

A Comparison of Computer Game Behaviour Control Systems for Background Characters in a Simulated Hospital Environment

Colm Sloan¹ and Brian Mac Namee² and John Kelleher³

Abstract. Utility-based control (UBC) has long been used in robotics but hasn't been widely adopted for use in commercial game AI. Some of the reasons for this are that UBC is presumed that to be: (1) too resource expensive, (2) too difficult to use complex behaviours with, and (3) incapable of scaling to cope well with complex environments. This paper investigates these presumptions by comparing the performance of agents using a UBC system, a finite state machines, and goal-oriented action planners - the latter two control systems are more frequently used in commercial games. These three control systems are compared based on: (1) quantitatively testing their resource requirements in terms of processing cycles and memory; (2) an evaluation of the desirability and complexity of the behaviours they afford and (3) an assessment of how easily each of these control architectures are implemented and extended. The experiments described in this paper were carried out in a virtual hospital environment.

1 INTRODUCTION

The Finite State Machine (FSM) [3] has long been a staple of the commercial video game industry. When simple scripted behaviours like those seen in early games such as Space Invaders weren't enough for the complexities newer games required, developers adopted a more powerful technique from robotics [22][21] - namely, the finite state machine to control the behaviours of computer controlled characters, commonly referred to as non-player characters (NPC). Finite state machines have endured due to their simplicity and power, despite a number of shortcomings. In fact, the finite state machine (and its many variations) are still in surprisingly wide use in the commercial video game industry today[23].

One of the directions in which this migration is taking place is towards the use of planners to control the behaviours of NPCs. Planners are used to create sequences of actions to meet the goals arising from the specific situation in which NPCs find themselves. An advantage of planning is that designers can more easily direct the behaviour of an agent using high-level goals, rather than the low-level instructions required in a finite state machine. Planners took longer to enter the commercial gaming landscape because plan construction was seen as too computationally expensive a process to run under the tight computational constraints of AI in commercial games. However, increases in hardware capabilities have, for the most part, removed this barrier.

Goal-Oriented Action Planning (GOAP) [27] is a form of planning devised with gaming in mind, and so works using few resources compared to planners used for other applications. GOAP uses a regressive cost-based search [25] that allows agents to select the most-efficient sequence of actions to complete a goal. GOAP has been compared to FSMs complexity of implementation and resultant agent behaviour [19] and the resource requirements of GOAP have been touched upon [28] but not compared plainly against other control systems.

The weaknesses of GOAP are that it can only attempt to achieve one goal at a time; is difficult to expand to allow only the partial completion of a goal.

Utility-based control (UBC) systems [2] do not share these weaknesses and are still capable of high-level planning. Rather than focusing on reaching a single goal state, a utility-based agent attempts to maximize a utility function and in this way can handle multiple goals and partial satisfaction of goals. UBCs offer an interesting approach to NPC control in games.

Contribution: The contribution of this paper is a detailed evaluation and comparison of three control systems used to drive the behaviours of NPCs in a game environment: FSMs, GOAP and UBC systems. The focus of this comparison is to determine the feasibility of using advanced planning techniques in commercial games. We analyze the differences between FSMs, GOAP and UBC by: (1) quantitatively testing their resource requirements in terms of processing and memory; (2) an evaluation of the complexity of the behaviours they afford, here emergent behaviours are deemed to be more complex, interesting and desirable; and (3) an assessment of how easily each of these control architectures are implemented and extended.

Overview: Section 2 will briefly describe the three control systems. Section 3 will describe implementations of these control systems. Section 4 will detail the results of the evaluation. Section 5 outlines conclusion of the research and explains the directions in which we intend to take this work in the future.

2 BACKGROUND

A *behaviour control system* directs the actions performed by an agent in a simulation such as an immersive game. Some of the structures that have been used as behaviour control systems in computer and video games are rule-based systems[16], expert systems [13], black-board systems [9], decision trees [30], Markov systems [31], fuzzy-logic based systems [18], hierarchical finite state machines[15][6] and behaviour trees [7]. This paper will focus on FSMs [11] (because these are still very popular within the computer and video game industry), GOAP [24] because it and its variations are being used more

¹ Dublin Institute of Technology, Ireland, email: colm.sloan@student.dit.ie

² Dublin Institute of Technology, Ireland, email: brian.macnamee@dit.ie

³ Dublin Institute of Technology, Ireland, email: john.d.kelleher@dit.ie

and more in computer games, and utility-based control systems because of their potential for computer games. The remainder of this section will detail how each of these control systems works, and review previous work in comparing the performance of behaviour control systems.

2.1 Finite State Machines

A finite state machine is a control system consisting of a predefined collection of states, each of which defines exactly how an agent should act when in that state. An agent may only occupy one state at a time [1]. Finite state machines have proven to be powerful, fast and flexible and many variations of the FSM have been developed within the games industry and academia [8][33][35][14].

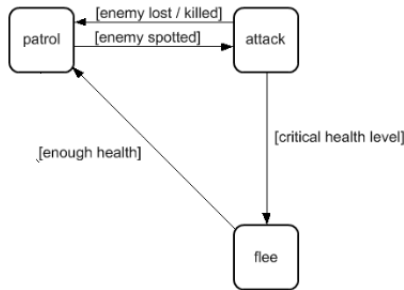


Figure 1. An example of a finite state machine

An example of a finite state machine for a typical guard NPC is shown in Figure 1. The rounded rectangles represent states, the arrows represent transitions between states and the words surrounded by square brackets represent the variable changes that trigger a transition.

The outward behaviours may be the same regardless of the control system being used but the control system changes how the designer will implement these behaviours. When a designer is giving a behaviour to an NPC using a simple finite state machine, the designer must specify every behaviour and every action in a behaviour before the program starts running. A transition to some state can be triggered by the satisfaction of some set of conditions. The NPC then carries out some predetermined fixed sequence of actions, giving the impression of an intelligent behaviour, such as running away from an enemy when health is below a certain level.

Problems arise with finite state machines as they scale up. All transitions between states need to be specified within the code while the FSM is being designed. This means that the designer must know all possible transitions from any state to any other at design-time. It can be very difficult to foresee all possible situations an NPC may find itself in when dealing with complex environments such as those in a commercial game. This means that designers can easily become tangled in a massive switch statement responsible for controlling NPC behaviour. To make a finite state machine work, it has to be programmed to work specifically for a particular environment, making it difficult for state machines to be transferred into a new environment, such as a new game. FSMs have a range of other drawbacks that are well documented [5] including the difficulty in adapting them to cope with goal-directed decisions; the fact that they are not standardized and that they don't handle concurrent behaviours well.

2.2 Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) [24] is a goal-oriented planning system that has been used as a behaviour control system in commercial games [26]. GOAP goes a step further than other behaviour control systems by allowing agents to decide not just what to do but also how it should be done at run time. Agents using GOAP periodically reevaluate their situation and choose the optimal behaviour to achieve their most relevant goal. A goal in GOAP is any condition that an agent wants to satisfy, be it to get a particular item or to kill a particular opponent.

GOAP is based on the early Stanford Research Institute Problem Solver (STRIPS) [10]. Unlike STRIPS, GOAP associates actions with costs, giving the planner the ability to know which actions are preferable. GOAP also has the ability to replan, i.e. the ability to make a new plan when a previous one has failed while holding knowledge of what action caused the plan to fail so that another can be selected.

GOAP agents have predefined goals that are activated by some condition being satisfied, such as seeing an enemy. Whichever goal is selected from a small, predefined list is passed to a plan formulation process to determine the optimal sequence of actions that will achieve the goal. Plan formulation is performed by searching backwards through a sequence of actions from the goal state to the current state using a search algorithm that can take the cost of each action into account, e.g. the A* algorithm [20]. The cost of a GOAP plan is the sum of the cost of all of its actions. The algorithm will return a plan with either the lowest cumulative cost sequence of actions that achieve the goal state, or no plan at all, if none is possible. If an action should fail because one of its conditions cannot be met, the planner will select the next cheapest action that satisfies the same condition in order to reach the goal state.

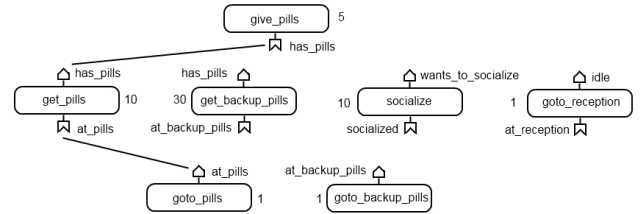


Figure 2. A representation of the GOAP plan formulation process

Figure 2 shows a representation of the GOAP plan formulation process where actions are shown as rounded rectangles, effect as arrowheads and conditions as inverted arrow heads, lines connecting actions show the action sequence taken to satisfy the goal. To satisfy the **give_pills** goal, the planner will have to look through all actions with effects that might be able to satisfy the goal conditions and then attempt to find the conditions needed for those actions.

GOAP is essentially an optimal, heuristically guided planner, like those introduced during the mid 1990s [36]. The only difference between such a planner and GOAP is that GOAP agents refer to a list of previously failed actions while formulating a plan, which itself has been done before in early planning research [32]. GOAP represents these ideas very nicely repackaged and presented for game development. Perhaps the greatest contribution of GOAP to game AI is the elegant separation of what a goal is and how it can be achieved.

One of the downsides of GOAP is that very specific behaviours are very difficult to craft as designers can only work with high-level

logic. This limitation can be overcome by using some scripted behaviours in parallel with GOAP but there's nothing within GOAP that deals with this problem. Another problem with *vanilla* GOAP is that because it starts with the goal and works backwards, there is no way to begin executing the first action in a plan while the other actions are still being calculated [24]. GOAP also only allows for the pursuit of one goal at a time and is difficult to extend for use with partially-satisfying plans, though it is possible [4].

2.3 Utility-Based Control

Utility is a representation of the desirability of a state [34]. A utility-based system attempts to maximize its utility by changing a state through the performance of actions. In a utility-based system, a goal is a function that returns a numeric value used by the agent to choose preferences for certain states or actions. A plan is a *policy* that determines which actions should be chosen in a particular state. It is a function that maps states to actions.

Utility theory enables reasoning over preferences. A utility-based agent [12] can have a set of goals, each of which is represented as a numeric value, usually between 0 and 1 where 1 represents a goal that is completely satisfied and therefore having perfect utility. Each time a utility-based agent is going to make a move, it must first calculate the utility of all of its goals by running a function or functions. When this is done, a *policy* then chooses which actions should be executed for a state. The policy can be made to favor some goals or actions over others. With this, it's possible to have predictable plans forming using a well-tuned policy.

For any state, in basic implementations, all of the applicable actions are checked to see how they would affect the overall utility of the agent if applied. Each action leads to another state. The plan with the largest positive impact on agent utility is selected.

Utility-based control systems can identify plans that only partially satisfy goals. A Markov Decision Process (MDP) [29] is the data structure most commonly used to implement utility-based control systems. However, this was not the structure tested in this research as the MDP proved too unwieldy for the environment and was judged to be too unintuitive for computer game AI programmers.

In order to compare the effectiveness of using the three behaviour control systems described, a game environment was developed populated by NPCs using the three different aforementioned control systems. The next section will describe this game environment.

3 IMPLEMENTATION

To compare the performance of FSMs, GOAP and utility-based control systems at driving the behaviours of NPCs in game environments a test simulation environment has been developed. The gameplay in the simulation takes place within a virtual hospital where NPCs play the roles of nurses, doctors, patients, and visitors. Figure 3 shows a screenshot of this environment.

Nurses care for patients by giving them food and medicine when necessary; visitors talk to patients; doctors occasionally check how things are going; and patients ask for food, medicine and company. The actions executed by the NPCs are governed by either an FSM, GOAP, or utility-based controller. When using an FSM, NPCs perform actions manually specified by the programmer at design time. When using GOAP, NPCs construct plans at run-time. When using UBC, NPCs calculate how a list of plans created at design-time would affect utility and select the plan that optimizes utility. This essentially treats each plan as an action.



Figure 3. A screenshot of the gameplay of the simulation

While this environment isn't quite as demanding as some found in commercial computer games it is sufficient to compare the performance of the three control systems.

The environment was created as a modification of the game *Half-Life 2* (www.half-life2.com). The particular implementation of GOAP used during this evaluation uses GOAP in its simplest form without any enhancements. The deliberation mechanism used for both the FSM and GOAP is a priority list. The algorithm works by checking which is the highest priority goal (or state for the FSM) that needs to be achieved from the predefined list of the agent goals (or states). When using UBC, NPCs calculate how a list of plans created at design-time would affect utility and select the plan that optimizes utility.

4 EVALUATION

In order to evaluate the feasibility of UBC as an NPC control architecture for games we compared its performance versus FSM and GOAP in terms of: (1) the processing and memory requirements of the three control structures, (2) the complexity of the behaviours they afford, and (3) our assessment of how easily each of these control architectures are implemented and extended. Comparing the FSM, GOAP and UBC approaches to control in games based on their processing and memory requirements is justified because these metrics are widely accepted as fundamentally important for real-time computational systems. Our motivation for assessing the complexity of the behaviours these systems can generate is an attempt to capture the potential of each system to generate more diverse gameplay. We believe that this assessment of the control structures is relevant from a game-player's perspective because the more complex the behaviours, the more challenging the gameplay will be, creating greater potential for replayability and hence a more desirable the system. Contrasting with this player's perspective, our analysis of how easily each of the control architectures are to implement and extend can be seen as taking the designer's perspective on the desirability of the architecture as a tool.

NPCs were made to perform a single thinking process with their control system every 0.2 seconds. The measurement concerns only how long it took their control system to finish performing whatever it needed to do to cause a behaviour in an agent. The sum of all processing time for each control system was recorded. The simulation was run five times for each of the control systems. The runs with lowest and highest average time for each type were discarded and the remaining three runs of each type were averaged to produce a single line representing the performance of each type of control

system. Each simulation lasted for four minutes.

4.1 Processing

The computer used for all of the tests described had an Intel Core 2 CPU 6600 at 2.4Ghz, 4096MB of RAM and ran Windows 7 Ultimate 64 bit. Figure 4 shows the processor usage over a fixed time frame for a single NPC using a finite state machine, GOAP and UBC behavior control systems.

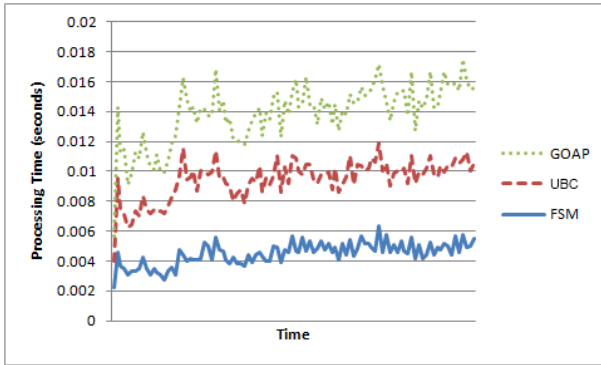


Figure 4. A comparison of processor usage for GOAP, the FSM and the UBC

Figure 4 shows the processor usage for UBC agents, GOAP agents, and agents using FSMs. Each control system was being used by 32 agents. The FSMs used the least processing time and although GOAP used the most processing time because of its A* search, the amount of processing wasn't prohibitive and didn't have any ill effects on the simulation. Since GOAP is the most computationally expensive the simulation runs well and UBC requires less processing time, this shows that UBC is a computationally viable behaviour control system for computer game agents.

4.2 Memory

The memory requirements for the finite state machine and UBC were negligibly small. For GOAP, a database of all goals, actions, and conditions and effects that GOAP will search through must be held in memory. This includes the conditions and effects of each goal and action. Each NPC in the simulation only had a small set of actions to choose from. The data that needs to be stored for each action is quite small. For example, an action may just be a *struct* with an integer for ID and a vector for conditions and effects, though something much more space efficient could be employed. Commercial games using GOAP have only needed a few dozen actions, so the memory footprint of the databases have been very small, as they were with this game. GOAP, using A*, had to keep an open and closed list during plan creation but because of the short plan length and the very limited number of actions examined to find a valid plan, very little memory was needed.

4.3 Behaviour

Following Juul's [17] analysis of online games we distinguish between open and closed games. Open games include emergent elements that introduce variability into the gameplay. Importantly, Juul notes that emergent games "tend to be replayable", an aspect that is desirable from a player's perspective.

There were no emergent behaviours or surprises of any kind observed when the simulation ran with agents controlled using GOAP. The finite state machine also acted exactly as it was scripted to do, as expected. The UBC, on the other hand, proved more dynamic and interesting. During the trials the FSM and GOAP controlled nurse NPCs consistently focused on first giving everyone their medication and then giving them food as these were respectively the two most important duties of the nurse NPCs. In contrast, the UBC controlled nurse NPCs judged whether satisfying the hunger of some patient was better to the overall utility of the nurse and would give food first to a really hungry patient rather than give medication to a patient that isn't in great need of it. This variability in behaviour wasn't possible to implement in GOAP as GOAP only considered the cost of plans rather than the benefit derived from executing a plan. Similarly, it would be very difficult to implement using an FSM. This is a real advantage of using utility-based approaches as they allow easy promotion of emergent behaviours.

4.4 Ease of Use

GOAP is remarkably modular. Everything can be broken down into very comprehensible chunks. The things that motivate behaviour are separated from plans, that are themselves separated from how a plan may be carried out. There is no point at which a designer may feel overwhelmed by the complexity of the behaviour. As a result of this modularity, there is no code replication in planning. The finite state machine and UBC were quite different. They seemed to grow beyond the capacity of what a designer could mentally retain at a discomfortingly fast rate. Each new behaviour added to a daunting collection of variable checks. The simulation only required a few behaviours for each NPC, but even in this case the readability of the code systems plummeted quickly. Because the order in which things occurred mattered in the finite state machine, great care had to be taken not to disturb the flow of the sequence of checks. For UBC, it was quite difficult to implement a policy that led to believable behaviour, even for this very simple environment, which is a real drawback to widespread implementation of this technique.

GOAP was able to share behaviours across agents. Visitors and nurses were given the low priority goal of socializing. Both used exactly the same planning logic but had different implementations of how they went about fulfilling that goal. Nurses would talk to doctors or other nurses, while visitors would talk to patients. There was no logic duplication for planning. Actions, such as talking, were also shared. GOAP is also flexible enough to allow goals and actions to be shared over different projects [25] because the logic may be domain-independent while the implementation of actions can be domain-specific. Finite state machines, on the other hand, are domain-specific. Moving a finite state machine from one game to another almost certainly won't work without great care.

GOAP extended very easily. The modularity meant that new actions could be added with little thought of consequences. It was difficult to understand how the whole system integrated its parts together at first but once this was established, productivity was greatly enhanced. Though, by the time the use of the system was mastered, most of the behaviours had already been implemented, owing to the very few NPC types in the simulation. However, once GOAP was set up, it was very easy to add behaviours to agents.

When a new behaviour was added to an agent using a UBC, the entire policy had to be tweaked again to get the desired behaviour. In short, the UBC scaled badly while the FSM was reasonable and GOAP scaled very well.

5 CONCLUSIONS AND FUTURE WORK

The goal of this paper was to evaluate the feasibility of using a UBC control architecture for driving the behaviours of NPCs in computer games. This evaluation consisted of a comparison of FSMs, GOAP and UBC control structures in terms of their processing and memory resource requirements, the variability of the behaviours they can dynamically generate, and our assessment of how easily each of these control structures are to implement and extend.

We found that a GOAP control system needs more processing time than a basic FSM or UBC system. However, although GOAP needed more, it still performed well in terms of required processing time. The simulation also established that FSMs and UBCs need very little memory to work while GOAP needed more, but still not a large amount. With respect to the emergent NPC behaviour, we conclude that the UBC structure is the most amenable to the development of emergent behaviours. Finally, as regards ease of use, the time needed to set up the FSM and UBC was initially very small but as more behaviours were added to the UBC, it took longer and longer to alter the policy to give the desired behaviour. By contrast, although GOAP took the longest to implement it was the most modular architecture and most easily extended.

From a game designer's perspective the small processing overhead GOAP needs compared to FSM and UBC systems is well worth the ease of design it gives. Its modularity makes it easier to read and understand and makes it very easy to create new NPC behaviours and alter behaviours to be specialized to specific NPCs and their circumstances. Taking a game player's perspective, the UBC system has the greatest potential to generate emergent behaviours opening the possibility for more challenging and interesting games. If a UBC agent architecture could be made that could scale as well as GOAP we feel that a UBC system would be superior to GOAP and FSMs. In the future we intend to develop such a scalable UBC system. Instead of accessing all plans, agents will access plans regarding objects that are closest to them before assessing others. The notion of resources (e.g. time, energy etc.) will also be used as goals can be expressed in a manner that makes implementing NPCs easier for programmers.

REFERENCES

- [1] D. Abreu and A. Rubinstein, 'The structure of nash equilibrium in repeated games with finite automata', *Econometrica: Journal of the Econometric Society*, 1259–1281, (1988).
- [2] B. Boehm and A. Jain, 'An initial theory of value-based software engineering', *Value-Based Software Engineering*, 15–37, (2006).
- [3] D. M. Bourg and G. Seemann, *AI for Game Developers*, O'Reilly Media, Inc., 2004.
- [4] David Hernandez Cerpa and Julio Obelleiro, *An Advanced Motivation-Driven Planning Architecture*, volume 4, 377 – 381.
- [5] Alex J. Champandard. 10 reasons the age of finite state machines is over, December 2007.
- [6] Alex J. Champandard. The gist of hierarchical fsm, September 2007.
- [7] Alex J. Champandard. Understanding behavior trees, September 2007.
- [8] E. Dybsand, 'A generic fuzzy state machine in c++', *Game Programming Gems*, 2, 337–341.
- [9] R. Englemore and T. Morgan, *Blackboard Systems*, Addison Wesley Publishing Company, 1988.
- [10] R.E. Fikes and N.J. Nilsson, 'STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving', *Artificial Intelligence*, 2(3-4), 189–208, (1971).
- [11] D. Fu and R. Houlette, 'The Ultimate Guide to FSMs in Games', *AI Game Programming Wisdom*, 2, (2003).
- [12] M. Ghallab, D.S. Nau, and P. Traverso, *Automated Planning: theory and practice*, Morgan Kaufmann Publishers, 2004.
- [13] J.C. Giarratano and G. Riley, *Expert Systems: Principles and Programming*, Brooks/Cole Publishing Co. Pacific Grove, CA, USA, 1989.
- [14] Elizabeth Gordon, *A Goal-Based, Multitasking Agent Architecture*, 265–274.
- [15] D. Harel, 'Statecharts: A visual formalism for complex systems', *Science of computer programming*, 8(3), 231–274, (1987).
- [16] F. Hayes-Roth, 'Rule-Based Systems', (1985).
- [17] J. Juul, 'The Open and the Closed: Games of Emergence and Games of Progression', in *Computer Game and Digital Cultures Conference Proceedings*, pp. 323–329, (2002).
- [18] C.C. Lee, 'Fuzzy logic in control systems: Fuzzy logic controller—part I', *IEEE Transactions on systems, man, and cybernetics*, 20(2), 404–418, (1990).
- [19] E. Long, 'Enhanced NPC behaviour using goal oriented action planning'. Citeseer, (2007).
- [20] J. MatthewsGenerations, 'Basic A* Pathfinding Made Simple', *AI Game Programming Wisdom*, 105, (2002).
- [21] G.H. Mealy, 'A method for synthesizing sequential circuits', *Bell System Technical Journal*, 34(5), 1045–1079, (1955).
- [22] E.F. Moore, 'Gedanken-experiments on sequential machines', *Automata studies*, 34, 129–153, (1956).
- [23] A. Nareyek, 'Game AI Is Dead. Long Live Game AI!', *IEEE intelligent Systems*, 9–11, (2007).
- [24] J. Orkin, 'Applying Goal-Oriented Action Planning to Games', *AI Game Programming Wisdom*, 2, 217–228, (2003).
- [25] J. Orkin, 'Symbolic representation of game world state: Toward real-time planning in games', in *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, (2004).
- [26] J. Orkin, 'Three States and a Plan: The AI of F.E.A.R.', in *Proceedings of the 2006 Game Developers Conference*, volume 12, pp. 13–14. Citeseer, (2006).
- [27] Jeff Orkin, *Applying Goal-Oriented Action Planning to Games*, volume 2, 217 – 227.
- [28] D.L. Pittman, 'PRACTICAL DEVELOPMENT OF GOAL-ORIENTED ACTION PLANNING AI'.
- [29] M.L. Puterman, *Markov decision processes: Discrete stochastic dynamic programming*, John Wiley & Sons, Inc. New York, NY, USA, 1994.
- [30] JR Quinlan, 'Induction of decision trees', *Machine learning*, 1(1), 81–106, (1986).
- [31] L.R. Rabiner and B.H. Juang, 'Introduction to hidden Markov models.', *IEEE ASSP MAG.*, 3(1), 4–16, (1986).
- [32] G.A. Reece and A. Tate, *Synthesizing Protection Monitors from Causal Structure*, Artificial Intelligence Applications Institute, University of Edinburgh, 1994.
- [33] G. Rosado, 'Implementing a Data-Driven Finite-State machine', *AI Game Programming Wisdom* 2, 307–318, (2004).
- [34] S.J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, Prentice hall, 2009.
- [35] P. Tozour, 'Stack-Based finite state machines', *AI Game Programming Wisdom* 2, 303–306, (2004).
- [36] M. Williamson and S. Hanks, 'Optimal Planning with a Goal-Directed Utility Model', in *In Proceedings of the Second International Conference of AI Planning Systems*, (1994).