Constraint-based Discourse Agents

Toby Donaldson and Robin Cohen
Department of Computer Science
University of Waterloo

Introduction

In this paper, we discuss how a discourse agent can manage multiple discourse goals using dynamic constraint optimization. We discuss *turn-taking* agents, which are agents that know how to take turns in dynamic, conversational settings. Conversation constrains an agent's behaviour in a number of ways; for example, a conversational agent must act quickly to avoid long pauses, and it must be prepared to sometimes act before it is completely finished thinking. Our solution to this turn-taking problem satisfies these constraints, and also provides for straightforward application of well-known heuristics for solving the particular optimization problem that arises.

Turn-taking Agents

During a conversation between two (or more) agents, the agents take *turns* speaking: one agent speaks, and then the other.[1] When one agent is speaking, we say it is that agent's *turn* to speak, and that the agent has the *floor*. In human-human conversation, turn coordination is a subtle and necessary[2] part of conversing. In general, people tend to avoid both long pauses and speaking at the same time (Oreström 1983; Goldman-Eisler 1968). Humans dynamically coordinate turn-taking by issuing *turn-taking signals*, either implicitly or explicitly, throughout the conversation (Oreström 1983). A person decides *when* to take a turn based not only on when she has something to say, but when the other conversant has given the appropriate turn-ending or turn-yielding signals. While people can interrupt at any time, interruptions should be used sparingly and in the right situations, to avoid annoying the other conversant.

We are interested in taking these features of human-human conversation and applying them to human-computer interfaces, such as advice-giving systems or software help systems. Many kinds of human-computer interaction can be treated as a conversation, and by identifying the relevant turn-ending and turn-yielding signals, we hope to enable computers to take turns in a more intelligent—and less irritating—manner than is typically the case.

[1] We are considering only *English* conversation, as some turn-taking habits are language or culture dependent.

[2] However, turn-taking is not *sufficient* for a conversation. For example, a play-by-play announcer and a colour commentator for a hockey game take turns speaking, yet they are not having a conversation with each other.

Architecture of a Turn-taking Agent

We will refer to conversational participants as *turn-taking*, or *conversational* agents. Conversational agents have the general structure shown in Figure 1. The three boxes represent parallel processes, and an utterance flows through the system from left to right. First, the utterance input filter does basic processing on the raw utterance received from another conversational participant; what this involves depends on what the utterance problem solver does. If the input is natural language in the form of a stream of characters, the utterance input filter might parse the input, and then give this parse tree (instead of the raw input) to the utterance problem solver. In general, the utterance problem solver can be solving any problem, so long as it generates a "current utterance" for the turn execution manager to use when the agent is required to speak. The job of the turn execution manager is then to "listen" to the conversation and watch for appropriate points to take a turn. In this paper, we will focus on describing one general approach for solving an agent's utterance generation problem.

Note that our general model of turn-taking allows the utterance generation problem to be whatever problem is most appropriate for the domain of discourse. For more complex turn-taking agents, such as advice-giving systems, plan-based models have frequently been used as the driving force behind the agent's behaviour, e.g. (Chu-Carroll & Carberry 1994; van Beek, Cohen, & Schmidt 1993). In this paper, we take an entirely constraint-based approach: we consider turn-taking agents that are driven mainly by solving constraint satisfaction problems, in particular *dynamic* constraint satisfaction that change over time through the introduction or deletion of turn-taking goals.

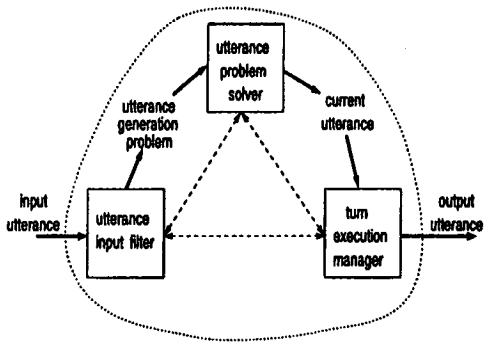

Figure 1: The Internal Structure of a Turn-taking Agent.

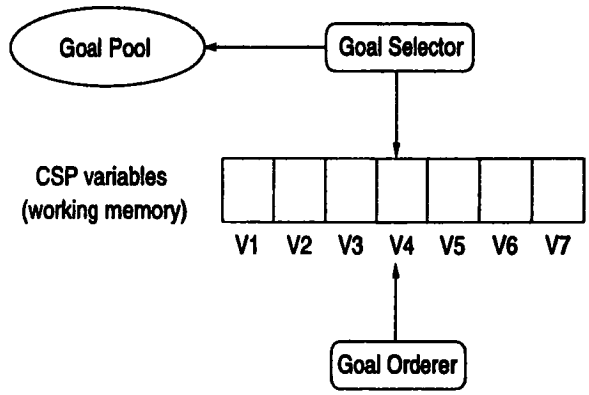

Figure 2: A constraint-based turn-taking agent

Controlling Agents with DCSPs

To highlight how we intend to use constraints to control turn-taking agents, we will use a simpler agent model than that of Figure 1. Figure 2 shows how we view the goal-management part of turn-taking agents. The agent perceives the outside world, and turn-taking goals are created and dumped into a *goal pool.* Goals are selected from the goal pool by the *goal selector* and placed into the "working memory" of the agent, which consists of a fixed number of slots that can hold one goal each. These slots correspond directly to the variables of the constraint satisfaction problem the agent solves, and the first variable holds the "current utterance", that is, the goal that the agent will try to achieve if it is required to act this instant. The working memory variables also have a fixed order of execution: if the agent is required to act, then it will first try to achieve the goal in V_1, then V_2, and so on. This order influences how the *goal orderer* performs its processing. The goal orderer continually works on arranging the goals in working memory in the order that breaks as few constraints as possible. We discuss the operation of the goal selector and the goal orderer more fully in a moment. Finally, we are not concerned directly with *when* to execute a goal; we leave that to the turn execution manager.[3]

Kinds of Goals

The particular types of turn-taking goals that are relevant depend upon the type of discourse in which one is engaging. We have studied advice-giving settings most closely, and have identified the following turn-taking goal-types (this is not intended to be an exhaustive list):

- repair goals:
  - information repair goals (i.e. fixing of faulty information that a conversant is believed to possess);
  - contradiction repair (i.e. reconciliation of contradictory utterances/beliefs);
- clarification goals:
  - domain plan clarification (i.e. determining the higher-level goals behind the utterance);
  - meaning clarification (e.g. requesting the meaning of an unfamiliar word used by the speaker);
  - linguistic clarification (i.e. cases where some sort of noise, such as a spelling mistake in written discourse, has prevented the listener from clearly interpreting the utterance);
- information-seeking goals;
- question-answering goals.

The various kinds of constraints that determine the ideal order of goal satisfaction are specified by the following penalty matrix:[4]

	1	2	3	4	5	6	7
1	·	p	p	·	·	·	·
2	·	·	p	·	·	·	·
3	·	·	·	·	·	·	·
4	p	p	p	·	·	p	p
5	p	p	p	p	·	p	p
6	·	·	·	·	·	·	·
7	·	·	·	·	·	·	·

1 - domain plan clar. 5 - answering
2 - meaning clar. 6 - info-repair
3 - linguistic clar. 7 - contra. repair
4 - info-seeking

In the matrix, a p means a penalty is assigned. A penalty is any positive value, and the penalties can be different values. Penalties are read off by row and then column. For example, if G_a is an information-seeking goal, and G_b is an information-repair goal, then putting G_a before G_b results in a penalty, but putting G_b before G_a does not.

[3]Specifying the behaviour is an important part of our research, but we do not address it in this paper.

[4]This matrix only gives penalties for what we call "goal-type" constraints. In (Donaldson & Cohen 1997), we also consider more general constraints related to sub-goals and temporal ordering.

Goal Structure of a Turn-taking Agent

A constraint-based turn-taking agent tries to find an ordering of a subset of goals from the goal pool that breaks as few constraints as possible. More precisely, we specify the problem as shown in Figure 3. Given a (possibly changing) pool of goals and a set of k working-memory CSP variables $V_1, \ldots, V_k$, select and order the k goals that result in the fewest penalty points. The penalty function $p(G_a, G_b)$ is the penalty received for G_a occurring before G_b, and is 0 if there is no penalty. The penalty function is not symmetric, i.e. $p(G_a, G_b)$ is not necessarily the same as $p(G_b, G_a)$. Given an ordered list of k goals, we calculate the total penalty by summing all penalties between all pairs of goals in a left-to-right manner.

Given n goals
k variables to fill, $k \leq n$
a penalty function $p(G_a, G_b) \geq 0$ that gives the penalty for G_a coming before G_b
a $1-1$ function $\sigma : \{1, \ldots, k\} \rightarrow \{1, \ldots, n\}$

$$\text{Minimize} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} p(G_{\sigma(i)}, G_{\sigma(j)})$$

Figure 3: The turn-taking problem.

We will refer to this as the *turn-taking problem*, and note that is a kind of linear assignment problem, similar to the travelling salesman problem (although the TSP has a different evaluation function, and you do not need to first select some subset of cities). During a conversation, and agent will be presented with many instances of the turn-taking problem to solve since goals will be appearing in the goal pool as the conversation continues, and goals will disappear when the agent has satisfied them.[5]

We are interested in solving not only particular instances of this problem, but the dynamic version, where the goals (and possibly even the penalty function) change over time. This sort of *dynamic CSP* has been proposed before as a way to handle changing environments (e.g. (Dechter & Dechter 1988; Verfaillie & Schiex 1994)), and in general this is a very difficult problem. Letting P_t be an instance of the turn-taking problem at time t, participation in a conversation can be treated as solving a series of turn-taking problems $P_1, P_2, \ldots, P_t, \ldots$, at least one for each new goal that appears in the goal pool. In the general dynamic CSP problem, there need be no relation between any two particular problem instances, although practically some sort of restriction is put on how much a problem can change from one time unit to the next (e.g. constraints can only be added, never removed). We avoid the general dynamic CSP problem by considering only the turn-taking problem of Figure 3, and also our application suggests that adjacent problems will be very similar, so we expect that the solution to the problem at $t-1$ will be a better starting point for solving the problem at t than would be an initial guess generated using some general-purpose heuristic.

[5] A rational agent may also drop a goal because it believes it can never be achieved. Further discussion of the *persistence* of goals can be found in(Cohen & Levesque 1990), and also (Donaldson & Cohen 1996).

Solving the Turn-taking Problem

A turn-taking agent is involved in a conversation, so it must take this fact into consideration when solving turn-taking problems. Ideally, we need a solution method with the following features:

- It should be an *anytime* algorithm, where the longer we let it run, the higher the quality of the solution. In a conversation, an agent cannot always know how long it will have to "think", and so it will sometimes be required to act before it has completely finished processing;
- It should be reasonably efficient, since conversation is ultimately a real-time process where acting quickly (to avoid a pause) is often more important than acting correctly. As an example of this, (Carletta, Caley, & Isard 1993) give instances of self-repairs in conversation where people say the opposite of what they mean (e.g. saying "turn left" when they meant "turn right"), but then quickly correct themselves. Humans can tolerate such mistakes in conversation because it is usually possibly to quickly repair them, or ask for clarification;
- The amount of time spent choosing goals for the V_i's should be sensitive to the fact that when the agent acts, it will first try to achieve the goal in V_1, and then V_2, and so on, lastly attempting to achieve the goal in V_k. If the agent has T seconds to make a decision, then the largest proportion of T should be spent deciding upon what goal to put in V_1, and the least on V_k (since it has the least chance of actually being acted upon);
- As mentioned in the previous section, we do not want to solve just single instances of the turn-taking problem, but multiple related instances $P_1, P_2, \ldots$. Thus, the solution method should not require major reworking to handle the dynamic case.

Figure 2 is a blueprint for our approach to this problem: goal selection and goal ordering are largely independent processes that run continuously. The goal ordering process looks only at the working memory variables, and continuously applies a local search procedure to maintain a reasonably good ordering of the working memory goals. The goal selection process decides what goals from the goal pool should be promoted to working memory, and what working memory goals should be demoted to the goal pool; it makes

no decisions about ordering the goals in working memory. Our decision to break the problem into two parts stems from the fact that working memory corresponds to CSP variables, and the goal pool is the domain for the variables.

While we are currently investigating the best methods to use to solve the turn-taking problem, Figure 4 gives pseudo-code for a simple implementation of a two-process turn-taking problem solver. The CSP variables form the major shared resource, so only one process can access working memory at once. We also use a shared flag to indicate when working memory has been changed through the addition or removal of a goal. While the two processes do communicate with each other, their operation is largely independent, and each one could continue processing if the other one were to stop.

Besides adding parallelism, this scheme addresses each of the issues listed above:

- Local search is used to order the goals, and this is a natural anytime procedure;
- The local search algorithms we expect to use are straightforward iterative procedures often able to quickly find reasonably good solutions. There are a number of useful local search heuristics for such assignment problems that we can essentially take off the shelf, e.g. the TSP 2-opt (and related) heuristics (Lin & Kernighan 1973), simulated annealing (Martin & Otto 1993; Nahar, Sahni, & Shragowitz 1986), and guided local search (Voudouris & Tsang 1995);
- An important consideration on the heuristics used is that the processing of the CSP variables go from V_1 to V_k. The pseudo-code in Figure 4 ensures this in a simple way for the goal orderer by filling the CSP variables in left to right order, and always restarting from the first variable when a new goal is added to working memory. This left-to-right requirement means some heuristics cannot be directly applied. For example, the basic min-conflicts heuristic (Minton *et al.* 1992) would be inappropriate since it works by randomly choosing a conflicted variable (i.e. a variable that participates in breaking a constraint), no special treatment will be given to the left-most variables, and it is possible for min-conflicts to spend all its time improving the solution without, for example, never getting to change a poor value in V_1. If the goal in V_1 is poor, then that can lead to trouble in the conversation, effectively wiping out the careful ordering of the goals after V_1;
- The goal selection and ordering modules are relatively robust in the face of changing goals, i.e. if a new goal appears or disappears, the modules essentially continue to apply their basic algorithms on whatever goals are there. Of course, if a new goal appears in the goal pool, then we must deal with the issue of when to analyze that goal, e.g. should the goal selector stop what its doing and immediately look at the new goal, or should the new goal wait for its turn to be analyzed? However, the basic heuristics and architecture need not be drastically changed to solve the dynamic turn-taking problem.

```
process GOAL-ORDERER()
  changed ← false
  loop forever
  /* FILL GOALS IN LEFT TO RIGHT ORDER */
    for i ← 1 to k do
      fill-vble(i)
  end loop
end goal-orderer

procedure FILL-VBLE(i)
  for j ← i+1 to k do
    if changed is true then
      /* RESTART THE GOAL-ORDERER */
      GOAL-ORDERER()
    else
      swap the goals in V_i and V_j according to chosen
        heuristic
end fill-vble

process GOAL-SELECTOR()
loop forever
  for each G_i in the goal pool & working memory do
    if G_i should be promoted to working memory then
      choose a goal G_wm to demote to goal pool
      swap G_i and G_wm
      changed ← true
    else if G_i must be removed from working memory then
      remove G_i from working memory
      choose a goal G_pool to promote to working memory
      put G_pool into working memory
      changed ← true
    end if
end goal-selector
```

Figure 4: Pseudo-code for solving the turn-taking problem.

Example

We now work through a course-advising example where a student asks a natural-language advice-giving system to help her schedule courses. For this example, we do not focus on the details of the natural language understanding and generation issues needed for such a system to work in practice, and so assume that the system is able to translate the student's input into turn-taking goals and tries to give helpful replies. We also assume that goals have "drop" conditions, so that if an agent is trying to achieve G, it will know when the goal is satisfied, and a goal can be removed from the goal pool and working memory when its drop conditions are achieved. Each broken constraint results in the addition of 5 penalty points, and its working memory holds at most three goals (but any number of goals can be in the goal pool).

To begin, we assume the system only knows that it is talking to a student, and that the student wants advice on which course to take. Suppose the student's first utterance is:

STUDENT: *Can I take MATH320 and MATH440 at the same time?*

Suppose as well that this utterance causes the following four turn-taking goals to be adopted and put into the goal pool:

G_1: a yes/no question-answering goal;

G_2: a domain plan clarification goal, to find out what degree the student is pursuing;[6]

G_3: a linguistic clarification goal, since there is no MATH440 (but there is MATH445);

G_4: an information-seeking goal to determine if the student has the correct pre-requisites for the mentioned courses; this can be thought of as a default pre-condition of G_1.

These four goals are added to the goal pool (in whatever order they arrive — they need not all appear at the same time), and we initialize working memory with arbitrary values: $GP = \{G_4\}, WM =< G_1, G_2, G_3 >, penalty = (5+5)+5 = 15$. The vector WM is ordered, and means variable V_1 contains G_1, V_2 contains G_2, and V_3 contains G_3. The penalty is calculated by first converting the goals of WM to their types as shown in the legend for the penalty matrix: $WM =< G_1, G_2, G_3 > \rightarrow < 5, 1, 3 >$. The pairs (5,1), (5,3), and (1,3) each break a constraint, so the penalty is $5+5+5 = 15$.

Now the goal selector and goal ordering processes operate continuously. Suppose that the goal selector decides G_4 should replace G_1 in working memory: $GP = \{G_1\}, WM =< G_4, G_2, G_3 >, penalty = (5)+5 = 10$. Now the goal orderer swaps the values of V_1 and V_3: $GP = \{G_1\}, WM =< G_3, G_2, G_4 >, penalty = 5$. The agents "plan of action" is this: it will first address the fact that there is no course called MATH440, and then, if that goal is satisfied, it will find out what degree the student is pursuing, and then go on to ask if the student has the correct pre-requisites.[7]

Having reached a reasonable goal-ordering, the system asks about MATH440:

SYSTEM: *There is no course called MATH440. Did you mean MATH445?*
STUDENT: *Yes.*

[6] Recognizing a user's plan is a common technique used to help produce cooperative responses in advice-giving situations. See, for example, (van Beek, Cohen, & Schmidt 1993).

[7] Being non-existent, MATH440 obviously has no pre-requisites. We assume, for simplicity, that the student really meant to say MATH445, and this correct value will replace the wrong value.

G_3 is satisfied, and so is deleted. We assume (for this example only, to keep it simple) that the orderer slides each remaining goal in working memory left one variable, and a new goal from the goal pool is put in V_3: $GP = \emptyset, WM =< G_2, G_4, G_1 >, penalty = 0$. Now the following exchange occurs:

SYSTEM: *Are you majoring in mathematics?*
STUDENT: *Yes.*

At this point the system has satisfied G_2, and so working memory can be updated: $GP = \emptyset, WM =< G_4, G_1, \times >, penalty = 0$. The student unexpectedly mentioned that she did poorly in MATH320, and we assume that this causes another goal to be adopted:

G_5 : information-seeking goal, to find out of if the student is generally doing poorly in math courses.

G_5 is put into the goal pool, and then put into the empty variable V_3: $GP = \emptyset, WM =< G_4, G_1, G_5 >, penalty = 5$. The goal orderer swaps G_1 and G_5, which does not affect the system's next question which attempts to satisfy G_4:

SYSTEM: *Have you taken MATH275, the pre-requisite for MATH320?*
STUDENT: *Yes. I passed, but I did really badly in it.*

G_2 is satisfied, so working memory is now: $GP = \emptyset, WM =< G_5, G_1, \times >, penalty = 0$. The system now takes the initiative and asks if the poor mark is typical of her performance in math classes:

SYSTEM: *Is your poor mark in MATH320 typical of your marks in mathematics?*
STUDENT: *No, I don't know what happened. That A- was the lowest mark I've ever gotten in a math course.*

The student is not doing all that badly, so the system simply drops G_5. Finally, the system is ready to answer the student's original question:

SYSTEM: *Since you are a math major and have the proper pre-requisites, taking both MATH445 and MATH320 is okay. They cover complementary material.*

The system now has nothing left to say, so at this point it is entirely up to the student to continue the conversation.

The anytime nature of the algorithm does not explicitly arise in this advice-giving example, although it is important in other domains. Imagine an intelligent navigation aid in your car that you can ask to find the shortest route from your house to, say, a downtown restaurant. It searches for the shortest route while you drive, and to make sure you do not miss a turn, the system should tell you the first part of the directions ("turn left at the next intersection") even before it has completely finished its plan.

Discussion

In this paper, we have presented a model of a constraint-based turn-taking agent. Turn-taking agents solve a particular dynamic constraint optimization problem, and we have proposed a solution framework for effectively solving this problem. Our solution takes into account a number of important constraints imposed by real-time discourse: decisions must be made quickly, agents must be prepared to sometimes act before they are finished deliberating, and an agent should give more thought to the goals it is likely to act upon. While we have not specified what exact combination of heuristics works best for a turn-taking agent (this will require experimentation), our solution framework allows us to easily apply various local search and selection heuristics.

One of the benefits of this constraint approach is the ease with which new constraints can be added by simply changing the penalty function. This suggests the possibility of learning new penalty values on the fly as the conversation is going on. For example, suppose you are speaking with a colleague who has strong opinions on the merits of object-oriented design. You have noticed in the past that even mentioning "objects" will cause this person to talk only about that. If you mean to speak to this person about something other than objects, then you might purposely avoid mentioning the word "object" in an effort to keep the conversation on topic. Our approach allows us to handle such global constraints by simply modifying the penalty function; we could add a penalty for any goal that requires mentioning objects before goals that do not mention objects.

Another feature of our approach is the explicit use of parallel operation. In human-human conversation, it takes time to say something, and the more you want to say, the more time it takes. In human-computer discourse, the time it takes for a computer to speak is almost instantaneous — e.g. the time it takes for a "print" statement to display a string. If, instead of printing its output the computer uses voice synthesis, then conversation is more symmetric, since it takes a significant amount of time for the computer to speak. When people speak, they generally do not stop thinking, and as self-repairs show, they can sometimes catch themselves in mid-utterance and start over. Since our architecture takes parallel operation into consideration, it allows for an agent to think while it is speaking, and, if warranted, interrupt itself and start over much like humans occasionally do. It seems odd to think of computers as making mistakes in this way, and we would like to avoid errors if possible. However, conversation is a complex enough process that it might not always be possible to quickly generate perfect output. Even if we have unlimited speed, an agent must still sometimes resort to making the best guess it can based on incomplete or even inaccurate information. It may turn out that a system that, every once in a while, makes a mistakes and then fixes it the way people do is more user-friendly than a fool-proof, HAL-like computer that thinks only humans can make mistakes.

References

Carletta, J.; Caley, R.; and Isard, S. 1993. A collection of self-repairs from the map task corpus. Technical Report tr-47, Human Communication Research Centre, University of Edinburgh.

Chu-Carroll, J., and Carberry, S. 1994. A plan-based model for response generation in collaborative task-oriented dialogues. In *Proceedings of AAAI-94*, 799–805.

Cohen, P., and Levesque, H. 1990. Intention is choice with commitment. *Artificial Intelligence* 42:213–261.

Dechter, R., and Dechter, A. 1988. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, 37–42.

Donaldson, T., and Cohen, R. 1996. Turn-taking in discourse and its application to intelligent agents. In *AAAI-96 Workshop on Agent Modeling.*

Donaldson, T., and Cohen, R. 1997. A constraint satisfaction framework for managing mixed-initiative discourse. Presented at AAAI-97 Symposium on Mixed-Initiative Interaction.

Goldman-Eisler, F. 1968. *Psycholinguistics, Experiments in Spontaneous Speech.* Academic Press.

Lin, S., and Kernighan, B. 1973. An effective heuristic algorithm for the travelling salesman problem. *Operations Research* 21:498–516.

Martin, O., and Otto, S. 1993. Combining simulated annealing with local search heuristics. Technical report, Oregon Graduate Institute of Science & Technology. CSE-94-016.

Minton, S.; Johnston, M.; Philips, A.; and Laird, P. 1992. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58:161–205.

Nahar, S.; Sahni, S.; and Shragowitz, E. 1986. Simulated annealing and combinatorial optimization. In *23rd Design Automation Conference*, 293–299.

Oreström, B. 1983. *Turn-taking In English Conversation.* CWK Gleerup.

van Beek, P.; Cohen, R.; and Schmidt, K. 1993. From plan critiquing to clarification dialogue for cooperative response generation. *Computational Intelligence* 9(2):132–154.

Verfaillie, G., and Schiex, T. 1994. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of AAAI-94*, 307–312.

Voudouris, C., and Tsang, E. 1995. Guided local search. Technical report, Department of Computer Science, Univestiy of Essex. CSM-247.