when carried out in the world. This implies that case-based systems should index their cases in ways that allow them to both come up with solutions and anticipate their potential for success.

Concluding Remarks

What should be taken away from this chapter? Those in the CBR community should see this as a first attempt at uncovering our tacit knowledge and in making the implicit explicit. This chapter is only a beginning to articulating the paradigm, the methodology, and the cognitive model. We need to be more aware of what our assumptions are, discussing them among ourselves and articulating them explicitly to the outside world. Case-based reasoning has the potential to change a lot in the way we look at intelligence and cognition. That change has already started. In order to reach full potential, we need to articulate case-based reasoning, its underlying assumptions, its conceptual foundations, and its implications in ways that the rest of the community will appreciate.

I hope those from the outside who are reading this take away two insights, first, the importance of focusing on representation and knowledge access both in building systems and in addressing new problems, and second, the notion that seeing case-based reasoning as a driver of cognition rather than an add-on allows us to look at important issues (such as chunking and knowledge access) in new and useful ways.

Notes

1. There's a difference, by the way, between "easily available" features (described in the previous paragraph) and "surface" features, but the principle relates to both: surface features make good indexes to the extent that they are predictive of something important or useful.

2. There's a challenge related to this, one I posed at the talk in summer, 1993: I'd like to see someone take a case-based approach to what Doug Lenat is doing with CYC. How would it be different? The major difference is that it would take access of knowledge (indexing) seriously.

Case-based reasoning: Experiences, Lessons, & Future Directions, David Leake (ed) MIT Press 1996 371-388

17 What Next? The Future of Case-Based Reasoning in Post-Modern AI

Christopher K. Riesbeck

Case-based reasoning, (CBR)once the rallying point for anti-rule revolutionaries, has become an increasingly accepted part of the modern artificial intelligence toolbox. For example, a recent report on the ARPA/Rome Planning Initiative included several examples of large AI systems that involved several rule-based and case-based reasoning subsystems interacting to solve complex real-world problems (Fowler et al. 1995). Most calls for papers in modern AI conferences include CBR as one of the relevant topics.

Some may view this acceptance and proliferation of systems as success for CBR. On the other hand, some of us view it as a sign that the revolution has been co-opted, that the real point of CBR has been lost. Of course, this then raises the question, "What is the real point of CBR?"

In this chapter, I will argue that the real shift in AI has yet to come, and that CBR was just the opening act. The argument will proceed by answering the following questions:

- What is the point of case-based reasoning?
- What is (the point of) AI?
- What's the future of AI?
- What is the role of CBR in that future?

What Is the Point of Case-based Reasoning?

The original point of case-based reasoning was that people really don't think all that much, they remember, in both senses of the word "remember." First, we remember the things we do, including the thinking we do. Second, most of the time we don't need to think, we just have to remember what we thought before (Schank 1982). The first point was an obvious but revolution-

ary idea when first presented. It was obvious in that of course people remember what they do, but revolutionary in that virtually no problem solvers and story understanders did it. The primary exceptions were in planning, where some planners stored plans under the goals they achieved, usually for efficiency. Most machine learning programs, interestingly enough, did not remember the examples that they saw, only the generalizations that could be inferred from those examples. While they changed their behavior as the result of their experiences, they didn't remember those experiences.

At Yale, we found it particularly embarrassing to realize that our story understanders would do exactly the same thing no matter how many times they read the same story. They never got bored, they never realized that one story contradicted another, they were never reminded of Mary's burnt hot dog when reading about John's overcooked hamburger. This was intelligent story understanding!

The second point was—and remains—more controversial. How often do we really think and how often do we just re-use? The claim in Riesbeck and Schank (1989) was that people rarely "think," in the sense of performing the logiclike inferencing common to most AI systems. Rather, people respond to new situations by reusing memories of similar old situations.

> Real thinking has nothing to do with logic at all. Real thinking means retrieval of the right information at the right time (Riesbeck and Schank 1989, p. 9).

For example, repeat trips to Burger King® are handled by reusing memories of previous trips to Burger King®, if any, or by adapting memories of trips to McDonalds®, if any, or by trying to apply memories of other kinds of restaurants, repairing the expectation failures that result, and remembering those repairs for future reuse.

Early development of CBR systems focused on this reuse of real memories. Such systems began with retrieved cases, i.e., memories of past experiences, and adapted them to fit new situations. For example, Simpson's Mediator (Kolodner and Simpson 1989) retrieved and adapted cases of mediations, Hammond's Chef (Hammond 1989c) retrieved and adapted complete recipes, and Kass, Leake and Owens' Swale (Schank et al. 1994) retrieved and adapted old explanations to explain new anomalies. All of these systems used adaptation processes that *replaced* inappropriate details in retrieved cases with details from the current situation. The adaptation process was often quite complex and rulelike, leading many of the early researchers to propose case-based methods of doing adaptation and repair.

Old habits die hard, however, and as the use of CBR became more widespread, it began to take on a more rulelike flavor. Most modern CBR systems do not use episodic cases. It's much more common for them to retrieve "abstract cases" that are generalizations of real cases. These abstract cases are retrieved and applied by refinement processes that *add* details from the current situation, rather than replace details of old situations.

Some researchers have characterized case-based reasoning as "rule-based reasoning with very big rules." This comment only really applies to this modern "abstract CBR." Old-fashioned "true CBR" has two central processes not found in rule-based reasoning or abstract CBR:

- *Partial matching:* in true CBR, you don't find a matching case, you find the case that matches *best.* No case matches exactly in all details. Patterns may be used to organize and store generalizations about cases, but they are not themselves considered to be cases.
- *Adaptation:* in true CBR, you don't apply a case by filling in the details, you have to decide which details to throw away, which to replace, and which to keep.

Partial matching implies adaptation. If you allow the retrieval of cases that don't match the input situation exactly, you need adaptation to resolve the discrepancies. In short, abstract CBR starts with a template for an answer, and fills it in, while true CBR starts with an old answer, then works its way towards a good answer. This difference will become relevant when we discuss the role of CBR in the future of AI.

To summarize: the original point of CBR—the radical point—was to replace reasoning with the recall and adaptation of episodic knowledge.

What is (the Point of) AI?

Everyone has their own definition of AI and reasons why the other definitions don't work. A fair number of them are discussed in Russell and Norvig (1995), who then provide their own definition, which will be discussed shortly. I'd like to motivate my own definition by contrasting it to the following three very typical definitions:

> Artificial intelligence (AI) may be defined as the branch of computer science that is concerned with the automation of intelligent behavior (Luger and Stubblefield 1993, p. 1).

> *Artificial Intelligence:* The field of research concerned with making machines do things that people consider to require intelligence (Minsky 1986, p. 326).

> Artificial intelligence is the study of mental faculties through the use of computational models (Charniak and McDermott 1987, p. 6).

The first and second definitions focus on the term "intelligent." The problem with focusing on intelligence is that many of the most interesting tasks in AI are those that any jerk can do (walk, talk, nod appropriately during a lecture, and so on). Furthermore, there are activities, such as equation solving, or even simply multiplying very large numbers, that are considered to require

intelligence if done by a human, but which, when performed on a computer, are not considered to be AI.

The third definition, perhaps in an attempt to be sensitive to the apparent irrelevance of intelligence, generalizes AI to cover any mental activity at all. But this seems to go too far, making AI a vehicle for any kind of psychology, and leaving out all the AI programs that bear no connection to modeling human cognitive processing.

In short, all of these definitions have the same basic problem: they include many things that are not AI, as it's conventionally construed, and exclude many things that are AI.

A New Definition of AI

My definition of AI comes at it from another angle, by focusing on what most people really want:

> Artificial intelligence is the search for answers to the eternal question: *Why are computers so stupid?*

That is, AI is a repair process. Cognitive science has, among others, the goal of understanding "what is intelligence?" AI has the goal of understanding "what is computer stupidity, and how can we get rid of it?"

AI as a repair process immediately explains the following oft-noted phenomenon: If it works, it's not AI any more. This effect occurs because once computers stop being stupid in that particular way, further work is not repairing the stupidity, but solving some other problem, e.g., making the process faster.

Focusing on repairing stupidity also explains why getting computers to do simple tasks like walking and talking seem more like real AI than tasks requiring great intelligence, for example:

- Not understanding simple sentences in context is stupid. Ergo, natural language understanding research is AI.
- Not being able to solve equations is not stupid. Ergo, numerical analysis is not AI.
- Not recognizing your own hand in front of your own face is stupid. Ergo, computer vision research is AI.
- Not learning from experience is stupid. Ergo, machine learning and case-based reasoning are AI.

There are some controversial implications of this definition:

- Not being able to play chess well is not stupid. Ergo, computer chess is not AI.
- Not being able to prove theorems is not stupid. Ergo, theorem proving is not AI.

- Not being an expert is not stupid. Ergo, expert systems are not AI.

I claim however that in fact the definition is right in these conclusions. These areas are AI by historical inertia. That is, the initial research was AI, but, like optical character recognition and symbolic equation solving, the AI motivation has long since been superseded by other goals. In particular:

- Building the world's best chess player is no more AI than building a video game.
- Building a powerful theorem prover is no more AI than building an equation solver.
- Building an "intelligent" job-shop scheduler is no more AI than building a normal job-shop scheduler

Saying that these things are not AI has nothing to do with whether they are worth doing. Being AI is not better or worse than not being AI. The goal of defining AI is not to make value judgments, but to 1) explain to ourselves and others what the point of AI is; 2) explain what makes the field coherent, i.e., how AI people working on very different problems can actually have something to say to each other; and 3) provide a vision as to where AI should go next.

With this definition, the answer to (1) is that as computers become ubiquitous in modern society, so do the effects of computer stupidity. More and more people are encountering stupidity in word processors, spreadsheets, payroll programs, Web browsers, educational software, and so on. The costs and dangers of computer stupidity are increasing daily. The point of AI, then, is to reduce those costs and dangers.

The answer to (2) is that the solutions to the many examples of stupidity come down to a small number of common basic techniques, such as explicit knowledge structures and case memory.

The answer to (3) is the topic of the next section.

What's the Future of AI?

I have a vision of *post-modern AI.* Obviously, this presumes a definition of modern AI. Fortunately, such a thing exists.

Modern AI: Intelligent Agents

In 1995, Russell and Norvig's *Artificial Intelligence: A Modern Approach* appeared, with the following statement:

> The unifying theme of the book is the concept of an *intelligent agent.* In this view, the problem of AI is to describe and build agents that receive percepts from the environment and perform actions (p. vii, [italics in original]).

Russell and Norvig use this theme to structure their textbook from beginning to end. Rather than simply reviewing the standard sequence of topics (search, knowledge representation, rule-based inference, etc.), everything is described in terms of how it can help build an autonomous intelligent agent.

Intelligent agents are indeed what most people, both inside and outside the field, probably view AI as being about. Robot vehicles, expert systems, chess players, automated text understanders, even software robots cruising the 'net—these all seem to qualify as intelligent agents. There's still that tricky bit about "intelligence," but we could reword the above to be "the problem of AI is to describe and build agents that aren't stupid." So let's assume that describing and building intelligent agents defines modern AI. What's wrong with that as a unifying goal for a field? Why do we need post-modern AI?

Problems with Intelligent Agents

I claim that there are two primary problems with making the development of autonomous intelligent agents the primary goal for AI:

- Intelligent agents are so far off that the goal doesn't help decision making in the here and now.
- Intelligent agents are not even what we want computers to be for a great many situations.

The goal of autonomous intelligent agents is too distant to drive near-term research and development. Viewed from the standpoint of this ultimate goal, all hard AI problems are of equal importance. It provides no guidance, no roadmap in trying to determine where the next big push should go. As a result, research decisions are made on the basis of where we are, rather than we want to be next. Someone with an inference engine will work on making it faster, adding new capabilities to it, or finding a new application for it. Someone with a natural language parser will make similar choices for it. It's hard to know what's really the best problem to tackle next, because the ultimate goal is so far off.

The intelligent agent theme is also inappropriate because it ignores a key difference between computers and people: computer programs rarely act as independent agents. They are components in larger systems in ways that people never are, except metaphorically. A computer in a car or a server on a network is not a free-standing entity. The communication protocols are far more rigid, and requests for real-time service far more demanding, than anything people can handle, or that an intelligent autonomous agent should want to put up with.

But if intelligent agents are both a distant and somewhat unnecessary goal, what should be the driving goal for AI?

Post-Modern AI: Intelligent Components

Following the textual form of Russell and Norvig, here is my theme for post-modern AI:

> The unifying theme of this chapter is the concept of *intelligent components*. In this view, the problem of AI is to describe and build components that reduce the stupidity of the systems in which they function.

That is, the goal should not be intelligent systems. The goal should be the improvement of how systems function through the development of intelligent parts to those systems. For example, I don't want an automated librarian. I want a library search program that isn't stupid. I want one that knows concepts, not keywords, that won't find James Cook when I'm researching great chefs in history. Central problems in post-modern AI are cases of stupidity that are common and severe across a large class of systems.

Though I've not seen the term "post-modern AI" before, much of this alternative view of AI can be found in other writings. For example, Chandrasekaran wrote the following in an editorial for *IEEE Expert* "AI, Knowledge, and the Quest for Smart Systems:"

> The public's definition of a smart product has nothing to do with ... the AI community's long-held goal of developing a general purpose intelligence. The public does not expect a smart system to do everything that people do. It does expect a smart product to be flexible, adaptive, and robust (p. 2).

If we differ, it would be on two points. First, I claim that the proper adjective isn't "smart" but "not stupid." For many reasons, some rational, some not, people don't want machines to be smart, but neither do they want them to be stupid. Second, I claim that this is not just a change in how AI should be applied in the real world, but in how the field of AI should define its ultimate goals and select strategies for achieving those goals.

Another point that distinguishes post-modern AI from modern AI is what each area means by "integration." Consider figure 1, taken from an article on an expert system for a data acquisition and control system for an electrical utility (Pfau-Wagenbauer 1993). The original caption said "The expert system *integrated* with Scada" [italics mine]. In the United States, this kind of integration leads to school bussing. That is, the systems reside in "separate but equal" areas, but they don't really have to talk to each other very much. This is exactly the level of integration that one would expect if the expert system is an autonomous intelligent agent. Integration for agents means "communicates with."

In post-modern AI, the AI becomes an invisible part of the overall system. In this, I am inspired in part by a vision articulated by Allen Newell at a panel, held at IJCAI–81, celebrating the twenty-fifth anniversary of the 1956 Dartmouth Conference. At one point, the participants were asked to predict the future of AI. Most gave the standard answer: intelligent thinking computers.

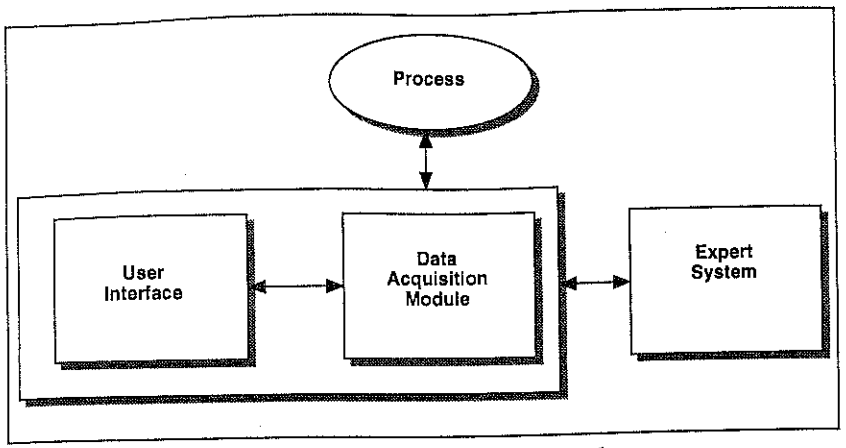


Figure 1. Integrating intelligent agents into other systems. (Pfau-Wagenbauer, 1995, p. 13)

Newell had a different view. He envisioned not a single intelligent entity, but "a cognitive city," where traffic lights understood the flow of traffic and street lamps knew when people stood below them. His examples, at least as I remember them, are paradigmatic cases for post-modern AI. The goal is not "smart" appliances and cars that talk to us. The goal is street lamps that don't waste electricity on totally deserted sidewalks, and traffic lights that don't turn green for streets closed for construction. That is, the primary goal is to develop systems that aren't stupid, not systems that are intelligent. The intelligence that makes systems not stupid will be as unremarkable in those systems as it is in people.

Examples of Intelligent Components

I will now describe a few examples of intelligent components in systems built at the Institute for the Learning Sciences, in order to illustrate the differences between the intelligent agent and intelligent component approach, some of the different demands intelligent components must deal with, and some of the techniques that work well for intelligent components. The third item will bring us to the future of CBR.

The Casper Parsing Component

The Casper system was developed at the Institute for the Learning Sciences for North West Water, a privatized British water utility, to teach their customer service representatives (CSRs) how to diagnose water quality problems (Kass 1994a).

Casper was built by first designing a number of scenarios, e.g., a customer with a water problem caused by rust in a hot water tank, another with a problem caused by the mains (hydrants) being flushed recently, and so on. Then content analysts determined all the relevant questions, both good and bad, that a novice CSR might ask in those scenarios. Finally, actors playing customers recorded answers to those questions. A scenario usually has several hundred such question-answer pairs.

A student, playing the part of a customer service representative, interacts with the pre-recorded customers. To ask a question, a student can use a menu-based "question constructor." This is however somewhat unnatural and interrupts the flow of the dialog. Alternatively, the student can simply type what they want to say. Casper then lists the questions it has that best match the student's input, and the student either picks one, tries again, or uses the menus.

The student input is "understood" by an *indexed concept parser* (Fitzgerald, 1995), an intelligent component that works as follows: 1) The stored questions are indexed in advance by the sets of the basic concepts those questions refer to. 2) The student input text is mapped to a set of index concepts. 3) The stored questions are sorted by how well their index sets overlap with the input index set, and the best matches are presented. Full details appear in Fitzgerald (1995).

The key points for our purposes are as follows.

There's enough AI to avoid being stupid:

- Matching is done on concepts, not words
- Ambiguous words and phrases are handled smoothly
- Matching takes into account ISA relationships between index concepts
- The scoring algorithm gives more weight to matches between less commonly seen concepts.

There's only *a little AI here:*

- The concepts and stored questions have no internal structure.

The needs and capabilities of the system determine the scope and power of the parser:

- The system needs the stored question closest to what the student wanted, therefore, the job of the parser is to find that question, and no more.
- The system needs only very simple concept structures and inference rules to connect student actions to scenario events and tutoring responses, therefore, the parser must make do with very limited knowledge representations.
- The system is a feasible solution only if content analysts, not programmers, can maintain and add new scenarios easily and quickly, therefore, the pars-

```
IF the student makes a diagnosis of the problem
   AND there is not enough evidence for it
   THEN
      1. Ask the student to justify his or her diagnosis.
      2. Explain why the diagnosis is premature.
      3. Ask the student to retract the diagnosis statement.
      4. Help the student with the next problem-solving step.
```

Figure 2. A critiquing rule example, summarized in English.

er has to be no harder to maintain than the rest of the system.

In short, on the one hand, the job of the parsing component is made harder by the fact that Casper needs reliable and robust handling of real user inputs, with serious limitations on how much knowledge is present in the system. On the other hand, the job of the parsing component is made simpler by the fact that the system requires only the selection of stored questions, not the generation of novel meaning structures.

The Casper Critiquer

There are many kinds of mistakes students can make in Casper. Some are mistakes in reasoning, such as coming to a conclusion unjustified or even inconsistent with the existing evidence. Some are mistakes in procedure, such as asking leading questions like "Is the water tea colored?" or performing some action before it makes sense.

Responses to classic student mistakes are generated in advance by content experts. These responses consist of text and video commentary on various actions, as well as stories of what happened when CSR's made similar mistakes in real life.

Mistakes are recognized by the Casper tutoring module, an intelligent component that selects an appropriate pre-defined response, based on the type of mistake, and where in the task it occurred. An example of a critiquing rule, summarized in English, is depicted in figure 2. Full details appear in Jona (1995).

The key points for our purposes are:

There's enough AI to avoid being stupid:

- Causal and diagnostic rules make the obvious connections, e.g., "rusty pipes cause rust flakes in water which causes brownish water," and "black bits in water implies possible flaking lead pipes."
- Evidence rules catch the obvious mistakes in hypothesis formation, such as failing to eliminate all other likely possibilities, or having only weak evidence for a conclusion.

There's only *a little AI here:*

- The critiques are basically canned templates.

The needs and capabilities of the system determine the scope and power of the critiquer:

- The system needs the most relevant of the pre-defined critiques retrieved at the right time. Therefore, the critiquer doesn't have to generate critiques, but it does have to avoid finding obviously irrelevant ones.
- The critiquing rules must be maintainable by the same people who maintain the rest of the system.

In short, on the one hand, the critiquing component has to be reliable and robust enough to help, rather than annoy. On the other hand, the critiquing component does not have to do anything more than what Casper needs, which is to point out mistakes that Casper knows about, and get the student back on track.

Casper differs from a classic intelligent tutoring system in several ways. First, Casper is not a model of a tutor, but of an environment. The environment has been modified to make learning easier, but the simulated environment remains the focal point. Second, Casper is not a domain expert. Casper has a crude causal model of the domain and of the diagnostic task. Finally, Casper does not try to model the student. It has only a simple ontology of actions and mistakes. In short, the *system* Casper is not intelligent. It does, however, have some not-so-stupid components, namely the parser and the critiquer.

The Creanimate Parser

Creanimate (Edelson 1995) is a program developed to get children to understand how the particular features and behaviors of different animals relate to the kinds of goals those animals have. Creanimate does this by engaging a child in a dialog about designing a new animal, based on modifying some existing animal, and showing lots of short, interesting videos about animals and the things they do.

One component of Creanimate is a DMAP parser (Martin 1993) that maps short phrases typed by students to internal frame structures, e.g., from "find female spiders" to find-mate (Fitzgerald 1995). As in Casper, the job of the parser isn't to construct meaning structures, but to find the most relevant existing concepts in memory. Full details appear in Edelson (1993) and Fitzgerald (1995). The key points for our purposes are:

There's enough AI to avoid being stupid:

- The parser can map the words such as "chase after gazelles" to the underlying concepts for hunting gazelles.

- Taxonomic relationships link points such as using speed to catch gazelles to using speed to catch prey in general.

There's only *a little AI here:*

- The videos are "black boxes" to the rest of the system. There's no image processing or detailed representation of the events in the videos.
- There's no model of the student.

The needs and capabilities of the system determine the scope and power of the parser:

- All "understanding" means here is finding the point stored in Creanimate closest enough to what the student meant to keep the dialog coherent.
- The parser must be maintainable by the same people who maintain the rest of the system.

In short, on the one hand, the parsing component has to be reliable and robust enough not to interfere with the flow of the dialog. On the other hand, the parsing component does not have to do any more than get to something Creanimate can talk about that's consistent with what the student just said.

Select and Adapt: The Secret of Intelligent Components

On the one hand, life is hard for intelligent component designers. They're not in control of what the component gets, what it has to produce, or how fast it has to produce it. Even worse, a component can only require a level of engineering and maintenance work commensurate with the value the component adds to the system as a whole. For example, the lexicon and knowledge representation aspects of the parsing components in Casper and Creanimate had to be relatively easy to engineer and maintain, because parsing was not a central component of those systems. Creanimate represented knowledge with a hierarchical frame system that the parser could take advantage of. Casper did not need such sophisticated representations. Therefore a parsing component had to be designed that could make do with less, because the effort and skills required to create such representations was more than the parsing component was worth.

On the other hand, designing an intelligent component can be orders of magnitude simpler than designing an intelligent agent, because of strong limitations in the needs and capabilities of the rest of the system. Thus, the parsing component in Creanimate didn't have to understand everything a kid could type, only those things that referred to concepts that Creanimate knew about. The parsing component in Casper didn't have to understand everything a customer service representative might ask, only those questions that Casper was prepared to handle.

Recognizing and taking advantage of the limits and capabilities of the bigger system is the secret of building intelligent components that are robust and

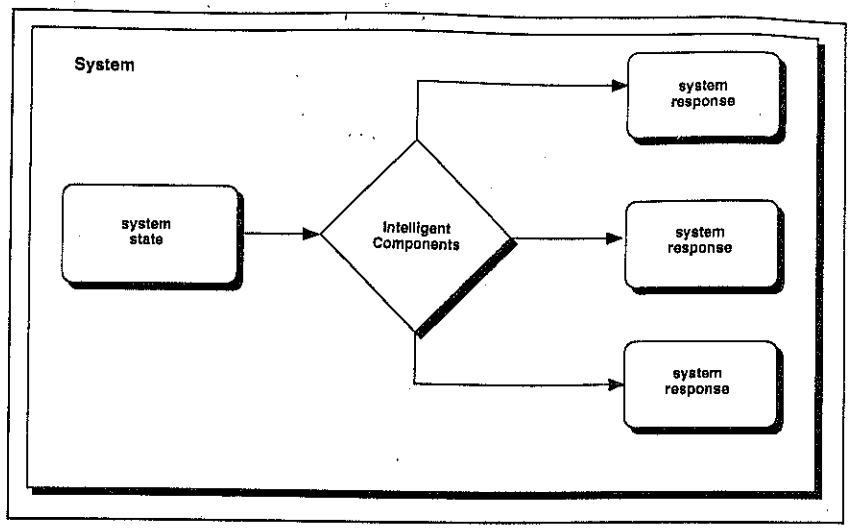


Figure 3. Integrating intelligent components into other systems.

successful. Artificial intelligence, like linguistics and computer science, has too long been concerned with being able to generate potentially infinite sets of responses In fact, many real systems are like Casper and Creanimate. They have very few things they can do. The problem is getting from a very large number of situations to the most appropriate response.

The key problem for systems like Casper and Creanimate is *selection,* not construction. That is, the job of the parser was not to construct a meaning, but to help the system select the most appropriate pre-defined response template, from a relatively small set. The job of the critiquer was to select the most appropriate pre-defined critiquing templates.

Figure 3 shows how an intelligent component is integrated, and is intended to contrast directly with figure 1. In figure 3, the component is an integral part of the larger system. The job of the component is to help the system choose a response good enough to meet the system's needs in a timely fashion. Figure 3 is, I believe, an accurate abstraction of how the parsers and critiquing components discussed earlier are integrated into Casper and Creanimate.

Select and Adapt Versus Generate and Test

There are two arguments for select and adapt rather than the classic AI paradigm of generate and test for intelligent components.

First, many systems have a finite set of possible response branches that they can follow, e.g., *x, y* and *z.* If a component generates (constructs) representational structures, then the rest of the system has to map those structures to

branch *x*, *y* or *z*. Such mapping is extra run-time work for the system, and extra build-time work for the system maintainers. The rules or tables that perform the mapping are one more place where things can break, and one more reason not to use "that AI stuff."

An intelligent component that *generates* responses, in other words, forces the rest of the system to become smarter, in order to understand those responses. A component that *selects* from the responses built into the system adds no extra effort to the rest of the system.

The second problem with generate and test is one of timeliness. It's often hard to put time-bounds on generative processes. Two general methods of generation are 1) assembly from parts, and 2) refinement from templates. If you prematurely stop either process, you get an incomplete, nonfunctional solution.

A selection processes, on the other hand, can be designed to start with a default complete answer. The answer may be wrong, but nothing is missing. Given more time, it can be replaced with better solutions, or the bad parts modified. At any point in time, there's an answer available. One might call this a "shoot first and ask questions later" approach.

Chess playing programs (whether they're AI or not) have this capability because they're always selecting moves from the set of possible legal moves. If you stop a chess program early, there's some move it can give that's the best choice so far. You could write a chess program that started with strategic goals (control center, reduce threat), and refined them into particular moves, but then you would no longer be able to interrupt it and ask it to move immediately.

In order for select and adapt to avoid the same problems as generate and test, adaptation has to be carefully limited. Many CBR systems avoid adaptation entirely. They are problem solving assistants that simply retrieve relevant prior examples for a human user to consider when solving some problem. The Casper and Creanimate systems also needed no adaptation after retrieving a stored question or concept. The Casper critiquer and the Creanimate dialog manager only had to instantiate text templates to create bridging introductions to the canned videos.

Ideally, then, two properties should be true of select and adapt algorithms used in intelligent components:

- The selection process should be an "anytime" algorithm (Dean and Boddy 1988) that quickly retrieves a real answer. It can replace that answer later with a better one, but it's never at a loss for some answer.
- The adaptation process, if needed, should be quick and never leave the adapted answer in an unusable state for very long.

CBR is a Select and Adapt Algorithm

CBR systems inspired by research on human reminding, i.e., the truest of the true CBR systems, typically have an "anytime" capability. That is, they find some answer almost immediately, then, given more time, they adapt it, or replace it with a better reminding. Examples of such systems include MEDIATOR (Kolodner and Simpson 1989), Protos (Bareiss 1989a) and Swale (Schank et al. 1994).

Saving and reusing adapted answers helps to overcome the strong limitations on adaptation just described. Even though any particular adaptation episode may be limited because of time and resource constraints, over time better and better answers are constructed, because later adaptations begin with the results of previous adaptations and repairs.

An early example of this is in CHEF (Hammond 1989c). CHEF's adaptation rules were probably more complex than appropriate for a small intelligent component, but they were still quite limited. Most plan step interactions were simply not recognized, such as the fact that stir frying beef and broccoli together might lead to soggy broccoli because of the water generated from the cooking beef. However, when those interactions led to execution failures, the repaired recipes, e.g., cook beef and broccoli separately, were stored in memory and made available to later problem solving situations. Thus, even though the adaptation process in CHEF was too limited to catch such interactions in general, common interactions were learned and added to the system's repertoire.

Case-based Intelligent Components

Case-based intelligent components, as illustrated in figure 4, select (and optionally adapt) system responses from an indexed store of response selections that is dynamically extended as the system runs over time.

I believe that there are some simple ways in which practical case-based components can be developed right now, but that the role of intelligent component also suggests some significant research problems. In the short term, I see three feasible kinds of case-based intelligent components: First, generalized situation-response caches, using surface feature indices to provide simple caching of answers to apply old answers to new situations that "any moron" would see are the same. Second, case memories for knowledge-based intelligent systems, including rule-based ones, using the deep feature indices already present in the larger system. Third, embedded browsable case bases, using topic indices and inter-case cross-links to support user-driven retrieval.

Situation-Response Caches

Caching responses to commonly asked requests is a well-known technique that is common in low-level processes, like disk accesses, but under-used in higher-level software. Caching reduces a major form of stupidity, namely redoing the same work every time a problem is solved, no matter how many times that kind of problem has already been solved.

One reason why caching is not used in systems supporting human problem solving is that as situations become more complex, the likelihood decreases of

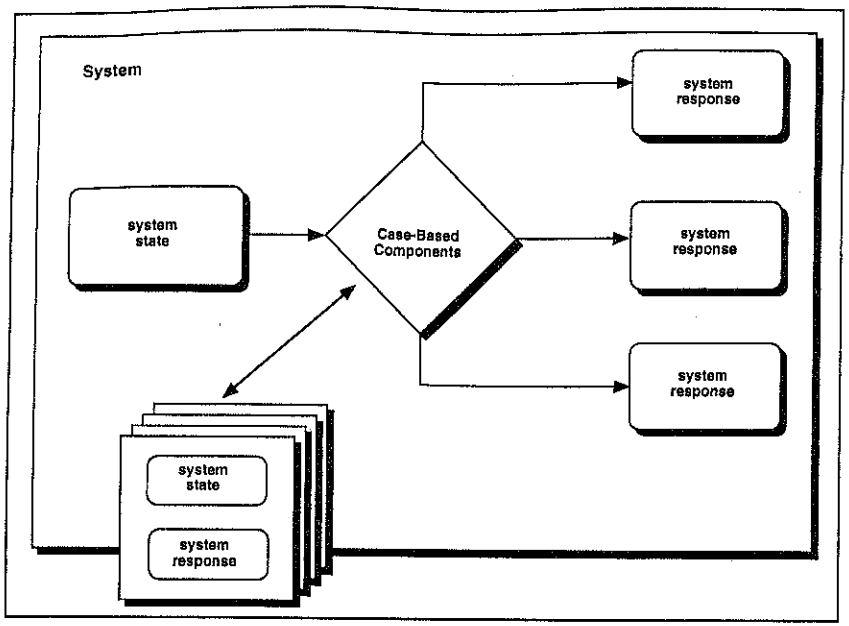


Figure 4. A case-based intelligent component.

seeing exactly the same situation again. This of course is what partial matching and adaptation in CBR is supposed to solve. Often the differences between two situations can be inferred to be irrelevant, using just a little bit of knowledge.

The inferencing needed to do the partial matching (and adaptation, if required) must be limited, otherwise the development and maintenance costs of adding a case-based cache will outweigh the benefits. For this reason, I think that approaches using surface features are currently the most appropriate. Several inductive techniques for case-based learning are discussed in Aha (1991), and a partial matcher for dealing with noise in DNA sequences is described in Shavlik (1991) These and similar approaches allow cases to be stored and retrieved with very little knowledge engineering effort. They extend exact-match caching to handle situations that are "obviously" the same. Such approaches are of course limited to very narrowly scoped case bases, but this is often exactly the kind of case base a larger system is generating.

Case Bases for Intelligent Systems

Case bases for intelligent systems are feasible because the representational work has already been done for the rest of the system. Adding the CBR component does not significantly increase knowledge maintenance needs. An early example of an adjunct system is Casey (Koton 1988c), where a CBR system ran in conjunction with a model-based diagnostic system.

One potential stupidity in this kind of system is spending more time adapting a retrieved case than would have been required to solve it using the rules. JUDGE (Riesbeck and Schank 1989) and PRODIGY (Veloso and Carbonell 1991) give two very different approaches to guessing when it isn't worth using a case.

Browsable Case Bases

Browsable case bases achieve feasibility by paring the notion of CBR to the bone. ASK systems (Ferguson et al. 1991), for example, are browsable corporate memories in which there is not only no adaptation, but no retrieval as well! Instead, there's a case-base, partially indexed with topics and richly indexed with inter-case links. The end user uses the topics to "zoom" to an initial case and then follows the links to other cases. The links in ASK systems are based on conversational coherence principles, but intercase links of some form are part of many early true CBR systems, such as MEDIATOR (Kolodner and Simpson 1989) and Protos (Bareiss 1989a).

ASK systems provide some of the "job aid" memory of a case-based retriever, in a form that allows systems to be built and maintained by content analysts, rather than AI programmers or knowledge engineers.

Research Goals for Case-Based Intelligent Components

Looking to the future, I see several research areas for CBR relevant to making CBR feasible for intelligent components in non-intelligent systems.

Indexing

Indexing in CBR tends to be either a major effort or almost no effort at all. Systems using surface features, as described above, require little knowledge engineering, but don't support cases from multiple domains. Systems like Swale (Schank et al. 1994) use fairly complex, inferred features as case indices, in order to support retrieval of the most relevant case from very disparate domains. This of course requires significant representational effort.

A research area then is broadening the range of case bases that can be indexed without incurring development and maintenance costs greater than the value added by the broader range. Such research includes the development of well-defined indexing methodologies, to lower the costs of developing and applying indexing vocabularies; libraries of indexing vocabularies, (and here I believe that many libraries of specific vocabularies will be more useful than a few libraries of very abstract concepts); and semi-automated indexing assistants, to assist indexers in applying indices to large case bases. One example is described in Osgood and Bareiss (1993).

Adaptation

Adaptation has always had a mixed status in CBR. On the one hand, adaptation is the "reasoning" part of "case-based reasoning." Furthermore, most early CBR work focussed on the development and application of adaptation strategies, such as parameterization and abstraction/respecialization (Riesbeck and Schank 1989). On the other hand, adaptation is usually the weak link in a CBR system. Adaptation techniques are hard to generalize, hard to implement, and quick to break. Furthermore, adaptation is often unnecessary. The originally retrieved case is often as useful to a human as any half-baked adaptation of it.

For intelligent components, adaptation techniques have to be far more robust than they currently are, far easier to define and support, and of far greater value to the system as a whole. Furthermore, the techniques have to work incrementally, so that there's an answer available any time one is asked for. For example, an adaptation technique that removes all details specific to the old situation before replacing them with details from the new situation would not be suitable, because the partially empty case would not be a usable intermediate answer.

A research area then is the development of incremental adaptation techniques. As with indexing vocabularies, I personally believe that developing libraries of fairly specific techniques is of greatest value.

An interesting approach to try is splitting adaptation techniques into quick fixers that rapidly fix problems in a retrieved case, and optimizers that remove inefficiencies in the results produced by the quick fixers. For example, when a recipe that chops vegetable is applied to situation calling for two vegetables, the quick fixer creates a recipe with two chopping steps. The quick fixer then calls its associated optimizer to see if the two chopping steps can be merged. The key point though is that the recipe with two steps is available for use, if the rest of the system needs it. The quick fixer makes sure that the adapter isn't so "stupid" that it doesn't even realize it has to chop two ingredients.

Conclusion

The argument above can be summarized as follows:

- AI should focus on the development of intelligent components rather than intelligent agents, because 1) we need systems that aren't stupid more than systems that are smart, and 2) we need nearer-term objectives than autonomous intelligent agents to focus current AI research strategies.
- Select and adapt is a better control structure for intelligent agents than generate and test.
- CBR's future is in the development of intelligent components that select and adapt true cases from dynamic memories.

18 Bibliography

Aamodt, A. 1994. Explanation-Driven Case-Based Reasoning. In *Topics in Case-Based Reasoning*, eds. S. Wess, K. Althoff, and M. Richter, 274–288. Berlin: Springer Verlag.

Aamodt, A., and Plaza, E. 1994. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications* 7(1): 39–52.

Acorn, T., and Walden, S. 1992. SMART: Support Management Automated Reasoning Technology for Compaq Customer Service. In *Innovative Applications of Artificial Intelligence 4*, 3–18. Menlo Park, Calif.: AAAI Press.

Aha, D., ed. 1996. *Artificial Intelligence Review* (Special Issue on Lazy Learning). Norwell, Mass.: Kluwer. Forthcoming.

Aha, D., ed. 1994. *Proceedings of the AAAI-94 Workshop on Case-Based Reasoning.* Technical Report WS-94-01. Menlo Park, Calif.: AAAI Press.

Aha, D. W. 1992. Generalizing from Case Studies: A Case Study. In Proceedings of the Ninth International Conference on Machine Learning, 1–10. San Francisco, Calif.: Morgan Kaufmann.

Aha, D. 1991. Case-Based Learning Algorithms. *Proceedings of the DARPA Case-Based Reasoning Workshop,* 147-158. San Francisco: Morgan Kaufmann.

Aha, D., and Branting, K. 1995. Stratified Case-Based Reasoning: Reusing Hierarchical Problem Solving Episodes. In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 384–390. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Aha, D., and Ram, A. 1995. Preface. Adaptation of Knowledge for Reuse: Papers from the 1994 Fall Symposium, v. Technical Report FS-95-04, American Association for Artificial Intelligence, Menlo Park, Calif.

Aha, D.; Kibler, D.; and Albert, M. 1991. Instance-Based Learning Algorithms. *Machine Learning* 6:37–66.

Allemang, D. 1993. Review of the First European Workshop on Case-Based Reasoning (EWCBR-93). *Case-Based Reasoning Newsletter* 2(3), December 22.

Alterman, R. 1988. Adaptive Planning. *Cognitive Science* 12:393–422.

Alterman, R. 1986. An Adaptive Planner. In Proceedings of the Fifth National Conference on Artificial Intelligence, 65–69. Menlo Park, Calif.: American Association for Artificial Intelligence.

Althoff, K.-D., and Wess, S. 1991. Case-Based Knowledge Acquisition, Learning, and Problem Solving for Diagnostic Real-World Tasks. In Proceedings of the European Knowledge Acquisition Workshop, 48–67. Glasgow, Scotland: EKAW.

7th Int. Conf. CBR, Belfast, UK
August 2007 164-178

Case-Based Planning and Execution for Real-Time Strategy Games

Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram

CCL, Cognitive Computing Lab
Georgia Institute of Technology
Atlanta, GA 303322/0280
{santi,kinshuk,nsugandh,ashwin}@cc.gatech.edu

Abstract. Artificial Intelligence techniques have been successfully applied to several computer games. However in some kinds of computer games, like real-time strategy (RTS) games, traditional artificial intelligence techniques fail to play at a human level because of the vast search spaces that they entail. In this paper we present a real-time case based planning and execution approach designed to deal with RTS games. We propose to extract behavioral knowledge from expert demonstrations in form of individual cases. This knowledge can be reused via a case based behavior generator that proposes behaviors to achieve the specific open goals in the current plan. Specifically, we applied our technique to the WARGUS domain with promising results.

Introduction

Artificial Intelligence (AI) techniques have been successfully applied to several computer games. However, in the vast majority of computer games traditional AI techniques fail to play at a human level because of the characteristics of the game. Most current commercial computer games have vast search spaces in which the AI has to make decisions in real-time, thus rendering traditional search based techniques inapplicable. For that reason, game developers need to spend a big effort in hand coding specific strategies that play at a reasonable level for each new game. One of the long term goals of our research is to develop artificial intelligence techniques that can be directly applied to such domains, alleviating the effort required by game developers to include advanced AI in their games.

Specifically, we are interested in real-time strategy (RTS) games, that have been shown to have huge decision spaces that cannot be dealt with search based AI techniques [2,3]. In this paper we will present a case-based planning architecture that integrates planning and execution and is capable of dealing with both the vast decision spaces and the real-time component of RTS games. Moreover, applying case-based planning to RTS games requires a set of cases with which to construct plans. To deal with this issue, we propose to extract behavioral knowledge from expert demonstrations (i.e. an expert plays the game and our system observes), and store it in the form of cases. Then, at performance time

the system will retrieve the most adequate behaviors observed from the expert and will adapt them to the situation at hand.

As we said before, one of the main goals of our research is to create AI techniques that can be used by game manufacturers to reduce the effort required to develop the AI component of their games. Developing the AI behavior for an automated agent that plays a RTS is not an easy task, and requires a large coding and debugging effort. Using the architecture presented in this paper the game developers will be able to specify the AI behavior just by demonstration; i.e. instead of having to code the behavior using a programming language, the behavior can be specified simply by *demonstrating* it to the system. If the system shows an incorrect behavior in any particular situation, instead of having to find the bug in the program and fix it, the game developers can simply demonstrate the correct action in the particular situation. The system will then incorporate that information in its case base and will behave better in the future.

Another contribution of the work presented in this paper is on presenting an integrated architecture for case-based planning and execution. In our architecture, plan retrieval, composition, adaptation, and execution are interleaved. The planner keeps track of all the open goals in the current plan (initially, the system starts with the goal of winning the game), and for each open goal, the system retrieves the most adequate behavior in the case base depending on the current game state. This behavior is then added into the current plan. When a particular behavior has to be executed, it is adapted to match the current game state and then it is executed. Moreover, each individual action or sub-plan inside the plan is constantly monitored for success or failure. When a failure occurs, the system attempts to retrieve a better behavior from the case base. This interleaved process of case based planning and execution allows the system to reuse the behaviors extracted from the expert and apply them to play the game.

The rest of the paper is organized as follows. Section 2 presents a summary of related work. Then, Section 3 introduces the proposed architecture and its main modules. After that, Section 4 briefly explains the behavior representation language used in our architecture. Section 5 explains the case extraction process. Then sections 6 and 7 present the planning module and the case based reasoning module respectively. Section 8 summarizes our experiments. Finally, the paper finishes with the conclusions section.

2 Related Work

Concerning the application of case-based reasoning techniques to computer games, Aha et al. [2] developed a case-based plan selection technique that learns how to select an appropriate strategy for each particular situation in the game of WARGUS. In their work, they have a library of previously encoded strategies, and the system learns which one of them is better for each game phase. In addition, they perform an interesting analysis on the complexity of real-time strategy games (focusing on WARGUS in particular). Another application of case based reasoning to real-time strategy games is that of Sharma et al. [15], where they

esent a hybrid case based reinforcement learning approach able to learn which e the best actions to apply in each situation (from a set of high level actions). ıe main difference between their work and ours is that they learn a case selec- ın policy, while our system constructs plans from the individual cases it has in e case base. Moreover, our architecture automatically extracts the plans from serving a human rather than having them coded in advance.

Ponsen et al [14] developed a hybrid evolutionary and reinforcement learning :ategy for automatically generating strategies for the game of WARGUS. In eir framework, they construct a set of rules using an evolutionary approach ach rule determines what to do in a set of particular situations). Then they use reinforcement learning technique called *dynamic scripting* to select a subset these evolved rules that achieve a good performance when playing the game. ıere are several differences between their approach and ours. First, they focus automatically generating strategies while we focus on acquiring them from expert. Moreover, each of their individual rules could be compared to one of r behaviors, but the difference is that their strategies are combined in a pure ıctive way, while our strategies are combined using a planning approach. For r planner to achieve that, we require each individual behavior to be annotated th the goal it pursues.

Hoang et al. [9] propose to use a hierarchical plan representation to encode :ategic game AI. In their work, they use HTN planning (inside the framework of ɔal-Oriented Action Planning [13]). Further, in [11] Muñoz and Aha propose way to use case based planning to the same HTN framework to deal with :ategy games. Moreover, they point out that case based reasoning provides way to generate explanations on the decisions (i.e. plans) generated by the stem. The HTN framework is very related to the work presented in this paper, ıere we use the task-method decomposition to represent plans. Moreover, in eir work they focus on the planing aspects of the problem while in this paper focus on the learning aspects of the problem, i.e. how to learn from expert monstrations.

The work presented in this paper is strongly related to existing work in case- sed planning [8]. Case Based Planning work is based on the idea of planning remembering instead of planning from scratch. Thus, a case based planner tains the plans it generates to reuse them in the future, uses planning failures opportunities for learning, and tries to retrieve plans in the past that satisfy many of the current goals as possible. Specifically, our work focuses on an :egrated planning and execution architecture, in which there has been little ɔrk in the case based planning community. A sample of such work is that Freßmann et al. [6], where they combine CBR with multi-agent systems to tomate the configuration and execution of workflows that have to be executed multiple agents.

Integrating planning and execution has been studied in the search based plan- ng community. For example, CPEF [12] is a framework for continuous planning d execution. CPEF shares a common assumption with our work, namely that ans are dynamic artifacts that must evolve with the changing environment in

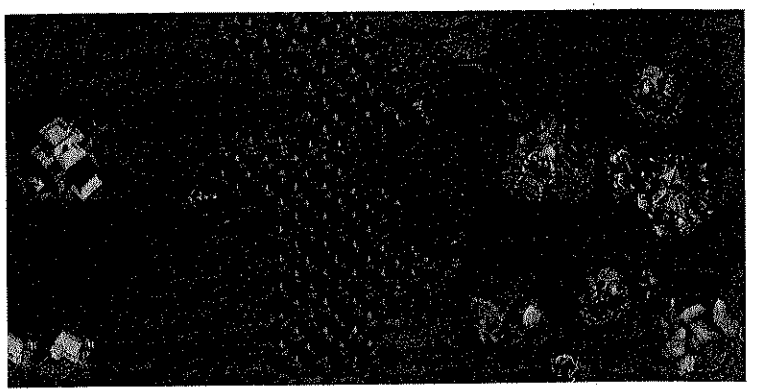


Fig. 1. A screenshot of the WARGUS game

which they are executing changes. However, the main difference is that in our approach we are interested in case based planning processes that are able to deal with the huge complexity of our application domain.

3 Case-Based Planning in WARGUS

WARGUS (Figure 1) is a real-time strategy game where each player's goal is to remain alive after destroying the rest of the players. Each player has a series of troops and buildings and gathers resources (gold, wood and oil) in order to produce more troops and buildings. Buildings are required to produce more advanced troops, and troops are required to attack the enemy. In addition, players can also build defensive buildings such as walls and towers. Therefore, WARGUS involves complex reasoning to determine where, when and which buildings and troops to build. For example, the map shown in Figure 1 is a 2-player version of the classical map "Nowhere to run nowhere to hide", with a wall of trees that separates the players. This maps leads to complex strategic reasoning, such as building long range units (such as catapults or ballistas) to attack the other player before the wall of trees has been destroyed, or tunneling early in the game through the wall of trees trying to catch the enemy by surprise.

Traditionally, games such as WARGUS use handcrafted behaviors for the built-in AI. Creating such behaviors requires a lot of effort, and even after that, the result is that the built-in AI is static and easy to beat (since humans can easily find holes in the computer strategy). The goal of the work presented in this paper is to ease the task of the game developers to create behaviors for these games, and to make them more adaptive. Our approach involves learning behaviors from expert demonstrations to reduce the effort of coding the behaviors, and use the learned behaviors inside a case-based planning system to reuse them for new situations. Figure 2 shows an overview of our case-based planning approach. Basically, we divide the process in two main stages:

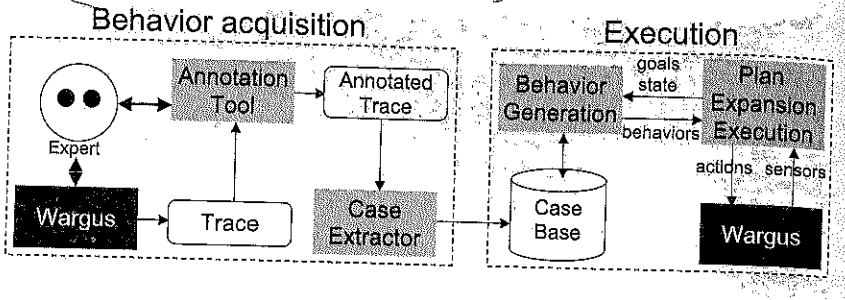


Fig. 2. Overview of the proposed case-based planning approach

- *Behavior acquisition*: During this first stage, an expert plays a game of WARGUS and the trace of that game is stored. Then, the expert annotates the trace explaining the goals he was pursuing with the actions he took while playing. Using those annotations, a set of behaviors are extracted from the trace and stored as a set of cases. Each case is a triple: situation/goal/behavior, representing that the expert used a particular behavior to achieve a certain goal in a particular situation.
- *Execution*: The execution engine consists of two main modules, a real-time plan expansion and execution (RTEE) module and a behavior generation (BG) module. The RTEE module maintains an execution tree of the current active goals and subgoals and which behaviors are being executed to achieve each of the goals. Each time there is an open goal, the RTEE queries the BG module to generate a behavior to solve it. The BG then retrieves the most appropriate behavior from its case base, and sends it to the RTEE. Finally, when the RTEE is about to start executing a behavior, it is sent back to the BG module for adaptation. Notice that this *delayed adaptation* is a key feature different from traditional CBR required for real-time domains where the environment continuously changes.

In the following sections we will present each of the individual components of r architecture.

A Behavior Reasoning Language

this section we will present the Behavior Reasoning Language used in our proach, designed to allow a system to learn behaviors, represent them, and to son about the behaviors and their intended and actual effects. Our language es ideas from the STRIPS [5] planning language, and from the ABL [10] iavior language, and further develops them to allow advanced reasoning and rning capabilities over the behavior language.

The basic constituent piece is the *behavior*. A behavior has two main parts: *eclarative* part and a *procedural* part. The declarative part has the purpose oroviding information to the system about the intended use of the behavior,

and the procedural part contains the executable behavior itself. The declarative part of a behavior consists of three parts:

- A *goal*, that is a representation of the intended goal of the behavior. For every domain, an ontology of possible goals has to be defined. For instance, a behavior might have the goal of "having a tower".
- A set of *preconditions* that must be satisfied before the behavior can be executed. For instance, a behavior can have as preconditions that a particular peasant exists and that a desired location is empty.
- A set of *alive conditions* that represent the conditions that must be satisfied during the execution of the behavior for it to have chances of success. If at some moment during the execution, the alive conditions are not met, the behavior can be stopped, since it will not achieve its intended goal. For instance, the peasant in charge of building a building must remain alive; if he is killed, the building will not be built.

Notice that unlike classical planning approaches, postconditions cannot be specified for behaviors, since a behavior is not guaranteed to succeed. Thus, we can only specify what goal a behavior pursues.

The procedural part of a behavior consists of executable code that can contain the following constructs: *sequence*, *parallel*, *action*, and *subgoal*, where an *action* represents the execution of a basic action in the domain of application (a set of basic actions must be defined for each domain), and a *subgoal* means that the execution engine must find another behavior that has to be executed to satisfy that particular subgoal. Specifically, three things need to be defined for using our language in a particular domain:

- A set of *basic actions* that can be used in the domain. For instance, in WARGUS we define actions such as *move*, *attack*, or *build*.
- A set of *sensors*, that are used in the behaviors to obtain information about the current state of the world. For instance, in WARGUS we might define sensors such as *numberOfTroops*, or *unitExists*. A sensor might return any of the standard basic data types, such as boolean or integer.
- A set of *goals*. Goals can be structured in a specialization hierarchy in order to specify the relations among them.

A goal might have parameters, and for each goal a function *generateSuccessTest* must be defined, that is able to generate a condition that is satisfied only when the goal is achieved. For instance, *HaveUnits(TOWER)* is a valid goal in our gaming domain and it should generate the condition *UnitExists(TOWER)*. Such condition is called the *success test* of the goal. Therefore, the goal definition can be used by the system to reason about the intended result of a behavior, while the success test is used by the execution engine to verify whether a particular behavior succeeds at run time.

Summarizing, our behavior language is strongly inspired by ABL, but expands it with declarative annotations (expanding the representation of goals and defining alive and success conditions) to allow reasoning.

Table 1. Snippet of a real trace generated after playing WARGUS

:le	*Player*	*Action*	*Annotation*
	1	Build(2,"pig-farm",26,20)	-
7	0	Build(5,"farm",4,22)	SetupResourceInfrastructure(0,5,2) WinWargus(0)
8	1	Train(4,"peon")	-
8	1	Build(2,"troll-lumber-mill",22,20)	-
8	0	Train(3,"peasant")	SetupResourceInfrastructure(0,5,2) WinWargus(0)
3	1	Train(4,"peon")	-
3	1	Resource(10,5)	-
7	0	Resource(5,0)	SetupResourceInfrastructure(0,5,2) WinWargus(0)
	...	...	...

Behavior Acquisition in WARGUS

igure 2 shows, the first stage of our case-based planning architecture consists quiring a set of behaviors from an expert demonstration. Let us present this in more detail.

ıe of the main goals of this work is to allow a system to learn a behavior by ly observing a human, in opposition to having a human encoding the behav- some form of programming language. To achieve that goal, the first step in rocess must be for the expert to provide the demonstration to the system. In articular application domain, WARGUS, an expert simply plays a game of RGUS (against the built-in AI, or against any other opponent). As a result at game, we obtain a game trace, consisting of the set of actions executed g the game. Table 1 shows a snippet of a real trace from playing a game of RGUS. As the table shows, each trace entry contains the particular cycle in ı an action was executed, which player executed the action, and the action

For instance, the first action in the game was executed at cycle 8, where r 1 made his unit number 2 build a "pig-farm" at the (26,20) coordinates.

Figure 2 shows, the next step is to annotate the trace. For this process, xpert uses a simple annotation tool that allows him to specify which goals ıe pursuing for each particular action. To use such an annotation tool, a set ıilable goals has to be defined for the WARGUS domain.

our approach, a *goal* $g = name(p_1, ..., p_n)$ consists of a goal name and a set rameters. For instance, in WARGUS, these are some of the goal types we defined:

- *WinWargus*(*player*): representing that the action had the intention of mak- ıg the player *player* win the game.
- *KillUnit*(*unit*): representing that the action had the intention of killing the nit *unit*.

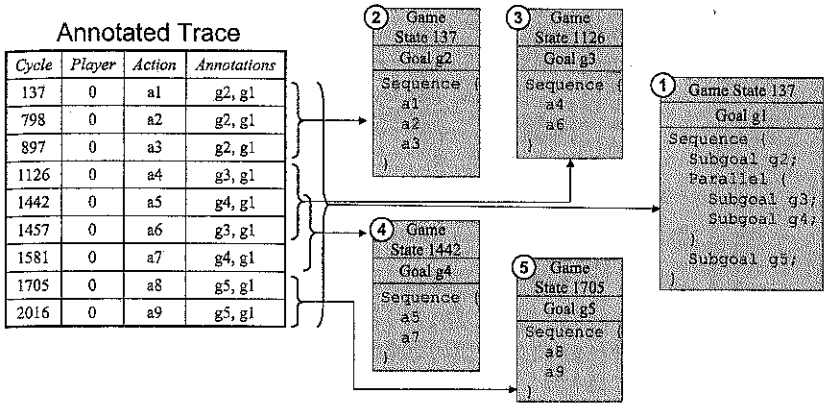


Fig. 3. Extraction of cases from the annotated trace

- *SetupResourceInfrastructure*(*player*, *peasants*, *farms*): indicates that the expert wanted to create a good resource infrastructure for player *player*, that at least included *peasants* number of peasants and *farms* number of farms.

The fourth column of Table 1 shows the annotations that the expert specified for his actions. Since the snippet shown corresponds to the beginning of the game, the expert specified that he was trying to create a resource infrastructure and, of course, he was trying to win the game.

Finally, as Figure 2 shows, the annotated trace is processed by the *case extractor* module, that encodes the strategy of the expert in this particular trace in a series of cases. Traditionally, in the CBR literature cases consist of a problem/solution pair; in our system we extended that representation due to the complexity of the domain of application. Specifically, a case in our system is defined as a triple consisting of a game state, a goal and a behavior. See Section 7 for a more detailed explanation of our case formalism.

In order to extract cases, the annotated trace is analyzed to determine the temporal relations among the individual goals appearing in the trace. For instance, if we look at the sample annotated trace in Figure 3, we can see that the goal *g2* was attempted *before* the goal *g3*, and that the goal *g3* was attempted *in parallel* with the goal *g4*. The kind of analysis required is a simplified version of the temporal reasoning framework presented by Allen [7], where the 13 basic different temporal relations among events were identified. In our framework, we are only interested in knowing if two goals are pursued in sequence, in parallel, or if one is a subgoal of the other. We assume that if the temporal relation between a particular goal g and another goal g' is that g happens *during* g', then g is a subgoal of g'. For instance, in Figure 3, *g2*, *g3*, *g4*, and *g5* happen *during* *g1*; thus they are considered subgoals of *g1*.

From temporal analysis, procedural descriptions of the behavior of the expert can be extracted. For instance, from the relations among all the goals in Figure 3,

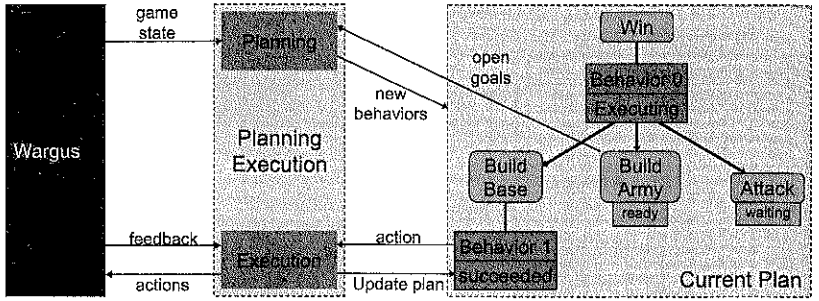


Fig. 4. Interleaved plan expansion and execution

umber 1 (shown in the figure) can be extracted, specifying that to achieve *g1* in the particular game state in which the game was at cycle 137, the t first tried to achieve goal *g2*, then attempted *g3* and *g4* in parallel, and hat *g5* was pursued. Then, for each one of the subgoals a similar analysis is med, leading to four more cases. For example, case 3 states that to achieve *2* in that particular game state, basic actions *a4* and *a6* should be executed ntially.

Real-Time Plan Expansion and Execution

g execution time, our system will use the set of cases collected from expert to play a game of WARGUS. In particular two modules are involved ecution: a real-time plan expansion and execution module (RTEE) and avior generation module (BG). Both modules collaborate to maintain a nt *partial plan tree* that the system is executing.

artial plan tree (that we will refer to as simply the "plan") in our framework resented as a tree consisting of two types of nodes: *goals* and *behaviors* (fol- g the same idea of the task/method decomposition [4]). Initially, the plan ts of a single goal: "win the game". Then, the RTEE asks the BG module erate a behavior for that goal. That behavior might have several subgoals, nich the RTEE will again ask the BG module to generate behaviors, and . For instance, on the right hand side of Figure 4 we can see a sample where the top goal is to "win". The behavior assigned to the "win" goal ree subgoals, namely "build base", "build army" and "attack". The "build goal has already a behavior assigned that has no subgoals, and the rest goals still don't have an assigned behavior. When a goal still doesn't have igned behavior, we say that the goal is *open*.

ditionally, each behavior in the plan has an associated state. The state of avior can be: *pending*, *executing*, *succeeded* or *failed*. A behavior is pending it still has not started execution, and its status is set to failed or succeeded its execution ends, depending on whether it has satisfied its goal or not.

A goal that has a behavior assigned and where the behavior has failed is also considered to be open (since a new behavior has to be found for this goal).

Open goals can be either *ready* or *waiting*. An open goal is ready when all the behaviors that had to be executed before this goal have succeeded, otherwise, it is waiting. For instance, in Figure 4, "behavior 0" is a sequential behavior and therefore the goal "build army" is ready since the "build base" goal has already succeeded and thus "build army" can be started. However, the goal "attack" is waiting, since "attack" has to be executed after "build army" succeeds.

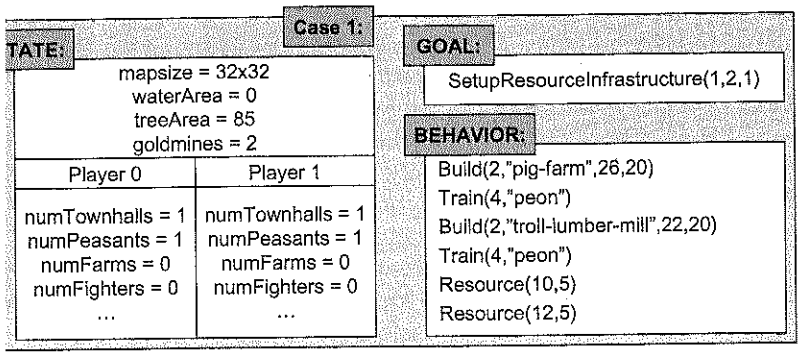
The RTEE is divided into two separate modules, that operate in parallel to update the current plan: the *plan expansion* module and the *plan execution* module. The plan expansion module is constantly querying the current plan to see if there is any ready open goal. When this happens, the open goal is sent to the BG module to generate a behavior for it. Then, that behavior is inserted in the current plan, and it is marked as pending.

The plan execution module has two main functionalities: a) check for basic actions that can be sent directly to the game engine, b) check the status of plans that are in execution:

- For each pending behavior, the execution module evaluates the preconditions, and as soon as they are met, the behavior starts its execution.
- If any of the execution behaviors have any basic actions, the execution module sends those actions to WARGUS to be executed.
- Whenever a basic action succeeds or fails, the execution module updates the status of the behavior that contained it. When a basic action succeeds, the executing behavior can continue to the next step. When a basic action fails, the behavior is marked as failed, and thus its corresponding goal is open again (thus, the system will have to find another plan for that goal).
- The execution module periodically evaluates the alive conditions and success conditions of each behavior. If the alive conditions of an executing behavior are not satisfied, the behavior is marked as failed, and its goal is open again. If the success conditions of a behavior are satisfied, the behavior is marked as succeeded.
- Finally, if a behavior is about to be executed and the current game state has changed since the time the BG module generated it, the behavior is handed back to the BG and it will pass again through the *adaptation* phase (see Section 7) to make sure that the plan is adequate for the current game state.

7 Behavior Generation

The goal of the BG module is to generate behaviors for specific goals in specific scenarios. Therefore, the input to the BG module is a particular scenario (i.e. the current game state in WARGUS) and a particular goal that has to be achieved (e.g. "Destroy The Enemy's Cannon Tower"). To achieve that task, the BG system uses two separate processes: *case retrieval* and *case adaptation* (that correspond to the first two processes of the 4R CBR model [1]).



5. Example of a case extracted from an expert trace for the WARGUS game

ice that to solve a complex planning task, several subproblems have to be . For instance, in our domain, the system has to solve problems such as build a proper base, how to gather the necessary resources, or how to de- each of the units of the enemy. All those individual problems are different ure, and in our case base we might have several cases that contain differ- haviors to solve each one of these problems under different circumstances. fore, in our system we will have an *heterogeneous* case base. To deal with sue, we propose to include in each case the particular *goal* that it tries to Therefore we represent cases as triples: $c = \langle S, G, B \rangle$, where S is a partic- ame state, G is a goal, and B is a behavior; representing that $c.B$ is a good ior to apply when we want to pursue goal $c.G$ in a game state similar to $c.S$. ure 5 shows an example of a case, where we can see the three elements: a description, that contains some general features about the map and some ation about each of the players in the game; a particular goal (in this building the resource infrastructure of player "1"); and finally a behavior ieve the specified goal in the given map. In particular, we have used a game definition composed of 35 features that try to represent each aspect of the GUS game. Twelve of them represent the number of troops (number of s, number of peasants, and so on), four of them represent the resources he player disposes of (gold, oil, wood and food), fourteen represent the ption of the buildings (number of town halls, number of barracks, and) and finally, five features represent the map (size in both dimensions, ntage of water, percentage of trees and number of gold mines).

e case retrieval process uses a standard nearest neighbor algorithm but a similarity metric that takes into account both the goal and the game Specifically, we use the following similarity metric:

$$d(c_1, c_2) = \alpha d_{GS}(c_1.S, c_2.S) + (1 - \alpha) d_G(c_1.G, c_2.G)$$

d_{GS} is a simple Euclidean distance between the game states of the two (where all the attributes are normalized between 0 and 1), d_G is the distance between goals, and α is a factor that controls the importance of the game

state in the retrieval process (in our experiments we used $\alpha = 0.5$). To measure distance between two goals $g_1 = name_1(p_1, ..., p_n)$ and $g_2 = name_2(q_1, ..., q_m)$ we use the following distance:

$$d_G(g_1, g_2) = \begin{cases} \sqrt{\sum_{i=1...n} \left(\frac{p_i - q_i}{P_i}\right)^2} & \text{if } name_1 = name_2 \\ 1 & \text{otherwise} \end{cases}$$

where P_i is the maximum value that the parameter i of a goal might take (we assume that all the parameters have positive values). Thus, when $name_1 = name_2$, the two goals will always have the same number of parameters and the distance can be computed using an Euclidean distance among the parameters. The distance is maximum (1) otherwise.

The result of the retrieval process is a case that contains a behavior that achieves a goal similar to the requested one by the RTEE, and that can be applied to a similar map than the current one (assuming that the case base contains cases applicable to the current map). The behavior contained in the retrieved case then needs to go through the adaptation process. However, our system requires *delayed adaptation* because adaptation is done according to the current game state, and the game state changes with time. Thus it is interesting that adaptation is done with the most up to date game state (ideally with the game state just before the behavior starts execution). For that reason, the behavior in the retrieved case is initially directly sent to the RTEE. Then, when the RTEE is just about to start the execution of a particular behavior, it is sent back to the BG module for adaptation.

The adaptation process consists of a series of rules that are applied to each one of the basic operators of a behavior so that it can be applied in the current game state. Specifically, we have used two adaptation rules in our system:

- *Unit adaptation*: each basic action sends a particular command to a given unit. For instance the first action in the behavior shown in Figure 5 commands the unit "2" to build a "pig-farm". However, when that case is retrieved and applied to a different map, that particular unit "2" might not correspond to a peon (the unit that can build farms) or might not even exist (the "2" is just an identifier). Thus, the unit adaptation rule finds the most similar unit to the one used in the case for this particular basic action. To perform that search, each unit is characterized by a set of 5 features: owner, type, position (x,y), hit-points, and status (that can be *idle*, *moving*, *attacking*, etc.) and then the most similar unit (according to an Euclidean distance using those 5 features) in the current map to the one specified in the basic action is used.
- *Coordinate adaptation*: some basic actions make reference to some particular coordinates in the map (such as the *move* or *build* commands). To adapt the coordinates, the BG module gets (from the case) how the map in the particular coordinates looks like by retrieving the content of the map in a 5x5 window surrounding the specified coordinates. Then, it looks in the current map for a spot in the map that is the most similar to that 5x5 window, and uses those coordinates.

2. Summary of the results of playing against the built-in AI of WARGUS in 1 2-player versions of "Nowhere to run nowhere to hide"

	map1	*map2*	*map3*
trace1	3 wins	3 wins	1 win, 1 loss, 1 tie
trace2	1 loss, 2 ties	2 wins, 1 ties	2 losses, 1 tie
trace1 & trace2	3 wins	3 wins	2 wins 1 tie

Experimental Results

aluate our approach, we used several variations of a 2-player version of the known map "Nowhere to run nowhere to hide", all of them of size 32x32. plained in Section 3, this map has the characteristic of having a wall of that separates the players and that leads to complex strategic reasonings. ifically, we used 3 different variations of the map (that we will refer as *map1*, ? and *map3*), where the initial placement of the buildings (a gold mine, a hall and a peasant in each side) varies strongly, and also the wall of trees separates both players is very different in shape (e.g. in one of the maps it very thin point that can be tunneled easily).

e recorded expert traces for the first two variants of the map (that we will as *trace1* and *trace2*). Specifically, *trace1* was recorded in *map1* and used a egy consisting on building a series of ballistas to fire over the wall of trees; *trace2* was recorded in *map2* and tries to build defense towers near the wall ees so that the enemy cannot chop wood from it. Each trace contains 50 to tions, and about 6 to 8 cases can be extracted from each of them. Moreover, ir current experiments, we have assumed that the expert wins the game, it ins as future work to analyze how much the quality of the expert trace ts the performance of the system.

/e tried the effect of playing with different combinations of them in the e variations of the map. For each combination, we allowed our system to against the built-in AI three times (since WARGUS has some stochastic ents), making a total of 27 games.

able 2 shows the obtained results when our system plays only extracting s from *trace1*, then only extracting cases from *trace2*, and finally extracting s from both. The table shows that the system plays the game at a decent l, managing to win 17 out of the 27 games it played. Moreover, notice that n the system uses several expert traces to draw cases from, its play level eases greatly. This can be seen in the table since from the 9 games the em played using both expert traces, it won 8 of them and never lost a game, g only once. Moreover, notice also that the system shows adaptive behavior e it was able to win in some maps using a trace recorded in a different map nks to the combination of planning, execution, and adaptation).

inally, we would like to remark the low time required to train our system lay in a particular map (versus the time required to write a handcrafted avior to play the same map). Specifically, to record a trace an expert has to play a complete game (that takes between 10 and 15 minutes in the maps we used) and then annotate it (to annotate our traces, the expert required about 25 minutes per trace). Therefore, in 35 to 40 minutes of time it is possible to train our architecture to play a set of WARGUS maps similar to the one where the trace was recorded (of the size of the maps we used). In contrast, one of our students required several weeks to hand code a strategy to play WARGUS at the level of play of our system. Moreover, this are preliminary results and we plan to systematically evaluate this issue in future work. Moreover, as we have seen our system is able to combine several traces and select cases from one or the other according to the current situation. Thus, an expert trace for each single map is not needed.

9 Conclusions

In this paper we have presented a case based planning framework for real-time strategy games. The main features of our approach are a) the capability to deal with the vast decision spaces required by RTS games, b) being able to deal with real-time problems by interleaving planning and execution in real-time, and, c) solving the knowledge acquisition problem by automatically extracting behavioral knowledge from annotated expert demonstrations in form of cases. We have evaluated our approach by applying it to the real-time strategy WARGUS with promising results.

The main contributions of this framework are: 1) a case based integrated real-time execution and planning framework; 2) the introduction of a behavior representation language that includes declarative knowledge as well as procedural knowledge to allow both reasoning and execution; 3) the idea of automatic extraction of behaviors from expert traces as a way to automatically extract domain knowledge from an expert; 4) the idea of heterogeneous case bases where cases that contain solutions for several different problems (characterized as *goals* in our framework) coexist and 5) the introduction of *delayed adaptation* to deal with dynamic environments (where adaptation has to be delayed as much as possible to adapt the behaviors with the most up to date information).

As future lines of research we plan to experiment with adding a case retention module in our system that retains automatically all the adapted behaviors that had successful results while playing, and also annotating all the cases in the case base with their rate of success and failure allowing the system to learn from experience. Additionally, we would like to systematically explore the transfer learning [15] capabilities of our approach by evaluating how the knowledge learnt (both from expert traces or by experience) in a set of maps can be applied to a different set of maps. We also plan to further explore the effect of adding more expert traces to the system and evaluate if the system is able to properly extract knowledge from each of them to deal with new scenarios.

Further, we would like to improve our current planning engine so that, in addition to sequential and parallel plans, it can also handle conditional plans. Specifically, one of the main challenges of this approach will be to detect and properly extract conditional behaviors from expert demonstrations.

owledgements. The authors would like to thank Kane Bonnette, for the VARGUS interface; and DARPA for their funding under the Integrated ing program TT0687481.

rences

amodt, A., Plaza, E.: Case-based reasoning: Foundational issues, methodological riations, and system approaches. Artificial Intelligence Communications 7(1), –59 (1994)

aa, D., Molineaux, M., Ponsen, M.: Learning to win: Case-based plan selection a real-time strategy game. In: Muñoz-Ávila, H., Ricci, F. (eds.) ICCBR 2005. NCS (LNAI), vol. 3620, pp. 5–20. Springer, Heidelberg (2005)

aro, M.: Real-time strategy games: A new AI research challenge. In: IJCAI'2003, . 1534–1535. Morgan Kaufmann, San Francisco (2003)

andrasekaran, B.: Design problem solving: a task analysis. AI Mag. 11(4), 59–71 990)

kes, R., Nilsson, N.J.: Strips: A new approach to the application of theorem oving to problem solving. Artificial Intellicence 2(3/4), 189–208 (1971)

eßmann, A., Maximini, K., Maximini, R., Sauer, T.: CBR-based execution and anning support for collaborative workflows. In: Muñoz-Ávila, H., Ricci, F. (eds.) CBR 2005. LNCS (LNAI), vol. 3620, pp. 271–280. Springer, Heidelberg (2005)

llen, F.G.: Maintaining knowledge about temporal intervals. Communications of e ACM 26(11), 832–843 (1983)

ammond, K.F.: Case based planning: A framework for planning from experience. ognitive Science 14(3), 385–443 (1990)

ang, M., Lee-Urban, S., Muñoz-Avila, H.: Hierarchical plan representations for coding strategic game ai. In: Proceedings of Artificial Intelligence and Interactive igital Entertainment Conference (AIIDE-05), AAAI Press, Stanford (2005)

ateas, M., Stern, A.: A behavior language for story-based believable agents. IEEE telligent systems and their applications 17(4), 39–47 (2002)

uñoz-Avila, H., Aha, D.: On the role of explanation for hierarchical case-based anning in real-time strategy games. In: Funk, P., González Calero, P.A. (eds.) CCBR 2004. LNCS (LNAI), vol. 3155, Springer, Heidelberg (2004)

yers, K.L.: CPEF: A continuous planning and execution framework. AI Magazine, (4) (1999)

rkin, J.: Applying goal-oriented action planning to games. In: AI Game Programming Wisdom II. Charles River Media (2003)

nsen, M., Muñoz-Avila, H., Spronck, P., Aha, D.: Automatically acquiring adaptive real-time strategy game opponents using evolutionary learning. In Proceedings the 20th National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence, pp. 1535–1540 (2005)

arma, M., Homes, M., Santamaria, J., Irani, A., Isbell, C., Ram, A.: Transfer arning in real time strategy games using hybrid CBR/RL. In: IJCAI'2007, page appear, Morgan Kaufmann, San Francisco (2007)

Case Authoring: From Textual Reports to Knowledge-Rich Cases

Stella Asiimwe[1], Susan Craw[1], Bruce Taylor[2], and Nirmalie Wiratunga[1]

[1] School of Computing
[2] Scott Sutherland School
The Robert Gordon University, Aberdeen, Scotland, UK
{sa, S.Craw, nw}@comp.rgu.ac.uk, B.Taylor@rgu.ac.uk

Abstract. SmartCAT is a Case Authoring Tool that creates knowledge-rich cases from textual reports. Knowledge is extracted from the reports and used to learn a concept hierarchy. The reports are mapped onto domain-specific concepts and the resulting cases are used to create a hierarchically organised case-based system. Indexing knowledge is acquired automatically unlike most textual case-based reasoning systems. Components of a solution are attached to nodes and relevant parts of a solution are retrieved and reused at different levels of abstraction. We evaluate SmartCAT on the SmartHouse domain looking at the usefulness of the cases, the structure of the case-base and the retrieval strategy in problem-solving. The system generated solutions compare well with those of a domain expert.

1 Introduction

Creating a case-based reasoning system can be quite challenging if the problem-solving experiences are captured as unstructured or semi-structured text [13]. This is because the system should be able to compare new problems with the textual case knowledge. Although IR-based techniques can be used to retrieve whole documents or snippets of documents, case comparison in this situation would only take place at word/phrase level. The features pertaining to the documents would still have to be compared using some domain/background knowledge or lexical source, in order to arrive at a useful ranking. Alternatively, a structured case representation can be created and the textual sources mapped onto it before they are used in reasoning [15]. This is quite difficult and the costs can be prohibitive if it is manually done by an expert.

It has been observed that humans do not interpret text at word-level but do so at a much higher level of abstraction where concepts are manipulated [4]. For example, an occupational therapist that reads about a *wheelchair user* immediately thinks about the person's *mobility*. Hierarchical organisation of cases enables effective retrieval at different levels of problem abstraction [2]. The humans' ability to organise information into concepts, in order to extract meaning that is beyond the words they read, is what we attempt to mimic in our work.

Our Case Authoring Tool SmartCAT creates knowledge-rich cases from textual SmartHouse reports. Figure 1 shows the expert interacting with SmartCAT to sanction authored cases. SmartCAT uses the information embedded in text to learn a concept