Chapter 7
Planning with applications to quests and story (DRAFT)

Yun-Gyung Cheong, Mark O. Riedl, Byung-Chull Bae, Mark J. Nelson

Games often have storylines. In some games, they are short backstories, serving to set up the action. The first-person shooter game *Doom*'s storyline, about a military science experiment that accidentally opens a portal to hell, is perhaps the canonical example of this kind of story: its main purpose is to set the mood and general theme of the game, and motivate why the player is navigating levels and shooting demons. In other games, the storyline structures the progression of the game more pervasively, providing a narrative arc within which the gameplay takes place. The *Final Fantasy* games are a prominent representative of this style of game storyline.

Since the theme of this book is to procedurally generate anything that goes into a game, it will not surprise the reader that we will now look at procedurally generating game storylines. As with procedural generation of game rules, discussed in the previous chapter, procedural generation of storylines is somewhat different from generation of other kinds of procedural *content*, because storylines are an unusual kind of content. They often intertwine pervasively with gameplay, and their role in a game can depend heavily on a game's genre and mechanics.

A common way of integrating a game's storyline with its gameplay, especially in adventure games and role-playing games, is the *quest* [23, 1]. In a quest, a player is given something to do in the game world, which is usually both motivated by the current state of the storyline, and upon completion will advance it in some way. For example, the player may be tasked with retrieving an item, helping an NPC, defeating a monster, or transporting some goods to another town. Some games (especially RPGs) may be structured as one large quest, broken down into smaller sub-quests that interleave gameplay and story progression.

There are several reasons a game designer might want to procedurally generate game stories, beyond the general arguments for procedural content generation discussed in Chapter 1. One reason is that procedurally generated game worlds can lack meaning or motivation to the player, unless they are tied into the game story by procedurally generating relevant parts of story along with the worlds. As Ashmore and Nitsche [2] argue, "without context and goals, the generated behaviors, graphics, and game spaces run the danger of becoming insubstantial and tedious".

A second reason is that proceduralizing quests can make them truly *playable*. Sullivan *et al.* [22] note that computer RPGs often have a particularly degenerate form of quest, “generally structured as a list of tasks or milestones”, rather than open-ended goals the player can creatively satisfy. Table-top RPGs have more complex and open-ended quests, since in those games, quests can be dynamically generated and adapted during gameplay by the human game-master, rather than being prewritten. Procedural quest generation gives a way to bring that flexibility back into videogame quests.

7.1 Procedural story generation via planning

One way to think about procedurally generating stories is to consider them to be a *planning* problem. In artificial intelligence, planning algorithms search for sequences of actions that satisfy a goal. A robot, for example, plans out the series of actuator movements necessary to pick up an object and carry it somewhere.

What are the sequences of actions for a story, and what is the goal? There are a number of ways to answer those questions, and researchers on procedural story generation started looking at them in the 1970s—at the time, generating purely text-based short stories, not game stories.

One answer is that a story is a sequence of events in a story world (in our case, a game world), which eventually lead, through the chain of events, to the story’s ending. This kind of story generation operates by a sort of world simulation: we tell a story by first simulating what happens as characters move around and take actions in the story world, and then we recount the events that happened. One of the first influential story-generation systems, *Tale-Spin* [14], takes this approach.

Generating stories by simulating a story world has some shortcomings, however. It does not take into account that a story is not exactly the same as a log of events. Stories are carefully crafted by authors to have a certain pace, dramatic tension, foreshadowing, a narrative arc, etc., whereas a simulation of a day in the life of a virtual character does not necessarily have any of these features of a good story, except by accident. Instead, we might want to look at the story-planning problem from the perspective of the author, rather than from the perspective of the protagonist in the story world. Story planning then becomes a problem of putting together a narrative sequence that fits the *author’s* goals [6]. *Universe* [12] and *Minstrel* [25] are two well-known story generators that take this author-oriented approach.

For videogame stories, planning from the perspective of an author can become a more problematic concept, because players act in the game’s story world, rather than in the author’s head. Procedurally generating stories using an approach more like *Tale-Spin*, that takes place within the story world, can be more straightforward, since it has the advantage of talking about the same place and events that the player will be interacting with. On the other hand, we may still want a narrative arc and other author-level goals, which may lead to hybrid systems that plan author-level goals on top of story-world events [13, 19]. Many questions remain open, so procedural story generation in games is an active area of research.

In the rest of this chapter, we'll introduce the concepts and algorithms behind story planning, and walk through examples of using planning to generate interactive stories.

7.2 Planning as search through plan space

Planning can be viewed a process that searches through a space of potential solutions to find a solution to a given problem, when knowledge about the problem domain is given. The problem is called a *planning problem* and consists of the *goal state* and the *initial state*. A solution in planning is called a plan which contains a sequence of actions. A plan is *sound* if it reaches the goal state starting from the initial state when executed. The *domain knowledge* is represented as a library of *plan operators* where each operator consists of a set of *preconditions* and a set of *effects*. Preconditions are just those conditions that must be established for the operator to be executed and effects are just those conditions that are updated by the execution of the plan operator.

A space of potential solutions can be represented as a state space or a plan space. A *state space* can be represented as a tree that consists of nodes and arcs where a node represents a state and an arc represents a state transition by the application of an operator. The root node of the space represents the initial state when the algorithm is forward progression search while the root node represents the goal state when the algorithm is backward regression search.

Here is the pseudo-code description of a state space algorithm.

```
1: construct the root node as the initial state
2: select a non-terminal node
    if non-terminal nodes are not found, return failure and exit
    if the goal state is true, return the path from the initial state
        up to the current node as a solution and exit
3: select an operator applicable
(its preconditions are true in forward progression search and
its effects are true in backward regression search)
    if no such operators are found, mark the node as terminal
        and go to step 2
4: construct child nodes by applying the operator
    if the number nodes in the graph exceeds a predefined
       maximum search nodes, return failure and exit
5. go to step 2
```

A *plan space* can be represented as a tree which consists of nodes and arcs. Unlike state space, however, the root node of the tree specifies the planning problem, the initial state and the goal state. Each leaf node represents a *complete plan* (i.e., solution) which can achieve the goal state from a given initial state when being executed or a partial plan that cannot be refined any more due to inconsistencies

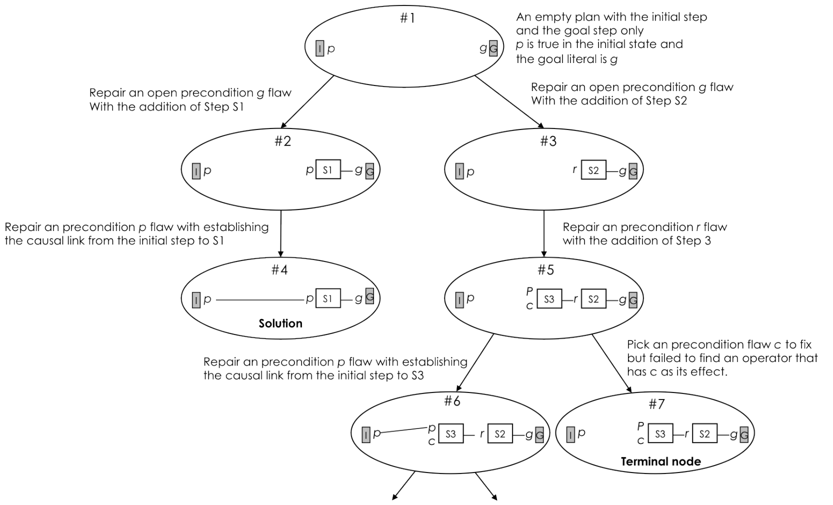

Fig. 7.1: A plan space graph. A box represents an event. A literal on the left side of the box denotes a precondition and a literal on the right side denotes an effect. An effect is omitted if it establishes a causal relationship to a precondition of another event. The root node #1 represents an empty plan that contains the initial and the goal step only. The initial state contains p as an effect and the goal step contains g as its precondition. Each arc between two nodes indicates a refinement of the parent node into a child node. The nodes #2 and #3 are partial plans that repairs the open precondition g by adding two different plan steps $S1$ and $S2$. The node #4 is a complete plan repairing an open precondition p by establishing a causal link from the initial step. The planning algorithm can return the node as a solution and exit. To find all the solutions, the refinement search process continues to generate more children (#5, #6, #7). Although the node #7 is not a complete plan, the algorithm mark this as terminal since no operators that are applicable to repair the open precondition c are found. The further search process to refine the node #6 is omitted due to space limit.

in the plan. Internal nodes represent *partial plans* that contain flaws. The search process can be viewed as refining the parent node into a plan that fixes a flaw of the parent node [10]. A *flaw* in the plan can be an *open precondition* that has not been established by a prior plan step or a *threat* that can undone an established causal relationship in the plan.

Here is the pseudo-code description of a partial-order planning algorithm.

```
1: construct the root node as the planning problem
2: select a non-terminal node (based on its heuristic value)
3: select a flaw in the node
    if no flaw is found, return the node as a solution and exit
4: construct children nodes by repairing the flaw
   if the flaw is an open precondition, either
```

```
       a) establishes a causal link from an existing plan step, or
       b) adds a new plan step whose effects establish the precondition
   if the flaw is a threat, either
       a) add a temporal ordering constraint
          so that the threatened causal link is not intervened, or
      b) add a binding constraint to separate the threatening step
          from the steps involved in the threatened causal link.
   if the flaw is not repairable, mark the node as terminal
        and go to 2
   if the number nodes in the graph exceeds a predefined
       maximum search nodes, return failure and exit
5. go to step 2
```

The complete plans generated by state-space search algorithm are *total-order plans*. A *total-order plan* structure specifies the temporal ordering constraint of every step in the plan while a *partial-order plan* specifies only those temporal orderings that must be established to resolve threats. For instance, imagine that you are given the goal of purchasing milk and bread in a grocery store. The goal can be successfully fulfilled without being worried about which one should be purchased first. And yet, a total-order plan specifies the order of these two purchasing actions and generates two plans: a) to purchase milk first and to purchase bread, and b) to purchase bread first and to purchase milk. On the other hand, a partial-order plan does not specify the ordering constraint and defers the decision until when it is necessary.

In a plan-space search, the search process can be guided by a *heuristic function* which estimates the length of the optimal complete plan, based on the number of the plan steps and the number of the flaws that the current plan contains.

While both state-space search and plan-space search algorithms have advantages, plan-space search planners have been favored in creating stories, because their representations are similar to the mental structure that humans construct when reading a story [24] and their search processes resemble the way humans reason to find a solution [17]. Furthermore, the causal relationships encoded in the plan structure allow further investigation of computational models of narrative, such as story summarization and affect creation [3, 5]. However, Partial-Order Planning (POP) is computationally expensive for its space exponentially grows as the length of the plan increases. Therefore, it has not been used in practical applications.

Hierarchical Task Network (HTN) [21, **?**] is a simple plan-space search that recursively replaces non-primitive actions into primitive actions. Figure 7.2 shows HTN *action schemas* that decompose abstract tasks into primitive tasks.

A simple HTN algorithm is described below. HTN is relevant to generate a story via generating character behaviors.

```
1: construct the root node with an abstract operator
2: select an abstract operator to expand
    if no abstract operators are found and
       all the preconditions are satisfied,
       return the network as a solution and exit
3: select an action schema whose preconditions are true
    if no such methods are found, return failure
4: decompose the abstract operator into sub-tasks
```

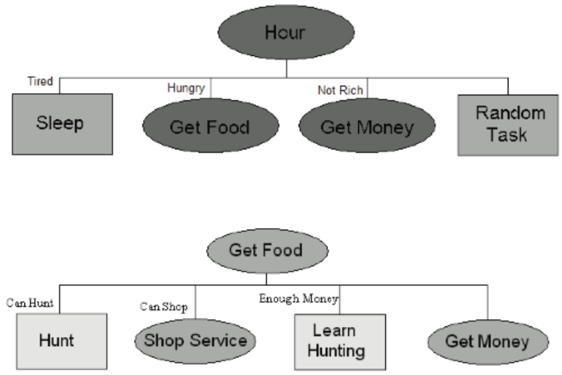

Fig. 7.2: The diagram shows an HTN action schema example where an oval denotes an abstract operator and a rectangle denotes a primitive operator. The method encodes an NPC's activity which can be done for the duration of an hour in the game world. The NPC can sleep if tired or perform a random task. He may want to *get food* if he is hungry. *Get Food* is an abstract task that needs to be decomposed into primitive tasks such as *Hunt* and *Learning Hunting*. [11]

```
     as encoded in the action schema
5. go to step 2
```

7.3 Domain Model

A *domain model* is the library of plan operator templates that encode knowledge in a particular domain (i.e., a story world in this chapter). Various formal languages have been proposed to describe planning problems in terms of states, actions, and goals. This section, among those formal planning languages, focuses on two planning languages (STRIPS and ADL), which have been widely used for classical planners.

Before we get to the formalism, let us take an example. Imagine that a character in a story, named Alex, is on the rooftop of a building. His goal is to be on the ground level of the building without being injured. Alex can think of several plans immediately. For instance, Alex can take a lift (Plan 1), can walk the stairs (Plan 2), or can jump from the roof (Plan 3). Making the decision will consider a variety of constraints such as his capability (e.g., Alex could be an old man having some mobility problems), the building's facility (e.g., lift), his preference (e.g., Alex always prefers walking down the stairs for exercise), etc. If the building has a lift and Alex wants to go to the ground level quickly, Plan 1 would be suitable. Alex may choose Plan 2 if there is no lift in the building. Alex may take Plan 3 if he has a parachute with him and a serial killer with knife is running toward him. As explained above, the goal of planning, often provided with limited resources, is to find a possibly optimal solution that minimizes the cost of executing the plan considering various conditions and preference. Thus, it is important to select a formal language that best expresses the problem domain.

7.3.1 STRIPS-style planning representation

STRIPS, introduced by Fikes and Nilson in 1971 [7], is like a forefather of modern formal languages in planning. In STRIPS-style, a state is represented by either a *propositional literal* or *first-order literal* where literals are ground (i.e., variable-free) and function-free. A propositional literal states a proposition which can be true or false (e.g., p, q, $PoorButler$). A first-order logic literal consists of a relation and objects (e.g., $At(Butler, House)$, $Lord(Higginbotham)$). In STRIPS-style representation, we make a *closed-world assumption* — any conditions that are not explicitly specified are considered as false. Thus only positive literals are used for the description of initial states, goal states, and preconditions. The effects of actions may include negative literals to assure the negativity of particular conditions. An exemplary planning representation of the previous example using STRIPS-style formalization is as below:

- Initial state representation
  $At(Alex, Rooftop)$ $\wedge$ *Alive(Alex)* $\wedge$ *Walkable (Rooftop, Ground)* $\wedge$ *Person(Alex)* $\wedge$ *Place(Rooftop)* $\wedge$ *Place(Ground)*
- Goal State representation
  At(Alex, Ground) $\wedge$ *Alive(Alex)*
- Action representation
  Action (WalkStairs (p, from, to))
  PRECONDITION: *At (p, from)* $\wedge$ *Walkable (from, to)* $\wedge$ *Person(p)* $\wedge$ *Place (from)* $\wedge$ *Place (to)*
  EFFECT: $\neg$*At(p, from)* $\wedge$ *At (p, to)*

In the above example, the initial state is represented by the conjunction of six first-order logic predicates. The goal state is represented by the conjunction of the two predicates in the same manner. In the action representation, the action named $WalkStairs$ has three variable parameters $(p, from, to)$; the action's preconditions are represented by the conjunction of five predicates; and the action's effects are denoted by the conjunction of two predicates including a negative literal. The action $WalkStairs$ will be applicable and executed only when its two preconditions are satisfied. And then after execution, the condition of $At(p, from)$ will be deleted from the current state of the world and the condition of $At(p, to)$ will be added to the current state of the world.

7.3.2 ADL (Action Description Language)

STRIPS is an efficient representation language for modeling states of the world. It can convert, using relatively simple logic description (e.g., a conjunction of positive and function-free literals), the states and actions of a particular domain in the real world into corresponding abstract planning problems. This simplicity, however, can be clear limitations to representing complicated planning problems. Therefore, as an effort to extend the expressiveness of STRIPS, ADL has been introduced as an advanced modification of STRIPS. [16]. Compared to original STRIPS representation, ADL can represent actions and states in a less restrictive way [20, 16]:

- Both positive and negative literals are allowed for the description of states, assuming open-world (that is, any unspecified conditions are considered as unknown).
- Quantified variables and the combination of conjunction and disjunction are allowed in the goal state description.
- Conditional effects are allowed.
- Equality and non-equality predicates (e.g., (from $\neq$ to)) and type in variable (e.g, (p: Person), (from: Location)) are supported.

An ADL-style planning representation of the previous example is shown below:

- Initial state representation
  At(Alex, Rooftop) $\land$ *¬Dead(Alex)* $\land$ *Walkable (Rooftop, Ground)* $\land$ *Person(Alex)* $\land$ *Place(Rooftop)* $\land$ *Place(Ground)* $\land$ *Wearing(Alex, Parachute)* $\land$ *¬Injured(Alex)* $\land$ *Thing(Parachute)*
- Goal State representation
  At(Alex, Ground) $\land$ *(¬Dead(Alex)* $\lor$ *¬Injured(Alex))*
- Action representation
  Action (WalkStairs (p: Person, from: Place, to: Place))
  PRECONDITION:*At (p, from)* $\land$ *(from* $\neq$ *to)* $\land$ *(Walkable (from, to))*
  EFFECT: *¬At(p, from)* $\land$ *At (p, to)*
  Action (JumpFromRooftop (p: Person, from: Place, to: Place, sth:Thing))
  PRECONDITION: *At (p, from)* $\land$ *(from* $\neq$ *to)* $\land$ *Emergent(p)*
  EFFECT: *¬At(p, from)* $\land$ *At (p, to)* $\land$ *(when Wearing(p, Parachute): ¬Dead(p))*

7.4 Planning a Story

A story can be represented as a partial-order plan, a tuple $< S, O, C >$ where

- S is a series of events (i.e., instantiated plan operators),
- O is temporal ordering information represented as ($s1$ ¡ $s2$) where $s1$ precedes $s2$,
- C is a list of causal links where a causal link is represented by (s, t; c) notating a plan step s establishes c, a precondition of a step t

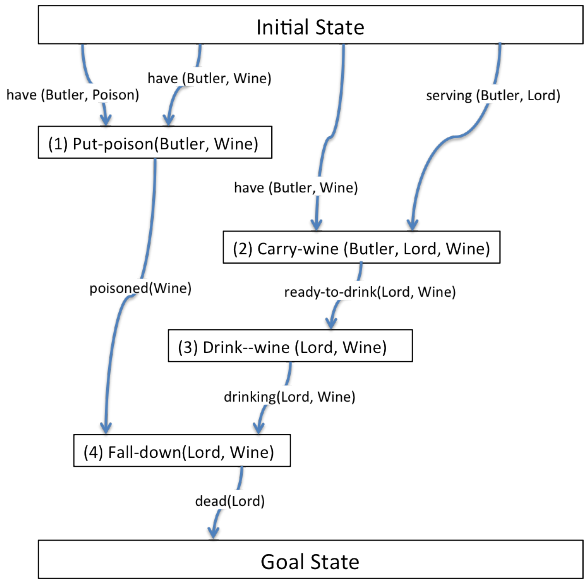

Fig. 7.3: The Butler story. A rectangle denotes an event and an arrow denotes a causal link where the event in the source establishes a condition for the event in the destination. The temporal ordering proceed from the top to the bottom. (The original story is from [4])

Figure 7.3 illustrates a story that consists of four events that fulfills the goal of $dead(Lord)$ starting from the initial state of $have(Butler, Wine) \wedge have(Butler, Poison)$ $\wedge$ $serving(Butler, Lord)$. The textual description of the plan can be read as: (1) Butler puts poison in wine. (2) Butler carries wine to Lord Higginbotham. (3) Lord Higginbotham drinks wine. (4) Lord Higginbotham falls down. (The original story is from [4])

The plan seems to be reasonable as a story. But, is it an optimal plan that has the minimum plan steps? What if the butler gives the poison to the Lord instead? Then, the plan would consist of three steps: 1) The butler carries the poison. 2) The lord drinks the poison. 3) The lord falls down.

As you may have sensed already, the new plan is logically sound but does not make a good story since the lord would not cooperate with the plan if he intends to be alive. This addresses the problem of the *author-centric story generation approach* which may ignore individual character's intention. The alternative approach, *character-centric story generation*, lets every character plan his/her own actions, expecting that some stories emerge from the character interaction. As one can easily imagine, however, a tellable situation rarely arises without the help of authorial goals. To tackle this issue, Riedl and Young have proposed an intent-driven planning algorithm to balance the author-centric approach and character-centric approach to story generation [19].

7.5 Generating Game Worlds and Stories Together

Many computer games engage players through interleaved periods of *story play* and *open-ended play*. Story play encompasses the activities of the players that promote the progression of the game world through a narrative sequence toward a desired conclusion. As laid out in this chapter, a story can be represented as a partially-ordered plan of actions that, when executed, transform the world progressively closer to a desired conclusion, represented by the goal situation. Open-ended play encompasses player activities that do not progress (nor inhibit) the story plan. Examples of open-ended include exploring the spatial environment, encountering random enemies, and finding treasure or items.

This section concerns itself with the generation of playable game experiences including both story play and open-ended play. Players expect to be immersed in a *game world*, a spatial environment encompassing all locations relevant to story play and open-ended play, and inhabited by the player character and all other non-player characters. Both story play and open-ended play are often tied to the spatial environment. Unfortunately, the generation of a story plan generator does not necessarily result in a playable experience without being tied to a spatial environment. In the case that a game world does not exist that suits the purposes of an automatically generated story plan, the game world may be automatically generated.

Table 7.1: Example plan with event locations.

1.	*Take* (paladin, water-bucket, palace)
2.	*Kill* (paladin, baba-yaga, water-bucket, graveyard1)
3.	*Drop* (baba-yaga, ruby-slippers, graveyard1)
4.	*Take* (paladin, shoes, graveyard1)
5.	*Gain-Trust* (paladin, king-alfred, shoes, palace)
6.	*Tell-About* (king-alfred, treasure, treasure-cave, paladin)
7.	*Take* (paladin, treasure, treasure-cave)
8.	*Trap-Closes* (paladin, treasure-cave)
9.	*Solve-Puzzle* (paladin, treasure-cave)
10.	*Trap-Opens* (paladin, treasure-cave)

To motivate the need for game world generation, consider the fully-ordered plan in Table 7.1. The plan involves a player character, the Paladin, performing a series of tasks to gain the King's trust, learn about a treasure cave, and escape a trap. Each action in the plan establishes a number of world conditions necessary for subsequent actions to occur. For example, the Witch will drop her shoes only once dead, and the King will trust the Paladin once he is presented with the shoes of the Witch. A story plan only provides the essential steps to progress toward a goal situation, but does not reason about player activities that do not otherwise impact the progression of the story.

A *domain model*—the library of plan operator templates—is an abstract model of a game world. The domain model abstracts away much of the moment to moment activity of the player and NPCs in order to focus on the aspects of the world that are most crucial for story progression. Game play, however, is not always a sequence of discrete operations. For example, solving a puzzle may require many levers to be triggered in the right sequence. For the purposes of this chapter, we will refer to operations in a story plan as *events* to highlight their abstract nature. Events are *temporally extended*; each event can take a continuous duration of time, and there may be large durations of time that take place between events. The plan also does not account for opportunities for open-ended play between events. For example, where is the graveyard relative to the castle, how long does it take to travel that distance, and what might the player see or experience along the way that is not directly relevant to the story plan?

If the game world is a given—i.e., there is a fixed world with a number of locations and NPCs—then there is a mapping of story events in the plan to virtual locations in the game world. For example, the game world for Table 7.1 requires a graveyard, a castle, and a treasure cave. However, due to the nature of automatically generated story plans, it is not always feasible to have a single fixed game world that meets the requirements of a story plan: Locations may be missing, there may be too many irrelevant locations, or locations may need to be reordered to make a more coherent and sensible flow. In the next section, we describe a technique to automatically generate a playable game world based on a story plan.

7.5.1 From Story to Space: Game World Generation

Recalling that games often interleave plot points and open-ended game play, the game world to be generated must ensure a coherent sequence of events are encountered in the world. The problem can be specified as follows: given a list of events that reference locations of known types, generate a game world that allows a linear progression through the events. To map from story to space, we will utilize a metaphor of *islands* and *bridges*. Islands are areas in the spatial environment where events occur. Bridges are areas of the world between islands where open-ended game play occurs. Bridges can branch, meaning there can be areas that the player does not necessarily need to visit in the course of the story. The length of bridges and the branching factor of bridges are parameters that can be set by the designer or dictated by a player model. A game world is generated in a 3-stage pipeline in which (1) a story plan is parsed for location information referenced by events, (2) an intermediate, abstract representation of the navigable space is generated, and (3) the graphical visualization of the navigable space is realized.

First, the generated story plan is parsed to extract a sequence of locations, each of which becomes an island. The story plan must be fully ordered to generate such a sequence (any partially ordered plan can be converted into a full-ordered plan). Each event in the story plan must be associated with a location. For example, in the story plan in Table 7.1, events occur at places referenced by the symbols *palace*, *graveyard1*, and *treasure-cave*. Each referenced location must have a type. This

Table 7.2: A portion of the initial state declaration for a planning domain.

Hero (paladin)	Thing (water-bucket)	Type (palace, castle)
NPC (baba-yaga)	Thing (treasure)	Type (graveyard1, graveyard)
NPC (king-alfred)	Thing (ruby-slippers)	Type (treasure-cave, cave)
Place (palace)	Evil (baba-yaga)	Type (water-bucket, bucket)
Place (graveyard1)	Type (baba-yaga, witch)	Type (ruby-slippers, shoes)
Place (treasure-cave)	Type (king-alfred, king)	Type (treasure, gold)

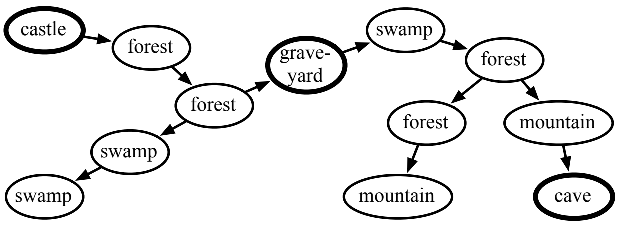

Fig. 7.4: An example space tree. Islands are marked with bold lines.

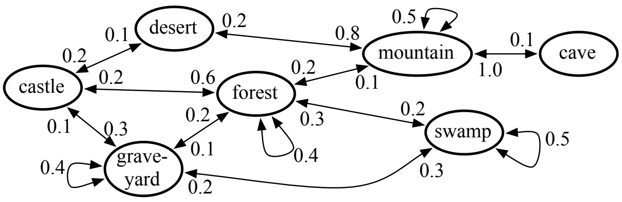

Fig. 7.5: An environment transition graph.

information is often found in the initial state declaration of the planning domain. Table 7.2 shows a portion of the initial state for the domain used to generate the example story plan. Thus the example story plan plays out in three locations: a castle (events 1, 5 and 6), a graveyard (events 2 through 4), and a cave (events 7 through 10).

The next stage is to generate an intermediate representation of the game world as a graph of location types called a *space tree*. A space tree is a discrete data structure that indicates how big the game world will be, how many unique locations, and which locations are adjacent to each other. Figure 7.4 shows an example of a space tree in which the nodes corresponding to island locations—where story plan events are to occur—are highlighted in bold and the rest of the nodes comprise the bridges.

The planning domain does not provide enough information to tell us what types of locations should be used for the bridges. We require an addition knowledge structure, called an *environment transition graph*. An environment transition graph is a data structure that captures the game designer's beliefs about good environment type transitions. Each node in an environment transition graph is a possible location type and edges indicate non-zero probability of transitioning from one location type to another. Figure 7.5 shows an example of an environment transition graph.

Space Tree generation can utilize any optimization search algorithm to find a space tree that meets the evaluation criteria. See Chapter 2 for the general search-based approach to procedural content generation, and see [8] for specific implementation details. The evaluation criteria are:

- Degree to which the number of bridges nodes in the space tree between islands have the preferred length.
- Whether the bridges have the preferred branching factor.
- Degree to which the length of side paths—branch nodes that are not directly between two islands—matches the preferred side path length.
- How closely environment type transitions between adjacent nodes match the environment transition graph probabilities.

These evaluation criteria make use of parameters set by the designer. Other evaluation criteria may be used as well.

Once the space tree has been generated via a search-based optimization process, the third stage is to *realize* the game world graphically. The space tree gives us an abstract representation of this game world but doesn't tell us what each locations should look like. Where should art assets be placed spatially to create the appearance of a forest, town, or graveyard, etc.?

We describe a graphical realization process that creates a 2-D, top-down, tile-based, graphical visualization of a game world described by a space tree. Starting with a grid of empty tiles, we will first map the space tree to the 2D grid and the choose tiles for each cell in the grid. If the grid is $m_{\text{world}} \times n_{\text{world}}$ tiles, then each $m_{\text{screen}} \times n_{\text{screen}}$ tiles is the number of tiles that can be displayed on the screen at any one time. Each node in the space tree will be mapped to a $m_{\text{location}} \times n_{\text{location}}$ grid of screens. In Figure 7.6, the world is 340×160 tiles, each screen is 34×16 tiles, and each location encompasses a 3×3 grid of screens (only a portion is shown). The mapping of space tree to grid is as follows. Use a depth-first traversal of the space tree, placing each child adjacent to its parent on a grid. In order to prevent an algorithmic bias toward growing the world in a certain direction (e.g. from left to right), one can randomize the order of cardinal directions it attempts to place each child. To minimize the likelihood that nodes will be mapped to the same portion of the grid, one can constrain the space tree such that nodes have no more than two children, for a total of three adjacent nodes. Backtrack if necessary. If there is no mapping solution, discard the space tree and return to the optimization search to generate the next best space tree.

Once each node in the space tree has been assigned a region on the grid, the module begins graphical instantiation of the world. Each node from the space tree has an environment type, which determines what *decorations* will be placed. Decorations are graphical assets that overlay tiles and visually depict the environment type. For a 2D tile-based realization of a game world, decorations are sprites that depict scenery found in different environment types. A forest environment has decorations consisting of grass, trees, and bushes, while a town has decorations that

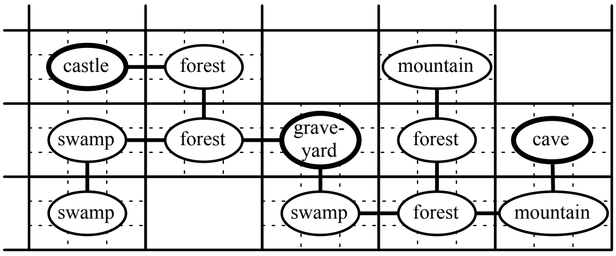

Fig. 7.6: A space tree mapped to a grid.

Fig. 7.7: A forest adjacent to a swamp, both with Gaussian distributions resulting in a blended transition.

look like buildings, castle walls, and street paving stones. But how does the system know where to place each decoration? This knowledge is also not present in the domain model, and a third type of external knowledge is necessary. Each environment type is associated with a function that maps decorations to a probability distribution over XY tile coordinates. We have identified two types of mapping functions. A *Gaussian distribution* defines the dispersement of decorations around the center point of a location such that decorations are placed more densely around the center point of each location. The advantage of a Gaussian distribution is that decorations can be placed in adjacent locations, creating the appearance that one location blends into the next, as in Fig 7.7. A *custom distribution* is an arbitrary, designer-specified function that returns the probability of placing a decoration at any XY coordinate. Fig 7.8 shows the custom distribution for a town location type such that buildings are likely arranged in a grid-like city blocks, paving stones make up streets between city blocks, and guard towers are arranged in a ring around the town perimeter.

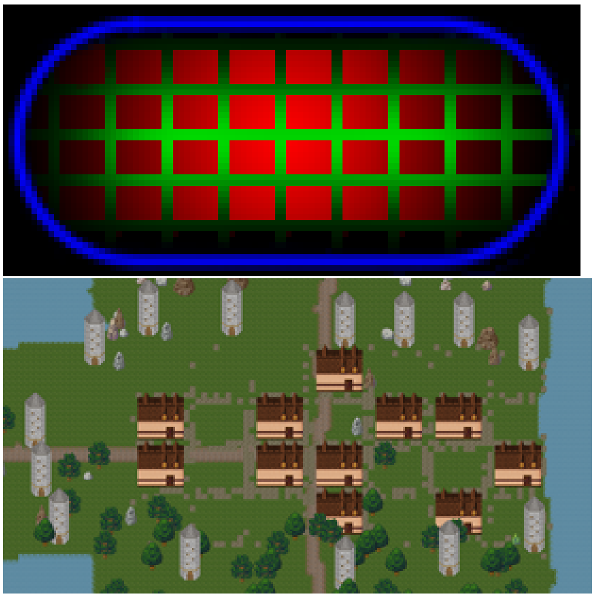

Fig. 7.8: A custom distribution for a town (left) and an example of the result (right). Brighter color indicates greater probability of a decoration, where red indicates buildings, green indicates paving stones, and blue indicates towers.

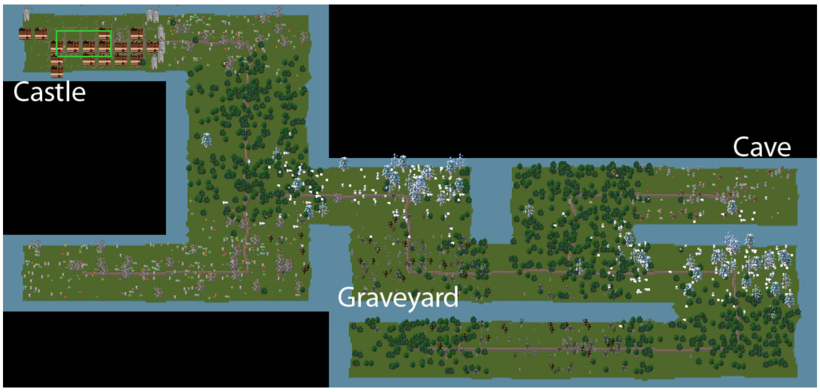

Fig. 7.9: Example game world generated from the islands in the plan in Table 7.1.

Fig 7.9 shows an example of a complete game world with three islands extracted from Table 7.1.

7.5.2 From Story to Time: Story Plan Execution

Once the space in which the story will unfold has been generated, there are two additional issues that must be addressed: (a) the world must be populated with NPCs, and (b) the NPCs must act out the story, which was not known prior to execution. Population of the world by NPCs is a simple process of parsing the story plan for references to NPCs and instantiating sprites (based on NPC type) in the location in which they are first required to participate in an event. Because of the temporal extension of events, NPCs must elaborate on events, including engaging in combat, engaging in dialogue, setting up and triggering traps (the world itself can be an NPC), etc. Because the story and world geometry are *a priori* unknown, the NPCs must be flexible enough to elaborate on an event under a wide range of conditions based on what events preceded the current time point and how the world is laid out.

One solution is to pair each event with a *reactive script* that decomposes the event into a number of primitive NPC behaviors. Roughly, a reactive script is an AND-OR tree structure in which internal nodes represent abstract behaviors—possibly joint between a number of characters—and leaf nodes represent primitive, executable behaviors such as animations. Reactive script execution is a walk of the tree implementing an event such that AND-nodes create sequences of sub-behaviors and OR-nodes express alternative means of decomposing achieving a behavior, implementing *if-then-else* decision-making logic. Internal nodes may implement applicability criteria (similar to preconditions) that are used to prune sub-trees that are not supported by the state of the virtual world at execution time. Examples of reactive script technologies include behavior trees [9], hierarchical finite state machines, hierarchical task networks [21] such as *SHOP 2* [15], and the ABL reactive behavior planner [13].

There are two types of reactive scripts necessary to execute an automatically generated story in an open-ended game world [18]: narrative directive behaviors and local autonomous behaviors. *Narrative directive behaviors* are reactive scripts associated with event templates in the domain model. They operate as above, decomposing events into primitive behaviors. Narrative directive behaviors enact an event as if it were a stage manager in a play; they are not associated directly with any one character, but may control many characters at once. *Local autonomous behaviors* are associated with NPC types and execute whenever an NPC is instantiated in the world but not otherwise playing a role in an event. Local autonomous behaviors create the appearance that NPCs have rich internal lives if they are encountered by the player during open-ended play.

7.6 Lab exercise: Write a story domain model

The purpose of this exercise is to let you be able to write a story domain model and characterize different planning algorithms.

1. Get familiarized with JSHOP2 - an off-shelf JAVA implementation of SHOP2 HTN planner (originally written in LISP).
   - Download and install JSHOP 2.0 (http://www.cs.umd.edu/projects/shop/)
   - Check out and test the sample examples included in the package.
2. Write a planning problem in terms of initial state, goal state, and actions by defining two story domains (Little Red Riding Hood and The Gift of the Magi) using either STRIPS-style or ADL-style representation. Discuss which representation is more suitable to describe the two storyworld domains and explain why.
3. Convert the above planning problems into HTN representation that is applicable in JSHOP2 planner, and execute them. Discuss the strength and weakness of HTN planning (or SHOP2 planner) as a story generation method/tool.
4. In the Butler story described in Section 1.4, suppose that the lord knows that the wine is poisoned and he just pretends to be dead, but the butler does not know that the lord knows. The new authorial goal is now reprensented as $\neg dead(Lord) \wedge Arrested(Butler)$. Make a complete story plan by adding additional actions (e.g., $Call-911(Lord)$, $Arrest(Police, Butler)$), states, and causal links. Do you think that it will make the story more interesting? Why or why not?
5. Discuss the overall advantages and limitations of planning-based story generation.
6. Discuss how planning-based story generation techniques can be effectively used in interactive storytelling systems and games.

References

1. Espen Aarseth. From Hunt the Wumpus to EverQuest: Introduction to quest theory. In *Proceedings of the 4th International Conference on Entertainment Computing*, pages 496–506, 2005.
2. Calvin Ashmore and Michael Nitsche. The quest in a generated world. In *Proceedings of the 2007 Digital Games Research Association Conference*, pages 503–509, 2007.
3. Byung-Chull Bae and R. Michael Young. A use of flashback and foreshadowing for surprise arousal in narrative using a plan-based approach. In *Proceedings of the 1st Joint International Conference on Interactive Digital Storytelling: Interactive Storytelling*, ICIDS '08, pages 156–167, Berlin, Heidelberg, 2008. Springer-Verlag.
4. W.F. Brewer and E.H. Lichtenstein. Event schemas, story schemas, and story grammars. In J.B. Long and A.D. Baddeley, editors, *Attention and Performance*, volume 9, pages 363–379. Lawrence Erlbaum Associates, Inc., 1981.
5. Yun-Gyung Cheong and R. Michael Young. Narrative generation for suspense: Modeling and evaluation. In Ulrike Spierling and Nicolas Szilas, editors, *Interactive Storytelling, First Joint International Conference on Interactive Digital Storytelling, ICIDS 2008, Erfurt, Germany, November 26-29, 2008, Proceedings*, volume 5334 of *Lecture Notes in Computer Science*. Springer, 2008.
6. Natalie Dehn. Story generation after TALE-SPIN. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 16–18, 1981.
7. Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. Technical Report 43R, AI Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, May 1971. SRI Project 8259.

8. Ken Hartsook, Alexander Zook, Sauvik Das, and Mark Riedl. Toward supporting storytellers with procedurally generated game worlds. In *Proceedings of the 2011 IEEE Conference on Computational Intelligence in Games*, pages 297–304, Seoul, South Korea, August 2011.
9. Damian Isla. Handling complexity in the Halo 2 AI. Presentation at the 2005 Game Developers Conference. Available at: http://www.naimadgames.com/publications/gdc05/gdc05.doc (retrieved September 9, 2013).
10. Subbarao Kambhampati, Craig A. Knoblock, and Qiang Yang. Planning as refinement search: A unified framework for evaluating the design tradeoffs in partial order planning. *Artificial Intelligence*, 76(1-2), 1995.
11. John Paul Kelly, Adi Botea, and Sven Koenig. Offline planning with hierarchical task networks in video games. In *AIIDE*, 2008.
12. Michael Lebowitz. Story-telling as planning and learning. *Poetics*, 14:483–502, 1985.
13. Michael Mateas and Andrew Stern. A Behavior Language: Joint action and behavior idioms. In Helmut Prendinger and Mitsuru Ishizuka, editors, *Life-like Characters: Tools, Affective Functions and Applications*. Springer, 2004.
14. James R. Meehan. *The Metanovel: Writing Stories by Computer*. PhD thesis, Department of Computer Science, Yale University, 1976.
15. Dana Nau, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.
16. Edwin P. D. Pednault. Formulating Multi-Agent Dynamic-World Problems in the Classical Planning Framework. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning About Actions and Plans: Proceedings of the 1986 Workshop*, pages 47–82, San Mateo, CA, 1987. Morgan Kaufmann Publishers.
17. Mary Jo Rattermann, Lee Spector, Jordan Grafman, Harvey Levin, and Harriet Harward. Partial and total-order planning: evidence from normal and prefrontally damaged populations. *Cognitive Science*, 25(6):941–975, 2001.
18. Mark O. Riedl, Andrew Stern, Don M. Dini, and Jason M. Alderman. Dynamic experience management in virtual worlds for entertainment, education, and training. *International Transactions on System Science and Applications*, 3(1):23–42, 2008.
19. Mark O. Riedl and R. Michael Young. Narrative planning: balancing plot and character. *Journal of Artificial Intelligence Research*, 39(1):217–268, 2010.
20. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.
21. Earl D. Sacerdoti. *A Structure for Plans and Behavior*. Elsevier, New York, 1977.
22. Anne Sullivan, Michael Mateas, and Noah Wardrip-Fruin. Making quests playable: Choices, CRPGs, and the Grail framework. *Leonardo Electronic Almanac*, 17(2):146–159, 2012.
23. Susana Tosca. The quest problem in computer games. In *Proceedings of the 1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, pages 69–81, 2003.
24. Tom Trabasso and Linda L. Sperry. Causal relatedness and importance of story events. *Journal of Memory and Language*, 24(5):595 – 611, 1985.
25. Scott R. Turner. *The Creative Process: A Computer Model of Storytelling and Creativity*. Psychology Press, 1994.