From: AAAI Technical Report WS-96-02. Compilation copyright © 1996, AAAI (www.aaai.org). All rights reserved.

Recognition and Critiquing of Erroneous Agent Actions

Ole Jakob Mengshoel and **David C. Wilkins**
Department of Computer Science and Beckman Institute
University of Illinois, Urbana-Champaign
Urbana, IL 61801
{mengshoe, wilkins}@cs.uiuc.edu

Abstract

An agent can perform erroneous actions. Despite such errors, one might want to understand what the agent tried to achieve. Such understanding is important, for example, in intelligent tutoring and expert critiquing systems. In intelligent tutoring, feedback to the agent—i.e. the student—should focus on which actions made sense and which did not. To achieve such understanding, we propose a combined planning and plan recognition approach. This paper focuses on plan-based plan recognition. We describe a generic recognition algorithm and show how it can be used for plan recognition and critiquing. The approach is illustrated using a real-world application in the area of multi-media tutoring of ship crisis management.

Introduction

An agent can perform erroneous actions. Despite such errors, one might want to understand what the agent tried to achieve. Such understanding is important, for example, in intelligent tutoring and expert critiquing systems. We are concerned with such systems in the context of crisis management. In particular, we are developing an intelligent tutoring system based on an interactive crisis simulator. The crisis simulator utilizes multi-media technology that immerses a ship crisis management expert in a very realistic simulation. Such a simulator raises a variant of the well-known problem of credit (or blame) assignment: Given a series of actions on part of an agent, which actions are to be given credit (or blame) for the success (or failure) of the agent, and to what degree?

Central in this question is the notion of action, and in particular reasoning about action in the form of planning and plan recognition. A number of plan recognition approaches have been proposed (Schmidt, Sridharan, & Goodson 1978) (Genesereth 1982) (Kautz & Allen 1986) (Charniak & Goldman 1993) (Gertner & Webber 1995). Plan recognition recognizes prototypical behavior, but typically assumes complete knowledge of plans in the domain (Kautz & Allen 1986). Planning, on the other hand, does not assume complete plan knowledge. Plans are constructed, tailored to the situation at hand (Fikes & Nilsson 1971) (Sacerdoti 1975) (Hammond 1986) (Simmons 1992) (Hanks & Weld 1995). Thus, an integration of plan recognition and planning seems fruitful, and the present work is a step in that direction.

Such an integrated approach to planning and plan recognition needs to be robust to agent error. Students typically commit a number of errors when using an interactive crisis simulator, and agents in general make errors. Errors have been examined in expert critiquing and intelligent tutoring system (ITS) research. In expert critiquing, human error has been investigated (Silverman 1992), however this line of research has generally, with some notable exceptions, not been within the realm of planning (Silverman 1992, p. 123). One exception is ONCOCIN, which performs skeletal-refinement planning in an expert critiquing context (Tu *et al.* 1989). Another exception is TraumAID, which combines progressive horizon planning with expert critiquing (Rymon, Webber, & Clarke 1993) (Gertner & Webber 1995). In intelligent tutoring, the notion of a bug or error library has been used to recognize erroneous student plans (Genesereth 1982) (Polson & Richardson 1988). A limitation is that it is very tedious to construct a bug library in a real-world application such as ours.

Plan recognition approaches have typically abstracted away the problem of erroneous plans (Kautz & Allen 1986). In planning, the notion of critic was introduced to detect helpful or harmful interactions in plans (Sacerdoti 1975). Both interaction types are similar to student errors. Similarly, the areas of transformational planning, adaptive planning, and case-based planning all contain transformational (i.e. refinement and retraction) steps that can remove 'errors' from a plan (Hammond 1986) (Simmons 1992) (Hanks & Weld 1995). Compared to operations needed to recognize and critique student errors, both plan refinement and retraction are very limited.

We believe that existing expert critiquing, ITS, plan recognition, transformational, adaptive and case-based planning approaches can be improved in situations where an agent's (e.g. a student's) actions contain

many errors. To achieve this, we propose a combined planning and plan recognition approach. This paper focuses on plan-based plan recognition. We describe a generic recognition algorithm and show how it can be used for plan recognition and critiquing. The approach is similar to the ITS notion of model tracing (Anderson *et al.* 1990) (Anderson *et al.* 1995) because a generative model of the cognitive skill, a planner, is employed. The approach is different from model tracing in that we use a planner and a plan representation and in that we avoid modeling erroneous student rules (or a bug library).

This paper is organized as follows. The first section describes the application, crisis management training, and what kinds of errors students perform. The second section presents our approach to plan recognition and critiquing. The third section gives an example of how the approach works for crisis management training. The final section concludes and mentions areas for future work.

Background on Damage Control

We present here the ship damage control domain, a multi-media tutoring system that has been developed for the domain, and some types of errors students perform when using the training system.

Damage Control Application

On Navy ships there is a designated officer, the damage control assistant (DCA), who is in charge of handling damage to the ship. The DCA reports to the commanding officer, and commands a number of repair teams that investigate, report, and repair damage to the ship as it occurs. A repair party belongs to a repair station, and the DCA commands repair parties by communicating with the repair station leader.

In order to train DCAs in a realistic way, an interactive courseware approach has been taken. In particular, an interactive multi-media system called integrated damage control team training (IDCTT) has recently been developed. IDCTT exposes the student DCA to a scenario containing a mine hit and a missile hit along with the resulting damage to the ship. Damage amounts to fire, smoke, and flooding of different spaces in the ship, and the student DCA directs virtual investigation and repair teams by using IDCTT. The scenario, which lasts approximately 20–25 minutes, is summarized in Figure 1. In the figure, approximate times for events and main events are shown to the left. The main part of the figure shows the six main goals ('Zebra reported to the bridge,' 'Aft flooding isolated,' ...) that the student needs to identify and achieve. Above each goal is an elongated rectangle, illustrating that some set of actions needs to be performed in order to achieve the goal. Above each action set there is a damage description ('Aft flooding,' 'Starboard FM low,' ...) or a status discrepancy ('Zebra not set').

The scenario goes through three phases, indicated by horizontal dotted lines in the figure. Consider the second phase, when a mine hits the ship, resulting in three problems. One of these, the 'Aft fire', needs to be followed up by the following orders (or, more generally, commands) on part of the student: 'Investigate space 3-370-0-E' sends an investigator to the space named 3-370-0-E with a fire alarm. In case of a report about a fire from the investigator, the DCA should order 'Set fire boundaries 442, 410, 370, 338', where the numbers represent bulkheads aft and forward of the fire in compartment 3-370-0-E. After hearing the report 'Fire boundaries set', the DCA should order 'Fight fire in space 3-370-0-E'. If the DCA then receives the report 'Fire out in space 3-370-0-E', the fire fighting effort succeeded. The scenario progresses according to which actions the student takes, and the ship either stays afloat or sinks.

The IDCTT system is currently used towards the end of a Navy course for student DCAs at the Navy's officer training school in Newport, Rhode Island. A group of two or three students cooperate and are supervised by one instructor when using the IDCTT system. One of the students plays the DCA, the others play his assistants. After one IDCTT session, the instructor gives feedback to the group of students.

From a planning perspective, the task of the student is to generate orders such that all problems are resolved—i.e. the implied goals are achieved. From a plan recognition perspective, the task of the instructor is to identify what the student's orders are meant to achieve, and to give feedback to the student concerning those actions. One goal of this research is to supplement or replace the critique as given by the instructor with automatically generated feedback.

Student Commands and Errors

Students control IDCTT, and similar simulation systems, using commands. A *command c* consists of a command type, command parameters and a command time. As an example, the command

> 10:00 REPAIR 3 - Ordered to set fire boundaries 442, 410, 370, 338

has command type 'set fire boundaries', command parameters 'REPAIR 3' and '442, 410, 370, 338', and command time '10:00'. Example commands are displayed, along with other IDCTT transcript messages, in Table 1. The commands of one student handling the fire in generator room 3-370-0-E are shown.[1]

Analysis of IDCTT transcripts of student DCAs shows that a number of errors are performed. Informally, we define an *error* as a change that can be made

[1] The message at 07:20 is actually 'REPAIR 3 - Reports investigators away', but the present critiquing system cannot handle such messages. Therefore we have made a simplification.

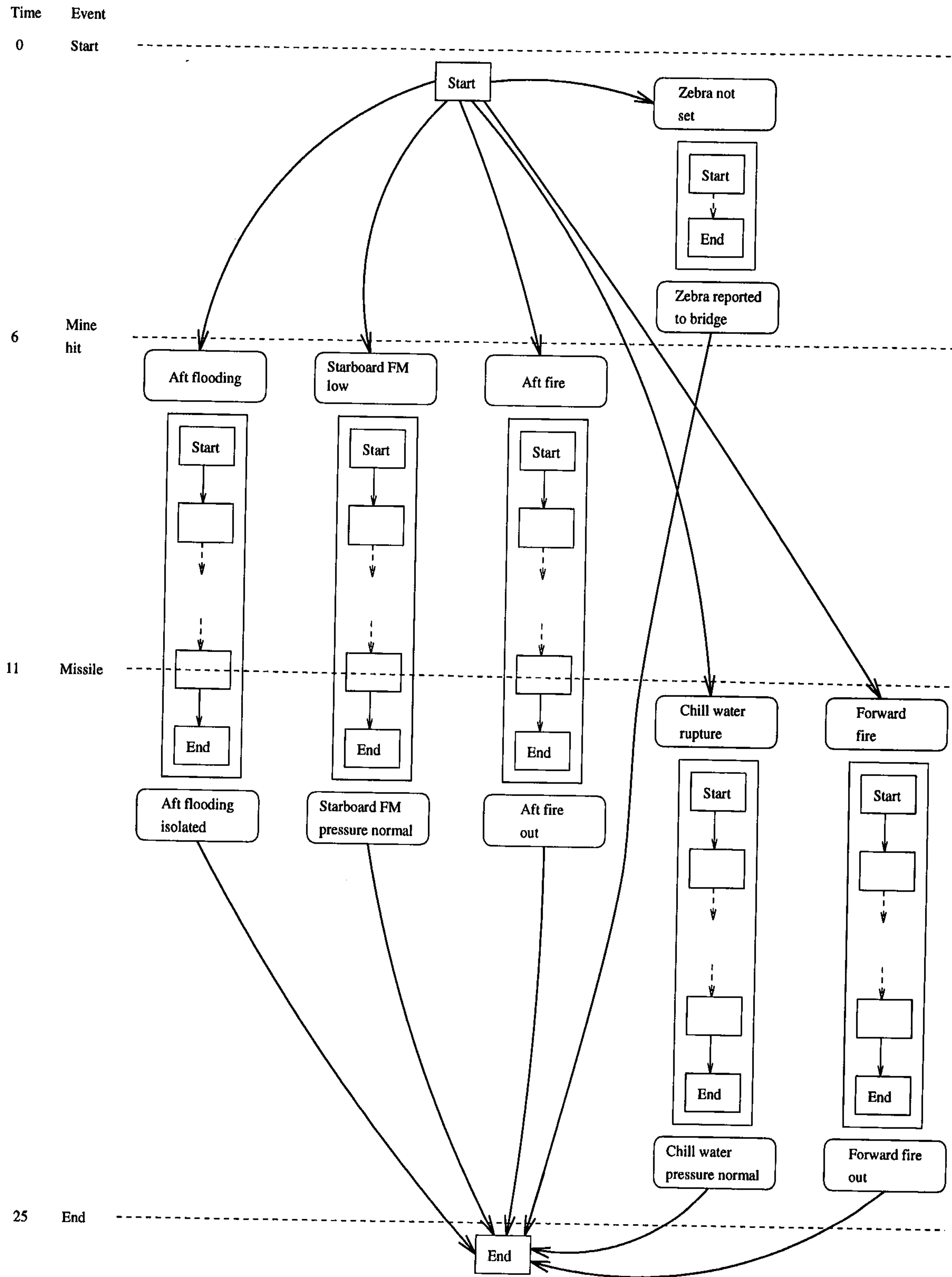

Figure 1: Overview of the IDCTT scenario.

Command time	Command parameter	Command type and parameters
06:45	DCCO	Reports fire alarm generator room 3-370-0-E
07:20	REPAIR 3	Ordered to investigate 3-370-0-E
09:27	DCCO	Reports halon alarm generator room 3-370-0-E
09:57	REPAIR 3	Reports fire in generator room 3-370-0-E, Halon activated, Fireparty en route.
10:09	REPAIR 5	Ordered to fight fire in 3-370-0-E
10:13	REPAIR 5	Recommends you review their last order
10:27	REPAIR 3	Reports halon ineffective in generator room 3-370-0-E
10:42	REPAIR 3	Ordered to fight fire in 3-370-0-E
11:40	REPAIR 3	Reports electrical and mechanical isolation is complete
11:44	REPAIR 3	Reports fire party OBA light off time
12:01	REPAIR 3	Ordered to fight fire in 3-370-0-E
14:53	REPAIR 3	Reports fire spreading rapidly aft of generator room
15:09	DCCO	Reports AFT VLS high temp alarm
15:14	CSMC	Reports AFT VLS high temp alarm, smoke reported, auto sprinkler failure

Table 1: Example commands issued by a student DCA.

Error type	Error subtype	Example
Command error	Missing command	Skipped 'set fire boundaries'
	Swapped commands	Swapped 'set fire boundaries' and 'fight fire'
	Redundant command	Included 'electrical and mechanical isolation'
Parameter error	Incorrect parameter	Incorrect fire boundaries: '480, 400, 350, 330'
	Incomplete parameter	Incomplete fire boundaries: ', 410, 310,'
	Swapped parameter	Swapped fire boundaries: '338,310,410,442'

Table 2: Types of errors performed by DCA students.

to an incorrect command, making it a correct command. A correct command is part of a correct set of commands, corresponding to a set of steps, i.e. a plan. In general, we are looking for smallest total error in comparison to some correct plan, according to some measure, to be described below. For the purpose of this paper, we are concerned with two types of student errors:

- Command error: There is an error in a command, or among a set of commands, when viewed as part of a procedure to achieve a goal.
- Parameter error: Correct command type is invoked, but with at least one erroneous parameter.

Examples of the two error types are presented in Table 2. The examples of parameter errors in the table relate to the correct fire boundaries '442, 410, 370, 338' to be used for the generator room fire in the scenario.

Plan Recognition and Critiquing

The combined planning and plan recognition approach is outlined in this section.

System Architecture and Overview

The system architecture consists of these components: planning, plan recognition, plan critiquing, and dialogue generation. Here we address plan recognition and critiquing. Plan recognition seeks to answer the question: What are the intentions of the student? Specifically, we interpret this as searching for a plausible plan, which is a correct solution representing what the student is trying to achieve (Mengshoel, Chauhan, & Kim 1996).

A student's interaction with a scenario is captured in the form of a transcript T. A transcript T consist of a sequence of transcript messages t_i, i.e. $T = (t_1, \ldots, t_n)$. Not all transcript messages are interesting from the perspective of plan recognition and critiquing. In particular, we focus on messages that are commands from the student, and denote these student commands C, where $C = (c_1, \ldots, c_m)$, where $m \leq n$.

A plan P is a tuple (S, K, L), where S is a set of steps (operators made unique); K is a set of constraints, ordering constraints or binding constraints; and L is s set of causal links, (s_i, q, s_j), where s_i and s_j are steps, q is an expression. An operator is a STRIPS operator (Fikes & Nilsson 1971) (Hanks & Weld 1995). Figure 2 shows example plans. A rectangle represents a STRIPS operator and a directed edge a causal link. A literal above a STRIPS operator is a precondition, a literal below (and at the tip of a causal link) is a postcondition. There are two dummy STRIPS steps START, with postconditions only, and END, with preconditions only, which represent the beginning and end of the plan respectively. The START step causes the

initial state, while the END step has the goal state as precondition.

This is the top-level algorithm:

Algorithm:

- Execute *Planner(I, O, G)* to compute plans PP
- Execute *Generic-Recognizer(C, SS, m, o, c)*, where $SS = \{S | P = (S, K, L) \in PP\}$, to compute closest match P' and matching R
- Execute *Plan-Critiquer(C, P',G,R)* to produce and present a critique

The *Planner* is given inputs initial state I, operators O, and goals G, and computes plans PP. The student's orders C as well as correct plans PP are input to the *Generic-Recognizer*, which computes a plausible plan P' as well as how it relates R to the student's commands C. The domain-specific functions m, o, and c are also input to this procedure. C, P', and R are then used by the *Plan-Critiquer* to generate a critique of the student.

We assume that a partial-order planner from the STRIPS tradition is used (Fikes & Nilsson 1971) (Hanks & Weld 1995), and do not discuss it below. The *Generic-Recognizer* and *Plan-Critiquer* algorithms are presented in the following.

Generic-Recognizer Algorithm

The essence of the algorithm is to transform C and $S \in SS$ into a weighted bipartite graph G with partite sets A and B (Cormen, Leiserson, & Rivest 1992) (Knuth 1993) (Mengshoel, Chauhan, & Kim 1996). Here, $SS = \{S_1, \ldots, S_m\}$, and $S_i = \{s_{i_1}, \ldots, s_{i_n}\}$. The *Generic-Recognizer* algorithm finds some solution S with highest total matching weight:

Input: Student's solution C, solutions SS, matching function m, projection function o, complexity function c

Output: Relation R between $C' \subseteq C$, $S' \subseteq S$, where $S \in SS$, and $w \in [0, 100]$

Algorithm:

- For each component F of C, create a vertex u in A
- For all $S_i \in SS$:
  - For each component H of S_i, create a vertex v in B_i
    - For each F (with vertex u) that corresponds (i.e. $o(F) = o(H)$) to H (with vertex v), create an edge $uv \in E_i$
    - Compute edge weight using matching score function m: $w(uv) = m(F, H)$ for all $uv \in E_i$
  - Execute *Weighted-Bipartite-Matching(G_i, w)* to calculate total weight $W(S_i)$ and matching N_i, where $G_i = (A \cup B_i, E_i)$
- Among all $S_i \in SS$, find the one with highest weight W, and the corresponding relation N from $C' \subseteq C$ to $S' \subseteq S$. If there is a tie between weights, pick the system solution S_i such that $c(S_i) \leq c(S_j)$. If there is a tie again, pick an arbitrary system solution.
- R is easily computed from N: If $(u, v) \in N$, then $uv \in E_i$ for some i, and $w(uv)$ exists. Let $(u, v, w(uv)) \in R$.

Weighted-Bipartite-Matching is a well-known algorithm used to compute the the weight of a matching; it is often called the 'Hungarian' algorithm (Knuth 1993). The functions m, o, and c depend on how the Generic-Recognizer algorithm is used. Intuitively, m expresses the degree of match between a component in C and a component in some S. The function o expresses the 'projection' or 'origination' of a component. For example, if we consider a command, the projection is merely the command type. The function c expresses the complexity of a solution. For example, a solution consisting of more components than another can be considered more complex. Thus the number of components can be used for c.

Command Recognition Functions

The *Generic-Recognizer* algorithm needs to be provided with the domain-specific functions m, o, and c. These are described below.

Let n and pre be, respectively, functions that return the name and preconditions of a step S. The name n of a step is the step type and step parameters—analogous to for a command. Now we define the origination function o as: $o(n(S)) = t$, where S is a step and t is the step's type.

Now consider the matching function m. These are the considerations for matching two actions. An action is an order generated by the student or a step generated by the planner. First, the actions match only if they have the same action type. So the order `Fight-fire(Repair-3, 3-370-0-E)` matches the step `Fight-fire(Repair-5, 3-370-0-E)`. However, `Fight-fire(Repair-3, 3-370-0-E)` does not match `Investigate-space(Repair-3, 3-370-0-E)`.

The second consideration for matching two actions is that parameters are matched on a parameter by parameter basis and according to their type. Example parameter types are space, like `3-370-0-E`, repair locker, like `Repair-3`, and boundaries, like `(442, 410, 310, 338)`. Parameters can be considered as numbers or strings. For example, consider the correct fire boundaries for the fire in space `3-370-0-E`, `(442, 410, 310, 338)`. Possible erroneous ways to specify this boundary are: `(338, 310, 410, 442)`, `(450, 420, 300, 330)` and `(142, 410, 170, 538)`. In the first case, the problem appears to be that the DCA has input the correct boundaries incorrectly. Here, we may consider the boundary as a string and do an edit distance operation. In the second case, the boundaries are reasonable but not quite correct, and the DCA has input them correctly. Here, it is appropriate to regard the boundaries

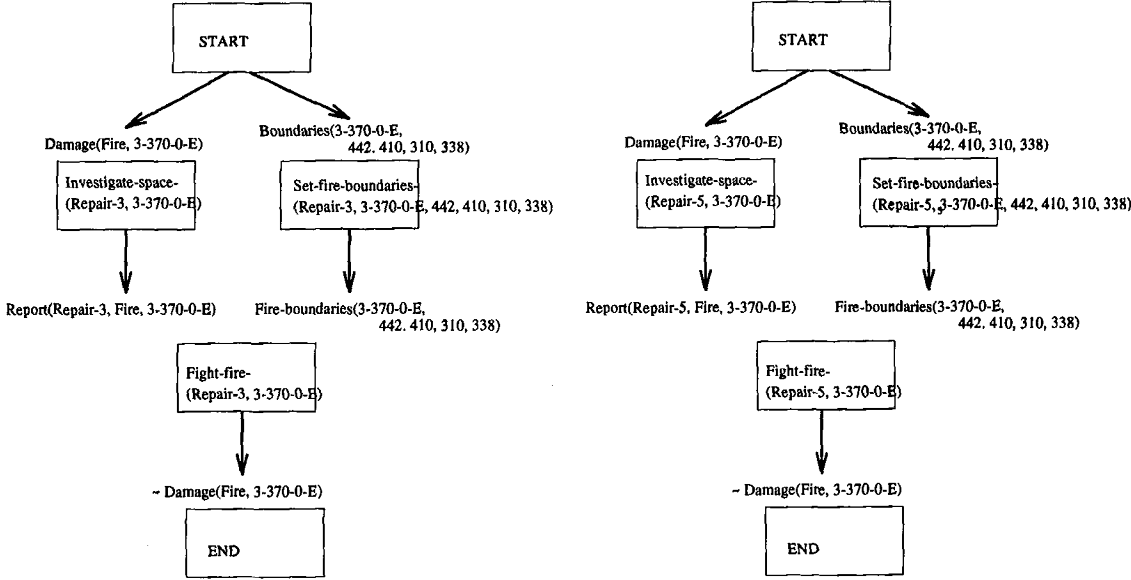

Figure 2: Example plans. The only difference between them is that in the leftmost plan, Repair-3 is ordered, while in the rightmost plan, Repair-5 is ordered. Repair-3 and Repair-5 are repair station which the DCA communicates with.

as a sequence of numbers, and perform arithmetic operations. In the third case, the boundaries do not make sense at all. So neither an edit distance transformation nor arithmetic operations would help.

More formally, let f and h be two actions to be matched, with

$$n(f) = t_f(p_{f_1}, \ldots, p_{f_n}),$$

and

$$n(h) = t_h(t_{p_1}, \ldots, p_{h_m}).$$

The function m is defined as follows:

$$m(f,h) = \begin{cases} 0 & \text{if } o(n(f)) \neq o(n(h)) \\ \sum_{i=1}^{n} \frac{100 + m'(t_{f_i}, t_{h_i})}{n+1} & \text{if } o(n(f)) = o(n(h)), \end{cases}$$

where

$$m'(s,t) = \begin{cases} m'_b(s,t) & \text{if } s,t \text{ are boundaries} \\ m'_s(s,t) & \text{if } s,t \text{ are spaces} \\ m'_o(s,t) & \text{otherwise} \end{cases}$$

So there is a number of parameter types, where for each there is a matching function such as m'_b, m'_s, or m'_o. Example function definitions follow, assuming that we perform matching on a scale from 0 to 100. Here 0 designates no match, 100 full match.

The general matching function $m'_o(s,t)$ is defined as:

$$m'_o(s,t) = \begin{cases} 100 & \text{if } s = t \\ 0 & \text{otherwise} \end{cases}$$

In the boundary matching function m'_b, the arithmetic distance operations a and the edit distance operation e are utilized:

$$m'_b(s,t) = \min(e(s,t), a(s,t))$$

Arithmetic distance a is defined as follows:

$$a(s,t) = 100 - \left\lfloor \frac{100}{4} \sum_{i=1}^{4} \frac{|s_i - t_i|}{f_{\max} - f_{\min}} \right\rfloor,$$

where $f_{\max}$ and $f_{\min}$ are maximal and minimal bulkhead numbers respectively (for the particular ship in question). u_i, where $u = s$ or $u = t$, designates bulkhead numbers in the boundary. u_1, u_2, u_3, and u_4 designate secondary aft, primary aft, primary forward, and secondary forward fire boundary respectively.

Finally, consider the complexity function c. We define c as:

$$c(S) = |S|,$$

where S is a set of steps.

Plan-Critiquer Algorithm

There are many ways to give feedback to a student, both in terms of content and structure. Content-wise, we base feedback on the matching plan and include negative feedback. Structure-wise, we structure the feedback according to goals.

The *Plan-Critiquer* algorithm is as follows:

Input: Student's solution C, matching plan P', goals G, matching R

Algorithm:

- *Critiquer*(END, C, P', R)
- For each command $c \in C$:
  - If $\exists r = (c, s, n) \in R \wedge n = 0$ then print("Redundant command", c)

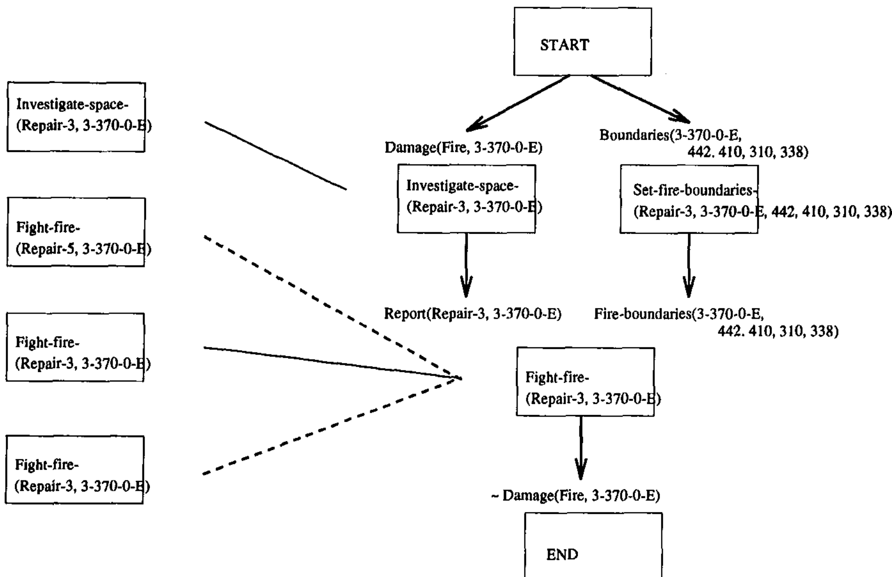

Figure 3: Example matching between commands and a plan. The commands C are shown to the left, the matching plan steps S in the plan P to the right. A solid line between a command and a step shows a correct order. A dotted line between a command and a step shows a redundant or incorrect command. This happens to be the best matching steps $S \in SS$, so the plan is called the plausible plan P'.

The *Critiquer* algorithm is as follows:

Input: Step s, student's solution C, plan $P' = (S, K, L)$, matching R
Algorithm:

- If marked(s) then return
- mark(s)
- For each goal $g_i \in$ pre(s)
  - If $(t, g_i, s) \in L$ and $t \neq$ START then *Critiquer*(t, C, P', R)
- If $\neg\exists r = (c, s, n) \in R$ then print("Missing command:", s)
- If $\exists r = (c, s, n) \in R \wedge n < 100$ then print("Incorrect command:", c)
- For each $(s_i, g, s) \in L$
  - If $\exists r_i = (c_i, s_i, n_i) \in R \wedge \exists r_j = (c_j, s, n_j) \in R$ $\wedge$ time(earliest(c_i)) > time(latest(c_j)) then print("Swapped commands:", c_i, c_j)

Here, mark and marked are procedures that respectively marks a step as visited and checks for such a mark; time is a function that returns the command time; earliest and latest are functions that return the earliest and latest command that is the same as their input parameter (modulo command time). The *Critiquer* first recursively critiques all steps with a causal link to the current step. Then it critiques the current step. The check for the Swapped Commands might be a bit subtle: Since there is a causal link from s_i to the current step s, the time of c_i (corresponding to s_i) should be less than the time of c_j (corresponding to s). So we critique when the time of c_i is greater than the time of c_j. The functions latest and earliest need to be used because there can be several identical commands—cf. the two `Fight-fire(Repair-3, 3-370-0-E)` commands in Figure 3.

Example of Approach

To illustrate our approach, consider how a critique for the student transcript fragment shown in Table 1 will be produced. The *Planner* step will create plans PP, examples of which are shown in Figure 2, The *Generic-Recognizer* will, based on commands C in the above transcript fragment and on correct plans SS, compute a plausible plan P'.

How is the plausible plan computed? The START operator (or initial state) comes from the simulator, the END operator (or goal state) comes either from the simulator or is computed by an as of yet unspecified algorithm. The initial state here is `Damage(Fire, 3-370-0-E)` and `Boundaries(3-370-0-E, 442, 410, 310, 338)`, the goal is `~Damage(Fire, 3-370-0-E)`. Given this, the *Planner* computes plans, of which two examples are shown in Figure 2. Based on matching with the student's commands, the leftmost plan in Figure 2 is chosen as shown in Figure 3.

The *Plan-Critiquing* algorithm will, based on the plausible plan P', generate this critique to the student:

1. Missing command: `Set-fire-boundaries(Repair-3, 3-370-E, 442, 410, 310, 338)`
2. Redundant command: `Fight-fire(Repair-5, 3--370-0-E)`

3. Redundant command: `Fight-fire(Repair-3, 3--370-0-E)`

How is the critique constructed? This is done by utilizing the matching between the student's commands and the plausible plan P' as depicted in Figure 3. In particular, it is easy to identify missing, swapped, incorrect, or redundant commands among the student's commands when compared to a plausible plan.

Conclusion and Future Work

The motivation for this research is the fact that agents, for instance damage control assistant students, make errors when acting. We have identified types of errors made by damage control assistant students for the purpose of recognizing and giving them feedback about those errors. We have described a generic recognition algorithm which can be utilized to recognize and critique erroneous student actions based on correct plans.

Areas for future work include the following. First, the dynamic nature of scenarios needs to be addressed. Second, while a 'pure' STRIPS language is used in this work, a more expressive language will probably be adopted. For example, there is a distinction between actions that have a causal effect (e.g. turning on a firepump) and those that have a communicative effect (e.g. ordering a repair party) that could be expressed using more expressive operators. Third, more attention must be given to the interaction between planning and plan recognition. For example, we have assumed that a complete set of ground (variable-free) plans are generated by the planner, and these assumptions might need to be lifted.

Acknowledgements: This work was supported in part by ONR Grant N00014-95-1-0749 and U.S. Army Research Lab Grant DAAL01-96-003. Thanks to Surya Ramachandran for clarifying discussions about the DCA domain and expert critiquing systems, and to Lisa Kaufman and the two reviewers for comments on the manuscript.

References

Anderson, J. R.; Boyle, C. F.; T., C. A.; and Lewis, M. W. 1990. Cognitive modeling and intelligent tutoring. *Artificial Intelligence* 42:7–49.

Anderson, J. R.; T., C. A.; Koedinger, K. R.; and Pelletier, R. 1995. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences* 4:167–207.

Charniak, E., and Goldman, R. P. 1993. A Bayesian model of plan recognition. *Artificial Intelligence* 64(1):53–79.

Cormen, T. H.; Leiserson, C. H.; and Rivest, R. L. 1992. *Introduction to Algorithms.* Cambridge, Massachusetts: MIT Press.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

Genesereth, M. R. 1982. The role of plans in intelligent teaching systems. In Sleeman, D., and Brown, J., eds., *Intelligent Tutoring Systems*, 137–152. London: Academic Press.

Gertner, A., and Webber, B. 1995. Recognizing and evaluating plans with diagnostic actions. In *Proc. of IJCAI-95 Workshop on Plan Recognition.*

Hammond, K. 1986. CHEF; a model of case-based planning. In *Proc. of AAAI-86*, 261–271.

Hanks, S., and Weld, D. S. 1995. A domain-independent algorithm for plan adaption. *Journal of Artificial Intelligence Research* 2:319–360.

Kautz, H. A., and Allen, J. F. 1986. Generalized plan recognition. In *Proc. of AAAI-86*, 32–37.

Knuth, D. E. 1993. *The Stanford GraphBase: A Platform for Combinatorial Computing.* New York: ACM Press, 1994 edition.

Mengshoel, O. J.; Chauhan, S.; and Kim, Y. S. 1996. Intelligent critiquing and tutoring of spatial reasoning skills. *Artificial Intelligence for Engineering Design Analysis, and Manufacturing.* Forthcoming.

Polson, M. C., and Richardson, J. J., eds. 1988. *Foundations of Intelligent Tutoring Systems.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Rymon, R.; Webber, B. L.; and Clarke, J. R. 1993. Progressive horizon planning — planning exploratory-corrective behavior. *IEEE Trans. on Systems, Man, and Cybernetics* 23(6):1551–1560.

Sacerdoti, E. D. 1975. The nonlinear nature of plans. In *Proc. of the 4th IJCAI.*

Schmidt, C. F.; Sridharan, N. S.; and Goodson, J. L. 1978. The plan recognition problem: An intersction of psychology and artificial intelligence. *Artificial Intelligence* 11:45–83.

Silverman, B. G. 1992. *Critiquing Human Error: A Knowledge Based Human-Computer Collaboration Apporach.* Knowledge Based Systems. London, Great Britain: Academic Proess.

Simmons, R. G. 1992. The roles of associational and causal reasoning in problem solving. *Artificial Intelligence* 53:159–207.

Tu, S. W.; Kahn, M. G.; Musen, M. A.; Ferguson, J. C.; Shortliffe, E. H.; and Fagan, L. M. 1989. Episodic skeletal-plan refinement based on temporal data. *Communications of the ACM* 32(12):1439–1455.