3] For SHOP2, our testbeds consisted of 60 randomly generated u-unground problems from the satellite domain used in the AIPS-2002 planning competition, and 80 randomly generated u-unground logistics problems. Graphplan cannot handle numeric values; so we only tested it on the logistics problems. In addition, Graphplan is much slower than SHOP2: it could solve only 18 of the 80 problems within the time limit of our experiments.[4] SNLP was too slow to solve even a single one of the logistics problems. So for both SNLP and Graphplan we used 20 u-unground problems from a simplified version of logistics domain that we called the travel domain. Each problem contained at most four unknowns; most unknowns had two possible values.

In order to test the two hypotheses, we measured the total number of queries and total running time for the wrapped planning procedures. We kept the lag times fixed at 0.1 seconds, and varied the expiration times from 0.2 to 1.5 seconds. For each combination of planner, problem, expiration time, and query management strategy we did five runs. Thus each data point for the satellite domain is the average of 300 runs, each data point for the logistics domain is the average of 400 runs of SHOP2 or 90 runs of Graphplan, and each data point for the travel domain is the average of 100 runs of SNLP or 100 runs of Graphplan. Since this gave us a total of more than 30,000 runs to perform, we needed to limit the running time of each procedure; we chose a limit of 5 minutes per run.

Figure 2 shows the total number of queries issued by the wrappers as a function of the expiration time. Note that the y axis is on a logarithmic scale that spans three orders of magnitude. In most cases the eager update strategy generates many times more queries than the lazy update strategy, especially when the expiration time is small. These results confirm Hypothesis 1.

Figure 3 shows the total running time for the wrapper as a function of the expiration time. Again the y axis is on a logarithmic scale.

[3] For this purpose we used a simulation. For a planner A, it is straightforward to develop a simulation of the wrapped planner $\hat{A}$ that gives very accurate results. The basic idea is to run invocations of A separately on each of the paths in the query tree and keep track of the timing data. This data can then be reused for several simulations, making it possible to simulate a large number of runs of the planner in a short amount of time.

[4] SHOP2 ran so much faster than the other planners because it can make use of domain-specific information

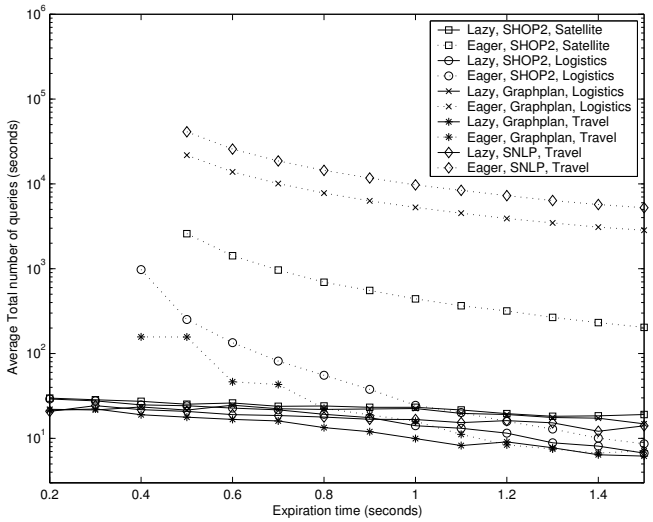

Figure 2. Total number of queries as a function of expiration time. Eager and lazy update strategies are denoted by dotted and solid lines, respectively. The data points for Graphplan and SNLP include only the problems that they could solve within our time bound.

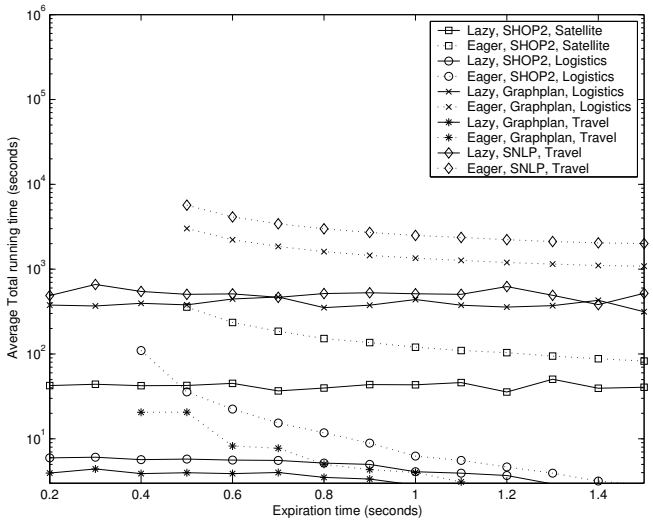

Figure 3. Total running time as a function of expiration time. Eager and lazy update strategies are denoted by dotted and solid lines, respectively. The data points for Graphplan and SNLP include only the problems that they could solve within our time bound.

In every case, the lazy update strategy has a smaller running time than the eager update strategy, regardless of the planner, domain, and expiration time. This confirms Hypothesis 2.

In addition to confirming the two hypotheses, Figures 2 and 3 suggest that of the three planners, SHOP2 is the one that is best suited for solving planning problems in which there is volatile external information. In the logistics domain, it consistently generated fewer queries than Graphplan (and the data points for SHOP2 include significantly more complicated problems than Graphplan, since Graphplan was only able to solve 20% of the logistics problems). SHOP2 would also have generated fewer queries than SNLP if we had run it on the problems in the travel domain.

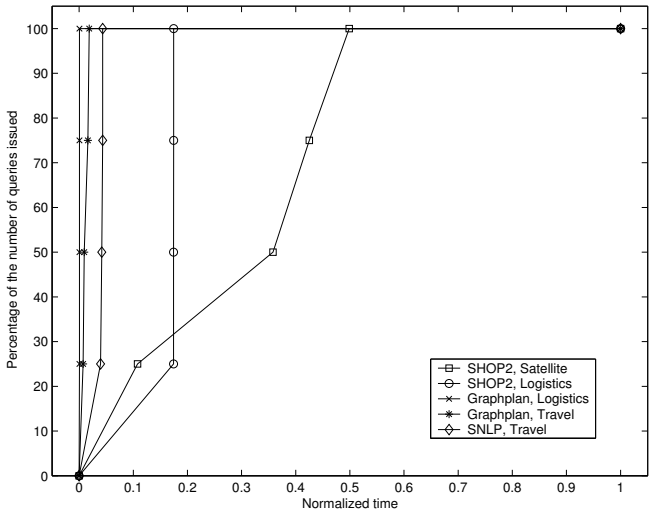

Figure 4. Fraction of the total number of queries issued at any point during the planning process if there is no expiration. All timelines are normalized to 1. Each line is the average over all possible execution traces.

To provide additional verification of which planner would be most suitable for solving in which there is volatile external information, we did one more experiment. In this one, we generated the entire query tree for each planner. Each path in the query tree represents what the planner's execution trace would be if no expirations ever occurred. For each of these paths, and for each query along that path, we computed the total number of queries that a planner has issued as a function of what percentage of the total planning time has elapsed. We averaged this data over all of the paths in the query tree: this consisted of 320 paths for SNLP and 384 paths for Graphplan in the travel domain, 288 paths for Graphplan and 2876 paths for SHOP2 in the logistics domain, and 2236 paths for SHOP2 in the satellite domain.

The figure shows that both Graphplan and SNLP issue almost all of their queries at the beginning of their planning process, while SHOP2's queries are somewhat more spread out. This provides additional confirmation that SHOP2 is better suited than SNLP or Graphplan for solving problems when there is volatile external information. If a planner issues all of its queries at once, this will temporarily increase the load on the communication network and the information sources, which is likely to increase the lag time for those queries. A planner that spreads out its queries will avoid this difficulty.

6 RELATED WORK

Our problem stems from the works on integrating planning system with multi-agent environments, in which a planning agent can interact with external agents, and make queries to distributed, heterogeneous information sources. There are different types of multi-agent planning problem, but our problem is most similar to the problem in which a single planner creates plans for several agents [4, 22, 26]. This type of problem arises in application areas such as multi-robot environments, distributed database management system, servers distributed over the Internet, logistics, manufacturing, evacuation operations and games.

Although we cannot find any work directly related to our prob-

lem, many works relate to different aspects of our problem. First, our problem is similar to contingent planning problem with partial observability, such as planning with information gathering and with sensing action [5, 14] and conditional planning [23]. The key difference is that the sensing actions of our planning agent—the issues of queries—-are executed during plan generation rather than during plan execution. Thus we gather information to learn what the planner does not know during planning. Second, the management of the expiration of answers in real time shares some aspects of the works in real time searching [16] and real-time path planning[15, 25]. Reactive planning [1, 6, 7, 19] handles real time information during plan execution rather than during the plan generation. Third, the adaption of new information makes use of the techniques in plan adaption [9, 10, 13], especially plan reuse [12], which is exactly how our wrappers resume previously saved runtime stacks of A. Fourth, the continuation of planning based on assumption making in lazy update strategy is like PUCCINI [8], a partial-order planner that allows the option of assuming that certain preconditions hold, performing the action, and verifying the preconditions afterward.

Sage [14] is an augmented version of UCPOP that constructs plans for how to gather information during plan execution. One distinction between their work and ours is that Sage does not do information-gathering during plan construction. However, Sage can do replanning, which in some cases could amount to the same thing. Another distinction is that Sage is a single planning algorithm, whereas we provide a wrapper that can be used with a large number of different planning algorithms, and we provide results about the performance of wrapped planners under different conditions.

7 CONCLUSIONS

In this paper, we have examined how to do planning in situations where the planner needs to get information from external sources and this information may change during the planning process.

We have described a general formulation for *wrappers* that may be placed around conventional planners (ones that do not query external information sources), in order to allow them to make such queries. The wrapper replaces some of the planner's memory accesses with queries to external information sources. When appropriate, the wrapper will automatically backtrack the planner to a previous point in its operation.

We have described two query-management strategies for wrappers: an *eager* update strategy that re-issues queries whenever the needed information expires, and a *lazy* update strategy that postpones re-issuing queries until later.

Our mathematical analyses of the query-management strategies suggest that the lazy update strategy usually issues fewer queries than the eager update strategy. The lazy update strategy will often require more CPU time than the eager update strategy—but the lazy strategy's total running time (its CPU time plus the lag times for its queries) is likely to be smaller than the total running time of the eager update strategy.

Our experimental tests of three different planning systems (SHOP2, Graphplan, and SNLP) confirm the analytical results. Our experiments also suggest that SHOP2 will be more suitable than Graphplan and SNLP for planning with volatile external information, for two reasons. First, SHOP2 issues fewer queries. Second, these queries are somewhat more spread out over time, which is preferable because it places a smaller transient load on the communication network and information sources.

ACKNOWLEDGEMENTS

This work was supported in part by Army Research Lab contracts DAAL0197K0135 and DAAD190320026, the CTAs on Advanced Decision Architectures and Telecommunications, ARO contract DAAD190010484, DARPA/RL contracts F306020020505 and F306029910552, and NSF grants 0205489 and IIS0329851.

REFERENCES

[1] M. Beetz and D. McDermott, 'Declarative Goals in Reactive Plans', in *AIPS*, (1992).
[2] A. L. Blum and M. L. Furst, 'Fast planning through planning graph analysis', *IJCAI*, 1636–1642, (1995).
[3] S. Chien, R. Hill, X. Wang, and T. Estlin, 'Why real-world planning is difficult: A tale of two applications', in *EWSP*, (1995).
[4] Jürgen Dix, Héctor Muñoz-Avila, Dana S. Nau, and LingLing Zhang, 'IMPACTing SHOP: Putting an AI planner into a multi-agent environment', *Annals of Mathematics and AI*, **37**(4), 381–407, (2003).
[5] D. Draper, S. Hanks, and D. Weld, 'Probabilistic planning with information gathering and contingent execution', in *AIPS-94*, pp. 31–36, (1994).
[6] M. Drummond, 'Situated control rules', in *KR*, pp. 103–113, (1989).
[7] J. R. Firby, 'An investigation into reactive planning in complex domains', *Artif. Intel.*, **3**, 251–288, (1987).
[8] Keith Golden, 'Leap Before You Look: Information Gathering in the PUCCINI planner', in *AIPS*, pp. 70–77, (1998).
[9] K. J. Hammond, *Case-Based Planning: viewing learning as a memory task*, Academic Press, New York, 1989.
[10] S. Hanks and D. S. Weld, 'A Domain-Independent Algorithm for Plan Adaptation', *JAIR*, **2**, 319–360, (1995).
[11] i2 Technologies. Reducing planning cycle time at Altera Corporation. http://www.i2.com/assets/pdf/96FDF2C7-71C7-43B5-906A01BAE2F0AE76.pdf, 2002.
[12] L. Ihrig and S. Kambhampati, 'Plan-space vs. State-space planning in reuse and replay', Technical report, Arizona State University, (1996).
[13] S. Kambhampati and J. A. Hendler, 'A validation structure based theory of plan modification and reuse', *Artif. Intel.*, **55**, 193–258, (1992).
[14] Craig A. Knoblock, 'Planning, executing, sensing, and replanning for information gathering', in *IJCAI*, ed., Chris Mellish, pp. 1686–1693, San Francisco, (1995). Morgan Kaufmann.
[15] S. Koenig and R. Simmons, 'Solving robot navigation problems with initial pose uncertainty using real-time heuristic search', in *AIPS*, (1998).
[16] R. Korf, 'Real-time heuristic search', *Artif. Intel.*, **42**, 189–211, (1990).
[17] D. McAllester and D. Rosenblitt, 'Systematic nonlinear planning', in *AAAI*, pp. 634–639, (July 1991).
[18] William H. McRaven, *Spec Ops : Case Studies in Special Operations Warfare: Theory and Practice*, Presidio Press, 1996.
[19] M.Schoppers, 'Universal plans for reactive robots in unpredictable environments', in *IJCAI*, pp. 1039–1046, (1987).
[20] Héctor Muñoz-Avila, D. Aha, L. Breslow, and Dana S. Nau, 'HICAP: an interactive case-based planning architecture and its application to noncombatant evacuation operations', in *IAAI*, pp. 870–875, (1999).
[21] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, William Murdock, Dan Wu, and Fusun Yaman, 'SHOP2: An HTN planning system', *JAIR*, **20**, 379–404, (December 2003).
[22] F. Pecora and A. Cesta, 'Planning and scheduling ingredients for a multi-agent system', in *ICMAS*, (2002).
[23] M. Peot and D. Smith, 'Conditional nonlinear planning', in *AIPS*, pp. 189–197, (1992).
[24] S. J. J. Smith, K. Hebbar, Dana S. Nau, and I. Minis, 'Integrating electrical and mechanical design and process planning', in *Knowledge Intensive CAD, Volume 2*, eds., Martti Mantyla, Susan Finger, and Tetsuo Tomiyama, 269–288, Chapman and Hall, (1997).
[25] Anthony Stentz, 'Optimal and efficient path planning for partially-known environment', in *ICRA-94*, pp. 3310–3317, (1994).
[26] V.S. Subrahmanian, Piero Bonatti, Júrgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross, *Heterogeneous Agent Systems*, The MIT Press, 2000.
[27] Dan Wu, Bijan Parsia, Evren Sirin, James Hendler, and Dana Nau, 'Automating daml-s web services composition using SHOP2', in *ISWC2003*, (2003).