Connecting PDDL-based off-the-shelf planners to an arcade game

Olivier Bartheye and Éric Jacopin [1]

Abstract. First, we explain the Iceblox game, which has its origin in the Pengo game. After carefully listing requirements on game playing, the contents of plans, their execution and planning problem generation, we design a set of benchmarks to select good playing candidates among currently available planners. We eventually selected two planners which are able to play the Iceblox video game well and mostly in real time. We outline our planning architecture and describe both the predicates and a couple of the operators we designed. We wish to report that we never tweaked the planners neither during the benchmarks nor during video game playing.

1 ICEBLOX

Description In the Iceblox video game [3], the player presses the arrow keys to move a penguin horizontally and vertically in rectangular mazes made of ice blocks and rocks, in order to collect coins. But flames patrol the maze (their speed is that of the penguin) and can kill the penguin in a collision; moreover, each coin is iced inside an ice block which must be cracked several times before the coin is ready for collection. The player can push (space bar) an ice block which will slide until it collides with another ice block, a rock or any of the four side of the game. A sliding ice block stops when it collides into another ice block, a rock or any of the four sides of the game and kills a flame when passing over it. If the player pushes an ice block which is next to another ice block, a rock or a side of the game, it cannot slide freely and shall begin to crack. Once cracked, an ice block cannot move any more and thus pushing a cracked ice block eventually results, after seven pushes, in its destruction. An ice block which contains a coin slides and cracks as does an ordinary ice block. A coin cannot slide; it can only be collected once revealed after the seventh push. The player gets to the next, randomly generated, level when all coins have been collected. Killed flames reappear from any side of the game and from time to time, as in Figure 1, it happens that locking the flames is a better strategy than killing them: since there is no constraint on time to finish any level of Iceblox, this strategy gives the player all the latitude to collect coins. The number of flames and rocks increase during several levels and then cycle back to their initial values. Finally, the player gets 200 points for each collected coin, 50 points for each killed flames and begins the game with three spare lives.

Why Iceblox? Iceblox is an open and widely available java implementation [1, pages 264–268] of the Pengo arcade game, published by Sega in 1982 (see Figure 2). In Pengo, there are pushable and crackable ice blocks but neither rocks nor coins; the main objective of the game is to kill all the bees patrolling the ice blocks mazes. Bees hatch from eggs contained in some ice blocks; the destruction of an ice block containing eggs results in the destruction of these eggs. In Pengo, bees can be killed with a sliding ice block and by a collision with the penguin when they are stunned. Pushing a side of the game when a bee is next to this side shall stun the bee for some short time. The player is given 60 seconds to kill all the bees of any of the 16 levels of the game (after the sixteenth level, the game cycles back to the first level). Bees accelerate when reaching the 60 seconds time limit and the player is given a short extra time to kill them before they vanish, thus making the level impossible to finish. There also are pushable diamonds which cannot be collected but must be aligned to score extra points. As in any arcade game of the time, there are many bonuses, making the high score a complex goal; and both the killing of the bees and the alignment of the diamonds score differently according to the situation. We refer the reader to [9] for further details about the game.

Figure 1. An Iceblox game in progress.

Iceblox obviously is easier than Pengo: no time constraint, uniform speed and scoring, no bonuses,... Main targets (coins) fixed rather than moving (bees or flames), flames cannot push ice blocks and fighting them is optional,... However, the locking of flames in a closed maze of ice blocks and rocks possess a geometrical spirit similar to that of the alignment of diamonds in the Pengo game.

Consequently, we chose Iceblox because it is a simpler start in the world of arcade video games than the commercial games of the time.

[1] MACCLIA, CREC Saint Cyr, Écoles de Coëtquidan, F-56381 GUER Cedex, email:{olivier.bartheye,eric.jacopin}@st-cyr.terre-defense.gouv.fr

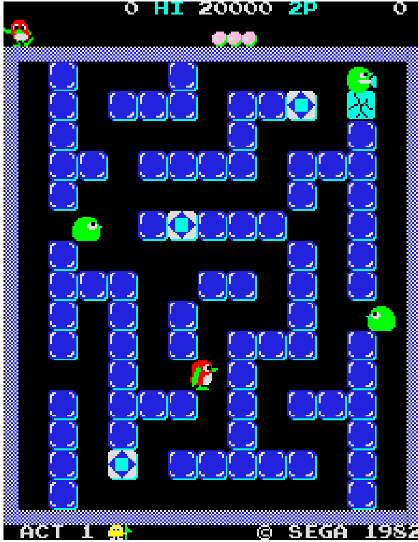

Figure 2. A Pengo game in progress [9].

2 REQUIREMENTS

Game playing On its way to collect coins, the planning system playing the Iceblox game shall badly either fight or avoid flames: disorganized movements or paths longer than necessary shall be allowed as long as they do not prevent the penguin from collecting coins. Both the fighting and avoidance of flames shall be realized in (near) real time: game animation might look sometimes slow or sometimes irregular but shall never stop; in particular, flames shall always be patrolling the maze, even when traditional problem solving (that is, Planning) shall take place.

Plans (what's in a Plan?) A Plan shall be a set of partially ordered operators which represent actions in the Iceblox domain. What kind of actions shall we allow to be part of a plan?

Let us distinguish five kinds of actions that may happen in an arcade video game such as (Pengo or) Iceblox:

1. Drawing a frame to animate a sprite; this is the pixel kind of action. Animation always takes place, even if the player does nothing: for instance, a flame is always animated in the Iceblox game, even in the case where it cannot move because it is surrounded by ice blocks.
2. Basic moving of one step left, right, up or down and pushing an ice block. This is the sprite kind of action which is uninterruptible for a certain number of pixels, usually the number of pixels of a side of the rectangle where the sprite is drawn (in the case of Iceblox, sprites are squares of 30 by 30 pixels). Each action of this kind exactly corresponds to a key pressed by the player: arrows keys to move left, right, up and down and the space bar to push an ice block.
3. Moving horizontally or vertically in a continuous manner, that is making several consecutive basic moves in the same direction; this is the path-oriented kind of action and means following a safe path in the maze.
4. Avoiding, fleeing, fighting or locking the flames in a closed maze of ice blocks and rocks. This is the problem solving kind of action and concerns survival and basic scoring. Ice block cracking until destruction to collect a coin also is a problem solving kind of action.
5. Defining goals and setting priorities on them; this is the game level strategy kind of action and is oriented towards finishing the game with the highest score. In the case of Iceblox, this means ordering the collection of coins, noticing the opportunity to lock the flames in a maze and setting its importance in finishing the level; or else seizing the opportunity to seek cover behind ice blocks of a geometrical configuration created during game playing.

Out of the five kinds of action, only push actions of kind 2, abstract Moves based on rectilinear moves of kind 3 and the coin collection action of kind 4 shall be represented as operators.

The Plans just described are *not* as simple as they may appear: it is *not* the case that either their construction shall be too simple or their use shall be redundant with the player's actions. First, these Plans look like the plans for the storage domain which is part of the deterministic IPC benchmarks [4]. Second, as Table 1 shows, current planners agree with their non easiness. Third, there are Plans (both constructed and executed [8] or only executed [7]) in commercial video-games which are shorter and simpler.

Plan execution The Plan execution system receives a plan from the Planning system in order to execute it in the Iceblox game. Execution of a Plan means executing the players actions (that is, actions of kind 2) corresponding to the operators of the plan, in an order compliant with the partial order of the plan. For instance, the Plan execution system shall decide of several basic moves actions to execute a Move operator in the Plan; which can include ice block pushing to facilitate the realization of the Move operator. The Plan execution system shall also decide of taking advantage of the current Iceblox situation to avoid pushing an ice block when a flame is no longer aligned with it, to choose a new weapon or to avoid unexpected flames. These decision shall result from a local analysis and no global situation assessment shall be realized to take such advantages. The plan execution system shall eventually warn the user when it terminates (whatever the outcome, success or failure). The player can terminate the Plan execution at any time by pressing any arrow key or else the space bar.

In case of emergency situations (e.g. no weapon is locally available, fleeing seems locally impossible, etc), the Plan execution system shall call for re-Planning.

Planning Planning refers to the plan formation activity. The player shall intentionally start the planning activity by pressing a designated key (e.g. the “p” key). Once running, planning shall not stop the Iceblox game: flames shall patrol the maze and sliding ice blocks shall continue to slide while the plan formation activity is running. The user shall be warned when the activity ends, either with success or else failure (e.g. a different sound for each case). If any of the arrow keys or else the space bar (push) is pressed before the end of the planning activity, then it is terminated. The user shall be able to play at any time.

Planners The main objective of the work reported in this paper is to determine whether any of today's planners is able to play the Iceblox game: no planner shall be specifically designed to play Iceblox and no existing planner shall be tweaked to play Iceblox.

Due to the on-going effort of the International Planning Competition (IPC) [4], many planners are available today and most of them

accept Planning data (i.e. states, operators) written in the PDDL language. Because of this wide acceptance, the overhead of both generation and processing of PDDL shall first be ignored; writing directly to a planner's data structures shall be considered only if game playing requirements are not met. Consequently, the Planning activity in the Iceblox domain shall take as input an initial state, a final state and a set of operators all written in the PDDL language.

Available planners are in general more ready to process PDDL text files than ready to be connected to a video game. We shall thus begin with the design of a set of Iceblox planning problems of increasing difficulty, in the spirit of the IPC: executable files for the planners, PDDL Iceblox problems files and scripts files to automate this Iceblox benchmarking process. If a planner fails these off line tests then it shall not be a good candidate for Iceblox video game playing.

What shall demonstrate that a planner is a good candidate for a connection to Iceblox? Solving the Iceblox benchmarks fast enough seems the obvious answer. Rather, the question should be: what is the time limit beyond which the current Iceblox situation is so dangerous that an action has to be taken right away, thus changing the initial state of the planning problem and consequently asking for replanning? Iceblox has a good playability when the frame rate is about 30 frames per second, that is, when the flames coordinates (in pixel) are updated about 30 times per second. Luckily, the sprites are 30 by 30 pixels; it then takes about 1 second for a flame to move from one crossroad to another. According to the Iceblox code, a flame gets a random new direction between 1 and 4 crossroads, while keeping in mind that the new direction at the fourth crossroad might just be the same than the previous one. We can then consider that the limit is when the penguin is 4 crossroads away from a flame, which gives a plan search runtime of at most 4 seconds. If a plan has been found then it needs to be executed, which undoubtedly takes time to trigger. Consequently, the flame should not reach the fourth crossroad and since movement between two crossroads is uninterruptible, the flame should not reach the third crossroad before the plan search ends, which gives us a time limit of 3 seconds.

A planner shall be a good candidate for Iceblox video-game playing if it can (off line) solve Iceblox planning problems within the time limit of 3 seconds. This time limit sorts out dangerous flames from harmless flames.

3 MINIMAL ICEBLOX SITUATIONS

Three examples We here describe three Iceblox planning problems of increasing difficulty which we designed in order to get a set of good candidate planners. Both plan length and planning problem size shall be used as criteria of difficulty: the larger number of predicates describing both the initial and the final states and the larger number of operators in the plan solution, the more difficult the problem.

It is necessary to collect a coin in each of the three problems. Figures 3, 4 and 5 contain an Iceblox screen shot illustrating the problem and its PDDL code. See Figure 3 for the simplest of our three problem; this is obviously a welcome-to-the-Iceblox-world problem. In Figure 4, we introduce danger from one flame with only one weapon to kill this flame. Finally, in Figure 5 we provide the case for two weapons to kill a flame guarding a coin. Let us sum up the properties of these three problems:

See Figure	3	4	5
Requires flame fighting?	No	Yes	Yes
Number of weapons	0	1	2
Number of predicates (initial and final states)	7	10	13
Number of paths leading to the coin	1	1	2
Length of solution plan	2	4	4 or 5

Why these three problems? First because their number of predicates all are very small and yet they cover the basics of Iceblox game playing: a planner failing at these problems shall not be able to play iceblox. Moreover, although more problems have been designed, they confirm the results of Table 1. Given as Iceblox levels to an Iceblox planning system, these problems can be solved in real time; screen shots of the final Iceblox situations are gathered in Figure 6.

Planning problems predicates The following predicates are used to describe both the initial and final states of the planning problems of Figures 3, 4 and 5 (crossroad$_{i,j}$ denotes the location at the intersection of line i and column j):

- (at i j): a sprite (penguin, ice block, iced coined, weapon) is at the crossroad$_{i,j}$.
- (extracted i j): the coin at the crossroad$_{i,j}$ has been collected by the player.
- (guard i_1 j_1 i_2 j_2): the flame at the crossroad$_{i_1,j_1}$ guards the coin at the crossroad$_{i_2,j_2}$. The idea of guarding a coin is linked to the 3 seconds time constraint (see the planners requirements): a flame makes dangerous the collection of a coin when it is within a range of 3 crossroads.
- (iced-coin i j): there is an ice block at the crossroad$_{i,j}$ which contains a coin.
- (protected-cell i j): there exists a path towards the crossroad$_{i,j}$. This path is safe: no flame makes this path dangerous.
- (reachable-cell i j): there exists a path towards the crossroad$_{i,j}$; there exists at least one flame putting this path in danger.
- (weapon i_1 j_1 i_2 j_2 i_3 j_3): there exists a weapon at the crossroad$_{i_2,j_2}$; the penguin should push this weapon from the crossroad$_{i_1,j_1}$. The weapon shall stop sliding at the crossroad$_{i_3,j_3}$; this is useful information when an ice block needs more than one push before the final kick at a flame.

Operators use two more predicates:

- (blocked-path i j): there is an ice block on the path to crossroad$_{i,j}$.
- (blocked-by-weapon i_1 j_1 i_2 j_2): the weapon at crossroad$_{i_2,j_2}$ is on the path to crossroad$_{i_1,j_1}$.

Over all the operators we designed, 4 proved to be critical for Iceblox game playing: move-to-crossroad, destroy-weapon, kick-to-kill-guard and extract. We hope their names are self explanatory. There are more (e.g. pushing a block to lock flames in a maze of ice blocks and rocks) but basic Iceblox playing is impossible if you don't get those 4 operators. Due to space limitations, we only give the PDDL code of the last two; these operators are given to the planners for benchmarking and playing:

```
(:action extract
   :parameters (?coinx - coord-i ?coiny - coord-j )
   :precondition (and (protected-cell ?coinx ?coiny)
         (iced-coin ?coinx ?coiny) (at ?coinx ?coiny)
```

```
            (reachable-cell ?coinx ?coiny))
    :effect (and (extracted ?coinx ?coiny)
            (not (iced-coin ?coinx ?coiny))
            (not (protected-cell ?coinx ?coiny))
            (not (reachable-cell ?coinx ?coiny)))))

(:action kick-to-kill-guard
    :parameters
        (?reachablewx - coord-i ?reachablewy - coord-j
        ?weaponx - coord-i ?weapony - coord-j
        ?newweaponx - coord-i ?newweapony - coord-j
        ?guardx - coord-i ?guardy - coord-j
        ?coinx - coord-i ?coiny - coord-j
        ?blockedx - coord-i ?blockedy - coord-j )
        :precondition (and (iced-coin ?coinx ?coiny)
        (at ?reachablewx ?reachablewy)
        (guard ?guardx ?guardy ?coinx ?coiny)
        (weapon ?reachablewx ?reachablewy ?weaponx
                ?weapony ?newweaponx ?newweapony)
        (reachable-cell ?weaponx ?weapony)
        (protected-cell ?weaponx ?weapony))
        :effect (and (at ?weaponx ?weapony)
                (protected-cell ?coinx ?coiny)
        (blocked-by-weapon ?blockedx ?blockedy
                ?newweaponx ?newweapony)
        (reachable-cell ?newweaponx ?newweapony)
        (protected-cell ?newweaponx ?newweapony)
        (not (reachable-cell ?weaponx ?weapony))
        (not (protected-cell ?weaponx ?weapony))
        (not (reachable-cell ?blockedx ?blockedy))
        (not (guard ?guardx ?guardy ?coinx ?coiny))
        (not (weapon ?reachablewx ?reachablewy
                ?weaponx ?weapony
                ?newweaponx ?newweapony))))
```

These operators can be much simpler and indeed we designed and used very simple versions of them. But of course, there is a compromise between simplicity which questions the overall utility of all this, and complexity which gets the planners nowhere. Believe us, the easiest thing to do is to put more information in the operators than necessary; and then: bye bye runtimes!

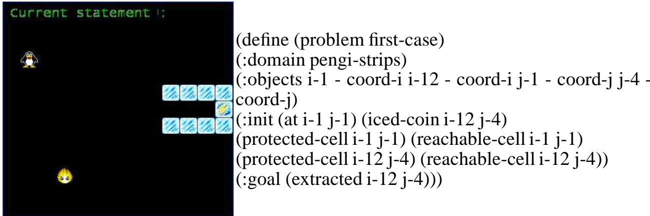

Figure 3. The danger is away, the path is obvious: change job if you don't solve this one.

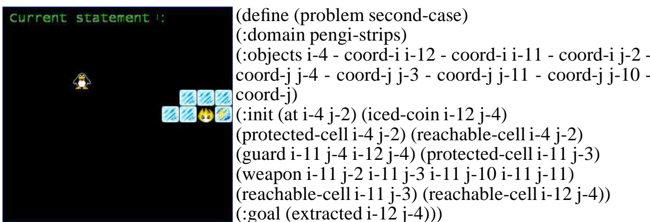

Figure 4. There is danger, but the fight is easy.

Results Several planners have been tested against these three problems (and others), with a 3.2 GHz Pentium 4 PC, loaded with 1Gb of memory. The runtimes, in seconds, averaged over ten runs, are gathered in Table 1. The planners appearing in Table 1 reflect all the cases which happened during the tests: some planners rejected part or all of the PDDL files, reporting PDDL mistakes and some planners found no solution to the problems. Hopefully, others correctly

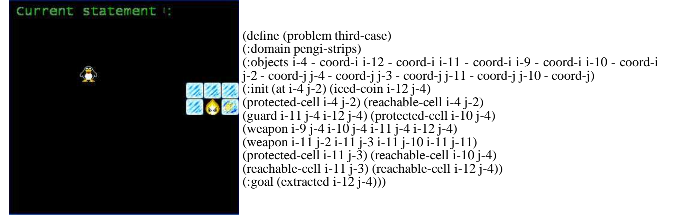

Figure 5. Two unsafe paths: one shorter, one longer. . .

Figure 6. When off line benchmarking becomes real time Iceblox playing. The red and white track materialize the path followed by the penguin.

read the PDDL files; of course, among those planners, some found solutions within the time limit and others did not. All the solutions found were correct (that is, no planner we tested returned an incorrect Plan).

Four planners, FF, Metric-FF, Qweak and SGPlan compose the set of good candidates for playing Iceblox. We eventually reduced the set to the well known and successful FF [2] and the less known and fastest but exotic Qweak [6].

Again, *no* planner was tweaked during both benchmarking and game playing (see the requirements for planners).

	Typed			Untyped		
	Fig. 3	Fig. 4	Fig. 5	Fig. 3	Fig. 4	Fig. 5
FF	0.01	0.49	1.83	0.01	> 120	
HSP2	PDDL Syntax errors					
Metric-FF	0.01	0.33	1.07	0.01	> 120	
Qweak	0.01	0.02	0.02	0.02	0.02	0.02
SGPlan	0.01	0.35	1.14	0.01	> 120	
STAN	0.18	PDDL Syntax errors		0.2	> 120	
TSGP	0.03	25	> 120	0.05	> 120	
YAHSP	0.01	No Plan found		0.01	No Plan found	

Table 1. Performances of several planners on PDDL files of the 3 problems of Figures 4, 5 and 6. A typed and untyped coordinates version was tested for each problem. All times, averaged over ten runs, are in seconds; planners were given 120 seconds to search for a solution although the required time limit is 3 seconds (see the requirements for planners).

4 GAME PLAYING ARCHITECTURE

There eventually are efficiency requirements to achieve the "in near real time" game playing requirement; something we clearly missed in our requirements. After several unsatisfactory prototypes, we implemented Iceblox in C++ using Microsoft's DirectX graphical API. This turned out to be an easy way to achieve real time playability on large game levels such as Figure 1 (since our goal was *not* to implement a video game *but* a testbed for today's planners). Both FF and Qweak were linked to the C++ code of the Iceblox game;

see Table 2 for the main game loop and Table 3 for the Plan execution process. Table 4 presents how the collection-of-a-single-coin is generated as a PDDL planning problem during game playing. This very fast procedure finds weapons by looking in the four directions around a dangerous flame. Due to space limitation, some procedures have been left aside, such as the path planning computation[2]. On the contrary of the Pengi system, our path planning system tries to produce rectilinear paths (see requirements for plans) and introduces ice block pushing to achieve shorter rectilinear paths.

While the Iceblox game is **not** over **loop**
  If the player presses a key $\in \{\rightarrow, \leftarrow, \uparrow, \downarrow, \sqcup\}$ **then**
    If currently running **then**
      Terminate Plan search task or else Plan execution task
    End if
    Check for collision; update the penguin data structure
    If the player pressed the space bar **then**
      Remember which ice block (maybe none)
        was pushed and in which direction
    End if
  Else if the player presses the "p" key **then**
    If the planning task is not running **then**
      Build a Planning problem and start the planning task
    End if
  End if
  For each sprite **do**
    Decide what to render in the next frame
  End for each
  Build and render the next frame on screen

  Warn the player **when**
    – Plan search task ends with failure
    – Plan execution task starts
    – Plan execution task ends
  End of warnings
End loop

Table 2. The main loop of the game playing system.

Repeat
  If the first operator of the Plan is Move-To-Crossroad(i,j) **then**
    Compute a path from current location to crossroad$_{i,j}$
    Repeat
      Execute one basic move following that path
    Until crossroad$_{i,j}$ is reached
  Else
    If the ice block at crossroad$_{i,j}$ is **not** destroyed
      Execute a push action in direction of crossroad$_{i,j}$
    Else
      Execute a basic move towards crossroad$_{i,j}$
    End if
  End if
  Delete the first operator of the Plan

  When the flame is no longer dangerous
    Ignore it
  When the flame is no longer aligned with the weapon **or**
      an unexpected flame becomes dangerous
    Find a weapon and use it
  When the penguin must leave the computed path
    Remember the current crossroad$_{i,j}$
    Get the penguin back to crossroad$_{i,j}$ as soon as possible
Until the Plan is empty

Table 3. The plan execution process.

[2] The interested reader should find an appropriate path planning algorithm in [5].

Generate PDDL planning problem header
Generate PDDL initial state header
Generate (at i-penguin j-penguin)
Find the (nearest **or** possibly flame-less) iced coin
Generate (iced-coin i-coin j-coin)
If there exists a path to this coin **then**
  Generate (reachable-cell i-coin j-coin)
Else
  Generate (blocked-cell i-coin j-coin)
End if
For each flames **do**
  If this flame is close to the iced coin **then**
    Generate (guard i-flame j-flame i-coin j-coin)
    For each nearest weapon in each of the four direction **do**
      Generate (weapon i-push j-push i-weapon j-weapon
          i-stop j-stop)
      Generate (protected-cell i-weapon j-weapon)
      Generate (reachable-cell i-weapon j-weapon)
    End for each
  End if
End for
Generate PDDL final state header and (extracted i-coin j-coin)

Table 4. The PDDL Planning problem generation for the collection of a single coin.

5 CONCLUSIONS

Both planners play large Iceblox levels well in near real time. Sometimes FF plays in real time while Qweak plays in real time most of the time. Due to the quality of the path planning computation and of the DirectX implementation, the penguin looks pretty rational, at the speed of the flames. Sincerely, we thought many times we'd never make it and the granularity of our Plans is crucial to achieve our goal: we just produce Planning problems with a small number of predicates. We now wish to work on the locking of flames and the handling of multiple flames per coin, before tackling the problem of reaching the high score.

ACKNOWLEDGEMENTS

This work is part of a 3 year project founded by the Fondation Saint-Cyr. Thanks to Rick Alterman, Marc Cavazza, Bernard Conein, Jon Gratch and David Kirsh for helpful discussions; to Saint-Cyr cadets Defacqz, Marty, Pertuisel and Yvinec for their implementation of Iceblox in Adobe's Flash; and to an anonymous reviewer for his constructive review.

REFERENCES

[1] Neil Bartlett, Steve Simkin, and Chris Stranc, *Java Game Programming*, Coriolis Group Books, 1996.
[2] Jorg Hoffmann, 'FF: The Fast-Forward planning system', *AI Magazine 22(3)*, 57–62, (2001).
[3] Karl Hornell. Iceblox. `http://www.javaonthebrain.com/-java/iceblox/`, 1996.
[4] International Planning Competition. `http://ipc.icaps-con-ference.org/`, 1998–2008.
[5] Steven LaValle, *Planning Algorithms*, Cambridge University Press, 2006.
[6] Alexander Nareyek, Robert Fourer, Enrico Giunchiglia, Robert Goldman, Henry Kautz, Jussi Rintanen, and Austin Tate, 'Constraints and AI Planning', *IEEE Intelligent Systems (March/April)*, 62–72, (2005).
[7] John O'Brien, *A Flexible Goal-Based Planning Architecture*, chapter 7.5, 375–383, Charles River Media, 2002.
[8] Jeff Orkin, 'Three States and a Plan: The A.I. of Fear', in *Proceedings of the Game Developper Conference*, p. 17 pages, (2006).
[9] Wikipedia. Pengo (arcade game). `http://www.wikipedia.org/`, 2007.