Planning with ChatScript

© Bruce Wilcox, gowilcox@gmail.com

Revision 7/3/2015 CS 5.5

Overview

Want your chatbot to control a physical body in either a real or a virtual world. Want it to perform tasks on your command beyond the immediately simple? You need a planner. ChatScript comes with a preexisting one you can try out called STOCKPILE. See the Bot Stockpile – planner doc for details.

Exploring alternatives is the domain of tree search, a mechanism which explicitly supports exploring hypothetical situations and backtracks when they fail. There are two kinds of tree search mechanisms.

First, there is the "alpha-beta" tree search, like those used for chess. It assumes an adversary and you take turns with him modifying the world to play a game. Imagine planning to go buy food at the grocery store. You will go down to your car, drive to the store, pick up the food and drive home.

But in a hostile world, you have to ask what happens if when you go to your car a criminal has broken into it and is about to drive off. That will be bad. But if you had a bat you could beat him and that would be OK. But if he had his gang with him you would be outnumbered and that would be bad. But if you had a cop with you that would be OK. But ... Clearly that branch of the tree has a lot of issues and if you can't explore them all, maybe you should take a taxi.

Fortunately the world is rarely actively hostile to you and your plans can be simpler. You can use a planner. It usually assumes there is no opponent and tries to find a sequence of actions to accomplish a goal.

Planners come in three flavors— *forward chaining*, *backward chaining*, and HTN (*hierarchical task network*).

A *forward chaining planner* is given an explicit appearance of what the initial world looks like (a set of facts currently true) and what the goal world looks like (a set of facts that need to become true), and it must search forward by legal operators (actions which change facts) trying to find a way to make the goal come true. This is more restrictive than a chess-like search because in chess we don't know what the board position for a win looks like (there is no one specific board image).

Forward chaining works well when there are few things you can do at any moment and you know what the answer looks like exactly. Knowing only the primitive actions that can be taken, the current state, and the goal state, forward planners can "discover" their way to the result with no further guidance by randomly

choosing actions and backtracking to an alternative when things don't work out. But this will not be efficient and generally forward-chaining planners need good heuristics to avoid wasting time researching bad moves.

A *backward chaining planner* is also given an explicit appearance of what initial and final worlds look like, but works backward trying to find what predecessor operator would have created the end state, searching its way back to the beginning.

Backward chaining has a strong goal-oriented behavior so it tends to involve less search than a forward-chainer. It, too, can work only knowing the start and end states. And it, too, may be inefficient without any heuristics to help it choose among multiple ways backward. And because it is working backward, backward chaining has no connection to data in the current world yet, so cannot make use of information there. This is a serious problem in any time-dependent planning task.

HTN (*Hierarchical Task Network*) planners are given templates of how to accomplish things, exactly like little scripts, and search by progressively refining high-level scripts into sequences of low-level scripts. Add in backtracking, and you provide the means to try out alternative scripts to see which one will really work in your current situation.

On one hand, having these HTN scripts makes for a very goal-oriented behavioral mechanism which can be counted on to efficiently find reasonable plans. On the other hand, scripters bear the responsibility for writing complete scripts. The planner cannot function by merely knowing the starting and ending states. If the planner comes to a step where it needs a sub plan to continue, there must be at least one usable sub plan already written.

HTN planners come in two flavors, *partial order* and *total order*.

The *partial order HTN planner* is somewhat like a backward chainer, in that it refines a plan by progressively reducing higher level plans to the next lower level. This allows it to do a broad-brush look at the solution, but leaves it isolated from current reality again, so that it can only discover critical details are missing when it finally reaches a bottom level.

A *total order HTN planner* is somewhat like a forward chainer, refining a plan by taking the first step in the top-level plan, and refining it all the way down to the current world, then taking the next piece in line at the deepest level, and refining it. That way the planner is progressively working on a plan in which it always knows what the current or modified world actually looks like. ChatScript uses a total order HTN planner.

Planner Script Basics

To support planning, ChatScript adds a new top-level element type, the `plan:`. It is a mutant hybrid of the outputmacro and the topic and looks like this:

```
plan: ^move (^from ^to)
    u: () $$distance = ^distance(^to ^from)
    u:() $$choice = horse
    u:() query(direct_sv $$choice range ?)
    u:() $$range = @0object
    u:() if ($$range >= $$distance) {end(PLAN)}
```

You declare a plan using a function name and a list of function arguments, just as you would an outputmacro. But the body of the plan is a series of rules, just like it was a topic. The rules are all `u:` responders because you don't want your plan to depend on what kind of sentence the user wrote and the topic will be invoked in responder mode.

Unlike a topic, which is looking for output to the user and will execute rules until that happens, a plan is looking for a declaration of success `^end(plan)` and will execute rules until that happens. Unlike the topic, which is not labeled a failure if it runs out of rules, a plan that runs through the last rule without success is a failure and so reports back to its caller.

If the plan above succeeds, the variable `$$choice` will be set to *horse* (and incidentally `$$distance` and `$$range` will also be set).

If a plan does fail, then the world reverts to a state where the plan was apparently never called. That is, any facts that were created are erased. Any variables that were changed are set back to their original state. Any facts that were deleted are restored. Any output generated by the plan will be discarded. So if the above plan fails (the horse can't go the needed distance) then the variable `$$choice` will be undefined.

Similarly when any rule fails, changes made by that rule are reverted as though the rule had not been tried. A call to plan looks like an ordinary function call, e.g.:

```
topic: ~test (test)
u: (test) ^move(Finland Russia)
```

Mostly plans should call plans and functions and manipulate facts and variables. And while you are in the midst of these calls, there are some things you cannot do. You cannot call `^gambit`. No rejoinders will be set on any rule that generates output. Plans never erase their rules (they are like a topic declared `SYSTEM` or `KEEP`). They never become an interesting topic. You cannot do a reset on a user, or an enable or a disable on a rule. You cannot call any print routines from within a plan. There are a host of other functions you SHOULDN'T call because they cannot be undone if the plan fails, but they are not explicitly blocked either.

Because a plan is a function, it returns to the caller whatever output it tries to generate, rather than putting that output directly to the user as a topic rule would. If multiple rules generate output, it will concatenate them together, separated by a space. The caller can thus print the output (default) or save it to a variable.

So far there is only minimal excitement to having a plan. If it fails, it doesn't change anything significant. But there's more.

Backtracking Plans

You can declare multiple copies of the same plan name. If the first copy of the plan fails, the system will find the next copy and try that, and so on. Plans are tried in the order they are compiled.

```
plan: ^move (^from ^to)

u: () $$distance = ^distance(^to ^from)
    $$choice = car
    query(direct_sv $$choice range ?)
    $$range = @0object
    if ($$range >= $$distance) {end(PLAN)}
```

The above plan is similar to the first, except I wrote the code differently (not significant) and I use a choice of car instead of horse. So if the horse can't go the distance, maybe a car can. So we can end up with two different forms of transport, depending on how far apart the endpoints of travel are. But there's more. Writing multiple copies of a plan just to change the resource to be used is redundant. Instead, you want an iterator with automatic backtracking within a plan.

Automatic backtracking

Trying different plans automatically is all very well and good for picking different actions, but what really makes a plan work is picking among a choice of resources (objects) to use without having to make separate plans to do it. So, here's how we do that in ChatScript.

A rule that fails does not just erase its changes and stop immediately. It sees if it can backtrack and try again. Perhaps this rule is known by the system to have gotten one of many possible values and so it could try to get a different value. With a new value in hand, it can try again. There are two kinds of backtracking operations, static and dynamic.

Simple Iterator

The first backtracking operator involves a new engine function `^iterator()` which takes three arguments, representing the subject, verb, and object of some series of facts. It's like a simple `^query()` function. You will name the verb to use, and either the subject or the object to base the set on. The remaining field will be `?`, meaning you are interested with results of that value.

```
concept: ~list0( car horse ~list1 carriage jumper)
concept: ~list1 (bird bee ~list2)
concept: ~list2 (animal vegetable)


plan: ^pickitem ()
    u: () $$tmp = ^Iterator(? member ~list0) Log(OUTPUT_ECHO $$tmp ^" " ) fail(rule)
```

Above is a hierarchically nested set of concepts and a plan to pick something from `~list0`. The rule here is a simple example of backtracking an iterator. The iterator says to return the subject of facts whose verb is member and whose object is `~list0`. Such facts, are the members of that concept, and this is exactly how the system stores concept members internally, as facts. The value from the iterator is returned and stored on `$$tmp`. The iterator will also have stored backtracking information. The rule then echos that value and a space into the log file (and onto the user's console in stand-alone mode). And then the rule fails. That's where it gets interesting.

When the rule fails, the plan will find the backtrack point of the iterator. It ignores the pattern component and merely re-executes the output side of the rule. After which it would continue normally with the rules after it (but there aren't any in this example). So what this iterator will do is walk the concept set and display all the members until it fails to find any more. The iterator is exhausted. What you would see is:

```
car horse ~list1 carriage jumper
```

It is called a *static backtrack* because all the data needed to move on to the next value is built into the system, you have nothing you have to maintain on your own. This is a very efficient operation with no chance for scripting errors.

Recursive Iterator

The iterator can also recursively walk the concept hierarchy, if you replace the ? value with ??. This tells it to go into any concept it sees rather than just retrieving the concept name.

```
plan: ^pickitem ()
    u: () $$tmp = ^Iterator(?? member ~list0) Log(OUTPUT_ECHO $$tmp ^" " ) fail(rule)
```

This plan would print out:

```
car horse bird bee animal vegetable carriage jumper
```

before failing.

Note: A loop blocks an iterator from backtracking in a rule. However it also establishes its own iteration context, so you can always use an iterator within a loop, even if a prior part of a rule has established a backtracking context.

Backtracking with fact-sets

The second backtracking collection happens with fact-sets. If you grab a fact out of a factset, there might have been other choices to grab. So the system will automatically back up to that rule and try the next fact from the set. So using `^pick`, `^first`, and `^last` will create automatic backtrack points. This will work correctly only if you don't go overwrite that factset in something called from that rule. And you want to be careful not to initialize the factset in the output part of the same rule as you use it. So a good plan might look like this:

```
plan: ^move(^from ^to)
    u: () $$distance = ^distance(^to ^from)

    u: (query(direct_sv ^from available ? -1 ? @1))
        $$currentVehicle = ^pick(@1)
        query(direct_sv $$currentVehicle range ?)
        $$range = @0object
        if ($$range < $$distance) {^fail(rule)} else {END(PLAN)}
```

The above plan selects from transport available at ^from that has the range (fuel presumably) to reach the goal. It does this by

1. computing the distance to the goal-oriented
2. acquiring in factset 1 all vehicles at the starting point and picking one at random
3. Getting the range of said vehicle and if it does not have the appropriate range, failing.

Failure will return to the automatic fail point of ^pick. It will re-execute the output of that rule (avoiding re-querying the available vehicles into set 1) and pick another vehicle. If that fails, the rule fails and it will move on to another rule.

If you can use an `^iterator()` instead of a fact-set you are better off. It will be more efficient and you can't make any logical scripting mistakes. For particularly large sets of facts, you don't really want to get them all into a factset at once. But if your facts are being dynamically located, fact-set backtracking may be your only choice.

Only One Backtrack point per Rule

Since the definition of backtracking is to return to the rule's output side and reexecute it from the start, do not try to have more than one backtrackable thing in a rule. It won't know how to handle it. In fact, it will declare it a runtime error when it happens.

When a rule is done, it's done. No backtracking from outside it.

When a rule is done, any choices it might have remaining are ignored. No failure of a later rule will trigger a backtrack into an earlier rule.

How to think about planning

To perform planning, the chatbot has to have a mental model of the world. This means creating facts. If you were building a Command and Conquer style game, you'd represent locations of resources like:

```
(city1 silver 5) - city1 has 5 units of silver
(city1 gold 1)
```

and costs of production like this:

```
(jewelry gold 1) - jewelry costs 1 unit of gold
(jewelry silver 1)
```

Your plan will be hunting for availability of resources and aiming to convert them into objects, such that when it completes its plan, it might have

```
(city1 jewelry 1)
```

You'd write scripts that handle conversion of resources into jewelry (all resources have to be in the same place) and scripts that can move resources from one location to another (maybe picking a vehicle and where to bring the resource from and where it is going to).

During the plan you'd be constantly representing the results of actions by killing off some facts and creating new ones. There is even a new function `^createAttribute()` to help you simultaneously do for some things. E.g., `(city1 gold 1)` means there is 1 unit of gold in city1. You can't simultaneously have 2 other units of gold there if we are treating this as the sum of all gold. So if you want to change to having 2 units of gold there as a result of dropping off another gold there, you have to kill off the old fact and create `(city1 gold 2)`. This is managed automatically by `^createattribute (city1 gold 2)`.

Nominally the execution of a plan will take time. Things are likely to change. So generating a plan that lists all the micro details is often a mistake. Consider, for example, a plan to go shopping at the grocery store. At a high level, there is how you get there, what you buy, and how you get home. How you get there will consider the current weather, do you have car, do you have the money for a

bus or taxi, etc. Once you decide there is at least one way to get there, you can move on to other parts of the plan. Your plan might simply be: go to the store, buy what's on the list, return home. This plan already confirmed you had a car, so you expected to use the car.

When you come to executing the plan, you might flesh out that you have to get the car keys, enter and start the car, and drive to the store. And while executing that sub-plan you discover the car won't start.

Since your original plan step was go to the store, you might have to substitute take a taxi. Then conduct the rest of the plan. So how you represent your overall plan becomes an important issue. If you plan had originally included all the steps of getting the car working, then when that fails, you may no longer understand your plan, because it was all too detailed. Yes, you need to check that likely you can fill in the details of some step, but that doesn't mean those details have to go in the master plan yet.

Fact Creation

When facts are created within a plan, they are all created TRANSIENT. They will disappear at the end of the volley unless you do something explicit to keep them.

Loops

Backtracking is a kind of loop. So, if you do something inside a loop that would mark a backtracking point, it will not do so. For example:

```
plan: ^acquire(^place ^resource ^quantity)

u: (^query(direct_vo ? member '~places -1 ? @4)) # find all places
loop()
    {
    $$place = ^first(@4subject) # get place
    if ($$place != ^place AND ^Has_some_resource($$place ^resource))
        {
        ^Transfer($$place ^resource ^place)
        ^End(PLAN)
        }
    }
```

Normally `^first` would be considered a backtrack point, but since it occurs within a loop, it does not set one up. If you wanted to do backtracking, the way to write this code is:

```
plan: ^acquire(^place ^resource ^quantity)
```

```
u: (^query(direct_vo ? member '~places -1 ? @4)) # find all places

$$place = first(@4subject) # get place
if ($$place != ^place AND ^Has_some_resource($$place ^resource))
    {
    ^Transfer($$place ^resource ^place)
    ^End(PLAN)
    }
```

Styles of Plans

There are a variety of styles of plans you can write. You can write a multiple plans as multiple solution ideas, like to acquire a resource you can have a plan to mine it, a plan to steal it, a plan to scrap something else to reuse a resource, etc. Or you can write a single plan that encompasses all the ways one might accomplish the goal.

Within a plan you can make each rule test conditions and select a particular method to solve the plan. Naturally you order these rules by the "best" first according to some criterion. Or rules of a plan might be progressive. Rules do some of the work and if the task is incomplete, later rules do more of the work. Or earlier rules might merely initialize data that later rules will depend upon, like selecting a vehicle that will be used by later rules to transport material from place to place.

Passing knowledge from a failed plan

The planner is designed for efficiency, leaving it to you to write plans that try things in some best order. But I could imagine a situation where you wanted to try out a bunch of plans, rate them, and then pick the best one on some criterion. Of course to try out a plan, at the end of it you have to fail it to let the world return to a state where you can try out the next plan.

Failing means you lose facts and variablesnyou might have used to store value information about the plan. So what can you do?

There is a cheat. The system does not restore changes to the _ match variables. So you can pass information around in them, concatenating text strings or whatever and then at the end decoding your message. A match var is only good for holding about 1500 bytes of data, so you do have limits, but match variables above 5 are rarely ever used by pattern matching, so they are free to be used by you during a volley any way you wish.