

EIGHT: STORY GENERATION WITH HTN PLANNING

Most programming languages provide you with some mechanism for breaking tasks up into simpler tasks, making a hierarchical network of tasks used to achieve an overall goal. When they allow you to specify several different ways of decomposing a task into other tasks – different methods – and leave it to the computer to try methods until it finds a sequence of methods that achieve the original goal, they're called **non-deterministic** languages. When they include operations like `set`, that allow you to update the value of a variable, or otherwise change the representations in the program, they're called **imperative** programs. When a language combines these two capabilities, as *Step* does, it is sometimes referred to as a **planner**. Indeed, *Step* stands for "Simple TExt Planner".

The notion of planning in AI comes from the desire to build things like robots that can do things in the world. At any moment, there are many different actions the robot can perform, and there are different goals it might want to achieve. However, the order of the actions matters. For example, if the robot is trying to make sautéed onions, it might put the onion in the pan, start heating it, and then decide to start chopping it in the pan. That would not work out well. It would work out much better if it chopped it first and then cooked it. The question is how the robot could figure that out if it didn't already have experience making sautéed onions.

The original idea of planning programs in AI was to run the actions in simulation, so if they did something stupid they could just undo it and try something different. They might still bumble around, but without negative consequences to the bumbling. At the end, when it's found a sequence of actions that works, it can run that sequence in the real world.

So a planner needs some way of:

- Representing the state of the world
- Specifying how actions modify that state (imperatives)
- Exploring possible sequences of actions until it finds a good one (non-determinism)

So that's the sense in which planners are non-deterministic, imperative languages.

This sort of planning capability is useful for story generation because characters are kind of like robots: they have goals they're pursuing and actions they can perform; but in the process of writing a story, you might "write yourself into a corner" where you realize late in the story that something earlier in the story needs to be changed to accommodate the ending that you want. Story planners try to do some version of this. Most of the story generators in AI research can be characterized as some sort of planner.

Planning algorithms can be very sophisticated, but *Step* is a fairly simple system. It gives you a way of specifying things to happen in the story, ways of specifying how the story world changes, and then a basic mechanism (backtracking) for deal with dead ends.

PLANNING IN STORY GENERATION

We saw a very simple version of this when we talked about generating pronouns based on discourse context. When you mention a character, that changes the discourse context of the story so that the next time that

character is mentioned, the story may be able to refer to them with a pronoun rather than their full name. The system does this by always trying to use a pronoun first, and only printing the full name if it can't use a pronoun. One could argue that this is too simple to really be called planning. Real planning is when it's possible for the system to generate a whole series of plot events, and then realize it needs to go back and change things its already generated. We'll talk about a more substantive example shortly.

FLUENTS

A planner needs some way of tracking how and when the story world has changed. You've already seen one way of doing that, using global variables and `set`. However, it's more common for planners to represent the world in terms of predicates, and that means that predicates need to be able to change their truth values over time as the plan being considered changes the world. These changing predicates are often called *fluents*, which just means a predicate whose truth values can change. In *Step*, you can change the value of predicates using an equivalent of `set` called `now`. As with `set`, the changes are undone if the system has to backtrack over the change.

The command `[now [fluent parameters ...]]` tells the system that subsequent calls to the *fluent* with the specified *parameters* should succeed. For example, if we want `Mention` to print the full name of a character the first time they're described, but just their first name after that, we could write it this way:

```
fluent Mentioned ?who.  
Mention ?who: [Mentioned ?who] ?who/FirstName.  
Mention ?who: ?who/FirstName+LastName [now [Mentioned ?who]].
```

The first line just tells the system that `Mentioned` is a fluent and that it takes one parameter. The second line gives a method for `Mention` that runs any time `Mention`'s parameter has already been `Mentioned`. The last line runs when we `Mention` something that hasn't already been `Mentioned`. It prints the character's full name and then notes that henceforth they're considered to have been `Mentioned`.

We can also reverse the sense of `now` by using `Not`:

```
Kill ?who: ?who died. [now [Not [Alive ?who]]]
```

So if David is currently alive, i.e. `[Alive david]` currently succeeds, then after running `[Kill david]`, it will print "David died," after which `[Alive david]` will fail.

FUNCTION FLUENTS

Predicates, and therefore fluents, represent **relationships** between their parameters, or if they only have one parameter, they represent yes/no properties of that parameter. But there's an important special case of relationships where one of the parameters is determined by the others. For example, if we want to represent the locations of objects using a predicate `[At ?what ?where]`, there should only be one *?where* for any given *?what*, since objects can't be in two places at once. So if we start out with `[At bill garden]`, meaning Bill's in the garden, and then we say `[now [At bill school]]`, we don't mean that Bill is now simultaneously in the garden and at school, we mean Bill is now *only* at school. So we want the system to understand that `[now [At bill school]]` also means `[now [Not [At bill garden]]]`.

These kinds of relationships where one parameter is determined by the other parameters are called **functions** in math and logic. And by convention, in languages like *Step*, when we have a function, we place the parameter that's determined by the other parameters at the end. In *Step*, you can tell the system that a fluent is also a

function, and it will understand that when you use `now` to update the fluent, it should remove any conflicting values for the final parameter. To tell the system that a fluent is a function, we just add the `[function]` annotation to it:

```
[function]
fluent At ?what ?where.
```

The `[function]` annotation will tell it that a given *?what* can only be `At` one *?where* at a time.

HIERARCHICAL TASK NETWORKS

Out of the box, *Step* behaves like a particular kind of planner, called a hierarchical task network or HTN planner. In particular, it's very similar to Nau et al.'s popular planner, *SHOP*. *SHOP* lets you describe a planning problem in terms of *tasks* to be achieved and *methods* for achieving the tasks, which can call other tasks. Sound familiar? You ask the planner to solve a problem by running the appropriate task, and it searches and backtracks until it finds a solution or gives up.

Here's a kind of standard example. Suppose we want to be able to plan how to go from place to place. Depending on how far the start and destination are, you might drive or fly. If we assume that you drive between places if they're the same general area, but fly if there in very different areas, then we need to keep track of what areas different locations are in:

```
[predicate]
MetroArea home chicago.
MetroArea northwestern chicago.
MetroArea field_museum chicago.
MetroArea tomate chicago.
MetroArea ohare chicago.
MetroArea moscone san_francisco.
MetroArea sfo san_francisco.
```

This says that home, Northwestern, the Field Museum, Tomate, and O'Hare are in the Chicago metro area, and so you can drive from one to the other. However, Moscone center (where the Game Developer Conference is held) and SFO (San Francisco airport) are in the SF metro area.

Now we can tell it what the airports for the different areas are:

```
[predicate]
Airport ohare.
Airport sfo.

[predicate]
AirportServing ?location ?airport:
  [MetroArea ?location ?city]
  [Airport ?airport]
  [MetroArea ?airport ?city]
[end]
```

The `Airport` predicate just tells which of the locations defined above are airports. The `AirportServing` predicate figures out from that what airport or airports serve a given location by finding an airport that is in the same metro area and the location we want to fly from.

Having done this, we can now describe different modes of transport:

```
# If you're already there, you're done
Goto ?who ?destination: [At ?who ?destination]

# If you're going to the same city, take a taxi
Goto ?who ?destination:
  [At ?who ?start]
  [MetroArea ?start ?city]
  [MetroArea ?destination ?city]
  ?who took a taxi from ?start to ?destination.
  [now [At ?who ?destination]]
[end]

# If it's in a different city, you need to fly there.
Goto ?who ?destination:
  [At ?who ?start]
  [AirportServing ?start ?startAirport]
  [AirportServing ?destination ?destAirport]
  [Goto ?who ?startAirport]
  ?who flew from ?startAirport to ?destAirport.
  [now [At ?who ?destAirport]]
  [Goto ?who ?destination]
[end]
```

Let's not worry about the first method here yet. The second one says if you're trying to go someplace, and your destination is in the same metro area as your current location, just drive there (or rather, print text saying you've driven there) and then update your location.

The third method only runs if you're trying to go someplace you can't drive to. It looks up what the airport is for your origin city and drives there by running `Goto` again. Then it flies to your destination city (prints out that you've flown there) and changes your location to that airport. Finally, it goes to your final destination.

So what's the first method there for? Well, if you were to say `[Goto jayden sfo]`, and Jayden was in Chicago, the system would have Jayden drive to O'Hare, then fly to SFO, but then the third method (which handles flying) would run `[Goto jayden sfo]` again, since that method flies to the relevant airport, but then `Gotos` to the actual destination, since those are usually different. That would cause it to print a spurious "Jayden took a taxi from sfo to sfo" message, which would look stupid. So we include an extra method at the start that checks if we're already at the destination, and if so just succeeds without printing anything.

This is a very simple example, but we could make it more complicated if we wanted to. For example, we could make it understand you might walk to nearby places rather than drive, or that you might take trains or subways to some places. But those make the example complicated enough that it isn't worth walking through the code in this reading.

HIERARCHICAL TROPE NETWORKS

Let's get back to story generation.

One way of crafting a story generator is to start with a common narrative structure and write each part of the structure as an HTN task. Let's start with this:

- There once was a heroine
- They have a magic item
- A monster killed their loved one
- In revenge, she killed the monster with the magic item

We can write this as:

```
MonsterStory:
  [IntroduceProtagonist]
  [Paragraph]
  [MonsterHurtsLovedOne]
  [ProtagonistKillsMonster]
  [Paragraph]
  [Coda]
[end]
```

INTRODUCING THE PROTAGONIST

Now let's fill in the gaps.

```
IntroduceProtagonist:
  There once was a girl who lived in Chicago. Her prized possession was a
  magic sword.
[end]
```

That's great, but it's now fixed that the protagonist is a girl who lives in Chicago with a magic sword. Let's change things so that we can pick random story worlds with different protagonists, locations, and magical artifacts (changes shown in **green**):

```
IntroduceProtagonist:
  [Protagonist ?p] [Home ?h] [MagicalArtifact ?a]
  There once was a ?p who lived in ?h. Their prized possession was ?a.
[end]
```

In order for that to work, we need to store the cast of the story in a fluent:

```
# Tracks what characters are cast in what roles.
fluent Cast ?role ?character.
```

We could then define Protagonist to just check this fluent to find out who the protagonist is:

```
Protagonist ?who: [Cast protagonist ?who]
```

CASTING CALL

The problem with this is that we then need some separate code to decide who the protagonist is and fill it into the fluent. So we can be a little fancier:

```

# Return the character cast in the role, or cast one if none has yet been cast
Role ?role ?character: [Cast ?role ?character]
Role ?role ?character:
    [Possible ?role ?character]      # Find someone who can do the role
    [Not [Cast ? ?character]]        # Who isn't already cast in another role
    [now [Cast ?role ?character]]    # And cast them
[end]

```

The first method here checks if someone has already been cast in the role. If so, it just returns them. The second method runs when nobody has been cast for that role. It picks a random character who is allowed to fill the role, makes sure they haven't already been cast in another role, and then updates the story world to reflect that henceforth they have that role. We can now define the Protagonist and friends in terms of Role:

```

Protagonist ?who: [Role protagonist ?who]
Monster ?who: [Role monster ?who]
LovedOne ?who: [Role lovedOne ?who]
Home ?where: [Role home ?where]
MagicalArtifact ?what: [Role artifact ?what]

```

And we can define the various candidate characters and objects for these roles by filling in facts for the Possible predicate:

```

[randomly] [predicate]
Possible protagonist girl.
Possible protagonist boy.
Possible protagonist prince.
Possible protagonist princess.

Possible home shoe.
Possible home cottage.
Possible home castle.

Possible monster vampire.
Possible monster werewolf.
Possible monster bandit.
Possible monster sheriff.

```

And so on (we won't list all the possibilities here). There are some problems with this, however. For example, we could cast a prince or princess as the protagonist but have them living in a cottage, which doesn't fit the usual rules of fairy tale story worlds. We can fix that by saying cottages aren't viable for royalty:

```

Possible home cottage: [Not [Protagonist prince]] [Not [Protagonist princess]]

```

Or define some general CanLiveIn relation and change the definition of Home to:

```

Home ?where: [Role home ?where] [Protagonist ?who] [CanLiveIn ?who ?where]

```

Which will force Role to recast the home role until it finds one that's genre-appropriate. Or you might not care. It's up to you as the author what kinds of weirdness you do or do not want to tolerate in your generated stories.

THE INCITING INCIDENT

Now the monster comes on the scene:

```

MonsterHurtsLovedOne:

```

```

[Monster ?m] [LovedOne ?l] [Protagonist ?p]
One day, a ?m killed ?l. ?p was distraught.
[end]

```

The monster shows up, kills a randomly selected loved one, and sadness ensues.

The only problem with that is that we get exactly the same story each time, with just the casting changing. So we'll let the story planner to switch things up. We'll put in two different versions of the loved one getting hurt, and let the system select choose randomly:

```

fluent Killed ?who.
fluent Infected ?who.

```

```

[randomly]
MonsterHurtsLovedOne:
  [Monster ?m] [LovedOne ?l] [Protagonist ?p]
  One day, a ?m killed ?l. [now [Killed ?l]]
  ?p was distraught.
[end]

```

```

Infectious vampire.
Infectious werewolf.
MonsterHurtsLovedOne:
  [Monster ?m] [Infectious ?m]
  [LovedOne ?l] [Protagonist ?p]
  One day, a ?m bit ?l, turning ?l into a ?m. [now [Infected ?l]]
  ?p was distraught.
[end]

```

So when it goes to generate the inciting incident, it will randomly choose which method to try first. But notice that the new method (listed second), where the monster bites the loved one, only applies to monsters that are Infectious. That's coordinated by the new code shown in **green**. So if the monster is just a bandit, then the system will skip over the biting version of the story, even if it tries that one first.

This change also means that the stakes are different in the two different version of the story. One version, where the loved one is killed, is a revenge story. The protagonist is killing the monster for payback. But the other, where the monster infects the loved one, is a rescue story: the protagonist is trying to kill the monster to save the loved one. So to handle that, we need some new fluents to let us keep track of what the current stakes are in the story world. This new code is shown in **blue**.

THE CLIMAX

In a real story, there would be a lot more going on. The protagonist might have to search for the monster, or seek guidance from an elder to learn how to defeat the monster. Many real horror stories are really mystery stories and so the protagonist would spend much of the story simply learning that monsters are a thing and that little Billie is acting so strangely because he was bit. However, since this reading is already 7 pages, we're going to jump straight to the climax where the protagonist dispatches the monster. Here we go:

```

ProtagonistKillsMonster:
  [Monster ?m] [Protagonist ?p] [MagicalArtifact ?a]
  ?p killed ?m with ?a.
[end]

```

This is straightforward: we look up the monster, protagonist, and artifact, and generate the text. The only problem is that while a bandit can probably be killed by anything, in most horror story worlds, vampires and werewolves can only be killed by specific kinds of objects. So this is only going to work if the protagonist's artifact happens to be the right kind of thing to kill that kind of monster. We can change the method to understand this:

```
[fallible]
ProtagonistKillsMonster:
  [Monster ?m] [Protagonist ?p] [MagicalArtifact ?a] [CanKill ?a ?m]
  ?p killed ?m with ?a.
[end]

[predicate]
CanKill ? (Human ?who).
CanKill silver_pistol ?.
CanKill battle_crucifix ?.
```

The **green** code here checks the monster and weapon are compatible. The methods for CanKill say that any weapon can kill a human (first rule) and that silver pistols and battle crucifixes can kill anything.

The only problem now is that if the artifact can't kill the monster, then ProtagonistKillsMonster fails completely because there's only one method for it. One could add other methods to ensure that every protagonist will somehow be able to kill every monster. But let's not do that. Let's just allow ProtagonistKillsMonster to fail. What happens then?

Well, normally, since ProtagonistKillsMonster is a normal task and not a predicate, Step would assume that was a bug and print an error message. But since that's the intended behavior here, we've added the annotation [fallible], which tells step we know it might fail and so that shouldn't generate an error.

When ProtagonistKillsMonster fails, *Step* falls back and tries to redo previous steps of the story, looking for something that will work. It will eventually go all the way back to when it chose what magical artifact to use, pick a different one, and then start generating the story again. It will keep doing that until it finds a monster/artifact combination that works.

So as a writer, you can write the code pretending that *Step* will always guess the right choice of weapon by looking into the future. It won't – it will pick randomly and keep trying things until it finds something that works – but as an author, that's hidden from you and you don't have to worry about it. You know that the system will make a correct choice provided there's a correct choice to be made.

THE EPILOG

Finally, we end the story with something about how the protagonist feels. This is a common structure called "scene and sequel", although here each is only a line or two. First, we have some action (the scene) and then we reflect on how it affected the characters and how they respond to it (the sequel).

In this case, what they feel will depend on what their motivation is. Their motivation depends on what the original harm was that the monster caused. So we have two different methods, that check which of the fluents, shown in **green**, was set previously in the story:

```
Coda:
  [Killed ?lovedOne]
```



```
[Protagonist ?p] [Monster ?m]
?p knew this act of revenge wouldn't bring ?lovedOne back. But at least
?m wouldn't be hurting any others.
[end]
```

Coda:

```
[Infected ?lovedOne]
[Protagonist ?p] [Monster ?m]
The moment ?m died, ?lovedOne passed out. ?lovedOne slept for many days,
but when ?lovedOne awoke, ?lovedOne was human again.

?lovedOne and ?p shared a tearful reunion.
[end]
```

This way, we get an ending appropriate to the earlier story events, whatever they may have been.

CRITIQUE

To keep things simple, we intentionally didn't include code for Mention, selection of pronouns, or printing things in different cases (subject, object, possessive). But when we add that stuff in, here are some stories you get from the generator:

- There once was a boy who lived in a castle. His prized possession was his battle crucifix. One day, a vampire bit his father, turning him into a vampire. The boy was distraught. The boy killed the vampire with his battle crucifix. The moment they died, his father passed out. He slept for many days, but when he awoke, he was human again. He and the boy shared a tearful reunion.
- There once was a boy who lived in a shoe. His prized possession was his silver pistol. One day, a werewolf bit his girlfriend, turning her into a werewolf. He was distraught. The boy killed the werewolf with his silver pistol. The moment they died, his girlfriend passed out. She slept for many days, but when she awoke, she was human again. She and he shared a tearful reunion.
- There once was a princess who lived in a castle. Her prized possession was her magic sword. One day, a sheriff killed her mother. The princess was distraught. The princess killed the sheriff with her magic sword. She knew this act of revenge wouldn't bring her mother back. But at least they wouldn't be hurting any others.
- There once was a princess who lived in a cottage. Her prized possession was her battle crucifix. One day, a dragon killed her boyfriend. She was distraught. The princess killed the dragon with her battle crucifix. She knew this act of revenge wouldn't bring her boyfriend back. But at least they wouldn't be hurting any others.

- There once was a prince who lived in a shoe. His prized possession was his battle crucifix. One day, a vampire bit his boyfriend, turning him into a vampire. The prince was distraught. The prince killed the vampire with his battle crucifix. The moment they died, his boyfriend passed out. He slept for many days, but when he awoke, he was human again. He and the prince shared a tearful reunion.

These would be impressive if they were made by a toddler, but nobody is going to consider them important literature or a gripping diversion. There are a bunch of issues with it.

One issue is that the stories are very short. They're barely stories at all. In some sense, that's a matter of just writing more text for the plot points (introducing the protagonist, etc.), and breaking them up into more steps so that more things happen. Rather than just saying that the prince's favorite possession was his silver pistol, we might say something about where it can from and what it looked like. Rather than just saying that he's distraught, we might narrate a montage of him crying himself to sleep, walking in the rain, and so on. That's obviously more work, but it's in some sense a straightforward matter of writing more text and more code for generating descriptions of random silver pistols.

Another issue is that there's basically just one story, or if you're generous, two stories: kill the monster for revenge and kill the monster to save your loved one. But apart from that, the only variation is in what characters and objects fill the different roles, and how those influence where the system can and can't use a pronoun. As it stands, it's a fancy MADLIB generator.

We can increase the range of variation in the stories by adding more methods for the different plot points and more methods for MonsterStory so there can be more than one basic story spine to choose from. Again, that's a lot more work. But it's certainly doable.

So we can make the stories longer and we can make them more varied. But in the end, this approach to story generation will always be about generating variations on a set of basic tropes that the author provides. The generator isn't going to innovate. One can argue that much popular fiction is really riffing off of well-established tropes, but AI story generation is still a long way from being able to do the kinds of things that even bad human authors can do.

WHAT'S IT GOOD FOR?

So why would you want to use this if it's only riffing off of story patterns provided by a human author? There are a few possible answers.

One is that it's common in digital games to automatically generate variations on fixed patterns to give the game replayability and a sense of freshness, even though much of it is the same. One example is quest generation. In games like Grinblat and Bucklew's *Caves of Qud*, the game automatically generates random quests for you. In fact, it generates the whole planet and the names for everything afresh each time you play it.

Quest generation comes up in many genres. There have been quest generators built into MMORPGs, and there are any number of dice-based and web-based random quest generators for table-top RPGs. The rule book for Hill and Young's *iHunt* RPG that we played in class includes such a dice-based quest generator, although in that story world, it's referred to as a gig generator.

There are also a number of tabletop games that use *modular narrative*, including Ken Hite's *The Dracula Dossier*, which we also played in class. In these games, the rule books provide a set of reusable characters, places, organizations, needs, monsters, and so on, along with guidelines for combining and recombining them into different overall stories. So for example, a horror game such as Robin Laws' *The Armitage Files* might describe an off-putting character who owns an art gallery that specializes in macabre art that involves occult themes, without specifying whether they're good (aware of the occult and trying to prevent the evil cults from bringing about Armageddon), evil (a member of the cult in question), or neutral (just a dweeb who's into creepy stuff or perhaps a huckster trying to make a buck, unaware that there really is an occult world).

These are all areas where story generators like this that are basically just recombining elements provided by a human author, could potentially be used.

The basic approach to story generation outlined here comes from a hybrid analog/digital part game, Horswill's *Dear Leader's Happy Story Time*. It used the kinds of techniques described here to make random stupid stories that humans then play out. For that game, the idea was to lean into the badness of AI-generated stories, and play them for camp, hence it's use as a party game.

EFFICIENCY AND SCALABILITY (FOR CS MAJORS)

Finally, computer science majors may be concerned about how long it takes a planner like this to actually generate a story. After all, our system's solution to cases where it's written itself into a corner, for example, but giving a weapon to the protagonist at the beginning of the story that can't actually be used to kill the antagonist at the end, is to use the backtracking mechanism built into *Step*. If the generator fails toward the end of the story, it backs up a little and tries again. If that doesn't work, it eventually backs up further, and then further still, until it finally picks a different weapon.

This is a simple, but stupid algorithm. It potentially forces the planner to generate huge numbers of partial stories that it just throws away. It's not much of a problem if the system only needs to backtrack a step or two to fix the problem. But if it really has to correct a decision it made 50 steps earlier, it won't succeed until it's exhaustively tried all the possibilities for the 49 decisions it tried after the offending decision. The number of possible combinations that need to be ruled out before revisiting the offending decision grows exponentially with number of decisions made between the initial bad decision and the moment when it catches the problem. That puts a practical limit on the length of story that planners like this can generate, unless we do something to limit the use of backtracking, which somewhat limits the usefulness of using a system like *Step* in the first place.

We can ameliorate this somewhat by rearranging things. For example, if we know the magical artifact will be used to kill the monster, then we don't have to wait until the end to specify that it should be able to kill the monster. We could specify at the start of the story, when we're first doing the casting. But that only works if we know in advance that the artifact will definitely be used to kill the monster. If it might be used to kill the monster or might be used for something else, then we need to wait to check the applicability of the artifact until we know what it's going to be used for.

It's for reasons like this that there's literally been 50 years worth of work done on designing smart planning algorithms. This is the naïve one that people thought of first, and it's one that fits easily into the same framework that we're using for inference and text generation. It's also very "authorable" in the sense that it's easy for you as an author to guide it in the directions you want. But there are more sophisticated ones that are more efficient.

One popular approach to designing efficient planners is to use *partial-order planning*. That means the planner doesn't construct the plan (the story) from start to end, but instead gradually makes note of events that it knows need to happen at some point, along with information about what events need to happen before what other events. Only once it's determined all the plot events, does it worry about what order to arrange them in. Unlike our story generator, a partial-order planner could have deferred even deciding there should be an artifact until it had already decided the monster should be killed by an artifact.

Another popular approach is SAT-based planning, where a planning problem is transformed into a SAT problem and then solved using a SAT solving algorithm, such as we discussed in class.