Scribe notes from Feb. 13, 2002: Graphplan

Amy Davis

February 15, 2002

1 Beginning Discussion - Random Problems

Even though they are not considered representative of real-world problems by most of the AI crowd, randomly generated problems are still required to demonstrate empirical evidence for new algorithms. David Poole argues that the real world has much more structure than perhaps one would dare introduce into a random problem. For more details on the David Poole statement, and others in Tobi Walsh's workshop on empirical AI, see the email sent from Choueiry to the class, and a paper by Paul Maurus and Sam Mondino.

A recent development in this field is the idea of structured random CSPs. (Kautz had something in IJCAI 2001 about it.) These include quasi-group completion. Another recent development is the idea that at least one solution should be preserved even in very dense, very tight problems. Problems in this range tend to have no solutions when they are randomly-generated. Xu Lin wrote a random generator for STPs that guarantees that at least one solution exists.

An STP is a simple temporal problem—a type of a CSP. The variables are points in time, and the constraints are intervals, defining the maximal and minimal distance between the points. A generalization of the TSP is the TCSP, which allows us to define a constraint as a disjuction of intervales—therefore, one can give two or more possible intervals. This allows much more flexibility in to the problem.

Xu Lin guarantees a solution in the STP by first placing each of the time points on a time line, and noting their distances. Then, with each constraint placed, the desired interval is some interval around their actual distance (for example, when distance is d, the interval may be $[\frac{d}{2}, 2d]$. He creates the constraints to the required density, and introduces an added degree of randomness by then randomly moving the constraints within the CSP.

In order for an STP to be solvable, we don't need path consistency. This is because the composition of constraints distributes over intersection[1] (Montanari'74) and the Floyd Warshall algorithm (from 310) guarantees that the network is minimal and decomposable, and also detect whether the network is not solvable.

2 Comments on GraphPlan

The paper had fewer typos than the Knoblock paper, however, the two papers (from IJCAI'95 and AIJ'97) are nearly identical. (Differences include the future work section, the monkey problem description and results, and hacks to make Graphplan more efficient.) Note in the copied version of the paper, Figure 3 has a truncated scale. It should read 5-10-15-20-etc. We should have discussed GraphPlan before discussing SATPlan. It would have made much of SATPlan easier to understand.

Section 4.1 tested Graphplan versus Prodigy and UCPOP, and reported that Graphplan was much better, even though they were written in different coding languages (LISP versus C), and run on different architectures. The runtime comparisons therefore may not be very reliable, and make the advantages less credible

[1] In class, we said that the property that guarantees this is the convexity of the constraints. This is incorrect.

in a real-life situation. Because all three were developed at the same university, it would have been likely that they could get the code to the others, and recompile so as to run on similar architecture. However, even with the less valid runtimes, the curves shown by GraphPlan are definitely better than those by UCPOP or Prodigy. Furthermore, a difference of more than 10 minutes to a few seconds is significant regardless of the dissimilar architecture.

Graphplan always returns the *shortest* possible plan, which can hurt the planner (because a longer plan may be easier to find), or waste time (if the shortest plan isn't desired), but does not produce plans of worse quality. This is because the shortest plan has the greatest level of parallelism, leaving the maximum number of serialisation of options for the one executing the plan. Returning the shortest possible plan also allows GraphPlan to avoid cycles.

The paper states that the ordering of the goals does not affect GraphPlan. We know that in backtracking search, the order of the variables does make a big difference. GraphPlan avoids the effects of variable (goal) ordering by performing an iterative-deepening breadth-first search. This sort of search doesn't affect completeness or optimality, and in practice backtracks less than a backtracking depth-first search does. It could be improved with look-ahead techniques, such as forward-checking. In the case that there is no solution at a particular level, then the ordering of the goals in that ordering has no effect. They will all be tried, and fail. This is stated in the paper after Theorem 2, on page 10. In the case where there are solutions, the savings that may be gained from a "good" goal-orderings are likely small.

Graphplan has a few techniques to avoid searching at every level for a solution. First, it performs no searches until it has determined that the goal is reachable. That is, when all the members of goal state are present at a given level, then we start identifying and propagating constraints of mutual exclusion (mutex). If any of the goals are found to be mutually exclusive, the search halts—the problem is unsolvable. Finally, when the literals that are present in the levels "level-off", that is, where there appear no more new literals in a level, then the problem is unsolvable. With these techniques Graphplan guarantees termination.

A level-off occurs when two consecutive proposition levels have exactly the same proposition. In this case, the effects of any action are already present at this state-level, so any action that is applied will produce no new effects. Therefore, we may as well just apply a no-op to every literal. Cycles are avoided because all possible values for the variables are included as separate literals (for example a and $\neg a$ would both be listed at a state, if they could both be reached by previously applied actions. Applying an action that negated a would not make a disappear, just put a $delete$ edge to it. Therefore, level-off is not susceptible to cycles. (Neither is graphplan as a whole, because it finds the shortest solution).

The complexity of building a graph in GraphPlan is linear in both time and space. However, searching for a solution remains exponential. More specifically, there are three phases that GraphPlan goes through.

1. Build L levels, until all the conditions in the goal are present. In order for an action to apply at level i, all the preconditions for that action must exist in the state level i. When building the levels, first check—are all the goals met? If no, continue building, otherwise proceed to propagate the mutex constraints. If the literals level off, stop—there's no solution.

2. Impose and propagate mutex constraints. Note that GraphPlan finds *most* mutex constraints, not all of them. Finding all mutex constraints is equivalent to solving a maximal cliques problem—it's not even approximable. The paper shows how it is equivalent to solving the problem. GraphPlan first assigns mutex between states that are obviously mutex—a and $\neg a$, for example. Then, it constrains actions that interfere with each other—where one deletes a precondition of another action at the same level. It also constrains actions that are competing, that is, they request preconditions that are mutually exclusive. Finally, two propositions, a and b are mutex if every to achieve a is mutually exclusive to every way to achieve b. Mutex constraints such as these are pure disjunctions and don't propagate well. During search, they only propagate when one of the variables has only one remaining value.

David Smith is quite fond on this sort of mutex. If any of the goals are found to be mutually exclusive, stop—there's no solution.

3. Backtrack search for a plan. If one is found, return it, if none is found, then add some more levels, and continue the process. This search is performed backwards from the goal, through the actions, and recursively generates solutions. Solutions are not found and returned until this point.

GraphPlan was not the first planner to use time stamping. In situation calculus, states and actions can both be associated with time stamps. Both planners consider time in discrete units (not continuous), and provide "snapshots" of time. Graphplan differs from situation calculus by allowing actions to happen in parallel. During the search for a solution, the search takes as many possible actions at once (Actions that occur on the same action level and are not mutually exclusive can potentially be taken in parallel). We determine that parallel actions are optimal by the length of the solution (in number of steps to the plan). The planner looks for only one solution, and different orderings may produce different solutions.

Part of what allows GraphPlan efficiency is memoizing. Graphplan keeps track of combinations that fail and will always fail. Therefore by memoizing—storing the information for use later—we have a readily accessible set of "no-goods". The space for storing the no-good sets trades off with the time of re-discovering the no-goods.

If backtracking is costing too much time in GraphPlan, one may limit the number of backtracks taken and force it to expand to another level, rather than making every search exhaustive.

GraphPlan, or a similar algorithm may be able to be employed in code verification to parallelize code. This (with finite state machines) is a similar problem domain to SAT planning.

GraphPlan's effort and space are actually expended during search. The graph causes no space problems, just searching through the graph. Kautz showed the failure points of GraphPlan in his paper. Exploitation of symmetries may improve GraphPlans performance. Derek Long and Maria Fox [2] have somewhat explored this domain, but it may benefit from interchangeabilities, dynamic symmetries, (or Markov Chains). Every step of GraphPlan is a CSP—each step tries to reach the goals of that step with actions from the previous step. In order to understand the efforts on symmetries in GraphPlan, we need to understand how GraphPlan can be formulated as a CSP. We don't desire to find all symmetries, or all interchangeabilities—this would be equivalent to graph isomorphism [1]. The detection of some symmetries is sufficient to improve the performance of search.

Next week we will continue to discuss this GraphPlan paper, moving to GraphPlan formulated as a CSP on Wednesday.

References

[1] James M. Crawford. A Theoretical Analysis of Reasoning by Symmetry in First-Order Logic (Extended Abstract). In *Working Notes of the Workshop on Tractable Reasoning, AAAI-92*, San Jose, CA, 1992.

[2] Maria Fox and Derek Long. The Detection and Exploitation of Symmetry in TIM. In *Proc. of the 16 th IJCAI*, pages 956–961, Stockholm, Sweden, 1999.