Appendix B: Implementation of Efficient Planning

B.1 Introduction

This appendix provides some details of how certain parts of the planning algorithms were implemented. Perusal may help to clarify why certain things were done the way they were, and give some insight into where difficulties remain.

These planning algorithms were designed to run essentially nonincrementally, e.g., they do not amortize their effort over the learning lifetime, but do all of it at the end. In a real agent, of course, a system that incrementally built or modified the data structures would be more appropriate.

Note that all of these algorithms were developed for use on a serial machine. Parallel machines may be able to finesse some of these problems by doing their planning in a parallel fashion—at least as long as the problem does not grow larger than the number of available processors, of course.

B.2 Reachability

The first requirement when attempting to plan a path from one schema to another is to be assured that there is, in fact, such a route. Simple explorations of a digraph composed of several thousand schemas will run exponentially slowly and are completely unsuitable; other algorithms, such as the iterative deepening algorithm discussed in Section B.4, on page 150, will fail to terminate if called to compute a path that does not, in fact, exist.

The general idea is to function like a depth-first, mark-sweep garbage collector. We build a *root set* of all schemas with null contexts and non-null results. (Such schemas must

be at the start of any chain, since, with null contexts, they cannot be chained to.) For each root object, we traverse the set of all schemas with non-null contexts. If we find an unmarked schema whose context chains from the result of the root object, we add it to the chain being built, mark the schema we just found, and then use the new schema as the base of a chain to be recursively extended. At any point during this process, we abandon the attempt to extend the chain if we either cannot find a next schema to chain to, or if the next schema is already marked (to prevent loops)—a marked schema must, by definition, have already had its chain traversed, or to be in the process of having its chain traversed at the moment. Since schemas are never duplicated and are therefore guaranteed unique [Drescher 91], we cannot find that the first schema we find to extend the chain is already marked, since the only way that two contextless schemas in the root set have the same next schema in the chain is if their results are identical, and we just assumed that this cannot be. Therefore, every schema in the root set that has any next schema will eventually wind up pointing at such a next schema.

While extending the chain is depth-first, we also go breadth-first in finding other schemas that might extend the chain from this schema for the case where this schema's result might chain to more than one other schema at this level (e.g., if the current schema is A/FOO/B, then valid next schemas are both B/BAR/C and B/BAZ/D, where there are two different actions leading from the same context to [same or different] results [C might or might not equal D]).

When we extend a chain from schema A to schema B, we push B onto a list maintained by A. At the end of this mark-sweep process, each schema contains a list of all other schemas that can possibly be reached, by any path, from that schema. This constitutes the *reachability information*. This is a relatively expensive operation; it can consume several minutes on a fast Lisp processor.[1]

Note that the root set does *not* need to be regenerated when performing or recovering from a lobotomy; this is explained further in Section B.3 below. This means that it may be generated once for some particular set of schemas, and never regenerated unless schemas are added to the knowledge base (schemas can never be subtracted).

The reachability results generated display interesting properties. In particular, there exist "islands" of connections, such that any schema in an island is reachable (via some, possibly quite convoluted and low-reliability) path from any other schema in the island. (This is all the more interesting when one considers that schema chaining generates a digraph, but islands of connections are generally connected in both directions.) As one increases the number of schemas in the knowledge base, the initially large number of small islands decreases, as the average size of each island increases. A knowledge base of 3000-4000 schemas typically displays on the order of half a dozen such islands, which among them completely partition the space.

B.3 Cached (lobotomized) reachable schemas

Given the above reachability information, we can relatively quickly compute a related concept, namely those schemas reachable from some given schema *when the knowledge base has been lobotomized*. For some schema still in the knowledge base, we can compute this by doing a mark-sweep along the schema's precomputed reachable schemas, immediately abandoning any path which mentions a schema in a lobotomized section.

In practice, we do not do this exhaustively for every lobotomy; instead, we maintain a hash table which caches lookups for any given schema. If some caller wishes the reachability information for a schema, it retrieves the cached information if present; otherwise, it computes it as described above and caches the result.

1. E.g., a Symbolics MacIvory Model 3.

Obviously, this cache must be flushed if the reachability criteria (e.g., the extent of the lobotomy) changes.

B.4 Computing paths from one arbitrary schema to another

The above cached reachability information now makes it possible to (relatively) quickly compute possible paths between any pair of schemas. In general, the number of possible paths between any given pair of schemas is quite large (perhaps millions in large runs); it would be helpful if we could quickly find just one path that is very likely to be the "best" we could do, especially since we must find such a path for *every* possible pair of schemas (e.g., hundreds of millions or billions of possible paths if looking at the full cross product).

Note that this is subtly different from the description of chaining (Section 4.3.2.4.1, on page 82) and path metrics (Section 4.3.2.4.2, on page 83) in the description of planning in Section 4.3.2.4. There, we were talking about picking a best path from amongst the thousands of possible paths from pairs of schemas the INITIAL and FINAL sets. Here, we are talking about picking a best path from amongst the millions of possible paths between some arbitrary pair of schemas. To put it another way, in Section 4.3.1.1, we were given (relatively) small sets of schemas—perhaps 50 to 100 schemas in each of INITIAL and FINAL. We had to find *one* path from *some* schema in INITIAL to *some* schema in FINAL; we were free to choose both ends of the path, plus the path itself, subject to the constraints of which schemas were in which set and the dictates of the path metric. Here, must find, *for each possible pair of schemas in the entire knowledge base*, some "best" path connecting that pair.

Thus, the problem here is as follows. We must compute, for the full cross product of every schema (call it A) to every other schema (call it B), what path to cache for the route

from A to B. Several approaches to the problem of computing a path from A to B are described below; any such approach would then have to be run several thousand or more times to generate the complete set of paths (e.g., a 3600-schema path could conceivably have 3600^2 or about 13 million paths; in practice, the graph is never that fully connected).

There are several approaches which are clear losers, as can be predicted from the AI literature on search problems (see, e.g., [Norvig 92], to pick a nice example). Depth-first search, for example, tends to favor long paths, which is a disaster in this system: the longer the path, the lower its expected reliability. A typical search in a large run might lead to a path several *hundred* actions long, when the "best" path (determined, say, by the algorithm described in Case 4 in Section 4.3.2.4.2, on page 83) might be only one or two actions.

Using A* search is an attractive possibility. Unfortunately, it is far too slow. Because we do not have a heuristic function available to guide the search, it takes far too long[2] to search the space, and conses enormous amounts intermediate garbage (in the form of path data structures and lists of them) in the process.

It turns out that using iterative deepening is in fact a very efficient way to generate short, useful paths in this case. This is where the cached reachability information becomes critically important: if we are guaranteed that *some* path exists between two schemas, then iterative deepening is guaranteed to find it. (And, in practice, the path found is generally half a dozen actions or less.) However, if *no* such path exists, then iterative deepening will run forever. Specifying an arbitrary upper limit on the length of a path may unnecessarily deprive us of a good path that is, e.g, just one action past our limit, but using reachability information, we may confidently set the upper limit at (effectively) infinity without worrying about failing to terminate.

2. Minutes to an hour or more on the aforementioned processor.

Using iterative deepening in this way, we stop computing the path between any given pair of schemas when we find the *first* one. Because of the way in which iterative deepening works, this path is guaranteed to be as short or shorter than any other possible path (if it were not, we would have found some different path at a shallower search depth). Rather than enumerate all such possible paths of this length, we simply assume that the first one we find is "good enough." In practice, this is perfectly reasonable behavior.

B.5 Computing a path between the INITIAL and FINAL sets

Given some combination of evaluation parameters as described in Section 4.3.2.9, on page 92 (e.g., the knowledge base of schemas to be evaluated, the extent of any lobotomies, and the path metric in use), we can build a cache of known-best paths from any pair of schemas (one in INITIAL, one in FINAL) that we have investigated. (This cache must be flushed if the evaluation parameters change.)

This works as follows. To recap the discussion in Section 4.3.2.5.4, on page 89, in the naive (e.g., no-cache) case, when planning a path to reach a goal, we must compute, for each possible pair of schemas in INITIAL and FINAL, the path from the selected schema (call it A) in INITIAL to the selected schema in FINAL (call it B). (The computation is performed, as it was in Section B.4 above, in an iterative depth-first manner.) Once we have computed all such paths, we then run the path metric on each such path, and pick the path with the best merit.

The iterative depth-first computation of the path from A to B will never change, however, as long as the evaluation parameters remain constant. We can therefore cache this work, so long as we are careful to flush the cache if these parameters change. Hence, the revised algorithm, before computing the path from A to B using iterative deepening, instead

checks the cache, and uses the cached path if it exists. Otherwise, it computes the path, and caches it.[3]

Using the cache in this way vastly speeds up computation of candidate paths. When the evaluation parameters are first changed (and the cache therefore flushed), the very first attempt at path planning generates several thousand to several tens of thousands of cached plans between different pairs of schemas. The next attempt at planning (after an action has been taken) generates a much smaller number of new entries in the cache, since many schemas that were in INITIAL and FINAL in the previous step may still be there (since only a few sensory bits in the world are likely to change at any given step). Empirically, the cache tends to grow asymptotically to about 5% of the total number of possible cross-products from all schemas to all other schemas, and does so in a small number (10 to 20) of actions executed. This is true because many schemas never wind up in INITIAL or FINAL in any given evaluation.

Hence, by caching the results of path lookup, we eliminate redundant calls to the iterative deepening algorithm. By computing the path between any given pair of schemas only on demand, instead of computing the entire cross-product, we decrease the required size of the cache and the effort of computing all those paths by a factor of 20 or so.

B.6 The promise of randomized algorithms

Despite the care with which the algorithms described above attempt to avoid combinatorial explosion and minimize redundant computation, they are still just barely in the bounds of reasonability for runs of several thousand schemas. On current hardware, generating the complete set of cached reachable schemas for a run of 3000-4000 schemas takes

3. The cache is implemented as a simple hash table, keyed by a number unique to each possible ordered pair of schema numbers. (For example, if *x* and *y* are the two schema numbers, a possible hash key could be *xn+y*, so long as *n>y*.) Keying by a number eliminates consing and allows use of an EQL hash table, which is extremely quick.\

most of an hour. Generating the majority of the cached INITIAL/FINAL paths for a run of that size can easily take another hour or more. Once several possible goals have been run in the same evaluation cycle, the results of the caching start to pay off, but planning is still somewhat slow, averaging a few seconds to half a minute or more (depending on the novelty of the situation, e.g, how many initial/final pairs were already cached) per plan generated.

It seems clear that, short of parallel implementation (which can at least pull one degree out of a polynomial blowup, at least until one runs out of processors), a better solution may to make use of *randomized algorithms* such as so-called *Monte Carlo* algorithms. This was not investigated in this research, but the significant speedups that randomized algorithms can offer in quickly finding a close-to-optimal path is probably well worth the small probability of not finding the truly optimal path. Given the somewhat ad-hoc nature of both the existing path metrics and the random nature of exactly which schemas have been created at any point in time, it is unlikely that a well-written randomized algorithm would noticeably degrade accuracy in path formation, and is likely to be orders of magnitude faster. This is an area deserving of future research.