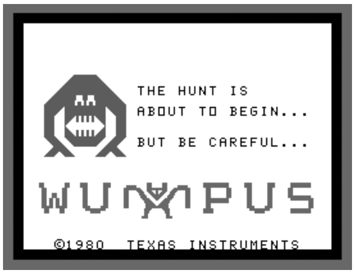
8

THIS AIN'T YOUR DADDY'S WUMPUS

In the previous chapter, we worked with mathematical graphs in a simple game. However, as an old-school geek, the first thing I think of when I see these graphs is the old game Hunt the Wumpus. When I was nine, I could think of nothing more fun than sitting in front of my TI-99/4A and playing this excellent game.

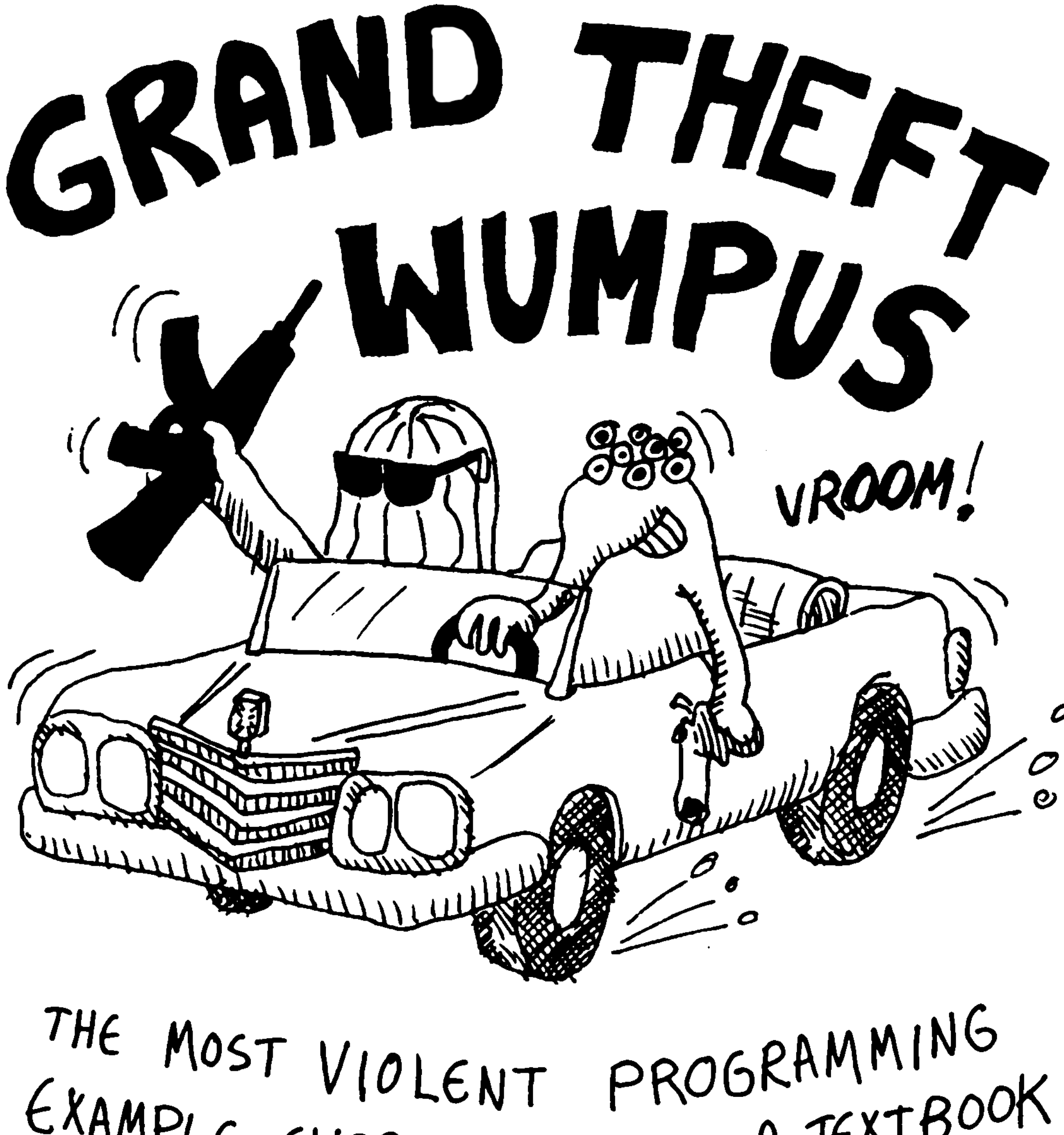
Here is the original title screen:



In Hunt the Wumpus, you are a hunter searching through a network of caves to find a mysterious monster—the fabled Wumpus. Along the way, you also deal with bats and tar pits. Ah, those were the days!

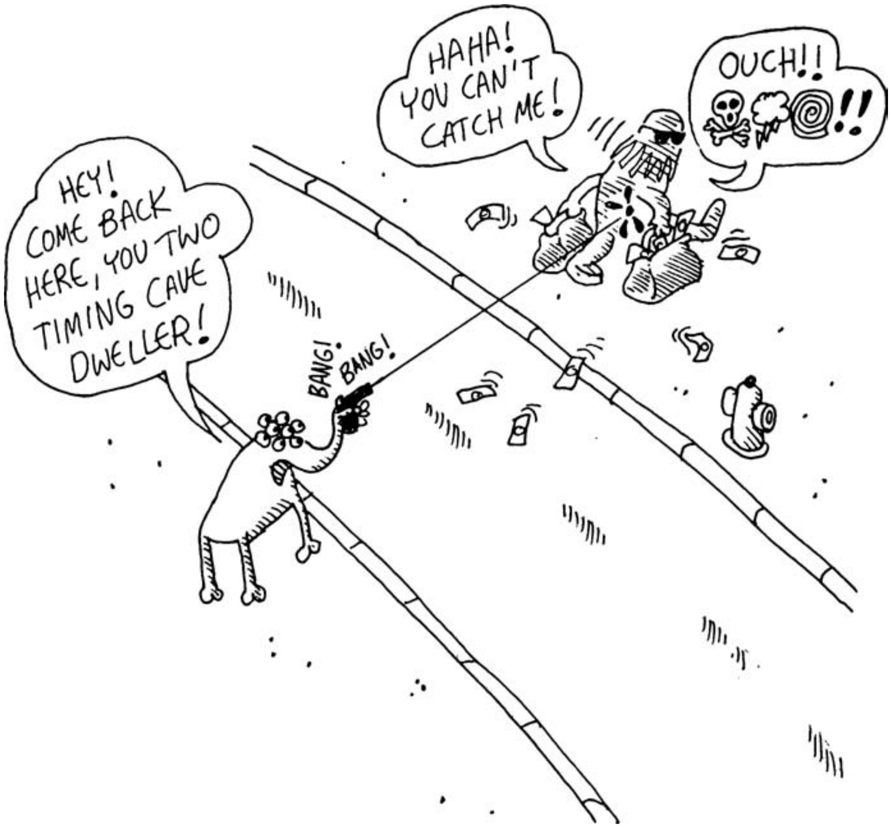
But, unfortunately, those days are long gone. We're in a new millennium now, and no one would be impressed by these crude graphics anymore. And the story line, well, let's just say it sounds a bit corny by modern standards. I think we can all agree that Hunt the Wumpus is in serious need of a makeover. That's quite a challenge, but one I think we can handle.

Therefore, I present to you . . .

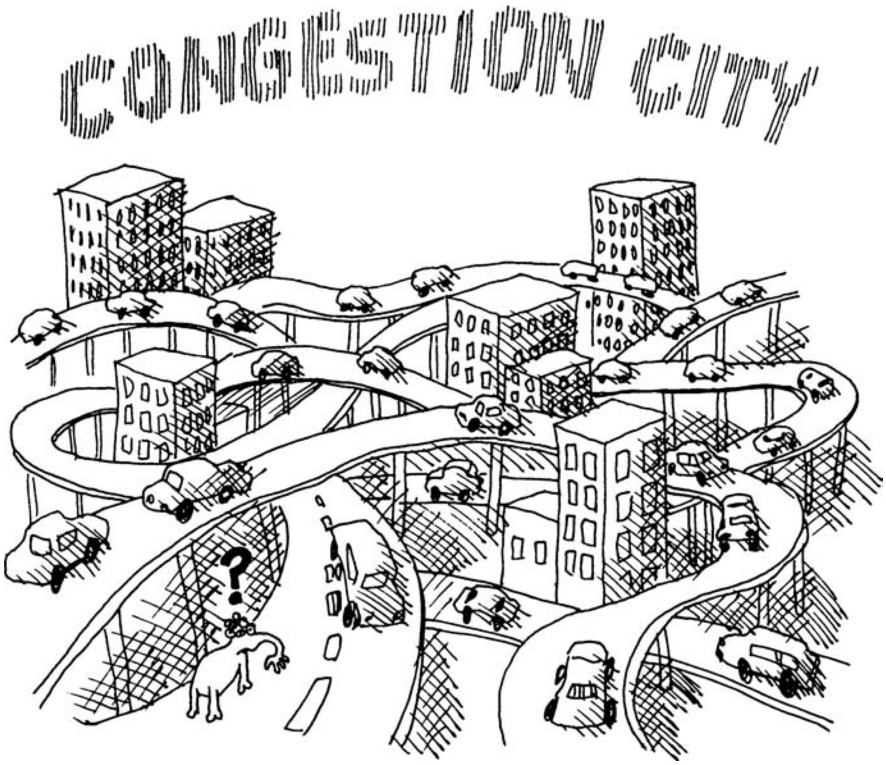

GRAND THEFT WUMPUS
VROOM!
THE MOST VIOLENT PROGRAMMING EXAMPLE EVER PUT INTO A TEXTBOOK

The Grand Theft Wumpus Game

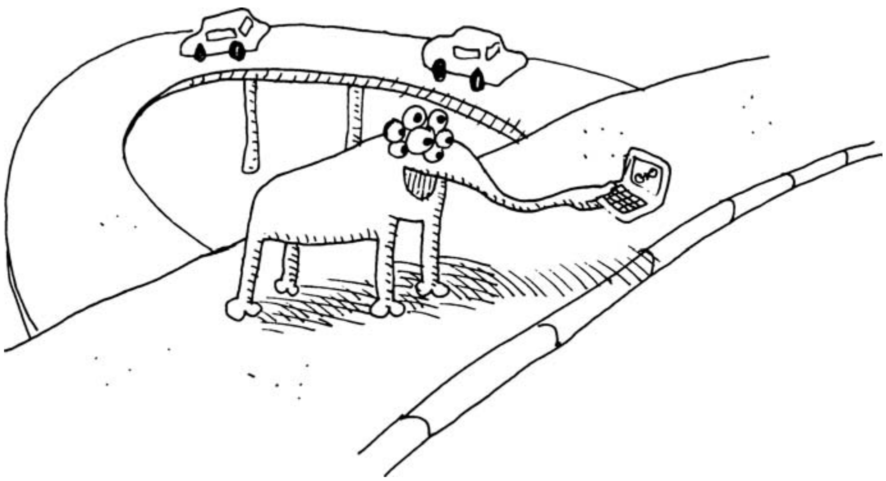
In this new version of Hunt the Wumpus, you are the Lisp alien. You and the Wumpus have just robbed a liquor store and made off with the loot. However, during the escape, the Wumpus decides to double-cross you and run off with the money and your car. But before he drives off, you manage to cap him a couple of times in the kidney.


HAHA! YOU CAN'T CATCH ME!
OUCH!! ☠ !!
HEY! COME BACK HERE, YOU TWO TIMING CAVE DWELLER!
BANG!
BANG!

Now you're in a pretty tough situation. You don't have a car or any money, and no way to track down your former partner in crime. But you also have no choice. You have your principles, so you're going to *hunt the Wumpus.* You know he won't be able to get very far with his injuries. He will most likely need to lie low for a few days to recover, which means he will still be somewhere in Congestion City. The problem is that the roads in this town are impossibly convoluted, and no one can find their way around, especially an out-of-towner like yourself. How are you ever going to find the Wumpus in this impossible maze?

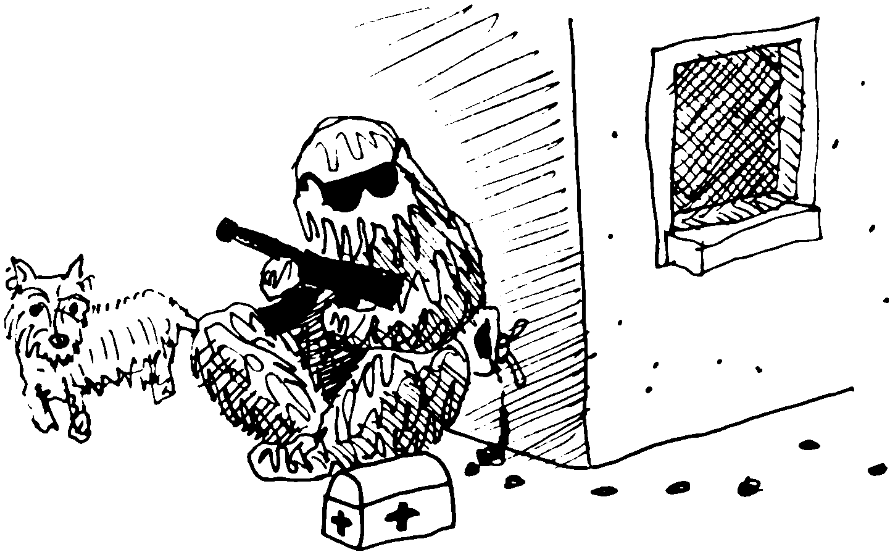


Luckily, being the Lisp alien, you always carry your trusty pocket computer. Using Lisp and your graph utilities, you're fully equipped to analyze complicated data such as the Congestion City roadways and intersections. Surely, you have the tools to conquer this impenetrable road system.



The Wumpus has been your partner in crime for a while now, so you know his MO quite well. He will always carefully scout out any new hiding place before he uses it. And since he is injured, any location one or two blocks away (that is, one or two graph edges away) from his hiding place should be marked with some telltale signs: his blood stains.

A problem is that he still has his trusty AK-47, while you have only a handgun with a single bullet. If you're going to take him out, you'll need to be absolutely sure you've tracked him down. You'll need to charge into his hideout and shoot him down immediately, and you'll have only one chance to pull this off.



Unfortunately, you and the Wumpus aren't the only criminals in this town. The most feared outlaw group in Congestion City is the Gruesome Glowworm Gang. These guys are a band of ruthless kidnappers. If you run into them, they will kidnap you, beat you up, rob you, blindfold you, and then kick you out of their car and leave you in some random part of town.

Luckily, they can be avoided if you know to keep an eye out for their glowing thoraxes (hence their name). If you see some blinking lights, you know that these guys are one street away from your current location. Also, you know the gang has exactly three separate teams that work the city from three separate locations.

Finally, you still need to contend with the cops. You know they've probably set up some roadblocks in town to try to catch you and the Wumpus. You should still be able to visit any location in Congestion City, but you need to be careful which streets you travel. (In other words, the cops will catch you if you travel along the wrong edge.) Unfortunately, you don't know how many of these roadblocks there may be.

As you can see, finding the Wumpus and getting back your money and car will be tough. If you think you're Lisp alien enough to take on the Wumpus, then let's write this game and hunt him down!

Defining the Edges of Congestion City

The map of Congestion City will be an undirected graph with data associated with each node stored in the variable `*congestion-city-nodes*`. The possible data at each node will include the presence of the Wumpus, a Glowworm team, and various danger signs.

A set of edges stored in `*congestion-city-edges*` will connect the nodes, and data linked to these edges will alert us to the presence of any police roadblocks. We declare these and other global variables at the top of our program using `defparameter`:

```
❶ (load "graph-util")

  (defparameter *congestion-city-nodes* nil)
  (defparameter *congestion-city-edges* nil)
  (defparameter *visited-nodes* nil)
❷ (defparameter *node-num* 30)
❸ (defparameter *edge-num* 45)
❹ (defparameter *worm-num* 3)
❺ (defparameter *cop-odds* 15)
```

We first load our graph utilities with the `load` command ❶, which evaluates all the code in *graph-util.lisp* (which we created in the previous chapter) so the graph utility functions will be available. Notice that Congestion City will have 30 locations ❷ (nodes, defined with `*node-num*`), 45 edges ❸ (roads, defined with `*edge-num*`), and 3 worm teams ❹ (defined with `*worm-num*`). Each street will have a 1-in-15 chance ❺ of containing a roadblock (defined with `*cop-odds*`).

Generating Random Edges

Next, we create a random list of edges to connect all the nodes:

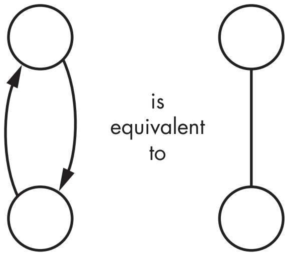
```
❶ (defun random-node ()
    (1+ (random *node-num*)))

❷ (defun edge-pair (a b)
    (unless (eql a b)
      (list (cons a b) (cons b a))))

❸ (defun make-edge-list ()
❹   (apply #'append (loop repeat *edge-num*
❺                         collect (edge-pair (random-node) (random-node)))))
```

First, we declare the `random-node` function ❶, which returns a random node identifier. It uses the `random` function, which returns a random natural number less than the integer you pass to it. Since we're going to be showing the node identifiers in our user interface, we use the `1+` function to number our nodes 1 through 30 (the upper limit because the `*node-num*` variable is set to `30`), instead of 0 through 29.

The `make-edge-list` function ❸ generates the actual list of random edges. It uses the `loop` command to loop `*edge-num*` times ❹, and then collects the requisite number of edges ❺. We'll take a closer look at the `loop` command in the next section. The graph of the city is undirected, so this function uses a helper function, `edge-pair` ❷, to create *two* directed edges between the randomly selected nodes. This extra step makes sense once you remember that an undirected graph is the same as a directed graph, with two opposing directed edges mirroring each undirected edge. (When we build our edges into an alist later in this chapter, this step will ensure that the list is properly formed.)



Let's try the `make-edge-list` function in the CLISP REPL:

```
> (make-edge-list)
((16 . 20) (20 . 16) (9 . 3) (3 . 9) (25 . 18) (18 . 25) (30 . 29) (29 . 30)
(26 . 13) (13 . 26) (12 . 25) (25 . 12) (26 . 22) (22 . 26) (30 . 29) (29 .
30) (3 . 14) (14 . 3) (28 . 6) (6 . 28) (4 . 8) (8 . 4) (27 . 8) (8 . 27) (3 .
30) (30 . 3) (25 . 16) (16 . 25) (5 . 21) (21 . 5) (11 . 24) (24 . 11) (14 .
1) (1 . 14) (25 . 11) (11 . 25) (21 . 9) (9 . 21) (12 . 22) (22 . 12) (21 .
11) (11 . 21) (11 . 17) (17 . 11) (30 . 21) (21 . 30) (3 . 11) (11 . 3) (24 .
23) (23 . 24) (1 . 24) (24 . 1) (21 . 19) (19 . 21) (25 . 29) (29 . 25) (1 .
26) (26 . 1) (28 . 24) (24 . 28) (20 . 15) (15 . 20) (28 . 25) (25 . 28) (2 .
11) (11 . 2) (11 . 24) (24 . 11) (29 . 24) (24 . 29) (18 . 28) (28 . 18) (14 .
15) (15 . 14) (16 . 10) (10 . 16) (3 . 26) (26 . 3) (18 . 9) (9 . 18) (5 . 12)
(12 . 5) (11 . 18) (18 . 11) (20 . 17) (17 . 20) (25 . 3) (3 . 25))
```

You see the pairs of node numbers that make up the edges. This list of edge pairs will form the skeleton of the Congestion City road system.

Looping with the loop Command

Our `make-edge-list` function employs the powerful `loop` command, which can be used to loop over various types of data. We'll be looking at `loop` in detail in Chapter 10. However, our game uses `loop` a few times, so let's consider some simple examples to clarify how it works.

One handy thing you can do with `loop` is create a list of numbers. For instance, the following command will create a list of 10 ones:

```
> (loop repeat 10
        collect 1)
(1 1 1 1 1 1 1 1 1 1)
```

Within the `loop` command, we specify how many times to `repeat`, and then specify an object to `collect` with every loop (in this case, the number 1).

Sometimes, we want to keep a running count as we're looping. We can do this with the following syntax:

```
> (loop for n from 1 to 10
        collect n)
(1 2 3 4 5 6 7 8 9 10)
```

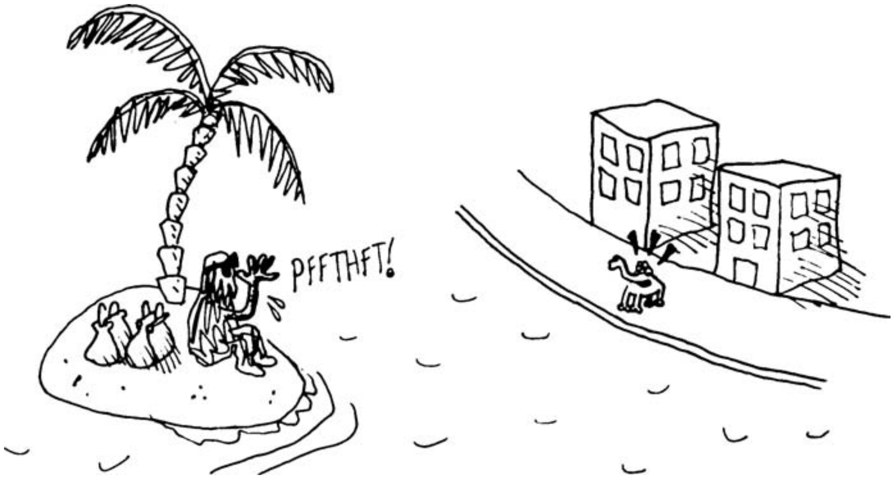
In this example, we are saying that `n` should loop from 1 to 10. Then we `collect` each `n` and return it as a list.

Actually, we can put any Lisp code in the `collect` part of the loop. In the following example, we add 100 as we do our collecting:

```
> (loop for n from 1 to 10
        collect (+ 100 n))
(101 102 103 104 105 106 107 108 109 110)
```

Preventing Islands

We now can generate random edges. Of course, if we just connect random nodes with random edges, there's no guarantee that all of Congestion City will be connected because of all that randomness. For example, some parts of the city might form an island, with no connections to the main road system.



To prevent this, we'll take our list of edges, find unconnected nodes, and connect these islands to the rest of the city network using this code:

```
❶ (defun direct-edges (node edge-list)
❷   (remove-if-not (lambda (x)
                     (eql (car x) node))
                   edge-list))

❸ (defun get-connected (node edge-list)
❹   (let ((visited nil))
      (labels ((traverse (node)
                 (unless (member node visited)
❺                  (push node visited)
❻                  (mapc (lambda (edge)
                           (traverse (cdr edge)))
                         (direct-edges node edge-list)))))
        (traverse node))
      visited))

  (defun find-islands (nodes edge-list)
    (let ((islands nil))
❼     (labels ((find-island (nodes)
                 (let* ((connected (get-connected (car nodes) edge-list))
                        (unconnected (set-difference nodes connected)))
                   (push connected islands)
❽                  (when unconnected
                     (find-island unconnected)))))
        (find-island nodes))
      islands))

  (defun connect-with-bridges (islands)
❾   (when (cdr islands)
      (append (edge-pair (caar islands) (caadr islands))
              (connect-with-bridges (cdr islands)))))

❿ (defun connect-all-islands (nodes edge-list)
    (append (connect-with-bridges (find-islands nodes edge-list)) edge-list))
```

First, we declare a utility function called `direct-edges` ❶, which finds all the edges in an edge list that start from a given node. It does this by creating a new list with all edges removed (using `remove-if-not` ❷) that don't have the current node in the `car` position.

To find islands, we write the `get-connected` function ❸. This function takes an edge list and a source node and builds a list of all nodes connected to that node, even if it requires walking across multiple edges.

The usual way to find connected nodes is to start a `visited` list ❹, and then perform a search along connected nodes, starting with the source node. Newly found nodes are added to the visited list with the `push` command ❺. We also traverse all the children of this found node, using `mapc` ❻.

If, on the other hand, we encounter a node that has already been visited, we know we can ignore it. Once the search is complete, the `visited` list will consist of all connected nodes.

Now that we have a function for finding nodes that are connected, we can use it to create a function that will find all the islands in our graph. The `find-islands` function first defines a local function, called `find-island` ❼. This function checks which nodes are connected to the first node in our list of nodes using the `connected` function. It then subtracts these nodes from the full list of nodes using the `set-difference` function. (`set-difference` takes two lists, and returns all items that are in the first list but not the second.)

Any remaining nodes are deemed unconnected. If any unconnected node exists ❽, we call the `find-islands` function again recursively to find additional islands.

Once we've found all the islands, we need a way of bridging them together. This is the job of the `connect-with-bridges` function. It returns a list of additional edges that join all the islands together. To do this, it takes the list of islands and checks if there is a `cdr` in this list ❾. If there is, it means there are at least two land masses, which can be connected with a bridge. It uses the `edge-pair` function to create this bridge, and then calls itself recursively on the tail of the island list, in case additional bridges are needed.

Finally, we tie all of our island prevention functions together using the function `connect-all-islands` ❿. It uses `find-islands` to find all the land masses, and then calls `connect-with-bridges` to build appropriate bridges. It then appends these bridges to the initial list of edges to produce a final, fully connected land mass.

Building the Final Edges for Congestion City

To complete our edges for Congestion City, we need to convert the edges from an edge list into an alist. We also will add the police roadblocks, which will appear randomly on some of the edges. For these tasks, we will create the `make-city-edges`, `edges-to-alist`, and `add-cops` functions:

```
  (defun make-city-edges ()
❶   (let* ((nodes (loop for i from 1 to *node-num*
                        collect i))
❷          (edge-list (connect-all-islands nodes (make-edge-list)))
❸          (cops (remove-if-not (lambda (x)
                                  (zerop (random *cop-odds*)))
                                edge-list)))
      (add-cops (edges-to-alist edge-list) cops)))

  (defun edges-to-alist (edge-list)
❺   (mapcar (lambda (node1)
              (cons node1
❻                   (mapcar (lambda (edge)
                              (list (cdr edge)))
                            (remove-duplicates (direct-edges node1 edge-list)
❼                                              :test #'equal))))
            (remove-duplicates (mapcar #'car edge-list))))
```

```
(defun add-cops (edge-alist edges-with-cops)
❽  (mapcar (lambda (x)
            (let ((node1 (car x))
                  (node1-edges (cdr x)))
              (cons node1
❾                   (mapcar (lambda (edge)
                              (let ((node2 (car edge)))
❿                               (if (intersection (edge-pair node1 node2)
                                                  edges-with-cops
                                                  :test #'equal)
                                    (list node2 'cops)
                                  edge)))
                            node1-edges))))
          edge-alist))
```

These are the most cumbersome functions in Grand Theft Wumpus. Let's take a closer look at them.

The make-city-edges Function

First, the `make-city-edges` function creates a list of nodes, using a `loop` ❶. (This is simply a list of numbers from 1 to `*node-num*`.) Next, it creates a random (but fully connected) edge list by calling the `make-edge-list` and `connect-edge-list` functions ❷. This result is stored in the `edge-list` variable. It then creates a random list of edges that contains `cops` ❸. We define these variables with the `let*` command, which allows us to reference previously defined variables.

The following example shows the difference between defining variables with `let` and `let*`:

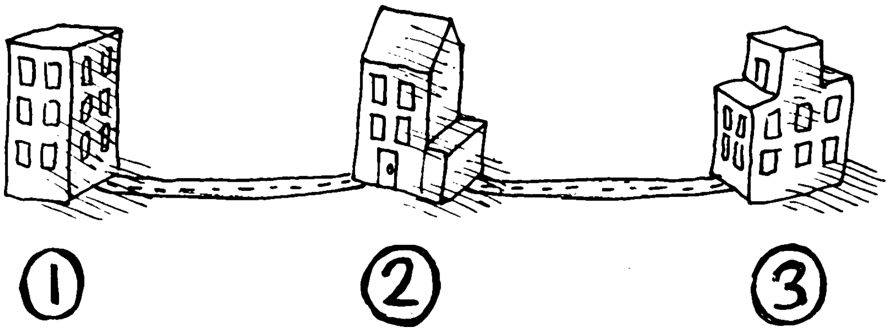
```
> (let ((a 5)
        (b (+ a 2)))
    b)
*** - EVAL: variable A has no value
> (let* ((a 5)
         (b (+ a 2)))
    b)
7
```

As you can see, `let` won't allow you to refer to other defined variables (the variable `b` can't reference the value of `a`). When defining variables with `let*`, on the other hand, this kind of reference is allowed. For our purposes, using `let*` allows our definition of `cops` ❸ to contain a reference to `edge-list`.

Once we've created the edge list and determined where the cops are, we need to convert our edge list into an alist and add the cops to it ❹. The edges are converted to an alist with the `edges-to-alist` function, and the cops are added with the `add-cops` function.

The edges-to-alist Function

The `edges-to-alist` function converts a list of edges into an alist of edges. For example, assume we have the following city, with only three locations and two edges connecting those locations:



We would describe this using an edge list as `'((1 . 2) (2 . 1) (2 . 3) (3 . 2))`. Remember that each of the edges is repeated, since the edges are undirected and can be used in both directions. If we described this same city as an alist, what would that look like?

Remember that an alist is a list that lets us look up a key (in this example, one of the three nodes in our city) and find the information associated with that key (in this case, a list of the roads connected to it). For this small city, the alist would be `'((1 (2)) (2 (1) (3)) (3 (2)))`.

To build this alist, the `edges-to-list` function first `mapcar`s ❺ over the nodes found in the edge list. To build the list of nodes, we use the `remove-duplicates` function, which removes duplicate items from a list. By default, `remove-duplicates` uses the `eql` function to check for equality, though it also allows you to choose a different test function using the `:test` keyword parameter. Since we're checking for equality of cons pairs in our `make-city-edges` function, we set `:test` to `#'equal` ❼.

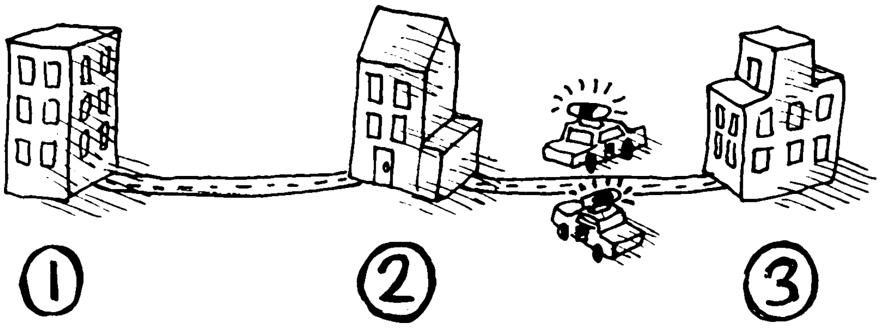
Within this outer `mapcar` ❺, we use another `mapcar` ❻ to map across all the `direct-edges` to this node. Together, these nested `mapcar` functions allow `edges-to-alist` to convert the edges of a city into an alist.

The add-cops Function

When we wrote the `make-city-edges` function, we randomly marked some of the edges to show that they have cops on them ❹. We are now going to use this list of cop edges to mark the edges in our alist that contain cops. This is the job of the `add-cops` function.

To do this, we use nested `mapcar` commands to map across the edges within each node ❽❾. We then check whether there are any cops on a given edge, using the `intersection` function ❿. (The `intersection` function tells us which items are shared between two lists.)

To understand exactly what the add-cops function is doing, it will help to once again imagine our city with only three locations and two streets. In this example, one of the streets has cops on it:



The generated alist for this city, created by add-cops, would look like this:

```
((1 (2)) (2 (1) (3 COPS)) (3 (2 COPS)))
```

This is actually a *nested* alist. The outer alist is organized based on the first node, and the inner alists are organized based on the second node.

With the edges in this format, we can easily find all edges connected to a given node by calling (cdr (assoc node1 edges)). To see if a given edge contains cops, we can call (cdr (assoc node2 (cdr (assoc node1 edges)))), which goes down two levels to grab the actual data linked to a specific edge between two nodes. (One additional benefit of using this nested alist format is that it is fully compatible with our graph libraries—a feature that we'll take advantage of shortly.)

Building the Nodes for Congestion City

Now we'll build an alist for the nodes in our city. These nodes may contain the Wumpus or the Glowworms, or they might contain various clues, such as blood, lights, or sirens.

Most of the clues in our game are based on proximity to another node, so we need to write some functions that tell us if two nodes are one node apart in the city graph. The neighbors function looks up the node's neighbors using the alist of edges. If the second node is in that list, we know we're one away.

```
(defun neighbors (node edge-alist)
 (mapcar #'car (cdr (assoc node edge-alist))))

(defun within-one (a b edge-alist)
 (member b (neighbors a edge-alist)))
```

First, this function looks up the first node (a) in the alist of edges with neighbors. Then it uses member to see if the other node (b) is among these nodes.

The blood stain clues for the Wumpus can also be seen from two nodes away. We can write a second function for checking two nodes like this:

```
  (defun within-two (a b edge-alist)
❶   (or (within-one a b edge-alist)
        (some (lambda (x)
❷               (within-one x b edge-alist))
❸             (neighbors a edge-alist))))
```

First, we check if we are within one node of our goal ❶, since if we're within one, we're also within two. Next, we extract all the nodes that are one away ❸ (similar to what we did in the `within-one` function). Finally, we check if any of *these* new nodes are within one ❷, which would make them within two of the original node.

Now that we have those utility functions, let's write the function that builds the final node alist (basically, the final map of our city.) Here's the listing:

```
  (defun make-city-nodes (edge-alist)
❶   (let ((wumpus (random-node))
❷         (glow-worms (loop for i below *worm-num*
                          collect (random-node))))
❸     (loop for n from 1 to *node-num*
❹         collect (append (list n)
❺                         (cond ((eql n wumpus) '(wumpus))
❻                               ((within-two n wumpus edge-alist) '(blood!)))
❼                         (cond ((member n glow-worms)
                                 '(glow-worm))
❽                               ((some (lambda (worm)
                                         (within-one n worm edge-alist))
                                       glow-worms)
                                 '(lights!)))
❾                         (when (some #'cdr (cdr (assoc n edge-alist)))
                            '(sirens!))))))
```

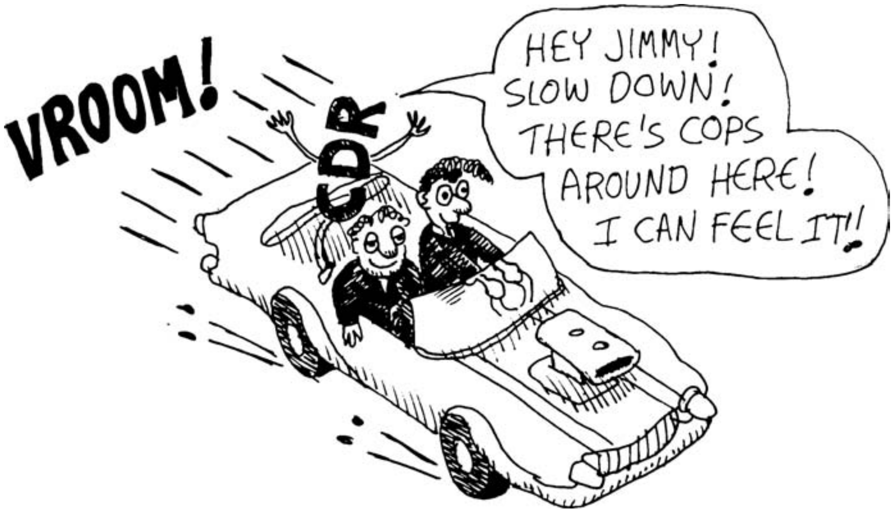
The `make-city-nodes` function first picks random nodes for the Wumpus ❶ and the Glowworms ❷, and then it uses `loop` ❸ to run through the node numbers. As it runs through the nodes, it builds a list describing each node in the city, `append`ed together from various sources ❹. By using `append`, each part of the code that describes these nodes (and is within the body of the `append`) can choose to add zero, one, or multiple items to the description, creating its own child lists with zero, one, or multiple items.

At the front of the list, we put the node name, `n` ❹. If the Wumpus is at the current node, we add the word *Wumpus* ❺ (but wrapped in a list, as we just described). If we're within two nodes of the Wumpus, we show its blood ❻. If the node has a Glowworm gang, we show it next ❼, and if the Glowworm gang is one node away, we show its lights ❽. Finally, if an edge from the node contains cops, we indicate that sirens can be heard ❾.

To check for the sirens clue, we simply grab the edges with `(cdr (assoc n edges))` and see if some of these nodes have a value in the `cdr`. The `'cops` symbol would be attached to the edges at the cdr. Since we have only one data point for edges in this game, looking for the presence of a cdr is an adequate check for the presence of cops. For example, if we use our earlier example of an alist with cops on it:

```
((1 (2)) (2 (1)❶ (3 COPS)❷) (3 (2 COPS)))
```

You can see that if an edge in the list has cops, such as here ❷, the cdr will lead to a non-nil value. An edge without cops ❶ will have a cdr that is nil.


VROOM!
CDR
HEY JIMMY! SLOW DOWN! THERE'S COPS AROUND HERE! I CAN FEEL IT!

Initializing a New Game of Grand Theft Wumpus

With our graph construction stuff out of the way, we can write a simple function that initializes a brand-new game of Grand Theft Wumpus:

```
(defun new-game ()
  (setf *congestion-city-edges* (make-city-edges))
  (setf *congestion-city-nodes* (make-city-nodes *congestion-city-edges*))
❶ (setf *player-pos* (find-empty-node))
  (setf *visited-nodes* (list *player-pos*))
  (draw-city))
```

There are two new functions here. One, the find-empty-node function ❶, ensures that the player doesn't end up on top of a bad guy right at the beginning of the game. Here's the code for that function:

```
(defun find-empty-node ()
❶ (let ((x (random-node)))
❷   (if (cdr (assoc x *congestion-city-nodes*))
❸       (find-empty-node)
        x)))
```

The `find-empty-node` function is pretty simple. First, it picks a random node ❶ to consider as the player's starting position. Then it checks whether it is a completely empty node ❷. If there's stuff in that node, it simply calls itself again, trying another random spot ❸.

WARNING *If you ever decide to modify the game and make it more crowded with bad guys, you could end up in a situation where no empty nodes exist. In that case, this function will search forever and lock up your Lisp REPL, since we didn't put in any checks to detect this situation.*

The other new function in our `new-game` command is `draw-city`, which we'll write next.

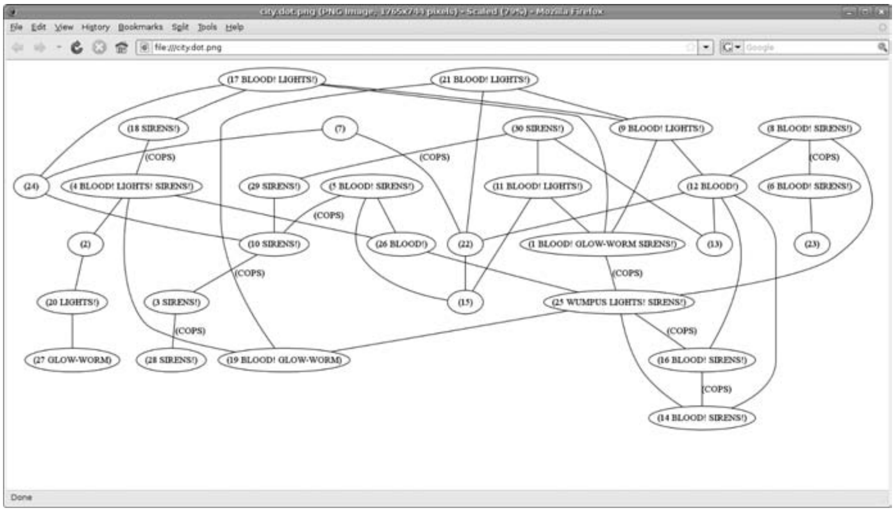
Drawing a Map of Our City

We're finally ready to draw a map of our new city. We're using a standard format for our graph data, so writing this function is a breeze:

```
(defun draw-city ()
  (ugraph->png "city" *congestion-city-nodes* *congestion-city-edges*))
```

We created the `ugraph->png` function in the previous chapter, as part of our graph library.

Now call `(new-game)` from the REPL, and open the *city.dot.png* picture in your web browser:



NOTE *Since every city map created by our code is unique, your map will look completely different from the one in this picture.*

Finally, we can marvel at the results of our urban planning!

Drawing a City from Partial Knowledge

Of course, it's awfully boring to hunt something if you already know where it is before the hunt starts. To solve this problem, we want a map of the city that shows only the nodes that we've visited so far. To that end, we use a global list called `*visited-nodes*` that is initially set to the player's position only, but which we'll update as we walk around the city visiting other nodes. Using this `*visited-nodes*` variable, we can calculate a smaller graph that includes only those parts of the city that are known to us.

Known Nodes

First, we can build an alist of just the known nodes:

```
(defun known-city-nodes ()
  (mapcar (lambda (node) ❶
            (if (member node *visited-nodes*) ❷
                (let ((n (assoc node *congestion-city-nodes*)))
                  (if (eql node *player-pos*) ❸
                      (append n '(*))
                      n))
                (list node '?))) ❹
          (remove-duplicates
              (append *visited-nodes* ❺
                      (mapcan (lambda (node) ❻
                                (mapcar #'car
                                       (cdr (assoc node
                                                   *congestion-city-edges*))))
                              *visited-nodes*)))))
```

At the bottom of `known-city-nodes`, we need to figure out which nodes we can "see" based on where we've been. We'll be able to see all visited nodes ❺, but we also want to track all nodes within one node of a visited node ❻. (We will discuss the `mapcan` function shortly.) We calculate who is "within one" using code similar to the previously discussed `within-one` function.

Next, we `mapcar` over this list of relevant nodes, processing each ❶. If the current node is occupied by the player, we mark it with an asterisk ❸. If the node hasn't been visited yet ❷, we mark it with a question mark ❹.

Known Edges

Now, we need to create an alist stripped of any cop sirens that we haven't reached yet:

```
(defun known-city-edges ()
  (mapcar (lambda (node)
            (cons node (mapcar (lambda (x)
                                 (if (member (car x) *visited-nodes*)
                                     x
                                     (list (car x)))) ❶
                               (cdr (assoc node *congestion-city-edges*)))))
          *visited-nodes*))
```

This function is similar to the known-city-nodes function. The noteworthy line of code is here ❶ where we strip the cdr from the edge list for edges so that cops are indicated on the map only if we've visited the nodes on both ends of an edge containing cops.

The mapcan Function

The mapcan function we used in known-city-nodes is a variant of mapcar. However, unlike mapcar, mapcan assumes that the values generated by the mapping function are all lists that should be appended together. This is useful when there isn't a one-to-one relationship between the items in a list and the result you want to generate.

For example, suppose we run a burger shop and sell three types of burgers: the single, the double, and the double cheese. To convert a list of burgers into a list of patties and cheese slices, we could write the following function:

```
> (defun ingredients (order)
     (mapcan (lambda (burger)
                (case burger
                   (single '(patty))
                   (double '(patty patty))
                   (double-cheese '(patty patty cheese))))
             order))
INGREDIENTS
> (ingredients '(single double-cheese double))
'(PATTY PATTY PATTY CHEESE PATTY PATTY)
```

Drawing Only the Known Parts of the City

Because we now have functions that can generate the known information about nodes and edges, we can write a function that turns this information into a picture, as follows:

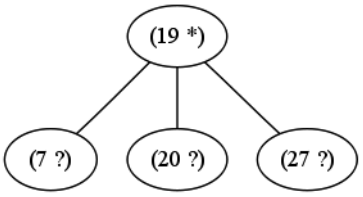
```
(defun draw-known-city ()
  (ugraph->png "known-city" (known-city-nodes) (known-city-edges)))
```

Now let's redefine our new-game function to draw the known city when the game starts:

```
(defun new-game ()
  (setf *congestion-city-edges* (make-city-edges))
  (setf *congestion-city-nodes* (make-city-nodes *congestion-city-edges*))
  (setf *player-pos* (find-empty-node))
  (setf *visited-nodes* (list *player-pos*))
  (draw-city)
❶ (draw-known-city))
```

This function is almost exactly the same as the previous version of new-game, except that we also create a drawing composed only of the known parts of the city ❶.

Now, if we call the new-game function from the REPL, we'll get a new picture named *known-city.dot.png* that we can view in our browser. It will look something like this:



Now we're ready to walk around our map of Congestion City!

Walking Around Town

We'll need two functions for traveling between the nodes in our city: a regular walk function and one for when we think we've found the Wumpus, and we want to charge that location with our final bullet. Since these two functions are very similar, we'll have both of them delegate the bulk of the work to a common handle-direction function:

```
(defun walk (pos)
  (handle-direction pos nil))

(defun charge (pos)
  (handle-direction pos t))
```

The only difference between these two functions is the flag they pass to handle-direction, which is set to either nil or t, depending on the kind of traveling.

The handle-direction function's main job is to make sure that a move is legal, which it does by checking the edges of the city:

```
  (defun handle-direction (pos charging)
    (let ((edge (assoc pos
❶                      (cdr (assoc *player-pos* *congestion-city-edges*)))))
      (if edge
❷         (handle-new-place edge pos charging)
❸         (princ "That location does not exist!"))))
```

First, this function looks up the legal directions players can move to from their current location ❶. It then uses the pos the player wants to move to and looks it up in that list of possible directions. Once we've determined that a direction is legal (that is, a node with that number shares an edge with the player's current position), we need to find out what surprises are waiting as the player travels to this new place, using the `handle-new-place` function, which we'll create next ❷. Otherwise, we display a helpful error message ❸.

Now let's create the `handle-new-place` function, which gets called when the player has traveled to a new place:

```
  (defun handle-new-place (edge pos charging)
❶   (let* ((node (assoc pos *congestion-city-nodes*))
❷          (has-worm (and (member 'glow-worm node)
                          (not (member pos *visited-nodes*)))))
❸     (pushnew pos *visited-nodes*)
❹     (setf *player-pos* pos)
❺     (draw-known-city)
❻     (cond ((member 'cops edge) (princ "You ran into the cops. Game Over."))
❼           ((member 'wumpus node) (if charging
                                       (princ "You found the Wumpus!")
                                       (princ "You ran into the Wumpus")))
❽           (charging (princ "You wasted your last bullet. Game Over."))
❾           (has-worm (let ((new-pos (random-node)))
                        (princ "You ran into a Glow Worm Gang! You're now at ")
                        (princ new-pos)
                        (handle-new-place nil new-pos nil))))))
```

First, we retrieve the node the player is traveling to from the alist of nodes ❶. Next, we figure out if the node contains a Glowworm gang ❷. We ignore the gang if they're in a node already visited, because they'll only attack once.

Next, the `handle-new-place` function updates `*visited-nodes*` ❸ (adding the new position to the list) and `*player-pos*` ❹. Then it calls `draw-known-city` ❺ again, since we now have a new place we know about.

Next, it checks to see if there are any cops on the edge ❻, and then whether the Wumpus is at that location ❼. If the player encounters the Wumpus, our `handle-new-place` function needs to know whether we were charging the location. If we are charging at the Wumpus, we win the game. Otherwise, the Wumpus kills us and the game is over.

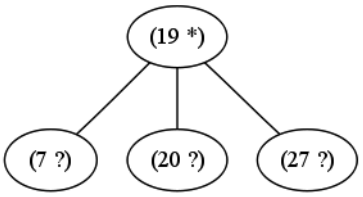
If, on the other hand, we charge at a location that does not contain the Wumpus, we waste our single bullet and we lose the game as well ❽. Finally, if the location has a previously unencountered Glowworm gang, jump to a random new location, calling `handle-new-place` recursively ❾.

Our game is now complete!

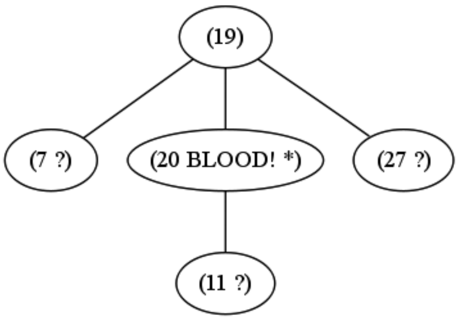
Let's Hunt Some Wumpus!

To play our game, simply enter the traveling commands we created (`walk` and `charge`) at the REPL, then switch to your browser and refresh *known-city.dot.png* to plan your next move.

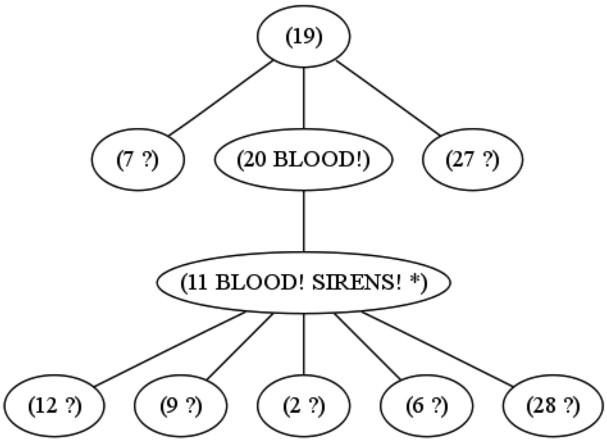
For example, here's where we left off in our sample game:



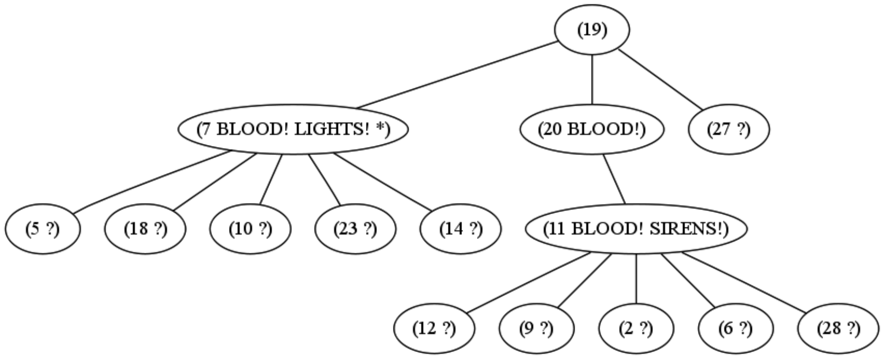
Since we have no clues, we know that any of these nodes will be safe to visit. Let's try `(walk 20)`:



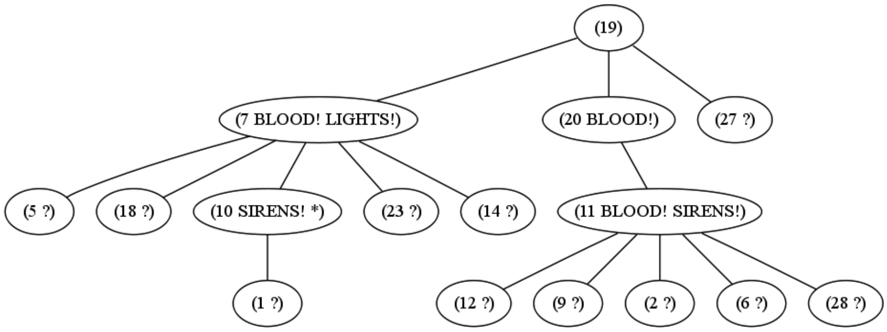
Uh oh! There's blood here. That means the Wumpus must be two nodes away! It should still be safe to `(walk 11)` though, because that's only one node away:



Oh no! One of these streets has a police roadblock. Let's backtrack with (walk 20) (walk 19), and then we can try (walk 7):



Darn! Now we have the Wumpus and some Glowworms nearby. Let's take a shot in the dark and try (walk 10):



Well, that didn't help, since there are cops down this path. However, because node 10 has only one other unexplored street, we can say with certainty that the street between 1 and 10 has cops on it.

You can see that it takes some serious thinking to become a master in Grand Theft Wumpus! Remember, you can always start a new game, with a new map, by using the new-game function. Once you've tracked down the Wumpus, use the charge function to attack him.

If you master the basic version of this game, try increasing the number of nodes, edges, cops, and Glowworms for an even greater challenge!

What You've Learned

In this chapter, we've used graph utilities with Lisp to make a more sophisticated game. Along the way, you learned the following:

- The `loop` function allows us to loop across various types of data. It will be discussed in more detail in Chapter 10.
- The `set-difference` function tells you which items are in one list but not in another list.
- The `intersection` function tells you which items are shared by lists.
- The `remove-duplicates` function removes duplicate items from a list.