Case study **14**

The book of which you are the hero

Objective. Game books are traditional text-based adventure games made up of a collection of pages (episodes) connected by references (branches). An episode comprises a text that describes a situation or an action and one or several branches that allow the gamer to jump to other episodes. Many of them are now available as *pdf* or *html* documents. In this study, we implement a simple game engine that automates such game books. This engine is based on a game database that can also be used to automatically generate stories.

Actually, this project is a nice opportunity to examine in some detail the concept of graph (a game book basically is a set of nodes and a set of edges) and to develop exploration and transformation algorithms. In particular, we study the structure of a game graph, we identify its anomalies, we extract its circuits, we build and count the different possible runs from the starting episode to an exit episode, we search for unreachable episodes and dead-end branches and we identify episodes that can be merged.

A representative heroic fantasy game book has been encoded and all the algorithms developed in the study are provided as SQLfast scripts.

Keywords. computer game, game engine, story generation, graph, cyclic graph, acyclic graph, graph transformation, Marimont algorithm, reachability, circuit, elementary circuit, transitive closure, cyclic kernel, set comparison.

14.1 Interactive adventure books and computer games

Those of us *over a certain age* should certainly fondly remember their first computer game, *Colossal Cave Adventure* (1977) or *Zork* (1979), both released on the PDP-10 mini-computer. These text-based adventure games relied on a primitive human-computer conversation: the computer writes a short description of a place or of actions taking place, then lets the user type commands, such as `go west`, `open door`, `take sword` or `kill troll with sword` (Figure 14.1).[1]

These games were then ported on personal computers. Some of them included simple graphics: instead of (or in addition to) being told that `In front of you an old house with a door and two broken windows`, the player can see a sketch of this house in B/W or in 16 colors (Figure 14.2). In addition, to make these games accessible to casual players, the available commands were written below the descriptive text, so that typing their sequence number, moving the cursor or, later, clicking on them was sufficient to move to the next step. In Figure 14.3, the user clicks on a verb, then on an object, to form a short `action-object` command.

In the 80's, descriptive texts and their list of commands (together forming an *episode*) were printed on paper and sold in bookshops. The reader was invited to follow the adventure by moving from episode to the next one, the page of which was mentioned in the command selected by the reader. These non-linear interactive books, where the reader plays an active role, were called *game books* or *books of which you are the hero*.

This is where things are getting interesting!

The structure of adventure games or game books as a set of episodes connected by commands submitted to the player is simple enough to think of some kind of automation through a database. Episodes and commands (we will call them *branches* from one episode to the next one) are stored in a database. Episodes and branches are extracted by SQL queries and presented to the player through a dialogue box.

14.2 Choosing sample game books

Quite surprisingly, nearly forty years after their first appearance, text-based adventure games still are very popular, gathering a worthy audience of retro-game amateurs[2]. Therefore, it shouldn't be too difficult to find one or two games for the sake of demonstration.

The site *Chronicles of Arborell* (http://www.arborell.com) provides a rich collection of small to large *Heroic Fantasy* game books written by Wayne Densley and available as pdf or html documents.

1. See http://classicgames.about.com/od/computergames/p/Adventures-In-Text-The-Origin-Of-Adventure-Games.htm
2. See http://textadventures.co.uk for instance.

```
You are facing the north side of a white house. There is no door here,
and all the windows are barred.

> go east

You are behind the white house. In one corner of the house
there is a window which is slightly ajar.

> open window

With great effort, you open the window far enough to allow passage.

> enter window

You are in the kitchen of the white house. A table seems to have
been used recently for the preparation of food. A passage leads to
the west, and a dark staircase can be seen leading upward. To the
east is a small window which is open.
On the table is an elongated brown sack, smelling of hot peppers.
A clear glass bottle is here.
The glass bottle contains:
A quantity of water.

> take sack

Taken.

> take bottle

Taken.

> go staircase

I don't understand that.

>
```

Figure 14.1 - Excerpt from Zork (1979)

Figure 14.2 - Mystery house (by On-Line Systems, Apple II, 1980)

Figure 14.3 - Maniac Mansion (by Lucasfilm Games, Commodore 64, 1987)

Amongst this collection, we have chosen two games,

– **TheWatchtower**[3], a small game book of 16 pages, 9 of which devoted to the description of 39 episodes (called *sections*) and 136 branches. The author modestly qualifies it *Micro-gamebook*, though we will show later that its structure may be quite complex.

– **WindhammeR**[4]: a large game book of 618 pages, 599 episodes and 993 branches.

In *TheWatchtower*, episode 15 (*section* 15 in the game) is written as follows (Figure 14.4):

> 15
>
> Moving towards the table in the south-west corner you find a collection of scrolls, journals and small paper packets filled with a grey powder. The scrolls are written in a dialect of the Hordim and you recognize them quickly for what they are.
>
> What you have found are research notes, and you understand now what the Hordim have been doing here. The grey powder is a poison and from the notes you can see that it is being altered so that it is specific to Humans alone. Luckily the Hordim have not yet perfected its chemistry and you cannot let them succeed. This is no simple raiding party. They are experimenting on human subjects and for you that presents a greater problem. Research is not a function of the Hresh for they serve as warriors alone. There is a Mutan Overseer here and he will not be far from his experiments.
>
> If you have not already done so and would begin a general search of this level *turn to section 22*.
>
> If you would make directly for the row of beds against the northern wall *turn to section 20*.

3. http://www.arborell.com/thewatchtower_text.html. "*This is a freeware title and can be copied and distributed free of charge, however all rights are reserved by the author.*"
4. http://www.arborell.com/windhammer.pdf. [same copyright as above]

If you would instead make for the lone bed in the south-eastern corner *turn to section 13*.

If there is good cause to make for the stairway to the battlements above *turn to section 17*.

Figure 14.4 - The contents of episode 15

This text comprises a short description of what appears in this episode, or, for action episodes, what happens, followed by a series of four possible decisions. Each one formulates a proposition of action and indicates the next episode number to move to, to continue the adventure.

14.3 Designing a text-based interactive adventure game

Our primary goal is to store the content of an interactive game book in a game database and to develop an SQLfast script (pompously called *game engine*!) that allows us to play this game.

We store the description of episodes in table EPISODE and the definition of branches in table BRANCH.

An episode is defined by its unique id (EPID) and the text to display (TEXT). A branch is described by the id of its source episode (EPID, a foreign key to EPISODE), its sequence number (SEQNBR) in the source episode, the text describing the decision (PROMPT) and the id of the target episode to jump to (TARGET, a foreign key to EPISODE). Branches are identified by the values of (EPID, SEQNBR).[5]

Script 14.1 shows the definition of the tables and Script 14.2 the loading of some representative objects of *TheWatchtower*.

The data have been extracted from the source text of the game and manually formatted into SQL statements. They have been remastered as follows:

– the text of some decisions has been adjusted to make them look more natural when they appear on the screen,

– artificial exit episode 0 has been added to collect all failure executions; similarly, exit episode 99 terminates all successful execution,

5. This representation is a faithful representation of the structure of the game as a graph, according to the mathematical definition of a graph as a set **V** of vertices (or nodes) and a set **E** of edges (or arcs), such that $E \subseteq \mathbf{V} \times \mathbf{V}$. Surprisingly, the textbooks and encyclopedia entries on *Graph theory* mention several data structures for representing graphs and developing algorithms, including the (elegant) incidence and adjacency matrices and the (awkward and irregular) incidence and adjacency lists, but never (as far as I know) the natural and efficient structure used in this study. This is unfortunate since it allows a more declarative expression of the main graph properties and algorithms.See [en.wikipedia.org/wiki/Graph_theory] for example.

```
create table EPISODE(
  EPID   integer not null primary key,
  TEXT   varchar(4096));
create table BRANCH(
  EPID   integer not null,
  SEQNBR integer not null,
  PROMPT varchar(256),
  TARGET integer not null,
  primary key (EPID,SEQNBR),
  foreign key (EPID)   references EPISODE on delete cascade,
  foreign key (TARGET) references EPISODE on delete cascade);
```

Script 14.1 - Data structures for the game

```
insert into EPISODE values (1,'\nIt is a truth of every
Ranger''s life that theirs'' must be a solitary existence.
...that it is time to begin.
\n\nWith the suns of Arborell ...
\n\nOutside the remains of the day flee westwards and with
their passing ... It is cold, it is dark, and without
hesitation you begin your mission.');
...
insert into EPISODE values (15,'\nMoving towards the table in
the south-west corner you find ... There is a Mutan Overseer
here and he will not be far from his experiments.');
...
insert into BRANCH values (15,1,'You have not already done so
and would begin a general search of this level',22);
...
insert into BRANCH values (15,4,'There is good cause to make
for the stairway to the battlements above',17);
...
```

Script 14.2 - Loading the description of episodes and branches

– episode id's have been transformed into purely numeric values: **?a** ⟶ **?01** and **?b** ⟶ **?02**; for example, **2b** is replaced by **202**,

– elementary formatting has been added, notably by insertion of *new lines* (`\n`).

The game engine displays the text of the current episode, followed by the list of possible decisions and lets the user select one of them. Then, the engine jumps to the selected episode and so on until an exit episode 0 or 99 has been reached.

The suggested dialogue box is shown in Figure 14.5. It is made up of two elementary boxes: a text window (`showText`) and a list of radio buttons (`selectOne`).

The player reads the text of the episode, selects one of the decisions then clicks on button OK, or Cancel to abort the game.

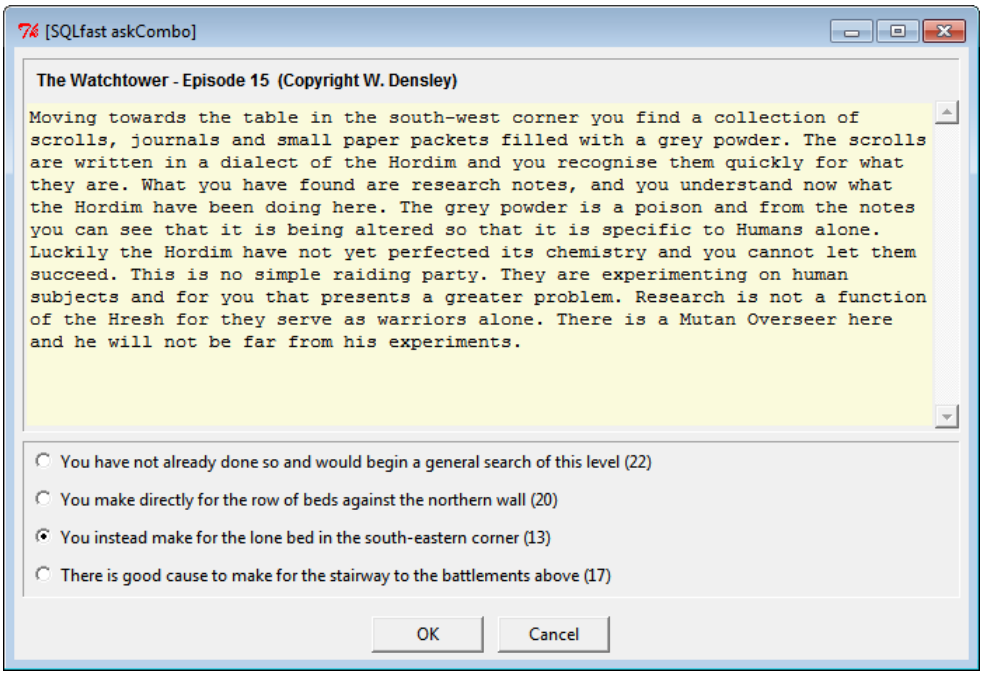

Figure 14.5 - The game dialogue box: visiting episode 15

An elementary game engine is proposed in Script 14.3. The current episode number is in variable E. The script extracts the text of the episode [1] and builds the list of decision items through a `group_concat` aggregation function [2].

```
set E = 1;
while ('$Ep$' <> '');
  extract txt = select TEXT from EPISODE where EPID = $E$;   [1]
  extract P   = select group_concat(PROMPT,'|') from BRANCH
                where  EPID = $E$ order by SEQNBR;           [2]
  askCombo [showText txt = [/b/x80/y14The Watchtower - Episode
                            $Ep$  (Copyright W. Densley)]]
           |[selectOne next = $P$];                          [3]
  if ('$DIALOGbutton$' = 'Cancel' or $Ep$ in (0,99)) exit;  [4]
   extract E = select TARGET from BRANCH
               where  EPID = $Ep$ and SEQNBR = $next$;       [5]
endwhile;
```

Script 14.3 - The game engine

Then, it displays the dialogue box [3]. It stops the game if requested to do so or if the current episode is an exit one (0 or 99) [4]. Finally, it extracts in variable E the id of the next episode [5], which becomes the new current episode. The loop of the engine comprises no more than **five statements**, among which **three SQL queries**.

This game is ready to play. Enjoy!

14.4 Story telling

Automatic novel writing is an old dream of computer scientists (those who have no talent for writing novels) and of lazy writers. Thanks to interactive game books, this dream comes true, practically at no cost.

When a game terminates, the sequence of the episodes visited forms a **story** driven by the decisions of the player. In fact, these decisions can be carried out by a script, which, for each episode, selects the output branch randomly.

For reasons we will analyze later, the probability that two runs would produce the same story is extremely low. So, each run of the game generates a new story. Running TheWatchtower game one thousand times probably will produce one thousand different stories!

Script 14.4 implements a *story generating procedure* that writes stories in external file A-TheWatchTower-Story.txt (to write in a file, the standard output is redirected to this file, through statement `outputOpen`).

```
outputOpen A-TheWatchTower-Story.txt;

set E = 1;
while ($E$ not in (0,99));
  extract txt = select TEXT from EPISODE where EPID = $E$;  [1]
  replace txt = \\n,@n;                                     [2]
  write $txt$;                                              [3]
  extract NB = select count(*) from BRANCH where EPID = $E$; [4]
  compute B = 1 + abs(random()) % $NB$;                     [5]
  extract B,PR,E = select SEQNBR,PROMPT,TARGET from BRANCH
                   where  EPID = $E$ and SEQNBR = $B$;      [6]
  if ('§PR§' <> 'Continue') write-b @n$PR$.;                [7]
endwhile;

outputAppend window;
```

Script 14.4 - The story generator

The current episode number is in variable E. The script extracts its text [1] in variable txt. In this text, new lines commands (**\n**)) are replaced by control characters **@n** [2], so that the **write** statement can manage them when writing in the external file [3]. The number of output branches of the current episode is computed and stored in

variable NB [4]. A random number in range [1,NB] is drawn and stored in variable B [5]. The branch of the current episode with sequence number B is extracted [6]. Its text, stored in variable PR, is written in the external file [7] if it is worth to (different from Continue). The generation is stopped when an exit episode (0 or 99) has been processed.

Just seven statements embedded in a loop for several million exciting stories![6]

The stories generated from *TheWatchtower* game look like this one (excerpt):

. . . .

The common area remains dark and quiet, the growing storm a rumbling backdrop that masks your footfalls upon the cold flagstones. In the shadows you consider what you should do next.

You would rather leave this floor and take to the stairs in the north-west corner.

Before you is a stairway that connects the Ground Floor to the First Level above. You listen intently but can hear nothing beyond the howl of the winds and the clatter of wooden shutters loose upon their mountings.

You would take these stairs and make for the First Level.

In the gloom of the First Level of the Watchtower you stand at the top of a stairway that leads down to the Ground Floor below. All about you is darkness, the First Floor veiled in shadows. It appears that this level served as a barracks, the eastern wall lined with a series of sleeping alcoves, the floor littered with broken furniture and discarded food. In the south-western corner you can also see the dark shadow of a stairway leading to the levels above.

You wish to explore this First Level.

Surveying the First Floor from the stairs you see evidence of a long occupation here. The Hordim have used the tower as a base of operations and it is an incursion that must be brought to an end. Outside a peal of thunder crashes against the stone walls about you and in its rolling aftermath you see a form rise from one of the sleeping alcoves. Immediately the Hresh sees you and starts to cry out, but its attempt at raising an alarm is smothered by another shuddering roll of thunder. Before the creature can fill its lungs with air again you charge it down, crashing into its side and splaying it bodily upon the floor. The warrior does not remain still. In one swift movement the Hresh regains its feet and attacks.

Clothed in a loose-fitting uniform and wielding a scimitar the Hordim closes upon you. The Hresh Warrior has a combat value of 14 and an endurance of 10. If you win this combat continue with your mission. If not, it will be here that your adventure will end.

You loose this combat.

Here ends your adventure.
It will be in a latter life that you will have to strive for greater success '

6. Let's remain modest: to generate *real novels* we should consider other aspects of the game, such as fighting (a random process) and acquiring resources!

14.5 Structure of a game

Beyond the fun of playing adventure games, there is the fun of analyzing and changing the structure of these games[7]. Basically, as we observed it above, games can be modeled as **graphs**, so that the concepts and techniques of graph theory can be applied, at least to some extent. However, we will mainly reason on game concepts instead of on abstract graph concepts. Let us first examine how games are structured.

14.5.1 Episodes, branches, paths and runs

The link from current episode E1 to episode E2, referenced by a decision of E1, is called a **branch**. E1 is the *source episode* of this branch while E2 is its *target episode*. E2 is a *successor* of E1 and E1 is a *predecessor* of E2.

Normally, the successors of an episode are distinct. However, some games may propose several branches to the same target episode for a source episode, as can be observed in WindhammeR. For the sake of simplicity, we will postulate the uniqueness hypothesis. If two decisions do lead to the same target episode, they are merged through an **or** logical connector.

The excerpt above shows that episode **15** is the source of four branches, the targets of which are episodes **22**, **20**, **13** and **17**. Episode **15** is also the target of five branches, namely from episode **17**, **20**, **22**, **26** and **201** (formerly **2a**).

The graph of Figure 14.6 depicts episode **15** with its predecessors and its successors. Let us call a **path** a suite of connected branches. For instance, the suite **26.15.13**, obtained by composing branch (**26**,**15**) with branch (**15**,**13**), is a path from episode **26** (the source of the path) to episode **13** (the target of the path). The **length** of a path is the number of branches it is made up of. The length of **26.15.13** is 2.

A **run** of the game is a path that starts at episode 1 and ends at exit episodes 0 (*fail*) or 99 (*win*).

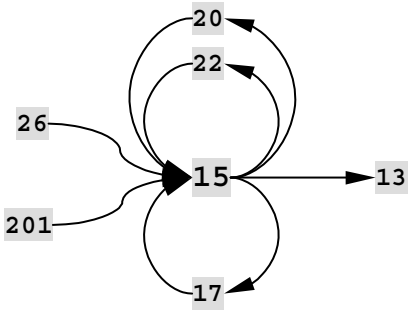

Figure 14.6 - Episode 15 together with its environment

7. Not to mention the fun of writing them.

14.5.2 Starting and ending episodes

A game book has one starting point, which is the first episode to visit. In *TheWatchtower* and *WindhammeR*, the starting point is episode 1. The game has at least one successful exit point and one failure exit point. They define the end of the game and the result of the run: *win* or *fail*.

14.5.3 Circuits

In most games, an episode can be visited more than once. There are several reasons for this:

- the episode is the source of several branches, each on them worth to be followed,
- some successors of an episode are made reachable only when the player has acquired some definite resources, such as a key or a special skill,
- the *mischief* of the game designer, who find it funny to lead the player to an episode already visited,
- or the player is a bit absent-minded or disoriented ...

A path the source and target of which are the same episode is called a **circuit**. Since there is no branch from an episode to itself, a circuit comprises at least two branches.

An episode can be visited several times in the same run if it appears in a circuit. Figure 14.6 makes it clear that *TheWatchtower* includes circuits. Actually, a circuit has no natural starting point, so that the paths E1.E2.E3, E2.E3.E1 and E3.E1.E2 all define the same circuit. However, we will find it convenient to designate one of its episode as its (arbitrary) starting point. This way, by choosing E1 as the starting point of the circuit, we can unambiguously denote the latter by path E1.E2.E3.

We could conclude from the observation of the graph of Figure 14.6 that it comprises three circuits of length 2, namely **15.20.15**, **15.22.15** and **15.17.15**.

However, it also comprises circuit **15.20.15.17.15** of length 4, as well as **15.20.15.20.15**. In fact, the graph includes an infinite number of circuits! As a consequence, there are quite a lot of paths (exactly an *infinity*) from episode **26** to episode **13**. Note that **26.15.20.15.13** is not a circuit but a path that includes a circuit.

We call **elementary** a circuit the episodes of which appear only once, except for its starting point. Circuit **15.20.15** is elementary while path **15.20.15.17.15** is not. The graph of Figure 14.6 includes three elementary circuits. We observe that an episode can be part of more than one circuit, as evidenced by episode **15**.

Two final observations: in a circuit, at least one episode must have two or more predecessors, otherwise the player would not be given the opportunity to enter the circuit! Similarly, at least one episode must have two or more successors, otherwise there would be no way to exit the circuit.

14.6 Analyzing a game

In this section, we will examine two properties of games: whether they are well-formed and whether they include circuits.

14.6.1 Searching for game abnormalities

A well-formed game must satisfy some structural properties. We will mention some of them, together with the SQL queries that can be used to check them.

a) The game has one and only one starting point

Checking: we count the episodes that have no predecessors. There should be only one.

```
select EPID,count(*) from EPISODE
where  EPID not in (select TARGET from BRANCH);
```

b) The game has at least one successful exit point

Checking: we count the branches the target of which is episode 99.

```
select count(*) from BRANCH where TARGET = 99;
```

c) The game has at least one failing exit point

Checking: we count the branches the target of which is episode 0.

```
select count(*) from BRANCH where TARGET = 0;
```

d) There is no isolated episode

Checking: every episode must appear in some branch.

```
select EPID from EPISODE
where  EPID not in (select EPID from BRANCH
                    union
                   select TARGET from BRANCH);
```

e) There is no circuit of length 1

Checking: we count the branches where the source is the target.

```
select count(*) from BRANCH where EPID = TARGET;
```

f) Every episode is reachable from the starting point

Checking: this one is a bit more complicated. We leave it for further investigation.

g) Each episode contributes to a run

Checking: same comment. We leave it for further investigation.

TheWatchtower appears to be devoid from these abnormalities.

14.6.2 Does the game include circuits?

We want to know whether the game includes circuits. If it doesn't, then it is called *acyclic*. It may be good news since then we are sure that we can finish the game in a finite number of steps, preventing us from spending the whole night to complete it. To cope with this question, we will first reason on two small games.

The first game is acyclic (Figure 14.7). It comprises 8 episodes, that cannot be visited more than once. Therefore, before examining the structure of the game, we are guaranteed that it can be completed in at most 7 steps (8 - 1, because only one exit episodes can be reached). Actually, a more thorough examination shows that it can be completed in 3, 4 or 5 steps.

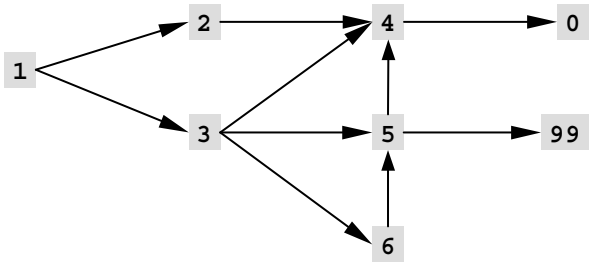

Figure 14.7 - An acyclic game

The second game, on the contrary, includes a circuit comprising episodes 2, 4 and 3 (Figure 14.8). The difference with acyclic games is that some episodes can be visited more than once. For this, they are called *cyclic episodes* and the game is said *cyclic* as well.

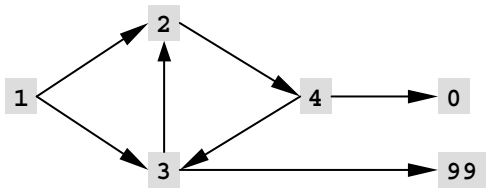

Figure 14.8 - A game including a circuit

Let us try to *prune* (i.e., to eliminate episodes from) the acyclic game as follows[8]:

– *step 1*: we remove all the entry episodes (if any), that is, episodes that have no predecessor

– *step 2*: if the remaining game still comprises entry episodes, we apply step 1.

Figure 14.9 shows what happens to acyclic game of Figure 14.7 when we apply this procedure. First, entry episode 1 is removed. Then, entry episodes 2 and 3 disappear, followed by episode 6, then 5, then 4 and 99. Finally, episode 0, which has become an entry episode, is removed. The result is an empty game.

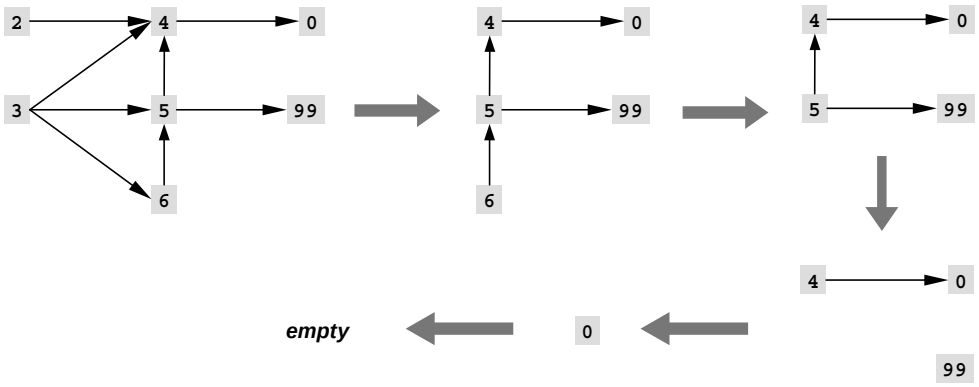

Figure 14.9 - Iteratively eliminating **entry** episodes until the game vanishes

The same result can be achieved by iteratively removing **exit** episodes. The following episodes are removed: first 0 and 999, then 4, then 2 and 5, then 6, then 3, and finally 1 to yield an empty game.

This experiment illustrates an interesting property of acyclic games: when we iteratively remove entry episodes, or exit episodes, or both, we finally remove all the episodes of the game.

How about applying this procedure to games with circuits? Let us try it with the game of Figure 14.8. First we remove episode 1, then ... things are blocked! Indeed, episodes 2 and 3 both have a predecessor and cannot be removed. Starting from the exit episodes does not help: we can remove episodes 0 and 99, but the remaining game now has no exit episode any more. Removing both entry and exit episodes yield the graph of Figure 14.10 (which is no longer a game since it has no entry nor exit episodes) which is cyclic.

8. This procedure is an application of *Marimont algorithm*, based on the property that a non empty graph that has no entry points and no exit points includes circuit(s).

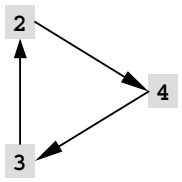

Figure 14.10 - Iteratively eliminating **entry** and **exit** episodes reduces the game to its cyclic episodes

This procedure is simple and intuitive. It allows us to discover whether a game is acyclic. Its SQL translation in particularly straightforward. The *entry episodes* of the game are defined by the following SQL expression,

```
select EPID from EPISODE
where  EPID not in (select TARGET from BRANCH)
```

and its *exit episodes* by,

```
select EPID from EPISODE
where  EPID not in (select EPID from BRANCH)
```

Therefore, the *entry* and *exit* episodes can be removed through these SQL queries:

```
delete from EPISODE
where  EPID not in (select TARGET from BRANCH);
delete from EPISODE
where  EPID not in (select EPID from BRANCH);
```

Hence the procedure of Script 14.5. The loop is controlled by variables **Nin** counting the *entry* episodes [1] and **Nex** counting the *exit* episodes [2]. When both of them have value zero, it is time to exit the loop [3]. The game is acyclic if BRANCH is empty, otherwise, this table contains all the branches of the circuits of the game. Applied to the game TheWatchtower, this procedure removes 9 episodes and 26 branches.

It is worth noticing that, though all the cyclic episodes belong to the remaining set, not all the remaining nodes can be qualified cyclic. Indeed a node that belongs to no circuit but that *bridges* two circuits cannot be eliminated and therefore belongs to the remaining set. In a further case study, *From data bulk loading to database book writing*, we will also exploit this elimination algorithm. This set of remaining nodes will be called the **cyclic kernel** of the graph.

```
while (True);
   extract Nin = select count(*) from BRANCH
           where EPID not in (select TARGET from BRANCH);   [1]
   extract Nex = select count(*) from BRANCH
           where TARGET not in (select EPID from BRANCH);   [2]
   if ($Nin$ = 0 and $Nex$ = 0) exit;                       [3]
   delete from EPISODE
   where  EPID not in (select TARGET from BRANCH);
   delete from EPISODE
   where  EPID not in (select EPID from BRANCH);
endwhile;
```

Script 14.5 - Reducing a game to its cyclic kernel

14.7 Extracting circuits

Identifying all the circuits of a game may appear a non trivial task. Actually, it involves some interesting challenges but, once carefully analyzed, step by step, it proves quite tractable. The basic idea is simple:

> *we build all the paths of the game by iteratively composing branches then we select those the first and last episodes of which are identical.*

This is a nice example of recursive CTE, expressed in Script 14.6. To simplify the construction, we represent a path by a sequence of episode id's separated by dots as we did before, but we also add a leading dot and a trailing dot so that each id is surrounded with dots. In addition, we consider that episode id's are character strings of length 2 (or more for very large games).[9]

The initialization query inserts in CTE PATHS columns EPID and TARGET of the BRANCH rows and builds from them the first element of the paths. The recursive query extends the paths in PATHS with the rows of BRANCH that match on PATHS.TARGET = BRANCH.EPID. They also add a new branch to existing paths. The paths built in this way are stored in table CIRCUIT defined as

```
CIRCUIT(Circuit)
```

Actually, this query does not work, and this for several reasons.

9. Otherwise, we would have to transform integers into fixed length strings through expression `cast(EPID as char(2))`, or, in SQLfast `frame(cast(EPID as char),2,'>','0')`. This would inevitably obscure the queries. The scripts of game analysis associated with this study use this convention.

```
with recursive PATHS(EPID,TARGET,Path) as
  (select EPID,TARGET,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH
      union
   select P.EPID,B.TARGET,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   )
insert into CIRCUIT
select Path from PATHS where EPID = TARGET;
```

Script 14.6 - Extracting the circuits of a game - First attempt

First, the graph of the game can be so large that computing the set of possible paths would take hours, if not days. So we build a safety mechanism that limits the length of the paths through CTE column Length. Its value is initialized to 1 and incremented by the recursive query. We also add column Length to CIRCUIT.

Secondly, when a circuit has been built in PATHS, new paths will be built by extending it with additional branches, which is useless. So, we freeze such paths by allowing extension of paths only if their first and last episodes are different.

Finally, since we are interested in *elementary circuits* only, we do not allow paths to be extended when they already include the node we intend to add, except of course when the latter is also the first one (in which case we are building a circuit). So, we add a constraint like this one:

```
P.Path not like '.%.'||B.TARGET||'.%'
```

What does this condition says? P.Path is the current path we intend to extend. B.TARGET is the candidate new episode. This episode may not appear in P.Path:

```
P.Path not like '%.'||B.TARGET||'.%'
```

except if it appears in the first position. So, to discard this case, we add a dot in front of the pattern to force the condition to ignore the first episode of the path:

```
'.%.'||B.TARGET||'.%'
```

The improved version of the algorithm is shown in Script 14.7.

Now, table CIRCUIT includes all the elementary circuits of the game. However, it also includes all the **permutations** of these circuits. Applied to the graph of Figure 14.8, that includes a single circuit, the query produces the content shown in Figure 14.11.

```
set len = 16;
with recursive PATHS(EPID,TARGET,Length,Path) as
  (select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH
      union
   select P.EPID,B.TARGET,P.Length+1,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   and    P.EPID <> P.TARGET
   and    P.Path not like '.%.'||B.TARGET||'.%'
   and    P.Length < $len$
   )
insert into CIRCUIT
select Path,Length from PATHS where EPID = TARGET;
```

Script 14.7 - Extracting the circuits of a game - Next try

```
+---------------+--------+
| Path          | Length |
+---------------+--------+
| .02.04.03.02. | 3      |
| .04.03.02.04. | 3      |
| .03.02.04.03. | 3      |
+---------------+--------+
```

Figure 14.11 - Permutations of the same circuit

So, the next problem we have to solve is to reduce all the permutations of a circuit to a single one. This does not seem obvious at first glance. Fortunately, this problem is a special case of a common, more general problem that we may encountered in many other situations: *comparing sets*. Indeed, the common property of these permutations is that they are defined on the same **set** of episodes.

Since SQL basically is a *set-oriented language*, we could express set comparison by an SQL query, for instance by checking that each element of the first one belongs to the second one and that they have the same size. However, such a query would be terribly inefficient in this situation, in which we have to compare hundreds of millions of pairs of sets. We will develop another technique based on a character string derived from the path and that uniquely denotes the set of its episodes, for instance this one, in which the episode id's are sorted:

```
02.03.04
```

This character string is common to all the permutations of a circuit, so that we can gather them with a simple `group by` and select, say, the permutation with the lowest rank in the alphanumeric order (`min(Path)`), that is, here, `.02.04.03.02.`. We suggest to add column EpSet (episode set) to table CIRCUIT, in which we will store the denotation of the set of episodes of each path:

```
+---------------+--------+----------+
| Path          | Length | EpSet    |
+---------------+--------+----------+
| .02.04.03.02. | 3      | 02.03.04 |
| .04.03.02.04. | 3      | 02.03.04 |
| .03.02.04.03. | 3      | 02.03.04 |
+---------------+--------+----------+
```

There is a last problem to solve: building this set denotation. We will derive it from the expression of the circuit. First, we remove the first and last dots:

```
trim(Path,'.')
```

Now, we have a dot-separated list of episode id's. We sort this list with SQLfast function itemSort. This function requires four parameters: the list to sort, the sorting direction (0 = ascending), the uniqueness (0 = duplicates allowed; 1 = duplicates discarded) and the separator:

```
itemSort(trim(Path,'.'),0,1,'.')
```

The completed code is shown in Script 14.8 and its result in Figure 14.12.

```
set len = 16;
with recursive PATHS(EPID,TARGET,Length,Path) as
  (select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH
      union
   select P.EPID,B.TARGET,P.Length+1,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   and    P.EPID <> P.TARGET
   and    P.Path not like '.%.'||B.TARGET||'.%'
   and    P.Length < $len$
   )
insert into CIRCUIT
select min(Path),Length,
       itemSort(trim(Path,'.'),0,1,'.') as EpSet
from   PATHS
where  EPID = TARGET group by EpSet;
```

Script 14.8 - Extracting the circuits of a game - Final version

Circuit	Length	EpSet
.06.18.06.	2	06.18
.06.21.06.	2	06.21
.06.28.06.	2	06.28
.06.30.06.	2	06.30
...	...	...
.02.24.29.02.	3	02.24.29
.02.25.29.02.	3	02.25.29
.02.27.29.02.	3	02.27.29
.03.09.14.03.	3	03.09.14
...	...	...
.02.12.24.29.02.	4	02.12.24.29
.02.24.25.29.02.	4	02.24.25.29
.02.24.27.29.02.	4	02.24.27.29
.02.24.34.29.02.	4	02.24.29.34
...	...	...
.02.12.24.25.29.02.	5	02.12.24.25.29
.02.12.24.27.29.02.	5	02.12.24.27.29
.02.12.24.34.29.02.	5	02.12.24.29.34
.02.24.25.27.29.02.	5	02.24.25.27.29
...	...	...
.02.12.24.25.27.29.02.	6	02.12.24.25.27.29
.02.12.24.25.34.29.02.	6	02.12.24.25.29.34
.02.12.24.27.23.29.02.	6	02.12.23.24.27.29
.02.12.24.34.27.29.02.	6	02.12.24.27.29.34
...	...	...
.02.12.07.11.37.34.29.02.	7	02.07.11.12.29.34.37
.02.12.24.25.27.23.29.02.	7	02.12.23.24.25.27.29
.02.12.24.25.34.27.29.02.	7	02.12.24.25.27.29.34
.02.12.24.34.27.23.29.02.	7	02.12.23.24.27.29.34
...	...	...

Figure 14.12 - Table CIRCUIT: elementary circuits of game TheWatchTower

The Watchtower game comprises 437 elementary circuits made up of 2 to 15 branches.

14.8 Counting runs

One of the important questions about a new game is whether the player would like to play it again. This refers to the variety of the runs, that is, to the number of the different paths starting at episode 1 and reaching exit episode 0 (fail) or 99 (win).

To answer this question for a definite game, we generate all the paths starting at episode 1, then we store the runs among them in new table RUN, defined as follows:

```
RUN(EXIT,Length,Path)
```

Column EXIT specifies the result of the run: *win* (99) or *fail* (0), Length the number of branches of the run and Path the sequence of episodes.

The general structure of the query that computes the runs (Script 14.9) is similar to that of Script 14.8. The main difference is that we initialize the process with the branches starting from episode 01 only.

A run can be *acyclic*, in which case all the episodes are distinct, or *cyclic* if the run includes one or several circuits. If we allow unconstrained runs, allowing them to follow any number of circuits, then the answer to the counting problem is clear and inexpensive: *infinite*.

So, we must control the number of circuits allowed in the runs computed by the query. The control mechanism is a condition symbolically called **<cyclic>** in Script 14.9.

```
set Len = 20;
with recursive PATHS(EPID,TARGET,Length,Path) as
  (select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH where EPID = '01'
      union
   select P.EPID,B.TARGET,P.Length+1,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   and    P.EPID <> P.TARGET
   and    <cyclic>
   and    P.Length < $Len$
   )
insert into RUN
select TARGET,Length,Path
from   PATHS where TARGET in ('00','99');
```

Script 14.9 - Computing the runs of a game

This condition is similar to that coded in Script 14.8. For linear runs, it tells that the path we intend to extend *may not include the new episode*, i.e.,

```
P.Path not like '%.'||B.TARGET||'.%'
```

For cyclic runs, we specify the number of episode instances allowed in runs. For instance, if we allow **one instance** of each elementary circuit,[10] the pattern becomes:

```
'%.'||B.TARGET||'.%.'||B.TARGET||'.%'
```

... for **two instances**,

```
'%.'||B.TARGET||'.%.'||B.TARGET||'.%.'||B.TARGET||'.%'
```

10. This is not quite exact. When a circuit has been followed, all the other circuits that share some episode with it will not be followed.

... and so on.

Linear runs

Let us first examine the result of this path generation procedure applied to the small acyclic game of Figure 14.7. It identifies 6 runs, 2 of which are successful and the other 4 fail (Figure 14.13).

EXIT	Length	Path
00	3	.01.02.04.00.
00	3	.01.03.04.00.
99	3	.01.03.05.99.
00	4	.01.03.05.04.00.
99	4	.01.03.06.05.99.
00	5	.01.03.06.05.04.00.

Figure 14.13 - The runs of acyclic game of Figure 14.7

Among the linear runs of the cyclic game of Figure 14.8, 2 are successful and 2 fail (Figure 14.14).

EXIT	Length	Path
99	2	.01.03.99.
00	3	.01.02.04.00.
00	4	.01.03.02.04.00.
99	4	.01.02.04.03.99.

Figure 14.14 - The linear runs of cyclic game of Figure 14.8

Cyclic runs

Now, let us allow runs to include at most **one instance** of each circuit. The application of the procedure to the cyclic game of Figure 14.8 generates the eight runs of Figure 14.14.

EXIT	Length	Path
99	2	.01.03.99.
00	3	.01.02.04.00.
00	4	.01.03.02.04.00.
99	4	.01.02.04.03.99.
99	5	.01.03.02.04.03.99.
00	6	.01.02.04.03.02.04.00.
00	7	.01.03.02.04.03.02.04.00.
99	7	.01.02.04.03.02.04.03.99.

Figure 14.15 - Acyclic and single cyclic runs of cyclic game of Figure 14.8

Analysis of TheWatchTower

Now, we consider a real but small game, *TheWatchTower*. Computing all the *linear paths* from episode 01, be they complete (i.e., runs) or incomplete, gives impressive results:

- 1,004,389,004 paths, that is, more than one billion.
- 213,508,282 runs
- 126,344,842 failing runs
- 87,163,440 successful runs
- the shortest run comprises 3 episodes (01.28.00)
- the longest run is made up of 38 episodes; there are 46,080 of them
- the most frequent run length (26,947,948) is 30 episodes.

One can guess that this result comes at a price: **more than 9 hours** of computing at full speed.[11] To get faster but more modest data we must limit the maximum length of the runs. The numbers of Figure 14.16 have been computed for runs from 2 to 20 branches. Failing runs are much more frequent than winning ones. An unfair habit of most games! One can fail after just two episodes but to win, one must visit at least 13 episodes.

Length	Win	Fail
2	0	1
3	0	3
4	0	11
5	0	31
...	...	...
12	0	9,341
13	1	22,567
14	13	51,870
15	95	113,343
16	514	236,366
17	2,281	471,998
18	8,712	905,093
19	29,367	1,670,646
20	88,753	2,975,483

Figure 14.16 - The runs of TheWatchTower, from 2 to 20 episodes

If you can complete a game in an average of 10 minutes, then *TheWatchtower* can give you 20,218 **years** of pleasure! Full time.

However, we are not done yet. So far, we have allowed *linear games* only, which is fairly unrealistic. We can use Script 14.9, allowing episodes to be visited twice,

11. Stored in a 105 GB database

three or four times (several episodes even have 5 or 6 successors). This will augment the number of runs considerably.

14.9 Are all episodes reachable?

We have left this question unanswered in Section 14.6.1 because we had not yet developed the necessary algorithm to solve it. Now we are ready to address this issue.

Basically, an episode is **reachable** if it belongs to at least one path (not necessarily a run) starting from episode 01. A first idea would be to identify the episodes that have no predecessor, that is, that do not appear as a target in any branch:

```
select EPID from EPISODE
where  EPID not in (select TARGET from BRANCH);
```

However, this query will not catch all unreachable episodes. For instance, all the episodes of a circuit *with no entry point* are targets but they are unreachable.

Then, we could adapt the algorithm of Script 14.9 a little bit, so that it generates all the paths of the game (that start at episode 1) and not only its runs. The TARGET components of these paths form the set of the reachable episodes. Therefore, the elements of EPISODE that are not in this set are not reachable and constitute abnormal episodes. Unfortunately, this technique is very expensive: it generates more than *one billion paths* in about *9 hours* for a game that comprises 39 episodes only. So, we must find a far more efficient algorithm.

When the CTE of Script 14.9 builds a path of the game, its last episode is a *reachable episode*. We can stop the extension of any other path with it since this would bring us information we already know. Practically, if episode E is known to be reachable, we can delete all the branches where TARGET = E.

Script 14.10 translates this idea. Since we will modify data in the process, we cannot express it through a CTE. So, we will simulate it with a loop as we already did in Section 12.2 of case study *Kings of France - Part 2*.

Table PATH, that will contain the paths to build, is initialized with the branches starting from episode 1. The next set of paths will be stored in table LAST which has the same structure as PATH. Table BBRANCH is a copy of BRANCH, the rows of which will be progressively deleted. The algorithm works as follows.

– Variable NE stores the number of episodes, which is the maximum number of iterations.

– Each iteration considers the *last paths* inserted in PATH (those with length = $length$) and extends them with the matching rows from BBRANCH. The extended paths are stored in table LAST then moved (`insert` + `delete`) to PATH.

```
insert into PATH(START,END,Length,Path)
   select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH where EPID = '001';
create table LAST as select * from PATH where 1 = 2;
insert into RBRANCH
   select EPID,TARGET from BRANCH where EPID <> '001';
extract NE = select count(*)-1 from EPISODE;
for length = [1,$NE$];
   insert into LAST
      select P.START, B.TARGET,
             $length$ + 1, P.Path||B.TARGET||'.'
      from   PATH P, RBRANCH B
      where  P.END = B.EPID
      and    P.Length = $length$
      and    P.Path not like '%.'||B.TARGET||'.%';
   extract N = select count(*) from LAST;
   if ($N$ = 0) exit;
   insert into PATH select * from LAST;
   delete from LAST;
   delete from RBRANCH where TARGET in (select END from PATH);
endfor;
```

Script 14.10 - Finding all the reachable episodes

– *Critical step*: at the end of each iteration, the rows of BBRANCH, the target of which is one of the episodes just added, are deleted.

– The reachable episodes are in column END of PATH:

```
select distinct END from PATH
```

The episodes of EPISODE that are not in PATH are unreachable. If the game is consistent, only episode 1 is unreachable!

The performances of this modified algorithm are very impressive. For game TheWatchTower (41 episode, 136 branches), table PATH is filled with 58 paths[12] in **0.02 s**. For WindhammeR (599 episodes, 993 branches), 64,289 paths have been generated in **0.92 s**.

12. We could wonder why this number is higher than that of episodes. Indeed, though no iteration identifies an episode that has already been identified by preceding iterations, the same episode may be identified more than once within the same iteration.

14.10 Do all episodes contribute to a solution?

An episode is said to *potentially contribute to a run* if it appears in at least one run. If it doesn't, even though it is reachable, then it belongs to a dead-end path and is useless. Just like the reachability problem, this one can be solved by a standard, but very expensive, algorithm based on the set of all the paths of the game. However, we can transform it into a known problem, by observing that this question is the inverse of the reachability problem. If we **inverse all the branches**, swapping EPID and TARGET values, we just have to prove that all episodes can be reached from episode 0 or episode 99.

14.11 Merging episodes

When an episode **E1** has only one successor **E2**, and if **E1** is the only predecessor of **E2**, we can think of simplifying the game by merging **E1** and **E2** into single episode **E12**. The text of **E12** is the text of **E1** followed by the text of **E2,** the predecessors of **E12** are those of **E1** and the successors of **E12** are those of **E2**. We call branch (**E1**,**E2**) *one-to-one*.

Figure 14.17 shows how episodes **7** and **11**, connected by *one-to-one* branch (**7**,**11**), are merged to form new episode **711**.

We discard artificial exit episodes **0** and **99** from this restructuring. In addition, **E1** cannot be the target of **E2**, otherwise, the merging would create a *reflexive* branch (here, from **711** to itself).

TheWatchtower includes 3 such patterns: (**7**,**11**), (**13**,**31**), and (**39**,**19**). They have been identified by Script 14.11, that works as follows. Episodes **E1** and **E2**, linked by a branch, can be merged if:

- **E2** is not an exit episode [1],
- **E1** has only one successor [2],
- **E2** has only one predecessor [3],
- **E1** is not a target of **E2** [4].

Merging these couples reduces the size of the game by 3 episodes and 3 branches.

In WindhammeR, 93 couples to merge have been found, among them 24 form 12 double chains (two successive one-to-one branches), 6 form 2 triple chains and 8 form 2 quadruple chains.

The writing of the transformation algorithm is not particularly difficult. It is left as an exercise.

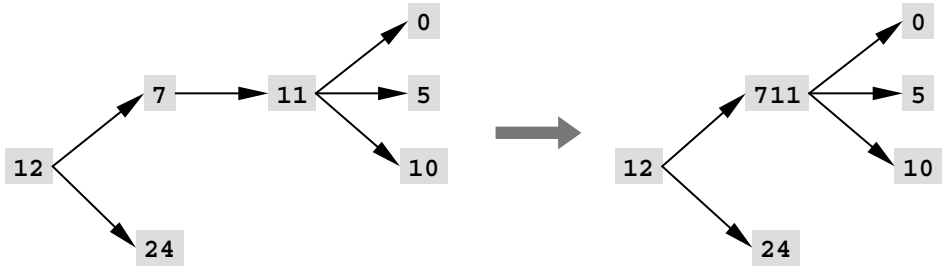

Figure 14.17 - Merging episodes 7 and 11

```
select EPID,TARGET
from   BRANCH B
where  TARGET not in ('00','99')                                [1]
and    (select count(*) from BRANCH
        where  EPID = B.EPID) = 1                               [2]
and    (select count(*) from BRANCH
        where  TARGET = B.TARGET) = 1                           [3]
and    (TARGET,EPID) not in (select EPID,TARGET from BRANCH); [4]
```

Script 14.11 - Identifying episodes that can be merged

14.12 Toward a better game engine

The game engine we have built is a very basic prototype. It can be improved in several ways. Let us begin with some goodies easy to develop.

– The database stores one game at a time. It would be nice to allow it to contain several games and to let the user choose which one she wants to play with. *The* user? Why not several users?

– There is no way to suspend the execution of a game, then to resume it later. This requires a *game saving* function, that saves the state of the game (aka a *save point*) and that restores it when the player wants to continue the game. A system allowing several saves of the game would be better.

– Why not show pictures or drawings in the episode display box (as in Figure 14.2)?

– Combat resolution requires the player to roll dices to determine who, among the player and the enemy, wins the combat. Dice rolling is simple to implement through SQLfast random() function.

– A game recording function would allow the player to replay, to undo (roll-back), to examine or to share a run.

– Some game books include an important descriptive part related to the world in which the game takes place including the actors, the arms, the resources, the language, the society rules and the history of this world. Large game books sometimes comprise a glossary of the specific terms. These resources can easily be implemented as help documents that can be opened from the dialogue box of the game.

Some improvements would require more effort but will make the games more fun to play.

– The player, and sometimes other characters of the game, can be assigned specific attributes or resources: strength, agility, endurance, weapons, health, energy, money, etc. These attributes generally are used to decide the way the player behaves when facing some problematic situations, such as combat, hunger or wound. These attributes can evolve: decreasing when used and increased when resources are encountered in some episodes (the ubiquitous *health packs* for example).

– What about a map of the episodes already visited?

– When the player can take objects found in some episodes, then an inventory system is required. These objects can be used to obtain some desired results, such as using a key to open a door or giving money to buy food. Figure 14.3 shows a nice way, easy to implement with two `selectOne` elementary boxes, to let the player perform an action on an object of the inventory.

– When the game model gets richer, storing its definition in the database through SQL `insert` queries may become more tricky. Why not design a GDL (*game definition language*)? And, of course, the program that translates it into SQL `insert` queries!

14.13 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Game_Book**. Actually, they can be run from main script **GameBook-MAIN.sql**, that displays the selection box of Figure 14.18.

Figure 14.19 shows the dialogue box that sets the parameters of the analysis of a game.

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

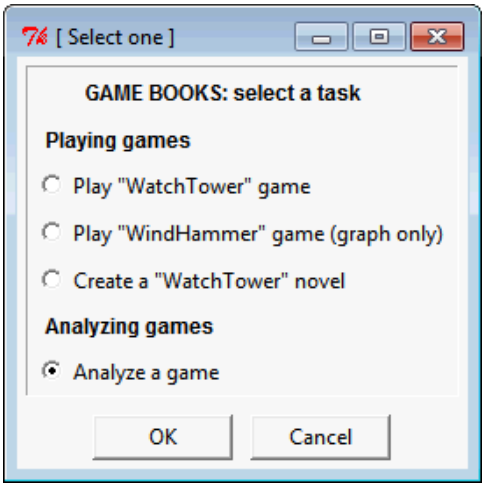

Figure 14.18 - Selecting an activity

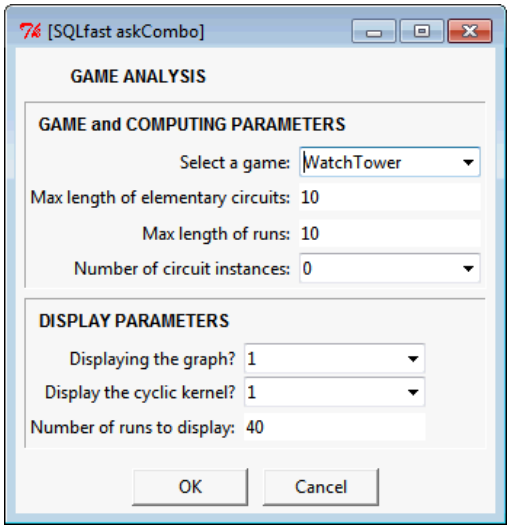

Figure 14.19 - Setting the parameters for game analysis