Jussi Rintanen, first version June 2010, updated in April 2015. (Also see tutorial at ECAI'14)

A brief overview of AI planning

The *planning* problem in Artificial Intelligence is about the decision making performed by intelligent creatures like robots, humans, or computer programs when trying to achieve some goal. It involves choosing a sequence of actions that will (with a high likelihood) transform the state of the world, step by step, so that it will satisfy the goal. The world is typically viewed to consist of atomic *facts* (state variables), and actions make some facts true and some facts false. In the following we discuss a number of ways of formalizing planning, and show how the planning problem can be solved automatically.

We will only focus on the simplest AI planning problem, characterized by the restriction to *one agent* in a *deterministic* environment that can be *fully observed*. More complex forms of planning can be formalized e.g. in the framework of Marvov decision processes, with uncertainty about the effects of actions and therefore without the possibility to predict the results of a plan with certainty.

The most basic planning problem is one instance of the general s-t reachability problem for succinctly represented transition graphs, which has other important applications in Computer Aided Verification (reachability analysis, model-checking), Intelligent Control, discrete event-systems diagnosis, and so on. All of the methods described below are equally applicable to all of these other problems as well, and many of these methods were initially developed and applied in the context of these other problems.

Further, more realistic planning and other problems can use the basic problem as a subprocedure, or more general problems can be reduced to it. For example, *temporal planning* can often be reduced to the base case directly (Cushing et al. 2007) or the base case can be used as a subprocedure (Rankooh & Ghassem-Sani, 2013)

See short overview of temporal planning (including planning by SMT.)

The methods discussed in this tutorial have strengths in different types of problems.

- Symbolic methods based on BDDs excel in problems with a relatively small number of state variables (up to one or two hundred), with a complex but regular state space.
- Explicit state-space search is generally limited to small state spaces, but the AI planning community has been successfully applying explicit state-space search also to very large state-spaces, when their structure is simple enough to allow useful heuristic distance estimates, and when there are plenty of plans to choose from.
- Methods based on logic and constraints (SAT, constraint programming) are strong on problems with relatively high numbers of state variables and not too long plans, especially when constraints about the structure of the solution plans and the reachable state-space are available.

Models of state transition systems

Most works on planning use a *state-transition system* model (we often just write *transition system*), in which the world/system consists of a (finite or infinite) number of states, and actions/transitions change the current state to a next state. (Exceptions to this are planning with Hierarchical Task Networks (HTN), and planning problems closer to Scheduling, in which the world states are not represented explicitly.)

There are several possible representations for state-transition systems. A state-transition system can be represented as an arc-labeled directed multi-graph. Each *node* of the graph is a *state*, and the arcs represent actions. An "action" is usually something that can be taken in several states, with the same or different effects. All the arcs corresponding to one action are *labeled* with the name of the action. There may be more than one action that moves us from state A to state B, and this means that there are more than one arc from A to B (this is why we said that the graph is a multi-graph.)

Succinct representations of transition systems

In planning (similarly to other areas, including Discrete Event Systems and Computer-Aided Verification), the transition systems are usually not represented explicitly as a graph, simply because those graphs could be far too big. Instead, a *compact* (*succinct*, *factored*) representation is used. These representations are typically based on *state variables* (corresponding to (atomic) facts). A state, instead of being an atomic object, is a *valuation* of a (finite) number of state variables.

If there are state variables A : {1,2,3}, B : {0,1} and C : {0,1}, then there are 3*2*2* states 100, 200, 300, 101, 201, 301, 110, 210, 310, 111, 211, and 311.

If the domain of each state variable is finite, and there are finitely many state variables, then the number of states is finite as well.

When states are represented as valuations of state variables, an action can be represented as a *procedure* or a *program* for changing the values of the state variables by making value assignments to them. Below we explain a couple of possibilities how these procedures can be defined.

Each of these action representations describes a *binary relation* on the set of all states: what are the possible pairs *(s,s')* of current state *s* and its successor state *s'*.

STRIPS

The simplest language used for formalizing actions is the STRIPS language. In STRIPS, the state variables have the domain {0,1} (equivalently {FALSE, TRUE}), and an action consists of three sets of state variables, the PRECONDITION, the ADD= $\{a_1,a_2,...,a_n\}$ list, and the DELETE=$\{d_1,d_2,...,d_m\}$ list (it is assumed that ADD and DELETE don't intersect.)

An action is possible in a state if all the variables in PRECONDITION have the value 1. Taking the action corresponds to executing the following program, consisting of assignment statements only:
a_1 := 1
a_2 := 1
...
a_n := 1
d_1 := 0
d_2 := 0
...
d_m := 0.

All the assignments are instantaneous and take place simultaneously. In STRIPS planning, a *goal* is usually expressed as a set of state variables. A state is a *goal state* if all of the goals have the value 1 in it.

Petri nets

Petri nets are a model of state transition systems in which several transitions may take place simultaneously in the sense that they are independent. Some of the restricted Petri net models (e.g. 1-safe state-transition nets) are equivalent to the propositional STRIPS model.

For Petri nets, the state variables are the *places*. In state-transition nets, each place can hold 0 or more *tokens*. Essentially, a place is a state variable with the set of natural numbers as its domain. The class of 1-safe Petri nets adds the condition that each place can only hold 0 or 1 tokens. This way each state variable has the domain {0,1}, and there will be only a finite number of states.

Transitions (actions) in Petri nets are described slightly differently. Each transition has a set of *predecessor* places and a set of *successor* places. The transition is possible if all predecessor places N have a token (this corresponds to the PRECONDITION in STRIPS). When the transition is *fired*, all the successor places will receive an additional token.

STRIPS actions can be relatively easily translated into Petri nets with the 1-safety property (see Hickmott et al., IJCAI'07), but there are some small complications because in STRIPS the assignment a_1 := 1 has the same result no matter what the old value of a_1 was, whereas in Petri nets if a_1 is a successor place, the number of tokens is *increased by one*, and may become something else than 1 if the place already had one or more tokens.

Petri nets in general cannot be translated into STRIPS, because in general there may be any number of tokens in a place, and consequently an infinite number of states. Petri nets with the 1-safety property however can be translated.

1. C. A. Petri, Kommunikation mit Automaten, Institut für instrumentelle Mathematik, Universität Bonn, Schriften des IIM, No. 2, Germany, 1962.
2. Murata, T., Petri nets: properties, analysis and applications, Proceedings of IEEE, 77(3), pp. 541-580, 1989.

PDDL/ADL

PDDL/ADL is a generalization of STRIPS. Here we describe a propositional variant of PDDL/ADL. (PDDL actually has a Lisp-like syntax, but we don't use it here.)

The differences of PDDL in comparison to STRIPS are

- The PRECONDITION may be an arbitrary *Boolean combination* of atomic facts about the state variables. Atomic facts say something about one state variable, for example a=0 or b=1. A precondition could for example be $(a=1) \vee \neg(b=1 \wedge c=0)$.
- Instead of the *unconditional* assignments represented by ADD and DELETE, the effects may be *conditional*. This means that the effects are of the form, IF condition THEN a := v
  where the condition is a Boolean combination of facts. STRIPS corresponds to PDDL with trivial conditions that are always true (the condition is the constant TRUE).
- Goals may be Boolean combinations of atomic facts (formulas).

PDDL can be reduced to STRIPS, but

- there is NO one to one correspondence between the PDDL actions and the STRIPS actions, because several STRIPS actions may be needed for one PDDL action, and
- the set of STRIPS actions may have a size that is *exponential* in the size of the set of PDDL actions. This is because a PDDL action may need to be reduced to an exponential number of STRIPS actions. For example, a PDDL action with effects

  IF a_1=0 THEN a_1 := 1
  IF a_1=1 THEN a_1 := 0

  IF a_2=0 THEN a_2 := 1
  IF a_2=1 THEN a_2 := 0

  ...

  IF a_n=0 THEN a_n := 1
  IF a_n=1 THEN a_n := 0

  corresponds to 2^n STRIPS actions because the effects of the actions are different in 2^n different classes of states, and for each class of state a different STRIPS action is needed.

Representation of actions in the propositional logic

All possible actions can be represented as formulas in the propositional logic. This includes all STRIPS and PDDL actions, and also further actions that cannot be represented in (deterministic variants) of PDDL. The logic representation of actions is useful for various powerful logic-based search methods which we will be describing later in detail.

Here we restrict to Boolean state variables, and will denote the atomic facts a=0 and a=1 respectively by a and ¬a. In the representation of actions and change in the propositional logic we need two sets of propositional variables: one expressing the values of the state variables in the predecessor state, and one for the successor state. We distinguish between the two by the ' sign so that the old value of x is denoted by the propositional variable x and the new value by the propositional variable x'.

All the above action representations (STRIPS, 1-safe Petri nets, PDDL) can be compactly represented in the propositional logic. Given an action, its logic representation consists of two parts:

1. the precondition formula P (exactly like in PDDL)
2. description of what happens to each state variable, which is a conjunction of formulas, one conjunct for each state variable. Each state variable a_i is represented as an equivalence $F(a_1,a_2,...,a_n) \leftrightarrow a'_i$ where F is a Boolean function on the state variables (a formula).

For STRIPS the functions F are trivial: for a state variable x they are either x, the constant TRUE, or the constant FALSE. This means that in STRIPS either the value of a state variable remains the same, or the variable unconditionally becomes TRUE or FALSE.

The reduction from 1-safe Petri nets to the propositional logic goes through the STRIPS representation.

For PDDL the functions F may be arbitrarily complex. We don't give a systematic mapping from PDDL to the propositional logic here, although the mapping is straightforward. An action consisting of the precondition d and the conditional effect IF a∧b THEN c is represented by the propositional formula $d \wedge (((a \wedge b) \vee c) \leftrightarrow c')$. The formula says that the new value of c will be TRUE if and only if a and b are both true or c was true already.

Explicit state-space search

Explicit state-space search, meaning the generation of states reachable from the initial state one by one, is the earliest and most straightforward method for solving some of the most important problems about transition systems, including model-checking (verification), planning, and others, widely used since at least the 1980s. All the current main techniques for it were fully developed by 1990s, including symmetry and partial order methods, informed search algorithms, and optimal search algorithms (A* already in the 1960s). It is relatively easy to implement efficiently, but, when the number of states is high, its applicability is limited by the necessity to do the search only one state at a time. However, when the number of states is less than some tens of millions, this approach is efficient and can give guarantees of finding solutions in a limited amount of time.

Search algorithms

There are several types of search algorithms that are routinely applied in planning. These include the following.

1. well-known uninformed search algorithms like depth-first search, breadth-first search
2. systematic heuristic search algorithms with optimality guarantees, for example A* and its variants like IDA*, WA*
3. systematic heuristic search algorithms without optimality guarantees, for example the standard "best-first" search algorithm which is like A* but ignores the cost-so-far component of the valuation function
4. incomplete unsystematic search algorithms, most notably stochastic search algorithms

Incomplete search algorithms can be useful for specific problems when good heuristics are available, or as a part of a portfolio of search algorithms (see portfolios below.)

References

1. P. E. Hart, N. J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum-cost paths, IEEE Transactions on System Sciences and Cybernetics, SSC-4(2), pp. 100-107, 1968.

2. J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984.
3. S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach.

Symmetry reduction

Symmetry reduction methods try to decrease the effective search space by recognizing *symmetries* in the state-space graph. In planning, symmetries are typically caused by the *interchangeability* of objects. If there is a plan that involves some interchangable objects A and B, there is a symmetric plan with the roles of A and B interchanged.

The standard works on symmetry reduction for state-space search are from early 1990s, and they are directly applicable to all of the explicit state-space search algorithms.

References

1. Starke, P. H., Reachability analysis of Petri nets using symmetries, Journal of Mathematical Modelling and Simulation in Systems Analysis, 8(4/5), pp. 293-303, 1991.

Partial-order reduction

Partial-order reduction methods try to decrease the number of search steps by recognizing different forms of *independence* of actions/transitions. The basic idea is that if actions/transitions A and B are independent in the sense that they can be taken in either order A;B or B;A, one only needs to consider one of these orderings.

Many of the existing partial-order reduction methods address *deadlock detection*, but there is a simple reduction from the problem of reaching a goal state to the problem of reaching a deadlock state, and therefore these methods are easily applicable to the planning problem, too.

1. Antti Valmari, Stubborn Sets for Reduced State Space Generation, Advances in Petri Nets 1990. 10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, Lecture Notes in Computer Science 483, pp. 491-515, Springer-Verlag, 1991.

Heuristics

There are several ways of systematically (automatically) creating heuristics. The main approaches are different forms of *relaxation*, in which either

- the original problem instance is simplified, and then the simplified problem instance is solved, or
- the problem instance is unmodified, but the definition of a solution is simplified to be efficiently solvable. (In many cases the same thing can be achieved in either of these ways.)

Well-known examples of this are the following.

- Computing the *max*-heuristic of Bonet and Geffner for STRIPS proceeds by ignoring all effects x := 0 and retaining all effects x := 1, and then finding a plan for this simplified problem instance. (This is easy to rephrase more generally for multi-valued and even real-valued variables.)
  This will give a lower bound for the plan length for the original instance, and can be used as a heuristic in a heuristic search algorithm such as A* to find shortest plans.
- Another, general method for deriving heuristics is *pattern databases* (Culberson & Schaeffer, 1998, with application to planning by Edelkamp in 2001). The idea is to relax the original problem instance in alternative ways, by eliminating all state variables except for a small number *n* of them (with *n* typically 10 or less, so that the problem becomes almost trivial), and then solving the simplified problem instances. The length of any of the optimal solutions is a lower bound for the length of the shortest plan of the original problem instance.

- *generalized abstractions* of Dräger et al. (2006), which can be viewed as a generalization of pattern databases that allows abstraction by arbitrary aggregation of states, not only by ignoring some of the state variables (this was called *combine-and-abstract* by Dräger et al., and planning researchers have renamed this to *merge-and-shrink*).

There are methods for deriving composite heuristics from several individual heuristics.

- The simplest one, used in connection with pattern databases, is to take the *maximum* of different heuristics. Since all of the constituent heuristics are lower bounds for the true shortest plan length, their maximum is a lower bound as well.
- There are variations of the "taking the maximum" scheme that try to be more accurate: when two heuristics respectively give lower bounds *n* and *m*, and their calculation (e.g. through the max-heuristics or pattern databases) involved completely disjoint sets of actions, then also the *sum* of the heuristics is still a lower bound for the length of the shortest plan of the original plan. The use of this strategy is more complicated, because it may be difficult to find a partition of the original planning problem in a way that the disjointness property can be guaranteed in a useful way.

The above discussion is about *admissible* heuristics, which represent true lower bounds for the actual shortest plan length. Admissible heuristics are needed when trying to find *optimal* plans with algorithms such as A*. When optimality is not required, heuristics don't have to be admissible, and then basically anything goes. Work on finding good non-admissible heuristics is typically driven by their performance w.r.t. standard benchmark problems, as they don't have to satisfy any special properties.

References

1. B. Bonet and H. Geffner, Planning as Heuristic Search, Artificial Intelligence Journal, 129(1-2), pp. 5-33, 2001.
2. K. Dräger, B. Finkbeiner and A. Podelski, Directed Model Checking with Distance-Preserving Abstractions, in Model Checking Software, LNCS 3925, pages 19-34, 2006.
3. J. C. Culberson and J. Schaeffer, Pattern Databases, Computational Intelligence, 14(4), pp. 318-334, 1998.
4. S. Edelkamp, Planning with Pattern Databases, Proceedings of the 6th European Conference on Planning (ECP-01), unpublished.

Planning by SAT and constraint satisfaction

Many transition systems have too many states to consider them explicitly one by one, and then factored representations that allow representing large numbers of states and state sequences may be a more efficient alternative. Such representations and search methods are known as *symbolic* or *factored*. The earliest symbolic search methods were based on Binary Decision Diagrams (BDDs, discussed below), pursued in state-space research since late 1980s. The currently most scalable method (since the late 1990s) is based on reduction to the propositional satisfiability problem SAT.

SAT methods are less prone to excessive memory consumption than BDDs. Unlike explicit state-space search, their memory consumption can be (far) less than linear in the number of visited (stored) states, and they can be implemented with a memory consumption that is linear in the length of a plan or transition sequence (as opposed to the in general exponential memory consumption of explicit state-space search.) However, the currently leading algorithms for the SAT problem gain much of their efficiency by consuming more memory than theoretically optimal algorithms, and in practice consume as much memory as explicit state-space search.

All the main improvements to explicit state-space search (symmetry reduction, partial-order reduction, heuristics) have counterparts in the SAT setting, for example in the form of *symmetry-breaking constraints* and *parallel or partially-ordered plans*. We will not be discussing these in more detail here.

(For a representative planning system (including downloads), see this page.)

SAT algorithms

For references to the technology underlying the current generation of efficient SAT algorithms see the following.

References

1. D. Mitchell, A SAT Solver Primer, EATCS Bulletin, 85, pp. 112-133, February 2005.
2. P. Beame, H. Kautz, and A. Sabharwal, Towards Understanding and Harnessing the Potential of Clause Learning, Journal of AI Research, 22, pp. 319-351, 2004.

Reduction to the SAT problem

Plans can be found by solving a satisfiability (SAT) problem, or solving a constraint satisfaction problem. We describe the SAT case; CSP and SAT frameworks are very much similar, and the ideas expressed in terms of SAT are directly applicable in many other constraint-based formalisms, including Constraint Programming, CSP, Integer Programming.

The idea is to have formulas P_T (where T is a positive integer) such that P_T is satisfiable if and only if there is a plan with a horizon 0,1,...,T. Depending on the definition of plans, at each time point 0,1,...,T-1 there may be one or more actions.

The formulas P_T can be constructed from the logic representation of actions we described earlier. Instead of using the propositional variables $A \cup A'$ where $A = \{ a_1,..., a_n \}$ and $A' = \{ a'_1,..., a'_n \}$ for the values of the state variables in the current and in the next time point, we have the propositional variables $A@i = \{ a_1@i,..., a_n@i \}$ for different time points $0 \leq i \leq T$.

Given a formula F for some action expressed in terms of the variables in $A \cup A'$, we obtain F@i by replacing all these variables by $a_1@i,..., a_n@i, a_1@(i+1),..., a_n@(i+1)$.

When we have formulas $F_1,...,F_n$ representing n different actions, we can represent the *transition relation* between states at time *i* as the formula $R@i = F_1@i \vee F_2@i \vee ... \vee F_n@i$.

Now a *sequence* of T actions can be represented as $R@0 \wedge R@1 \wedge ... \wedge R@(T-1)$.

Finally,finding a plan (of length T) from any state satisfying a formula I to any state satisfying formula G can be expressed as the formula $P_T = I@0 \wedge R@0 \wedge R@1 \wedge ... \wedge R@(T-1) \wedge G@T$, where I@0 and G@T are obtained from I and G by replacing each state variable a in these formulas respectively by a@0 and a@T.

The time it takes to test the satisfiability of the formulas P_T strongly depends on the number of propositional variables in the formulas, which, for a given planning problem, is determined by the parameter T. Most works that improve the efficiency of the satisfiability tests do this by decreasing T by allowing more than one action at each time point. See the references and my planning as satisfiability page for more details, including high-performance implementations.

References

1. H. Kautz and B. Selman, Pushing the envelope: planning, propositional logic, and stochastic search, AAAI'96, pp. 1194-1201, AAAI Press, 1996.
2. J. Rintanen, K. Heljanko and I. Niemelä, Planning as Satisfiability: parallel plans and algorithms for plan search, *Artificial Intelligence*, 170(12-13), pages 1031-1080, 2006.
3. J. Rintanen. Planning as satisfiability: heuristics, Artificial Intelligence Journal, 2012.
4. J. Rintanen, *Planning and SAT*, in A. Biere, H. van Maaren, M. Heule and Toby Walsh, Eds., Handbook of Satisfiability, pp. 483-504, IOS Press, 2009.
5. J. Rintanen. Planning with SAT, admissible heuristics and A*. In *Proceedings of the International Joint Conference on Artificial Intelligence*, AAAI Press, pages 2015-2020, 2011. (© 2011 American Association for Artificial Intelligence. AAAI)
6. Jussi Rintanen. Engineering efficient planners with SAT, In *ECAI 2012. Proceedings of the 20th European Conference on Artificial Intelligence*, IOS Press, 2012.

Invariants (state constraints, mutexes)

Explicitly representing interdependencies of state variables is often critical for efficient planning with SAT, and without them a SAT solver will be spending a lot of effort inferring the implicit interdependencies. Any dependency that holds in all reachable states is an *invariant* of the problem instance (a given initial state and actions). The most primitive non-trivial invariant has the form $\neg a \vee \neg b$, saying that *a* and *b* cannot be true simultaneously.

The first method that produced invariants for pruning the search space was the planning graph construction of GraphPlan (Blum and Furst, 1997). The invariants were a by-product of an approximate reachability computation, with invariants obtained as the fixpoint of the computation. Later algorithms have first made the computation more explicit (Rintanen KR'98) and then generalized (Rintanen, ECAI'08) to much more general forms of actions. The general fixpoint construction, though conceptually simple and natural, can be moderately expensive to implement, and many implementations have used simpler methods that find specific classes of invariants more efficiently.

References

1. A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. Artificial Intelligence Journal 90(1-2), pages 281-300, 1997.
2. J. Rintanen. A planning algorithm not based on directional search. in *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*, A. G. Cohn, L. K. Schubert, and S. C. Shapiro, eds., pages 617-624, Trento, Italy, June 1998. Morgan Kaufmann Publishers, San Francisco, California.
3. J. Rintanen. Regression for classical and nondeterministic planning. In Malik Ghallab, Constantine D. Spyropoulos, and Nikos Fakotakis, editors, *ECAI 2008. Proceedings of the 18th European Conference on Artificial Intelligence*. pages 568-571, IOS Press, 2008.

Planning by "symbolic" search methods such as Binary Decision Diagrams

Symbolic search methods here means the use of Binary Decision Diagrams (BDD) and similar normal forms (such as DNNF) for propositional formulas for representing sets of states and transition relations compactly.

"Symbolic" planning uses exactly the same formulas we used above with SAT-based planning. The difference is that instead of constructing the formulas $P_T = I@0 \wedge R@0 \wedge R@1 \wedge \ldots \wedge R@(T-1) \wedge G@T$ for different values of T>0 and testing their satisfiability, we incrementally and implicitly compute from the formulas

```
I@0
I@0∧R@0
I@0∧R@0∧R@1
I@0∧R@0∧R@1∧R@2
```

the sets of states that are respectively reachable by 0, 1, 2 or more actions, and see if some of the goal states are reachable. Instead of the timed variables a@i we only use the variables a and a' for the current and the next state.

I represents the initial state(s). Let $R = F_1 \vee F_2 \vee \ldots \vee F_n$ where the F_i formulas are as defined earlier. Now $I \wedge R$ represents the set S of state pairs (s,s') where state s' follows s when one of the actions is taken and s is an initial state. To compute the set { s' | (s,s') ∈S } we eliminate from $I \wedge R$ all propositional variables in A. For this we use the *existential abstraction* operation defined by $\exists x.F = F[0/x] \vee F[1/x]$. For sets $X = \{ x_1,\ldots,x_k \}$ of propositional variables we define $\exists X.F = \exists x_1.\exists x_2.\ldots.\exists x_k.F$.

We also define a renaming operation: F[A'/A] means that all occurrences of variables in A' are replaced by the corresponding variable in A (for example a' is replaced by a.)

Now sets of states reachable by 0,1,2 and more steps are represented by the following formulas.

```
I
(∃A.(I∧R))[A'/A]
(∃A.(((∃A.(I∧R))[A'/A])∧R))[A'/A]
(∃A.((∃A.(((∃A.(I∧R))[A'/A])∧R))[A'/A])∧R)[A'/A]
```

This can be more clearly written as a small program which computes the sets of states reachable with 0,1,2 and more steps until a set that intersects the goals G is reached.

```
states[0] := I;
t := 0;
while states[t]∧G is unsatisfiable do
 t := t+1;
 states[t] := (∃A.(states[t-1]∧R))[A'/A];
end while
```

The above algorithm terminates at iteration t iff t is the length of the shortest action sequence from a state in I to a state in G. At this point a plan can be output by the following algorithm.

```
goal := any valuation that satisfies states[t]∧G;
while goal does not equal I do
  i := any action index i such that states[t]∧F_i↓(goal[A/A']) is satisfiable;
  output action i;
  goal := any valuation (on A) that satisfies states[t]∧F_i↓(goal[A/A']);
end while
```

Above, goal[A/A'] is a valuation like goal, except that it is defined on A' instead of A, that is. goal[A/A'](a') = goal(a) for all a∈A. And R↓v for a valuation v on A is a formula obtained from R by replacing each occurrence of a∈A by TRUE if v(a)=true and by FALSE if v(a)=false.

The scalability of BDD-based traversal methods can be dramatically improved by doing away with the optimality of the solution length. The *saturation* method of Ciardo et al. (2001, 2007) replaces the breadth-first style search with a far more efficient traversal order.

References

1. Bryant, Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams}, ACM Computing Surveys, 24(3), 1992.
2. Adnan Darwiche, Decomposable negation normal form, Journal of the ACM, 48(4), 2001. *This paper proposed another normal form that is often more compact than BDDs.*
3. Coudert, Berthet and Madre, Verification of Synchronous Sequential Machines Based on Symbolic Execution}, in *Automatic Verification Methods for Finite State Systems*, LNCS 407, Springer-Verlag, 1990.
   This paper and the next are the first ones to use BDDs for solving state-space reachability problems.
4. Burch, Clarke, Long, MacMillan and Dill, Symbolic Model Checking for Sequential Circuit Verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13(4), 1994.
5. Yu, Andy Jinqing, Gianfranco Ciardo, and Gerald Lüttgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. International Journal on Software Tools for Technology Transfer 11.2 (2009): 117-131.
   Dramatically improved BDD-based state-space search, with scalability sometimes matching SAT-based methods.

Meta-level search strategies

When solving a planning problem, there are some meta-level strategies that can be applied to any type of search algorithm (or their combinations).

Algorithm portfolios

The idea of using a combination of several techniques has already been mentioned in the context of heuristics, where aggregates of two or more unrelated heuristics can be formed. Algorithm portfolios do the same at the highest level of a program that solves a planning problem. The general idea is to utilize the complementarities between different search methods. If the strengths of algorithm 1 are sufficiently complementary to the strengths of algorithm 2, the combination of these two algorithms may be much stronger than either of the components.

A portfolio can be used in different ways.

- **Selection:** Choose one algorithm from the portfolio, based on the properties of the problem instance, and run it.
- **Sequential composition:** Run several algorithms according to a given schedule, for example the first algorithm for *n* seconds or until some termination criterion is satisfied, and if no solution has been found, terminate the run and continue with the next algorithm.
- **Parallel composition:** Run several algorithms in parallel (in one or several CPUs/cores), with the same or different rates, until one of them finds a solution.

The advantage of parallel composition is that it finds a solution quickly if one of the component algorithms finds a solution quickly. Its disadvantage is that all the algorithms are run simultaneously, requiring more memory.

The first planner that extensively used portfolios was Kautz and Selman's Blackbox planner from 1999 (others, with fixed portfolios, include LPG and FF). In the planning area, the construction of portfolios has been more of an art than a science, typically constructing a fixed portfolio by selecting a suitable combination of algorithms that have been experimentally (or trial and error) found to perform well with the target problem set.

References

1. Gomes & Selman, Algorithm portfolio design: theory vs. practice, Uncertainty in AI, 1997.

Further Reading

See Temporal Planning, Temporal Planning by SAT Modulo Theories (SMT) as well as Planning as Satisfiability.

Jussi Rintanen