Notes from the paper *Recent advances in AI Planning by Daniel S Weld*

This paper, written in 1999, has a nice summary of major developments in AI Planning till that point. I have excerpted three developments which I felt were major milestones in the field

GraphPlan

The GraphPlan algorithm came out in 1997 and has had a huge impact on the field for two main reasons.

1. First, it solved problems upto an order of magnitude faster than the other state of the art planning systems existing at that time (like PRODIGY, SNLP and UCPOP)
2. Second, it proved to be a natural stepping stone to SAT based planning approaches (which we also discuss in this excerpt)

The planning graph is a way of representing a planning problem such that a plan can then be extracted from it. It has two kinds of nodes - proposition nodes that represent the ground literals pertaining to the state of the world and actions that represent transformations in the world. These two kinds of nodes are laid out in alternate levels. Level 0 has proposition nodes pertaining to the preconditions of the problem. Each odd level starting with Level 1 has actions that are applicable to the world in the previous even level. Each even level starting with level 2 has proposition nodes derived from the actions in the previous level. The levels are expanded till the levels *level off* i.e. the same level keeps getting generated further. Special implicit actions and proposition nodes are added to deal with four kinds of situations: *Inconsistent effects*, *Interference*, *Competing Needs*, *Inconsistent Support*

Once the planning graph is constructed, we now try to extract a plan from it to satisfy our objective. The original GraphPlan paper uses a backtracking approach to extract a plan where it starts from the highest level and tries to come up with a plan by picking appropriate actions in the previous levels and backtracking if it hits a dead-end. It is thus seen that the original GraphPlan system is polynomial time during the graph expansion phase and is worst case exponential time during the plan extraction phase.

There has been a rich body of research on GraphPlan since it first came out. We mention a few such strands next.

- Using search based approaches for plan extraction instead of backtracking. This is also the approach presented in the Russell and Norvig textbook. It is possible to come up with some heuristics and then use those heuristics to run a A* search algorithm to discover a plan. Some of the more well known heuristics are *level sum*, *ignore preconditions* and *ignore delete list*
- The graph expansion phase as described above is agnostic of the end goals. So it is possible to waste a lot of time expanding nodes / actions in the

planning graph that eventually have no bearing on the end goal and plan. There is a strand of research known as *regression focussing* that aims to be smarter about how the graph expansion unfolds so that such unrelated expansions are minimized

- Using a richer language.
  - The original GraphPlan paper describes the preconditions and goals in propositional logic CNF form. Since then, research has focussed on using a richer languge like allowing disjunctions and so on.
  - Using Action schemata. This is analogous to usage of parameters in functions in a normal programming language. We see an example of this in the PDDL language used in the Russell and Norvig textbook. In classical propositional logic, there is no notion of quantifying a proposition over several states. An action schemata is a way of doing so (and just to clarify, the system will still use propositional logic and not first order logic). For e.g. consider a world with vehicles and locations. Our task is to drive a vehicle from a given source location to a give destination location. Now in traditional propositional logic, we would construct all the ground fluents of this world by explicitly stating the effect of a drive action for all possible (vehicle,source,destination) tuples. However, it is possible to specify the drive action in abstract using three parameters for the vehicle, source and destination. After then, one can envision instantiating this schema with all possible (vehicle,source,destination) tuples to obtain the fluents pertaining to this world. Using Action schemata like this has some advantages:
    - Better expressivity. This is similar to why we abstract common functionality into functions in programming languages
    - We can now use type theory based approaches to guide and restrict the instantiations. Just as when using types in programming languages, it is possible to determne the set of types that an expression can possible have at any point in the program, similarly, it is possible to come up with a restricted set of ground states for which an action schemata needs to be instantiated at any level in the graph. This potentially reduces the work done during the graph expansion phase of the planning graph

SATPlan

In the initial days of Planning research, the belief was that a system designed specifically for Planning (like GraphPlan above) would outperform generic systems designed to solve boolean satisfiability problems (aka SAT solvers). This was also corroborated by the early systems. However as the state of the art for SAT solvers improved, it was later realized (and is now accepted) that translating a planning problem to a SAT problem and then solving it via a SAT solver is a much better way to approach Planning

The main issue here is how to translate a given Planning problem into a SAT problem. This has been an active area of research and is variously referred to as either *Compiling a Planning problem to SAT* or *encoding a Planning problem as a SAT problem* in the literature.

It is known that the speed of a SAT solver is exponential in the size of the formula being solved. But this is a nuanced notion since the performance of any individual solver depends a lot on what notion of size is being used - number of variables or number of clauses or number of ground literals summed over all the clauses. So, even though we want the compilation to produce a *small* formula, it is hard/impossible to theoretically justify the *best* compiler. We instead have to rely on actual performance on real life problems.

The two main issues to consider during the compilation are:

- How to represent action schemata. The paper talks about four different kinds of representations - *Regular*, *Simply Split*, *Overloaded Split* and *Bitwise* - these basically represent various points in the tradeoff between number of variables and number of clauses in a formula.
- How to confront the so called *Frame problem.* The problem is - How to represent fluents that do not change as a result of an action. The paper explains two main ways to do so - *Classical* frame axioms and *Explanatory* frame axioms.

The paper describes **BlackBox** - a compiler that first uses the normal graph expansion to construct a Planning Graph and then compiles it to a CNF formula and solves it with a SAT solver. In 1999, (the time that the paper was written), this was the considered to be the fastest and best planner.

Now, it is a moot point that without an efficient SAT solver, the above compilation is of no use. The paper describes some of the best known SAT solving techniques. Of particular interest to us are solvers like MODOC that try to incorporate Planning specific heuristics in generic SAT solving algorithms. An example of such a heuristic is that in Planning problems, generally speaking Action fluents are more interesting than the State fluents since the later follow unconditionally from the former. So in the generic DPLL SAT solving algorithm, the choice of the *splitting* variable is constrained to the Action fluents that exist at that time.

Reactive control system in the *Deep Space One* mission

This was one of the most exciting events in the world of AI planning in the late 1990's. NASA's Deep Space One mission incorporated a reactive control system for configuration planning and execution monitoring that incorporated ideas from AI planning research.

The spacecraft's configuration management system must satisfy high level planning goals (for e.g. achieve thrust before orbital insertion) and must incorporate known failures when computing a plan to achieve the goals. Deep space can be a very harsh environment so robust execution in the face of failures is a particularly challenging problem and one that is best approached as an online Planning problem (instead of trying to account for all possible failure combinations upfront).

The paper next describes the three main components of this system. An *execution monitor* interprets sensor readings to determine the current state of the aircraft. A *goal interpreter* determines the set of spacecraft states that achieve the high-level goals and are reachable from the current state and returns the lowest cost such state. Finally, the *incremental replanner* calculates the first action of a plan to reach the state by goal interpretation. Because of the possibility of component failures, it is not advisable to calculate an entire plan to achieve a goal. Instead the plan is recomputed at each step after incorporating inputs from the execution monitor.

The incremental replanner had an interesting insight. It was realized that spacecraft configuration goals are serializable and hence subgoals can be solved independently. (In other words a situation like the *Sussman anomaly* does not usally arise in this domain). This made it possible for the incremental replanning algorithm to be extremely efficient.