Further Issues in Planning

Alan Bundy

School of Informatics

(slides courtesy of Robert Wilensky)

Robust Execution of Plans

- Plans are typically executed in a dynamic environment.
- It may be more congenial than we planned for,
  - e.g a cooperative agent may have achieved some subgoals already.
- It may be more uncongenial than we planned for,
  - e.g. a hostile agent may undo some achieved goals.

Triangle Tables

	0	1	2
0	At(Robot,m)	Goto(m,n)	
1	At(k,n) Box(k)	At(Robot,n)	Push(k,n,o)
2			At(Robot,n) At(k,o)

The row to the left of an operator are its preconditions.
The column below an operator are its effects.

Execution of Triangle Tables

- Always start from the end and work back.
- If environment congenial may omit some steps.
- If environment uncongenial may need to repeat some steps.
- Need to constantly monitor environment to update world model.
- Keep triangle tables as macro operators to insert in new plans.

Example of Omitting Steps

- Suppose plan for **At(Box1,C)** is formed in expectation that robot and box are in different places: **At(Robot,A), At(Box1,B)**
- But, in practice, box is in *same* place as robot: **At(Robot,A), At(Box1,A)**
- So plan executed is: **Push(Box1,A,C).**
- Rather than: **Goto(A,B), Push(Box1,B,C).**

Example of Repeating Steps

- Suppose plan for **At(Box1,C)** is formed in expectation that box is in one place: **At(Box1,B).**
- Whereas it is actually at another place: **At(Box1,D)**
- So plan executed is: **Goto(A,B), Goto(B,D), Push(Box1,D,C).**
- Rather than: **Goto(A,B), Push(Box1,B,C).**

Another General Problem That Arises In Planning

- Can we really list all the preconditions for an action?
- E.g., we forgot to mention that **Move** requires that
  - The block not be nailed down.
  - The block not be glued to the table.
  - There not be a force field holding a block in place.
- This is (part of) the '*Qualification Problem'*.

The Qualification Problem

- A Solution: State conditions at right level of generality; inherit (or otherwise reason about) them.
- E.g., state that to move a block, it must not be attached.
  - State that gluing, nailing, etc., are ways of attaching.
- But we still have to check a lot of preconditions!
- Solution: Have some theory of *when* a precondition is worth checking; prove only that the plan will work given the assumptions.
- E.g., plan to turn on light by flipping switch has *lots* of preconditions, most of which we never check.

Toward Subtler Preconditions: An Example

- We'll introduce one simple distinction,
  - between preconditions worth attempting to achieve and those never worth attempting.
- Just this simple change will make our planner enormously faster.
- Of course, more subtle distinctions might be even nicer.

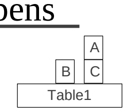
Subtler Preconditions: A Motivating Example

- Consider some plausible operators for stacking blocks.
- Might need several operators, because they have different effects and preconditions:
  - One for each of
    - moving a block *from* on top of a *block to* on top of another *block*
    - Moving a block *from* on top of the *table* to on top of a *block*.
    - moving a block *from* on top of a *block* to on top of the *table*.

Preconditions: Block World Example

- **ReStack(obj,from,to)**; stack a block that is on a block
  - Pre: **On(obj,from) ∧ Clear(obj) ∧ Clear(to) ∧ Block(obj) ∧ Block(from) ∧ Block(to)**
  - Add: **On(obj,to) ∧ Clear(from)**
  - Del: **On(obj,from) ∧ Clear(to)**
- **Stack-from-table(obj,from,to)**; stack a block that is on the table
  - Pre: **On(obj,from) ∧ Clear(obj) ∧ Clear(to) ∧ Block(obj) ∧ Table(from) ∧ Block(to)**
  - Add: **On(obj,to)**
  - Del: **On(obj,from) ∧ Clear(to)**
- **Un-Stack(obj,from,to)**; move a block from on top a block to the table
  - Pre: **On(obj,from) ∧ Clear(obj) ∧ Block(obj) ∧ Table(to) ∧ Block(from)**
  - Add: **On(obj,to) ∧ Clear(from)**
  - Del: **On(obj,from)**

Now Look What Happens



- Suppose goal is **On(A,B)** where
  Block(A) Block(B) Block(C) Table(Table1)
  On(B,Table1) On(C,Table1) On(A,C)
- What operator does the planner choose?
- Intuitively, **ReStack** is the only choice.
- However, *all* the operators seem helpful!
  - I.e., along with **ReStack**, both **Stack-from-table** and **Un-Stack** have **On(obj,to)** on their Add lists.
- Might end up with plans such as
  - **Stack-from-table A** onto **B**, having first done **Un-Stack A** onto **Table1**.
  - **Stack-from-table A** onto **B**, having first turned **C** into a table!
- In fact, the planner will consider a huge number of silly plans, most of which die a natural death; some make bad plans; increases possibility of loops in our search.

Solutions

- Distinguish those precondition literals that are useful to consider changing from those that simply need to be true.
- Call the latter 'filter conditions'.
- E.g., **ReStack(obj,from,to)**
  Filter: **On(obj,from) ∧ Block(from) ∧ Block(to) ∧ Block(obj)**
  Pre:**Clear(obj) ∧ Clear(to)**
  Add: **On(obj,to) ∧ Clear(from)**
  Del:**On(obj,from) ∧ Clear(to)**
- Here we divided preconditions into
  - 'Filter', i.e., literals not worth trying to change
  - 'Pre', i.e., literals that get turned into subgoals if not true, as before.

How Much Does This Help?

- In our planner, without filter conditions, the M&B problem takes about 1 hour to solve
  - on a 1 gigahertz PC.
  - It used 2,713,146,715 cons cells.
  - And it produces a plan with a silly step in it.
- With filter conditions, it is too fast to time.
  - It used 10,100 cons cells.
  - And produced a fine plan.

(Why is the difference *so* great?)

Operator Selection

- This is part of 'operator selection'.
- In general, might have conditions that aren't preconditions here.
  - In general, might know complex conditions under which various operators should be considered.
    - E.g., if my goal is 'satisfy hunger', I consider different actions at different times of the day, where I am, etc.
  - If so, might need to do less planning and more remembering.

Operator Selection and Knowledge

- One might learn these associations from experience.
  - E.g., if turning something into a table always fails, make it a 'Filter' rather than a 'Pre'.
- Maintain 'plan library' of previous plans.
  - Have ways of generalizing actual plans to be applicable to something other than the exact situation they were created for.
  - Have ways of indexing them so they are considered at the right time. E.g., store plans that were expensive to compute.
- How to generalize from experience is an interesting problem!

Yet Another Problem

- Can we really list all the results of an action?
  - E.g., if we move a block, its shadow will move too.
  - This is the '*Ramification Problem*'.
  - With respect to planning, this is the '*plan projection problem*'.
- As with preconditions, we don't want to list all the effects with each operator, but instead, have separate, general facts that allow us to make inferences.
  - E.g., if light source, then shadow attached to object.
  - Moving just changes location of object moved.
  - We reason by inference that shadow is moved too.

Ramifications and Planning

- Note that not listing a ramification of an action complicates operator selection.
  - Before, we just looked for an operator whose ADD list unifies with a subgoal.
  - Now, we would need to consider operators whose ADD list, plus other things, *entails* a subgoal.
- Moreover, if we *add* ramifications to a KB via inference, interesting problems arise.

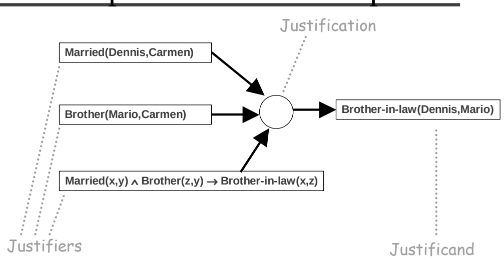
Ramification Example

- Suppose we knew
  - **Married(Dennis,Carmen)**
  - **Brother(Mario,Carmen)**
  - **Brother-in-law(Dennis,Mario)**
- Now suppose there is a **Divorce** operator, which, among other things, removes
  - **Married(Dennis,Carmen)**
- But now **Brother-in-law(Dennis,Mario)** becomes false too.
- Just as a **Marriage** operator couldn't list all the states to add, a **Divorce** operator couldn't explicitly list all the operators to remove.

Solution

- Maintain 'data dependencies'
  - In effect, keep history of how things got in the KB.
  - When a conclusion is no longer justified, withdraw the conclusion.

Data Dependency Representation Example

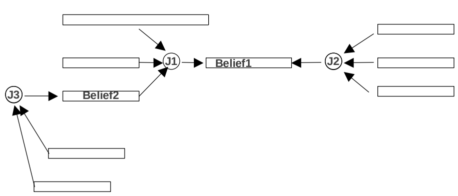


Removing any justifier invalidates the justification.

Complication

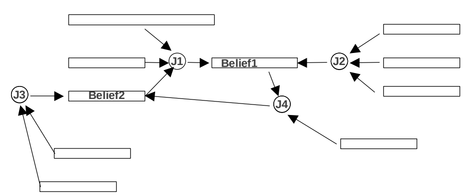
- Can have more than one justification for the same belief.
- Only remove a belief if *all* its justifications are no longer valid,
  - properly handling circular justifications.

Data Dependencies



- Removing a justifier of J3 will invalidate J1 also.
- Can only remove Belief1 when J1 and J2 are no longer valid.

Data Dependencies



- J4 is circular, but legitimate. (E.g., can deduce 'married' again from 'brother-in-law' and 'no sisters'; so, invalidating J1 and J3 doesn't invalidate Belief2.)
- Invalidating J3 doesn't invalidate J1.
- But invalidating J3 and J2 does, as the only justification is circular.

Truth Maintenance

- A system that maintains dependencies between sentences is called a *truth maintenance system*.
- Generally, a TMS doesn't actually remove something from a KB; instead, it marks statements as being 'in' or 'out'.
  - This way, it is relatively easy to recompute implications when assumptions change.
- What we have described is a *justification-based* TMS.
- Some TMSs keep track not of which justifications support which propositions, but which sets of assumptions support which propositions.
  - This '*assumption-based truth maintenance system'* (ATMS) is a finer-grain level of support, which, in effect, maintains at the same time all the situations that have ever been considered.
- Maintaining data dependencies is expensive (NP-hard).
  - People don't do this well either.

Truth Maintenance and Planning

- Whether we need a TMS for planning is related to the nature of how we determine the full consequences of operators.
- If we allow operators to have effects via deduction, then we need a TMS to reason about the changes from one situation to another.
- This is hard, but should be.

Summary

- We can get more robust plan execution by storing them in triangle tables.
- We can improve the performance of STRIPS further by having a more refined idea of preconditions.
  - Or being smart about operator selection generally.
- Reasoning about results can still be tricky.
  - Maintaining data dependencies can help.