CAUSATIVE FORCES IN MULTI-AGENT PLANNING[1]

David R. TRAUM and James F. ALLEN

Computer Science Department
University of Rochester
Rochester, NY 14627
USA

A general purpose reasoning agent may come in contact with many types of external forces which have differing types of effects on the world. Different types of forces require different types of reasoning about them. We present a classification scheme for planning domains, based on differentiating the types of causative forces present along dimensions of the degree of interaction and of how cognitive they are. We present some speculations on how best to reason in these domains. Example problems are illustrated using the ARMTRAK domain [7].

1 Introduction

In traditional domain-independent planning models (e.g. [3], [4], [8] [9] [1]) an important assumption has been that the planner itself is the only cause of any changes in the world (events). While this approach has simplified reasoning about actions, it is obviously inadequate for modeling many realistic domains. In the real world, there are a variety of forces, ranging from natural forces to other intelligent agents, which can cause changes.

For the purposes of this paper, we will assume that events are caused by one or more *causers*. We use this term instead of the term *agent*, since the latter generally implies some amount of volition. In this paper, a *causer* is a force which can change the state of the world, or interact with another causer in changing the world, while the term *agent* is used for a causer which has goals and intentions and acts according to them. Any change in the world is called an event, and an action is an event caused by a causer. Example causers include: people, natural forces (gravity, electricity, wind, waves), and machines.

Different types of forces can be most profitably reasoned about using different methods. There is little point in reasoning about the "goals" of the sun in figuring out if is going to rise at 6:45am tomorrow. Similarly, trying to figure out what a person is

[1]Appears in Y Demazeau and J. P. Muller, editors, *Decentralized A.I. 2.* Elsevier Science Publishers B. V., pages 89-105, 1991.

going to do without taking account of his or her goals is going to be unfruitful in many circumstances.

2 Types of Planning Domains

2.1 Dimensions for categorizing causative forces

There are a number of different features which we could use to classify causers. Some which seem useful are:

Interaction With Other Causers A causer can either influence or not influence the results of the actions of other causers. Conversely, a causer can either be influenced or not influenced by the actions of other causers. Generally, if a causer can neither influence nor be influenced by us we need not consider it in reasoning to satisfy our goals. Similarly, if it can not influence us, we can generally ignore a causer even if we do affect it (though we must be careful here, in that causers which have no short term effects on us may have long term effects).

Goals A causer may have goals and perform actions in an attempt to satisfy those goals, or may be acting (or reacting) unpurposefully.

Awareness An agent may or may not be aware of other causers. Its awareness can range from awareness of the physical capabilities of the other causers to awareness of goal-directed behavior, to actual knowledge of the goal-directed reasoning of the other agents.

Communication An agent may or may not communicate with other agents. Communication may be defined in a Gricean way as an attempt to change the beliefs, goals, or intentions of another agent.

Social relationship This is really a categorization of relationships between agents rather than the capabilities of a single agent. Two Agents may be cooperative, competitive, or neutral with respect to each other. Agents may also have non-equitable relationships with respect to the priority of each others goals; one agent may be subservient to the other to a varying degree.

2.2 Types of Causers a Planner Might Want to Model

Using the above dimensions, we can come up with several different types of causers with which a planner may have to contend.

1. uninfluenced, no goals
   This type of causer represents an inanimate force which can't (within the scope of the planning problem) be affected by actions of the planner. Example: the sun rising – it doesn't matter what the planner does, the sun will still rise and set at

the same time. The sun might affect an agent's plan, however (e.g. a plan to read a newspaper outside at 7:00am).

2. two-way interaction, no goals
   Here there is interaction both ways, but the causer is more a force than an agent with goals. Example: a falling ball – gravity pulls it but it can be stopped or redirected by putting out your hand.

3. goals, no awareness
   This type of causer can be fruitfully seen as having goals which it tries to accomplish by choosing to perform some subset of the possible actions, although it doesn't reason about other agents. Example: traditional planner – it has goals, but it doesn't reason about other agents, just executes plans based on observed state and goal state. (Perhaps insects are like this?)

4. awareness, no communication
   This agent is aware of other agents and reasons about their goals and beliefs in order to figure out what they are going to do so it can plan to achieve its own goals. Still, there is no direct communication or attempt to change their beliefs and goals. Example: game playing – you need to reason about other agents goals, beliefs, and plans, but you don't generally talk to them, you just make the moves based on what you think they are going to do; in a purely competitive zero-sum situation, there is no purpose in communicating, since you would only help your enemy by giving information.

5. communication
   This is really a whole range of types, based on the social relationships between the agents. On one end, the other agent is like a tool, having no goals but what the agent gives it. This is like the distributed AI paradigm, where the primary agent can insure that the subsidiary agents will do what it wants by giving them the right information. On the other end, are independent negotiating agents, where one can't even necessarily trust what the other agent says.

With the above types of causers, we can think of several types of planning environments, classified by which types of other causers exist in the model.

3 A Test Domain: Armtrak

The domain we have chosen in which to try out these classifications and demonstrate their usefulness is that of model railroads. This domain allows the conceptual simplicity of the blocks world, yet allows for a complex range of purposeful behavior. [7] describes the domain and associated research at the University of Rochester more completely.

The Armtrak domain has a number of types of forces, including changing the power on a train, and setting the direction of a track switch. These forces can have different effects based on conditions in the environment (e.g. a train moving or derailing, switches changing states).

Aside from the actual toy trains, there is a simulator, written by Nat Martin and augmented by Steven Feist, which allows rapid prototyping and testing of planners. The Armtrak simulation allows multiple programs running on different machines to issue commands, and thus allows one to have multiple autonomous agents operating in a given domain problem. For this project we are using the simulator.

3.1 Armtrak Primitives

An Armtrak situation is composed of the following basic elements:

- tracks
  There are track types conforming to pieces of rail of various shapes. Each track is connected to other tracks at each end. Tracks include the following special types:
  - buffer – the end of a line, no train may be on it
  - switch – has a connection to one of two different tracks at one side depending on the state of the switch (open or closed)
- trains
  trains are physically on a track, one car per track. Trains are a sequence of one or more connected cars. Trains can be coupled to other trains, both in front and behind, by moving one train into another. As a result of error conditions, a train can leave the track and become derailed, and then be unable to move until put back on the track. An engine is a special type of car which can be given power and can move, towing any other attached cars.

3.2 Armtrak Commands

The Armtrak simulator is given an initial set up consisting of the track layouts and the trains and their locations. Commands can be sent to the simulator from Lisp or C programs from any machines on the local network. Up to 16 independent programs may send signals. The Lisp primitives include:

1. Effectors
   - (set-power engine power) adds *power* to the current power of the engine. An engine will move (if not blocked or derailed) with a speed proportional to the power and inversely proportional to the weight of the train (including cars it is pulling). The direction of travel depends on the sign of the power, positive is forward and negative is backward (with respect to the orientation of the train). If there is no next track to go to (end of the line, or a switch going the wrong way) or if it collides at too high a speed, the train will derail.

- (set-switch switch state) sets the switch to the desired state (open or closed)
- (rerail-train car) puts the train back on the rails

2. Perception Queries

- (get-time) look at the clock
- (get-power engine) returns the power of the train
- (get-switch switch) returns the state of the switch
- (get-location car) returns the name of the track the car is on
- (get-derailed car) returns true if the car is derailed
- (get-track-object track) returns the object on the track, or false if nothing is there

3.3 An Armtrak Situation: Robin Hood Meets Little John

For demonstration purposes we have chosen a very simple Armtrak situation which is still complex enough to require reasoning about the interactions of multiple causers. The set-up is as shown in Figure 1. There are two engines, "Robin Hood" and "Little John", and two switches, allowing two paths through the center. The goal is that Robin Hood must get from Nottingham to Sherwood. An obstacle in achieving this goal is that Little John is in the way. The name comes from the story of Robin Hood's first meeting with Little John, in which they both tried to get over a bridge at the same time, going opposite ways. Instead of making sure that only one of them was on it at a time, or one person going back until the other was across, they ended up fighting until Robin Hood fell in the river. The goal of the current situation is to improve Robin Hood's "plan", so that he can get safely across.

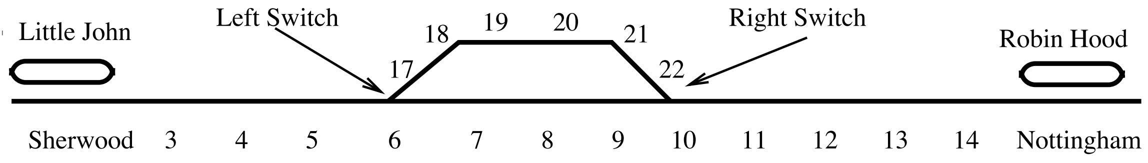

Figure 1: An Armtrak Situation

3.4 Scenarios of the Little John Problem

While keeping the ARMTRAK set up and the goal of Robin Hood constant, we can vary the causers present in the situation to end up with different environments. The primitive actions possible are: setting the states of the two switches and setting the powers of the two trains. We can also have higher level abstractions and combinations of these for plans to move from place to place. By changing the capabilities of the controllers of these actions we can achieve the desired environments. Here is a set of planning problems corresponding to types of planning environments composed of allowing different causers of the types outlined in section 2.2 above:

Level 0 Traditional Planning, No other causative forces
Both trains and the switches are under the planner's control. The planner must reason merely about plan operators, preconditions and effects.

Level 1 Only type 1 causers
The trains are both under the planner's control, but the switches are flipped according to a schedule, they are controlled by type 1 causers (the states of the switches can affect the movement of the trains, but not vice versa). We must also extend the reasoning to also cover events.

Level 2 Only types 1 and 2 causers
Robin is controlled by the planner, but John moves on a fixed schedule. John is controlled by a type 2 causer; the Little John train's movement affects and is affected by the Robin train. Now we actually have to reason about actions and their relationship to events.

Level 3 Type 3 causer
John has a goal to get to Nottingham. There is a simple agent (such as that controlling Robin in Level 0) attempting to get him there. The planner must reason about the beliefs, goals and intentions of the other agent.

Level 4 Type 4 causer
John's controller has a capacity to reason about Robin's plans. This requires reasoning about shared and nested beliefs (John's beliefs about Robin's beliefs about John's beliefs about ...).

Level 5 Type 5 causer
Robin and John talk to each other with a simple language to negotiate a joint plan which will accomplish the goals of each. This will require some notion of speech acts, shared plans, Gricean NN-meaning and M-intentions.

A planning logic and execution architecture is being designed for each of these problems. The rest of this paper will discuss these implementations in detail. At this writing, only the first four scenarios (levels 0, 1, 2, and 3) have been fully implemented.

4 Solving the Little John Problem

4.1 Level 0

For this scenario we can make several simplifying assumptions. Since there are no external causers, we don't need to worry about simultaneous actions or explicit representations of time. The planner is the only causer, so it can execute an action and just wait for it to finish before executing another action. A state based world representation is sufficient, with a new state occurring after each action. We use a plan representation based on STRIPS [3]. A sample plan operator is **move-one-space**, given below:

move-one-space (x:train from-loc:track to-loc:track)
Pre: (at x from-loc)
(adjoined to-loc from-loc)
(clear to-loc)
(¬ (derailed x))

Delete: (at x from-loc)
Add: (at x to-loc)
Decomposition:
(move-1 x (dir-fn x from-loc to-loc))

In the above operator, **at, adjoined,** and **clear** are predicates which can be tested using programs built from armtrak primitives. They have the following meanings: **at** is true if its first argument is at the location of the second argument. **adjoined** is true if its two arguments are adjacent tracks and any switches which connect them are set the proper way so that a train could move from one to the next. **clear** is true if there are no trains on its argument. **dir-fn** is a function which returns the direction of the third argument from the second argument, relative to the orientation of the first argument. **move-1** is a primitive program, below the level of planning here, which sets the power of the train to the appropriate magnitude, waits until it has arrived at the next track or has derailed, and then stops the train. Any of the standard Search or Theorem Proving methods can do for a planner. For an executer, all that is necessary is to take the sequence of plans generated by the planner and execute them sequentially.

A sample plan for solving the level 0 problem is given below, where **move-along-clear-path** is a higher level plan operator which uses repeated calls to **move-one-space** and **align-switches** (a plan operator to set switches the right ways for travel between two adjacent tracks) to move a train along a path of successively adjacent empty tracks.

Level 0 Sample Solution Plan
(move-along-clear-path john '(sherwood 3 4 5 6 17 18))
(move-along-clear-path robin '(nottingham 14 13 12 11 10 9 8 7 6 5 4 3 sherwood))

4.2 Level 1

In this scenario we need to worry about when external events occur, and how they will affect our plans. We must bring in a representation of time and time intervals (for calculating durations of actions and how long particular conditions will hold). We must also add events to our ontology. Events are not selected as operations by the planner, but need to be reasoned about in any plans. Since no actions by other causers can be affected by our actions, we still do not need to reason explicitly about the relationship between actions and events. We can figure out which events will be caused by any actions of other causers and just add those events into the planning structure. As long as we know all the events that will happen, and when, we can maintain an accurate world model. Plans look pretty much the same as in level 0, except that some of the preconditions must be altered to take account of possible events occurring during the action, and a duration field is added to operators which tells how long the plan takes to execute. Many predicates also have a temporal argument, which was unnecessary in level 0. Also, it no longer makes sense to talk about **add** and **delete** lists, because of the temporal arguments. We no longer delete a predicate, merely say that something else is true at the **end** of the action, if the action is performed when the preconditions hold. The **move-one-space** plan above, modified for the new environment is,

move-one-space (x:train from-loc:track to-loc:track)
Pre: (at x from-loc begin)
(adjoined to-loc from-loc [begin,end])
(clear to-loc begin)
(¬ (derailed x begin))
Effects: (at x to-loc end)
Duration: move-one-time
Decomposition:
(move-1 x (dir-fn x from-loc to-loc begin) begin)

begin and **end** are special temporal functions on actions and events which represent the time of commencement and conclusion of the action. [begin,end] represents the interval from **begin** to **end** (ie the entire duration of the action).

A planning system such as DEVISER [10] is adequate to handle this kind of planning environment. We can use a chart based representation for planning, where we first add the events which will be caused by the external causer, and then fill in the actions as they are planned for, using standard techniques to reason about preconditions. For execution, we use a priority-queue type scheduler, which has a sequence of actions to be performed, as well as the times at which they should be performed.

If we started with a schedule of the switching events such as:

time	**event**
1000	(set-switch 6 open)
1250	(set-switch 10 open)
1500	(set-switch 6 closed)
1750	(set-switch 10 closed)
2000	(set-switch 6 open)
2250	(set-switch 10 open)
2500	(set-switch 6 closed)
2750	(set-switch 10 closed)

Then our solution would be to hand the following plan to the scheduler, where **move-along-clear-simple-path** is a compound plan which moves a train along a path with no switches along it, space by space using repeated calls to **move-one-space**:

time	**action**
950	(move-along-clear-simple-path john '(sherwood 3 4 5 6))
1350	(move-one-space john :from 6 :to 17)
1500	(move-along-clear-simple-path robin '(nottingham 14 13 12 11 10))
2050	(move-one-space robin :from 10 :to 9)
2350	(move-along-clear-simple-path robin '(9 8 7))
2600	(move-one-space robin :from 7 :to 6)
2750	(move-along-clear-simple-path robin '(6 5 4 3 sherwood))

4.3 Level 2

This level adds more difficult interactions because both Little John and Robin Hood may act simultaneously, creating a host of possible mutual interactions. We must explicitly encode that we can't be completely sure of what the effects of an action will be unless we know about all the other actions which might affect it. We can no longer rely on the STRIPS assumption to solve the frame problem and must use other techniques.

Implicit in the preconditions for the operators in the previous levels is a world property that no more than one train can be on a particular track at a given time. In the previous levels it could remain implicit, because no other causers could move the trains, and the planner only would if the preconditions held. Here, we must explicitly encode that fact, using a domain axiom like:

$$\forall_{x:train}\forall_{y:train}\forall_{loc:track}\forall_{t:time}\ (\text{at x loc t}) \wedge \text{x} \neq \text{y} \quad \supset \quad \neg\ (\text{at y loc t})$$

Any kind of sound planning formalism becomes much more complicated. For instance, if we tried to just import **move-one-space** we find that the precondition **(clear to-loc begin)** is neither necessary nor sufficient for success of the action. It is not necessary, since there may be a train initially on **to-loc** which is moving in the other direction at the same speed or faster than **x**. It is not sufficient because, even if **to-loc** is clear, there may be another train moving onto **to-loc** from the other direction, so that the two trains will crash, and maybe **x** might not get there. In level 1 we could ignore this problem, since the planner had control of both of the trains and would only move one at a time.

Our solution is to add a predicate **moving** which takes a train, and a direction as arguments. **towards** is a partial function over pairs of tracks which returns the direction in which the second is adjacent to the first. Assuming we have a sequence of tracks: **from-loc, to-loc, next, next+1**, we could write the preconditions for **move-one-space** (x:train from-loc:track to-loc:track) as something like the following, which requires either that **to-loc** is clear and that nothing can move onto it during the action, or that it is not clear but the car there is moving away from train x.

```
(at x from-loc begin)
(adjoined to-loc from-loc [begin,end])
(¬ (derailed x begin))
(or (and (clear to-loc begin)
         (or    (clear next begin)
                (∃ y: (and (at y next begin) (¬ (moving y (towards next to-loc)))))))
    (∃ y: (and  (at y to-loc begin)
                (moving y (towards to-loc next))
                (adjoined to-loc next [begin,end])
                (¬ (derailed y begin))
                (clear next begin))))
```

Even this precondition is slightly too strong. It is fine for a problem such as ours where there are only two trains, but in the general case, we could have trains at next AND next+1, each moving away from to-loc (and associated other conditions to make sure they get there). In general we can have an unbounded number of trains in a row all moving away, and as long as they can get there (nothing in the way at the end or on a collision course) then **x** can move to **to-loc**. The preconditions get even harder to write down (and understand and reason about) as we move to higher level abstractions, such as moving several tracks along a path, where at the time at which we are moving to a particular track that track must be uncontested, but we can say very little about conditions on the whole sequence of tracks.

We need to be more explicit about the relationship between actions and events. For this purpose, we add a type **causer** and a predicate **perform** which takes a causer, an action (or plan operator), and a time as arguments. Now we can talk about events happening if specific conditions hold, and these conditions may include agents performing actions at specified times. We can also now describe motion at a lower level. If we think of movement as an event which doesn't take an action by the agent (once the power has been set), we can model the actual nature of the *move-1* program used above: starting the train, waiting for it to get to its destination, and stopping it. We have an event **move** which takes place whenever its conditions are met (train is moving, tracks are adjoined, and nothing is in the way) and actions **start** and **stop** which will affect whether trains are moving. These are shown below, where **next-track** is a function which returns the track or tracks which are next in its direction argument from its track argument:

start (x:train d:direction)
Pre: ($\neg$ (derailed x begin))
(stopped x begin)
Effects: (moving x d end)
Duration: start-time
Decomposition: (set-power x (dir-fn d))

stop (x:train)
Pre: (moving x ?dir begin)
Effects: (stopped x end)
Duration: start-time
Decomposition: (set-power x $-$(get-power x))

move (x:train from-loc:track to-loc:track)
Conditions: (moving x ?d:direction begin)
(at x from-loc)
(to-loc $\in$ (next-track from-loc ?d))
(adjoined to-loc from-loc [begin,end])
($\neg$ (derailed x begin))
($\neg$ (perform ?c (stop x) [begin,end]))
(clear to-loc begin)
$\forall$loc:track (connected loc to-loc) $\supset$
(at x loc begin) $\vee$ (clear loc begin)
Effects: (at x to-loc end)
Duration: move-one-time

This **move** event description says that whenever a train x is moving in direction d, and is at from-loc and to-loc is adjoined to from-loc in direction d, and x is not derailed and no causer performs a stop action on x, and to-loc is clear, and no other trains aside from x are within one space of to-loc then at move-one-time later, x will be at to-loc. The main formal distinction between the **move** event and actions such as **start** or **move-one-space** are that for an event, the effects will happen whenever the conditions are met, while for an action, the effects will happen only when the action is performed - thus the difference in notation, labeling one **Pre** and the other just **Conditions**.

In previous levels, our plan could only be foiled by incorrect action or acting at an improper time. Now, inaction can be just as dangerous. It is not enough to simply specify the conditions for the event **move** under which a train will successfully move to the next track, we need to know what happens if some of the conditions are not met. If a train is **moving** then it will either go to the next track (if it can), or it will derail. It won't just stay where it is until the conditions to move are fulfilled. If it happens that some preconditions for **move** will not be met when the train is moving, it is important that the planner **stop** the train and **start** it again when the conditions are met, or the train will derail and the whole plan will fail. Our preliminary solution is to give the following domain axiom, which states that when a train is moving throughout an interval of **move-one-time** duration, it either moves to a track which is adjoined and in the proper direction, or it derails:

```
(at x loc1 t) ∧ (moving x d [t,t+move-one-time])    ⊃

    (or   (and  (at x loc2 t+move-one-time)
                (loc2 ∈ (next-track loc d))
                (adjoined loc1 loc2 [t,t+move-one-time]))
          (derailed x t+move-one-time))
```

With such an axiom, if we can't prove that the train will move (say because of another train nearby or a switch going the wrong way), we must **stop** the train and wait for conditions to improve. This will generate an overly conservative world model, since in fact some of the times when the conditions for a **move** event are not met it will move anyway.

For an executer, we generalize the scheduler from Level 1. Instead of just waiting for a particular time, it executes an action whenever a specified set of conditions holds. The scheduler can now deal with more primitive (and quicker) actions, and thus deal with near simultaneous actions. The waiting for a train to get to the new location that before was done as part of the **move-one-space** action is now incorporated into the scheduler by decomposing the action into a **start** action followed by a **stop** action to be executed under the proper conditions (when the train has moved or derailed).

The solution plan for level 2 is given below. Intuitively, we have one high-level plan to follow a path, and other reactive plans to set the switches under the proper conditions to keep john out of his way. We start out with John's controller performing the action (start john (towards sherwood 3)) at time 1000. Then we construct the following Level 2 sample solution plan:

condition	**action**
(> (get-time) 1000)	(follow-path robin '(nottingham 14 13 12 11 10 9 8 7 6 5 4 3 sherwood))
(at john 6)	(set-switch 6 open)
(at john 22)	(set-switch 10 open)

follow-path is a compound plan which waits for the train to be at the first track in the path, and then puts at least two sub-plans on the schedule, one to **move-one** to the next track, one to **follow-path** to subsequent tracks on the path, and plans to **set-switch** any switches connecting the first two tracks, if there are any. **move-one** is in turn a plan to **start** the train in the right direction, and then **stop** it when it gets to the next track. Thus we can monitor execution at multiple levels of abstraction, using the same general scheduler mechanism. The **adjoined** preconditions are satisfied by the plans to set the switches. The switch setting based on john's position will guarantee that it is out of the way, satisfying the track **clear** preconditions on robin's movement.

4.4 Level 3

With Level 3, the causer controlling Little John is actually a full-blown (albeit naive) agent, with it's own goals, plans and beliefs. In principle, all of our planning and reasoning could be carried out at level 2, although with cognitive agents the notion of a plan becomes useful and allows good predictions which would otherwise be unavailable about which actions a causer will perform. Even though a causer may be operating according to a plan, a Level 3 analysis still may not be called for. If the entire plan is precomputed and known ahead of time, a level 2 analysis can be given: no explicit reasoning about the plan need be made, just calculate what it will do when. In cases of less than complete information, plan reasoning can be useful. Maybe just the goal and the available operators are known, but not the entire plan. Then, by simulating the planning process of the other agent, the plan, and thus the actions, can be determined. Also, using plan recognition techniques as in [5], observation of some actions can lead to inferences about the goals and what other actions will be taken.

Furthermore, even though a causer may not be operating based on goals and plans (as in level 2) one could still use a level 3 analysis and attribute a plan to it which would describe its functioning. This kind of thing happens all the time in using anthropomorphic descriptions of inanimate activity. The acid-test will be whether such rationality assumptions prove useful in predicting the actions of the causer.

Simulating the reasoning of Little John, we can come up with two possible minimal (in the sense that no track is unnecessarily traveled over twice) abstract plans. Where
path1 = (Sherwood 3 4 5 6 17 18 19 20 21 22 10 11 12 13 14 Nottingham)
and path2 = (Sherwood 3 4 5 6 7 8 9 10 11 12 13 14 Nottingham),
he can perform either p1 = **(Follow-path john path1)** or p2 = **(Follow-path john path2)**. We can also see that the only possible minimal plans for Robin Hood are p1$'$ = **(Follow-path robin path1$'$)** or p2$'$ = **(Follow-path robin path2$'$)** (where path1$'$ and path2$'$ are the inverses of path1 and path2, respectively). A payoff matrix can be calculated for each of the parties for each combination of plans, with success (John gets to Nottingham, or Robin gets to Sherwood) represented by a 1 and failure (not getting to the destination) represented by 0. For this problem, the payoff matrix is shown in figure 2, where Robin Hood's payoffs are in the lower left corner, and Little John's are in the upper right corner of each box.

		Little John's Plan	
		p1	p2
Robin Hood's Plan	p1′	0, 0	1, 1
	p2′	1, 1	0, 0

Figure 2

We can see that if Little John chooses p1, Robin Hood must choose p2′ while if Little John chooses p2, Robin Hood must choose p1′. Unfortunately, there is no way for Robin Hood to know ahead of time which one John will choose. To Robin Hood, Little John's choice will appear random. The solution is to monitor Little John's actions, and so be able to tell which choice he has made. Only certain actions will be consistent with any particular plan. When an action has been observed which is inconsistent with a plan, that plan can be removed from the list of possibilities.

Although there are no direct perception calls for actions, actions can be recognized by making repeated queries about interesting information and noting when the value changes. Thus we can recognize a **move-one-space** action by Little John, by monitoring the position of the train and noting when it has moved one space from its previous position. If we notice that Little John has moved from 6 to 17 we can be sure that his plan is p1, since **(move-one-space John from 6 to 17)** is not a step in p2.

Robin Hood cannot simply wait until it discovers Little John's plan, and then act, because in that case, Little John will be past the middle section before Robin can get to the Right Switch. Fortunately there is a middle ground between blindly guessing and waiting for perfect knowledge. We have chosen the following algorithm: At any time that Robin is not acting, try to choose an action subject to the following constraints:

1. It is the next action in some possible plan
2. For each possible combination of plans of the other agents, the action is part of a plan which is successful given that combination

This strategy will preserve success while still allowing actions to be made before complete information is known. In our example, we start out with Little John's plan being **(or p1 p2)**, while Robin Hood's plan is **(or p1′ p2′)**. Now, based on the payoff matrix in Figure 2, we know that we must maintain both p1′ and p2′ until we have discovered Little John's plan. But we can still act, because the first few actions (e.g. **(move-one-space Robin from Nottingham to 3)**, **(move-one-space Robin from 3 to 4)**) are common to both plans. If Robin Hood gets to the Right Switch before Little John commits to a plan, then there will be no safe actions, and Robin Hood will wait. This is a safe strategy at Level 3, because Little John is unaware of Robin Hood, and therefore will not base its decision on any behavior of Robin Hood. Eventually Little John will move either to 17 or to 7, allowing Robin Hood to recognize his plan as either p1 or p2, which will thus mandate a safe choice of action.

This strategy can be implemented straightforwardly in the level 2 style executer. We can precompute the possible plans for both Robin Hood and Little John, and the payoff matrix for Robin Hood. We then have two top level activities in our scheduler, the first observes the salient portions of the world and recognizes actions and updates the possible plans for Little John by eliminating any which do not include the observed actions, the second waits until Robin is not performing an action and there is a safe next action to take (based on the criteria above) and then adds the action to the schedule and updates the possible plans for Robin, eliminating any which do not have this action as a part.

4.5 Level 4

In Level 4, we have increased the capabilities of the other agent, so that it can now reason about the planner's plans and beliefs. The planner now needs to reason about nested and mutual belief and knowledge. This is no longer just straightforward simulation of another agent's planning process: the other agent will also be trying to figure out what the planner is doing. For a very small game with a definite solution, (such as NIM), a kind of exhaustive search can be used so that it doesn't matter so much what the other agent believes, but this is impractical for any but the smallest problems. There is a danger of infinite recursion, with each agent reasoning about what the others are reasoning about its own reasoning, and so on.

We cannot simply use the solution from level 3, because if both agents use this strategy, a deadlock will occur: each will be waiting indefinitely for the other to commit to a plan. The way out of this is to take a chance and, if the others agent's plan has not been discovered, just pick one at random. This is relatively safe, since the payoff matrix indicates a cooperative situation. Each can reason confidantly that if the other recognizes his plan, it will act so as to further it by taking the other path. This way, the first one to come to the middle section will choose a path, and the other will take the other path. A problem occurs if they both come to the middle path at the same time. In this case there is a 50% chance that they will choose the same path and thus fail.

While it is easy enough to engineer the solution to this problem, it is not so easy to see how to achieve it as a result of reasoning about nested beliefs. Unfortunately, if one believes that the other agent will choose a path, it pays to wait for that choice, as in the level 3 solution. But if one believes the other will wait, it pays to choose one, because the other agent will then pick the other path. There is an infinite alternating series here with deadlock if they both choose to wait, and a possibility of crashing if both choose to go ahead.

Unfortunately, there is no way to do better in a totally unfamiliar situation without dynamic replanning. If replanning were allowed, the trains could realize that they are on a collision course, and back up and try again. Eventually, they will choose different paths (if they are using truly random selection, this is guaranteed, but even if they are using psuedorandom numbers with the same seed, eventually they will get out of synch due to realtime interactions, so that one will be on a path while the other isn't). Such replanning makes plan recognition much harder in the first place, however.

Another way out of the problem, if such situations occur frequently in a community of agents, is to set up a convention. This could be something as general as "always take the rightmost path" or as specific as, for this particular situation, "Robin Hood takes the lower path while Little John takes the Upper Path". As long as the convention is agreed upon, each agent can easily pick the correct path. This amounts to mutual knowledge about the expectations of individuals to take certain actions rather than others, which combined with the payoff matrix will yield a preference to follow the convention. [6] provides other examples of this type of phenomenon.

4.6 Level 5

At this level, full-scale communication between agents occurs. Agents can use language to find out about the world (things that perhaps they can't see) or intentions of the other agents, etc. They can also inform other agents about their own goals and beliefs, and perhaps make it easier to satisfy their goals by changing other agents plans, beliefs, and goals. We can also have negotiating agents, where communication is used not just for passing information but also to form mutual plans among the agents for cooperative activity. This can range from a plan which both contribute towards fulfilling two individual goals, to the two agents coming up with one compound goal which benefits both.

Specifically, in this example, we can use communication to arrive at the mutual expectations that will allow a way out of the level 4 problem. Communicated intentions will allow each agent to choose a path confident that the other will be aware of this choice and act accordingly, in the same way as if a convention already existed. How to incorporate communication of intention into a theory of rational action is still an object of much study [2].

5 Conclusion and Future Work

In a particular planning problem, an autonomous agent may be faced with several different types of forces. The classifications described in this paper may help in doing the most appropriate reasoning about such a problem. Type 1 causers are easy to plan around. Type 2 causers require more care in checking the interactions between planned actions and actions of other causers. A type 3 causer will be easier to deal with if its plan can be figured out, also some sorts of minimal rationality assumptions may help here. A type 3 causer is not going to concern itself directly with other agents, whereas a type 4 causer will. A type 5 causer can be communicated with to smooth over potential interactions.

An interesting follow-up would be to see how this classification scheme interacts with uncertainty. It would be interesting to contrast how using higher level models of causers may give better information, and at which point one model is preferable to another.

The ARMTRAK simulator also provides a good platform for this study, since it can introduce additional probabilities of failure of actions which are distinct from the failure due to interactions of particular causers.

Acknowledgements

This material is based upon work supported by ONR/DARPA under research contract numbers N00014-82-K-0193 and N00014-80-C-0197. We would like to thank Nat Martin for many helpful suggestions, including the choice of domain problem and the "Robin Hood" name, Steven Feist for maintaining the Armtrak simulator, and Lou Hoebel, Josh Tenenberg, Leo Hartman, Len Schubert, Henry Hexmoor, and the participants of the MAAMAW90 Workshop for comments on earlier versions of this work.

References

[1] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[2] P. R. Cohen, J. Morgan, and M. E. Pollack, editors. *Intentions in Communication.* MIT Press, 1990.

[3] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem sloving. *Artificial Intelligence*, 2:189–208, 1971.

[4] Cordell Green. Application of theorem proving to problem solving. In *IJCAI-69*, pages 219–239, 1969.

[5] Henry Kautz. *A Formal Theory of Plan Recognition.* PhD thesis, University of Rochester, Rochester, NY 14627, 1987.

[6] David Lewis. *Convention: A Philosophical Study.* Harvard University Press, 1969.

[7] Nathaniel G. Martin, James F. Allen, and Christopher M. Brown. Armtrak: A domain for the unified study of natural language, planning, and active vision. Technical Report TR324, University of Rochester, January 1990.

[8] E. D. Sacerdoti. Planning in a hierarchy of abstraction. *Artificial Intelligence*, 5:115–135, 1974. Abstrips.

[9] E. D. Sacerdoti. The nonlinear nature of plans. In *IJCAI-75*, pages 206–214, 1975. Noah.

[10] Steven A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246–267, 1983.