Optimizing Practical Planning for Game AI

Éric JACOPIN

[1 leave this spacer to make page count accurate]
[2 leave this spacer to make page count accurate]
[3 leave this spacer to make page count accurate]
[4 leave this spacer to make page count accurate]
[5 leave this spacer to make page count accurate]
[6 leave this spacer to make page count accurate]
[7 leave this spacer to make page count accurate]
[8 leave this spacer to make page count accurate]
[9 leave this spacer to make page count accurate]
[10 leave this spacer to make page count accurate]
[11 leave this spacer to make page count accurate]
[12 leave this spacer to make page count accurate]
[13 leave this spacer to make page count accurate]
[14 leave this spacer to make page count accurate]
[15 leave this spacer to make page count accurate]
[16 leave this spacer to make page count accurate]
[17 leave this spacer to make page count accurate]
[18 leave this spacer to make page count accurate]
[19 leave this spacer to make page count accurate]
[20 leave this spacer to make page count accurate]

1 Prologue

The reader is expected to have a basic knowledge of GOAP [Orkin 04], predicate-based state and action representation [Cheng 05], [Ghallab 04, Chapter 2] and of basic search techniques (e.g. breadth-first search) applied to planning [Ghallab 04, Chapter 4].

2 Introduction

Planning generates sequences of actions called plans. Practical planning for game Artificial Intelligence (AI) refers to a planning procedure which fits in the AI budget of a game and supports playability so that Non-Player Characters (NPCs) execute actions from the plans generated by this planning procedure.

Jeff Orkin developed Goal-Oriented Action Planning (GOAP) [Orkin 04] as the first ever implementation of practical planning for the game [FEAR]. GOAP implements practical planning with: (i) actions as C++ classes, (ii) plans as paths in a space of states and (iii) search as path planning in a space of states, applying actions backwardly from the goal state to the initial state; moreover, GOAP introduced actions costs as a search heuristic. Many games used GOAP since 2005 and it is still used today, e.g. [DeusEx3DC], [TombRaider], sometime with forward search, which seems easier to debug, as the most noticeable change.

In this paper, we present how to optimize a GOAP-like planning procedure with actions as text files [Cheng 05] and forward breadth-first search (cf. Section A.2 of [Ghallab 04]) so that it becomes practical planning. Actions as text files allow non-programmers to provide actions iteratively without recompiling the game project: non-programmers can modify and update the action files during game development and debugging. That is, planning ideas can be developed and validated offline. Moreover, if needed, C++ can always be generated from the text files and included in the code of your planner at any time during game development. Forward breadth-first search is one of the simplest search algorithm: it is easy to understand. It needs not building extra data structure prior to search which begins immediately: short plans shall be found faster and with less memory than other appealing plan-graph-based planning procedures [Ghallab 04, Chapter 6]. And it is also a complete search procedure which returns the shortest plans: NPCs won't get redundant or useless actions to execute.

This paper is organized as follows. The next section presents the necessary steps before going into any optimization campaign, with examples specific to practical planning. The following section described what can be optimized in practical planning. Finally, we detail practical planning data structures which lead to both runtime and memory footprint improvements.

3 How can you optimize?

Consider a function with various parameters and a set of criteria which tells you when the values returned by this function can be qualified as the best returned values. Then, optimizing the function means looking for the values of the parameters which gets you the function's best values.

The function we here consider is practical planning, and time and memory are our optimization criteria: the lower the runtime and memory footprint of practical planning, the better.

3.1 Measure it!

Your first step is to get some code in order to measure both time and memory usage.

The objective here is to instrument your practical planning code easily and quickly to show improvements on runtime and memory usage and respect of the allowed budgets.

Runtime measurement is not easy as it sounds. It is such that several measures in the same testing conditions rarely give the exact same value. So the most important here is probably to decide for a unit of time: several measures in the same testing conditions should have enough significant digits, and their numerical value should be very close. If you're going to improve runtime by 2 orders of magnitude, you need to start with at least 4 significant digits. C++11 provides the flexible `std::chrono` library [Josuttis 13] which should fit most of your needs but any platform-specific library providing a reliable high resolution counter should do the job. Microsecond seems a good start.

With no (unpredictable) memory leak, memory measures are stable and must return the exact same value in the same testing conditions. The first step here is to measure memory overhead; for instance, an empty `std::vector` takes 16 bytes with Microsoft's Visual C++ 2013 while an empty `std::valarray` takes only 8 bytes if you can use it instead (they only store numeric values, they cannot grow but they can be resized); and there's no memory overhead for an instance of `std::array`, again if you can use it (they are C-Style arrays: their size is fixed). The second step is to decide whether any measure is at all relevant; for instance, do you want to count distinct structures or the whole memory page which was allocated to store these structures?

Finally, you'll have to decide between using conditional compiling to switch on and off the call to measures, assuming the linker shall not include the unnecessary measurement code when switched off, or a specific version of your practical planner that shall have to be synchronized with further updated versions of the planner.

3.2 Design valuable tests!

The second step is to design a set of planning problems.

A first set of planning problems is necessary in order to attest the planner generates correct plans; that is, plans that are solutions to the planning problems of your game. If your planner uses a complete search procedure such as breadth-first search, the correct plans shall also be the shortest ones. By running your planner against these gaming problems your objective is to show this planner can be used in your game. Do not consider only planning problems related to your game, because these problems certainly are too small to stress the full power of a GOAP-like planner: on one hand, a GOAP-like planner generates less than one plan per second per NPC on average and on another hand, these plans are very short, say, at most 4 actions. Consequently, a second set of stressing planning problems is needed to show any improvement in the optimization process. Runtime for such stressing problems can be up to several minutes, whereas in-game planning runtime is at most several milliseconds. There are two kinds of stressing problems: scaling problems and competition problems.

Firstly, scaling problems provide an increasing number of one specific game object: box, creature, location, vehicle, weapon, and so forth. Solutions plans to scaling problems can be short and plan length is expected to be the same for all of the scaling problems; the idea is to provide more objects of one kind than would ever happen in a gaming situation so that the branching factor in the search space explodes although the solution is the same. For instance, and this is valid for a forward state space GOAP-like procedure, take an in-game planning problem such that the goal situation involves only one box; assume that this box, say `box-1`, has to be picked up at one location and moved to another location. Then build an initial situation with increasing number of boxes: `box-2`, `box-3`, and so on, although `box-1` is still the only box which appears in the goal situation. As the planner can pick up one box, it shall try to pick up and then move each box until the goal situation, which only requires `box-1`, is reached.

Secondly, competition problems are stressing problems whose objective is to swallow a lot of computing resources with the help of a high branching factor and a solution with a long sequence of actions [IPC 14]. For instance, it is now time to move several boxes to the final location: allow for picking up and moving only one box at a time and do not impose priorities between boxes. There are consequently many solutions of the exact same length. Each of these solutions reflects the order in which the boxes reach the goal location, thus entailing a huge search space.

Of course, if any updated version of the planning code shows a performance decrease against these planning problems, then this update is not an improvement.

3.3 Use profilers!

There is no way to escape the use of profilers.

With the first two steps, you are able to show your practical planner is usable for your game and that your latest code update is either an improvement or else a bad idea. But how are you going to improve more, avoid bad ideas, or discover unexpected and eventually fruitful paths? Runtime and memory profilers are here to help.

On the contrary to the quick and easy first two steps which both are matters of days, this third step is a matter of weeks and months. Either you have access to business tools (I use Intel® VTunes™ and IBM® Rational® Purify Plus which both allow to profile source code and binaries) and then this, as usual, begins with learning to master these tools (if you can afford training, then do not hesitate); or else you'll want to develop specific in-game profiling [Rabin 00] [Lung 11], and this definitively takes time (at the obvious advantage of mastering every part of it).

By reporting where the computing resources go, these tools tell you where to focus your improvement effort and this is invaluable. If, on this improvement path, you reach a point where no part of your code seems to stand out as a candidate for improvement, then, beyond changing the profiling scheme and counters and running the tests again, it might be time to settle down and think of your planning data structures and your planning algorithms.

4 Practical planning data structures

From a far viewpoint, anything can be optimized, from parsing the actions text files to the data structure holding the solution to the planning problem. There is also the planning domain (how actions are encoded) and the planning problems (how states, either initial or goal, are encoded) using properties of your game world [Cheng 05] (pp. 342–343). For instance, you may not wish to represent the details of navigation from one location to another (e.g. navigating into a building, unlocking and opening doors, avoiding obstacles) or the necessary actions to solve a puzzle in the planning problem; instead, encode only one action which magically reaches the goal location or magically solves the puzzle; then, delegate the execution of this action to a specific routine. Our focus here is different.

From a closer viewpoint, we learn from computational complexity that time complexity cannot be strictly less than space complexity [Garey 79, p. 170]; that is, at best, the computation time shall grow (with respect to a given parameter, e.g. the number of predicates) as the amount of memory needed for this computation, but never less. Consequently, you can design better algorithms to achieve better runtimes, but you can also design better data structures and start with shrinking memory: use as less memory as you can and use it as best as you can [Rabin 11].

First make sure to choose the smallest structure that supports the features you need. Then, avoid multiple copies of these structures by storing information only once and share it everywhere it is needed [Noble 01, pp. 182-190]. For instance, the actions text files may contain several occurrences of the same identifier; when reading the first occurrence of this identifier, push it in an `std::vector` and share its position:

```
std::vector<Identifier> theIdentifiers;

size_t AddIdentifier(Identifier& id)
{
    size_t position = theIdentifiers.size();
    theIdentifiers.push_back(id);
    return position;
}
```

Of course, the next occurrence of `id` in the text file must not be pushed at the back of `theIdentifiers` but must be retrieved in `theIdentifiers` in order to share its position. So you may want instead to hash identifiers in an `std::unordered_map`, storing an integer value at the hashed position, and increment this value each time a new identifier is added to the table:

```
std::unordered_map<Identifier, size_t> theIdentifiers;

size_t AddIdentifier(Identifier& id)
{
    size_t position = theIdentifiers.size();
    theIdentifiers[id] = position;
    return position;
}
```

Lookup and sharing is then achieved through an `iterator`:

```
size_t shared_position;
std::unordered_map<Identifier, size_t>::iterator it;

it = theIdentifiers.find(id);
if (theIdentifiers.end() == it)
   shared_position = AddIdentifier(id);
else
  shared_position = it->second;
```

When the parsing of the action text file ends, we know exactly how many distinct identifiers are in the action text file and thus can allocate an array and move the identifiers from the hash table to their position in the array. More space can be saved as soon as we know the number of distinct identifiers: that is, use an `unsigned char`, instead of `size_t` in the code above, to share the positions when there are less than 256 identifiers for your planning problems, and so on.

Finally, consider a custom memory allocator (even for the STL [Isensee 03]) to access memory quicker than the classical memory allocation routines (e.g. `malloc`). Several high performance memory allocators are available [Berger 14], [Lazarov 08], [Lea 12], [Masmano 08] with various licensing schemes. Try them or any other one before embarking into developing your own.

Assuming that predicates can share their position to other planning data structures, the rest of this section discusses the use of the sharing pattern [Noble 01, pp. 182-190] to actions, plans and states.

4.1 Actions

An action is made of two sets of predicates, in the spirit of IF/THEN rules [Wilhelm 08]: the set of preconditions predicates and the set of post-conditions predicates (effects). A predicate can only occur in both sets if and only if it is negative (prefixed by `not`) in one set and positive in the other set. For instance, the positive predicate `hold(gun)` can appear as a precondition of the action `Drop` if the negative predicate `not(hold(gun))` is one of its effects; accordingly (that is, symmetrically), `hold(gun)` can be an effect of the action `Take` if `not(hold(gun))` is one of its preconditions.

Assume an array of shared positions of predicates, ranging from 0 to (p-1). Then the action predicates can be ordered in the array so that they occur only once:

- Let a, b, c, d, e and p such that $0 \leq a \leq b \leq c \leq d \leq e \leq (p-1)$,
- [0, a-1] is the range of positive preconditions,
- [a, b-1] is the range of positive preconditions which occur as negative effects,
- [b, c-1] is the range of negative effects,
- [c, d-1] is the range of positive effects,
- [d, e-1] is the range of positive effects which occur as negative preconditions,
- [e, p-1] is the range of negative preconditions.

Consequently preconditions are in the range [0, b-1] and in the range [d, p-1], and effects are in the range [a, e-1] as illustrated in the following figure:

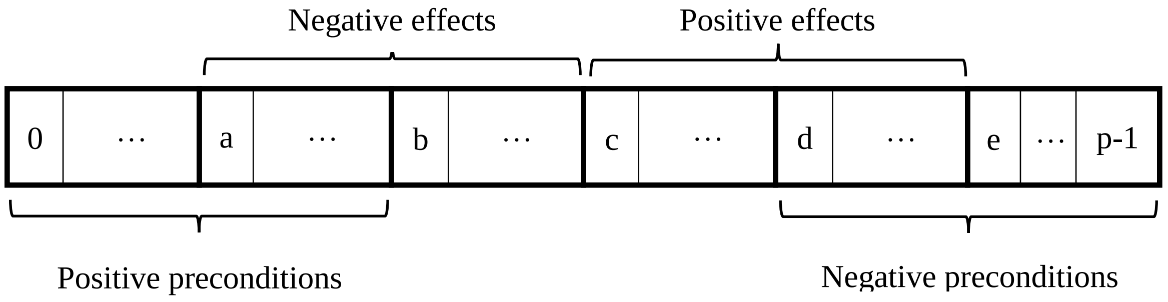


Figure 1 Action predicates occur only once when preconditions and effects overlap.

For instance, the positive predicate `hold(gun)` of the action `Drop` shall occur in the range [a, b-1] whereas it shall occur in the range [d, e-1] for the action `Take` (which is detailed in section **5.1**).

Moreover, an action has parameters which are shared among its predicates; for instance, both the actions `Drop` and `Take` have an `object` as a parameter for the predicate `hold`. If we further constrain all predicates parameters to be gathered in the set of parameters of an action then all predicates parameters only need to be integer values pointing to the positions of the parameters of their action; the type `unsigned char` can be used for these integer values if we limit the number of parameters to 256, which is safe.

4.2 Plans

A GOAP-like plan is a totally ordered set of actions. An action is uniquely represented by action identifier, e.g. `Drop`, and its parameters (once they all have a value), the so-called action signature, e.g. `Drop(gun)`. An action signature is a good candidate for sharing its position in an array. Consequently, a plan is an array of shared positions of action signatures.

As plans grow along search, using an `std::vector` is practical, while keeping in mind the memory overhead for `std::vector`s. Assume the 16 bytes of Visual C++ 2013, at most 256 action signatures, and a maximum plan length of 4 actions [FEAR]: `std::array<unsigned char,4>` (4 bytes) can safely replace `std::vector<unsigned char>` (at least 16 bytes and at most 20 bytes for plans of length 4). Assuming 65535 action signatures and a maximum plan length of 8 actions, then `std::array<short,8>` still saves memory over `std::vector<short>`.

4.3 States

The planning problem defines the initial and the goal states. Forward breadth-first search applies actions to states in order to produce new states, and hopefully the goal state. Indeed, if the preconditions of an action, say `A`, are satisfied in a state, say `s`, then the resulting state, say `r`, is obtained by first applying set-difference (–) with the negated effects of `A` and second to `s`, and then by applying set-union (+) with the positive effects of `A`: r = (s – (negative effects of `A`)) + (positive effects of `A`).

States are made of predicates, and as for actions, it is obvious to substitute predicates by their shared positions.

Set operations can be easily implemented with the member operations of `std::set` or `std::unordered_set`. The memory overhead of these STL containers is of 12 bytes for the former and 40 bytes for the latter with Visual C++ 2013; if you want to store all generated states in order to check whether the resulting state r has previously been generated, then a thousand states means 12 Kb or else 40 Kb of memory overhead.

Set operations can also be implemented with bitwise operations where one bit is set to 1 if the predicate belong to the state and set to 0 otherwise. `std::bitset` provides the bitwise machinery and a thousand states over (at most) 256 predicates then means 32 Kb.

Runtime measurement can help you make the final decision but combining the two representations provides an interesting tradeoff. For instance, you may want to use an `std::array`, which has no memory overhead, to represent a state and convert it to an `std::bitset` when computing the resulting state; then, convert the resulting state back to an `std:array`. In this case, 10 shared positions of predicates in a state on average means 10 Kb for a thousand states which is less than 12 Kb, while 32 Kb would allow for the storing of more than three thousands states.

5 Practical planning algorithms

Runtime profiling should report at least the two following hot spots for forward breadth-first search with actions as text files: (i) checking which predicates of a given state can satisfy the preconditions of an action, and (ii) unifying the precondition predicates of an action with predicates of a given state in order to assign a value to each parameter of this action (consequently, we can compute the resulting state).

5.1 Iterating over subsets of state predicates

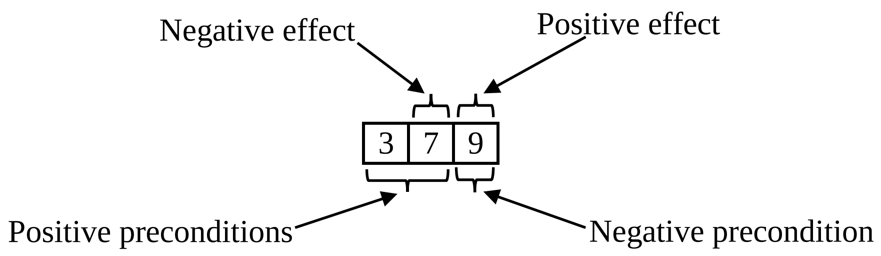
First, consider the following description for action `Take`, stated with simple logic expressions (conjunction and negation) and written with keywords (`:action`, `:parameters`, `:preconditions`, `:effects`), a query mark to prefix variable identifier, and parenthesis, to ease parsing and the reading of the action:

```
(:action Take
    :parameters (location ?l, creature ?c, object ?o)
    :preconditions (not(hold(?object, ?c))
                        and at-c(?l, ?c)
                        and at-o(?l, ?o))
    :effects (not(at-o(?l, ?o)) and hold(?object, ?c))
)
```

Assume the shared positions of the predicates of the action `Take` are the following:

Predicate	**Shared position**
`at-c(?l, ?c)`	3
`at-o(?l, ?o)`	7
`hold(?object, ?c)`	9

If predicates are shared in the order they are read from the actions file, the values of the shared positions above means `at-c(?l, ?c)` is the third predicate which was read from the file, while `at-o(?l, ?o)` and `hold(?object, ?c)` were the seventh and ninth, respectively. With these shared positions, the array (cf. Figure 1) representing the 3 predicates of the action `Take` is the following:



This array means the following values: a = 1, b = c = d = 2, and e = p = 3.

Second, consider the following initial state:

```
(:initial (at-c(loc1, c1) and at-c(loc1, c2)
           and at-c(loc2, c3) and at-c(loc3, c4)
           and at-o(loc1, o1) and at-o(loc1, o3)
           and at-o(loc3, o4) and at-o(loc5, o2)
           and at-o(loc5, o5))
)
```

For instance, the action represented by the action signature `Take(loc1,c1,o1)` can be applied to the initial state because all its preconditions are satisfied in the initial state. But there are 4 more actions which can be applied to this initial state, represented by the four following action signatures: `Take(loc1,c1,o3)`, `Take(loc1,c2,o1)`, `Take(loc1,c2,o3)`, and `Take(loc3,c4,o4)`.

We can firstly remark that each positive precondition identifier must match at least one state predicate identifier. We can secondly note that no state predicate identifier can be a negative precondition identifier. When these two quick tests are passed, we can further test the applicability of an action pushing further the use of the predicate identifiers.

To test the applicability of action `Take` in any state and in particular in the initial state above, we can sort the predicates of the state according to their identifier (cf. Figure 2). It is consequently possible to test only 20 pairs (4 instances of `at-c` × 5 instances of `at-o`) of predicates from the initial states instead of the 36 pairs (choosing any two elements in a set of 9 elements = (9×8)/2 = 36 pairs) which can be built from the initial state.

In the Figure 2 which follows, the top array containing 3 and 7 represents the positive preconditions of action `Take`. We then iterate over the predicates of the initial state. If an initial state predicate has the same predicate identifier than a precondition predicate, then this initial state predicate is pushed at the back of the appropriate column. If the identifier of an initial state predicate does not correspond to any identifier of a precondition predicate, then this initial state predicate is ignored. Finally, iterating in both columns, we can make pairs of predicates as indicated:

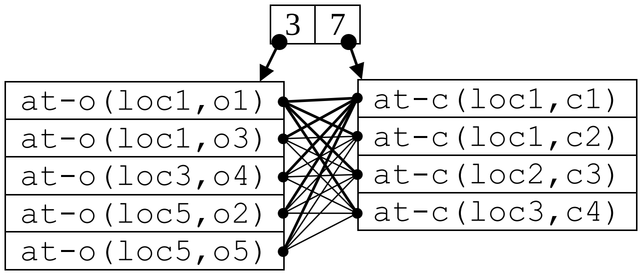


Figure 2 Sorting state predicates with respect to their identifiers.

Note that if the language of the actions file allows predicates with the same identifier but with a different number of parameters, then the number of parameters must also be checked to build both columns above.

There are various ways of achieving the iterations over both columns. For instance, we begin by forming the two-digit number made of the size of above columns: 54. Then we start with the two-digit value 00 and increase the right-most digit; when this value reaches 4 we rewrite the two-digit number to 10 and increase it until the right-most digit reaches 4 again. We then rewrite this number to 20 and so on until 54 is reached. This procedure builds the following list of two-digit numbers: 00, 01, 02, 03, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 40, 41, 42 and 43. Alternatively, we can start from 43 and decrease until 00 is reached, thus producing the two-digit numbers of the previous list in the reverse order. Each of these 20 two-digit numbers can be used to access a position in both columns in order to make a pair of state predicates. This iterating procedure works for any number of positive preconditions:

```
For each predicate of the current state
   Push the predicate back to the list which corresponds to its identifier
End For each;
Make the number n with as many digits as there are non-empty lists;
Set each digit of n to 0;
Repeat
   Access each of the lists with respect to the digits of n and
      make the tuple of predicates of s;
   Increase the least significant digit of n by 1;
   For each digit d of n, starting with the least significant digit,
      If the digit d is equal to the size of the dth list Then
         If d is the most significant digit of n Then
            Break the enclosing For each loop;
         End if;
         Reset digit d to 0;
         Increase digit (d+1) of n by 1
      Else
         Break the enclosing For each loop;
      End if;
   End For each;
Until the value of n is made of the size of the n lists.
```

5.2 Recording where the actions parameters occur in the action predicates

The previous procedure generates tuples such that each state predicate identifier matches the identifier of a precondition predicate. Consequently, the unification procedure need only checking whether the state predicates parameters unify with the action parameters.

We know from Figure 2 that the positive precondition predicate whose shared position is 3, i.e. `at-o(?l,?o)` is unified with state predicate `at-o(loc5,o5)`, then the parameter `?l` of action `Take` gets the value `loc5`, and the parameter `?o` of action `Take` gets the value `o5`. The positive precondition predicate whose shared position is 7, i.e. `at-c(?l,?c)` with the parameter `?l` equal to `loc5`, must now unify with state predicate `at-c(loc3,c4)`. This fails because `loc3` is different from `loc5`.

The idea is to record all the positions where an action parameter occurs in positive preconditions predicates and then check that the parameters at these positions in the state predicates have the same value. For instance, the parameter `?l` of action `Take` occurs as the first parameter of both positive precondition predicates. If the values at these positions in the state predicates are equal (which can be trivially achieved by testing the value of the first position against all other positions), then we can check for the equality of the occurrences of the next parameter. Recording the positions can be achieved once for all when parsing the action files.

6 Conclusion

A Visual C++ 2013 project is available from the book web site which implements a practical planner with the features (that is, actions as text files and forward breadth-first search) and the data structures and algorithms described in this paper.

Although planning is known to be very hard in theory, even the simplest planning algorithm can be implemented in a practical planner which can be used for your gaming purposes, providing you focus on shrinking both memory and runtime. Quick runtime and memory measurement routines, relevant testing and systematic profiling can hopefully lead you to your gaming purposes.

All this requires is perseverance, optimism, and imagination. This is good news, actually.

7 References

[Berger 14] Berger, E. 2014. The Hoard Memory Allocator. http://emeryberger.github.io/Hoard/ (accessed May 26th, 2014).
[Cheng 05] Cheng, J. and Southey, F. 2005. Implementing Practical Planning for Game AI. In *Game Programming Gems 5*, ed. K. Pallister, 329–343. Hingham: Charles River Media.
[DeusEx3DC] Deus Ex Human Revolution – Director's Cut. Square Enix, 2013.
[F.E.A.R.] F.E.A.R. – First Encounter Assault Recon. Vivendi Universal, 2005.
[Garey 79] Garey, M., and Johnson, D. 1979. *Computers and intractability – A Guide to the Theory of NP-Completeness*. New-York: W.H. Freeman & Co Ltd.
[Ghallab 04] Ghallab, M., Nau, D., and Traverso, P. 2004. *Automated Planning – Theory and Practice*. San Francisco: Morgan Kaufmann.
[IPC 14] International Planning Competition, 2014. http://ipc.icaps-conference.org/ (accessed May 28th, 2014).
[Isensee 03] Isensee, P. 2003. Custom STL Allocators. In *Game Programming Gems 3*, ed. D. Treglia , 49–58. Hingham: Charles River Media.
[Josuttis 13] Josuttis, N. 2013. The C++ Standard Library. Upper Saddle River: Pearson Education.
[Lazarov 08] Lazarov, D. 2008. High Performance Heap Allocator. In *Game Programming Gems 7*, ed. S. Jacobs, 15–23. Hingham: Charles River Media.
[Lea 12] Lea, D. 2012. A Memory Allocator (2.8.6). ftp://g.oswego.edu/pub/misc/malloc.c (accessed May 26th, 2014).
[Lung 11] Lung, R. 2011. Design and Implementation of an In-Game Memory Profiler. In *Game Programming Gems 8*, ed. A. Lake, 402–408. Boston: Course Technology.
[Masmano 08] Masmano, M., Ripoli., I., Balbastre, P., and Crespo, A. 2008. A Constant-Time Dynamic Storage Allocator for Realt-Time Systems. *Real-Time Systems*, 40(2): 149-

179.
[Noble 01] Noble, J. and Weir, C. 2001. Small Software Memory – Patterns for systems with limited memory. Harlow: Pearson Education Ltd.
[Orkin 04] Orkin, J. 2004. Applying Goal-Oriented Action Planning to Games. In *AI Game Programming Wisdom 2*, ed. S. Rabin, 217–227. Hingham: Charles River Media.
[Rabin 00] Rabin, S. 2000. Real-Time In-Game Profiling. In *Game Programming Gems*, ed. M. DeLoura, 120–130. Rockland: Charles River Media.
[Rabin 11] Rabin, S. 2011. Game Optimization through the Lens of Memory and Data Access. In *Game Programming Gems 8*, ed. A. Lake, 385–392. Boston: Course Technology.
[TombRaider] Tomb Raider – Definitive Edition. Square Enix, 2013.
[Wilhelm 08] Wilhelm, D. 2008. Practical Logic-Based Planning. In *AI Game Programming Wisdom 4*, ed. S. Rabin, 355–403. Boston: Course Technology.

10 Biography

Éric Jacopin is a professor at the French military academy of Saint-Cyr were he headed the computer science research laboratory from 1998 to 2012; this includes teaching Turing machines, war games and project management, and the management of international internships for computer science cadets. His research has been in the area of Planning for the past 25 years, not only from the viewpoint of artificial intelligence but also from everyday life perspectives. He received his Ph.D. (1993) and his Habilitation to Direct Research (1999) both from the Pierre & Marie Curie University (Paris VI).