Search in Real-Time Video Games

Peter I. Cowling[1], Michael Buro[2], Michal Bida[3], Adi Botea[4], Bruno Bouzy[5], Martin V. Butz[6], Philip Hingston[7], Héctor Muñoz-Avila[8], Dana Nau[9], and Moshe Sipper[10]

1 **University of York, UK**
peter.cowling@york.ac.uk
2 **University of Alberta, Canada**
mburo@cs.ualberta.ca
3 **Charles University in Prague, Czech Republic**
Michal.Bida@mff.cuni.cz
4 **IBM Research, Dublin, Ireland**
adibotea@ie.ibm.com
5 **Université Paris Descartes, France**
bruno.bouzy@parisdescartes.fr
6 **Eberhard Karls Universität Tübingen, Germany**
butz@informatik.uni-tuebingen.de
7 **Edith Cowan University, Australia**
p.hingston@ecu.edu.au
8 **Lehigh University, USA**
munoz@eecs.lehigh.edu
9 **University of Maryland, USA**
nau@cs.umd.edu
10 **Ben-Gurion University, Israel**
sipper@cs.bgu.ac.il

Abstract

This chapter arises from the discussions of an experienced international group of researchers interested in the potential for creative application of algorithms for searching finite discrete graphs, which have been highly successful in a wide range of application areas, to address a broad range of problems arising in video games. The chapter first summarises the state of the art in search algorithms for games. It then considers the challenges in implementing these algorithms in video games (particularly real time strategy and first-person games) and ways of creating searchable discrete representations of video game decisions (for example as state-action graphs). Finally the chapter looks forward to promising techniques which might bring some of the success achieved in games such as Go and Chess, to real-time video games. For simplicity, we will consider primarily the objective of maximising playing strength, and consider games where this is a challenging task, which results in interesting gameplay.

1998 ACM Subject Classification I.2.8 Problem Solving, Control Methods, and Search

Keywords and phrases search algorithms, real-time video games, Monte Carlo tree search, minimax search, game theory

Digital Object Identifier 10.4230/DFU.Vol6.12191.1

Artificial and Computational Intelligence in Games. *Dagstuhl Follow-Ups*, Volume 6, ISBN 978-3-939897-62-0.
Editors: Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius; pp. 1–19

Dagstuhl Publishing
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany

1 Introduction

Search algorithms have achieved massive success across a very wide range of domains, and particular success in board and card games such as Go, chess, checkers, bridge and poker. In each of these games there is a reasonably well-defined state-action graph (possibly with information sets in the games of imperfect information games such as bridge and poker). The success and generality of search for producing apparently strategic and human-competitive behaviours points to the possibility that search might be a powerful tool in finding strategies in video game AI. This remains true in multiplayer online strategy games where AI players need to consistently make effective decisions to provide player fun, for example in the role of supporting non-player character. Search is already very well embedded in the AI of most video games, with A* pathfinding present in most games, and ideas such as procedural content generation [78] gaining traction.

However, video games provide a new level of challenge when it comes to thinking about the sort of strategic behaviours where search has worked so well in board and card games. Principally this challenge is that the complexity of the naïvely defined state-action graph has both a branching factor and a depth that is orders of magnitude greater than that for even the most complex board games (e.g. Go), since we must make sometimes complex decisions (such as choice of animation and path) for a large number of agents at a rate of anything up to 60 times per second. Currently these problems are overcome using painstakingly hand-designed rule-based systems which may result in rich gameplay, but which scale rather poorly with game complexity and are inflexible when dealing with situations not foreseen by the designer.

In this article, we point towards the possibility that power and ubiquity of search algorithms for card and board games (and a massive number of other applications) might be used to search for strategic behaviours in video games, if only we can find sensible, general ways of abstracting the complexity of states and actions, for example by aggregation and hierarchical ideas. In some ways, we are drawing a parallel with the early work on chess, where it was felt that capturing expert knowledge via rules was likely to be the most effective high-level method. While capturing expert rules is currently the best way to build decision AI in games, we can see a bright future where search may become a powerful tool of choice for video game strategy.

In order to provide a coherent treatment of this wide area, we have focussed on those video games which have the most in common strategically with board and card games, particularly Real Time Strategy (RTS) and to a lesser extent First Person games. For these games a challenging, strong AI which assumes rational play from all players is a goal which is beyond the grasp of current research (although closer than for other video games where speech and emotion are needed), but which would likely be of interest to the games industry while providing a measurable outcome (playing strength) to facilitate research developments.

The paper is structured as follows: in section 2 we consider the current state of the art in relevant research areas, in section 3 we point to some of the research challenges posed by video games, and in section 4 we discuss promising approaches to tackling them. We conclude in section 5. The paper arose from the extensive, wide ranging and frank discussions of a group of thought leaders at the Artificial and Computational Intelligence in Games summit at Schloss Dagstuhl, Germany, in May 2012. We hope it may provide some inspiration and interesting directions for future research.

2 State of the Art

2.1 Search

Alpha Beta Minimax Search

Alpha-beta is a very famous tree search algorithm in computer games. Its history is strongly linked with the successes obtained in Computer Chess between 1950, with the Shannon's original paper [65], and 1997 when Deep Blue surpassed Gary Kasparov, the human world champion [1], [17], or indeed when Shaeffer and co-workers showed that with perfect play the game of Checkers is a draw [63], which is one of the largest computational problems ever solved. Alpha-beta [35] is an enhancement over Minimax which is a fixed-depth tree search algorithm [81]. At fixed depth d, Minimax evaluates nonterminal game positions with a domain-dependent evaluation function. It backs up the minimax values with either a min rule at odd depths or a max rule at even depths, and obtains a minimax value at the root. Minimax explores b^d nodes, where b is the search tree branching factor.

At each node, alpha-beta uses two values, alpha and beta. Without entering into the details, alpha is the current best value found until now, and beta is the maximal value that cannot be surpassed. During the exploration of the branches below a given node, when the value returned by a branch is superior to the beta value of the node, the algorithm safely cuts the other branches in the search tree, and stops the exploration below the node. Alpha-beta keeps minimax optimality [35]. Its efficiency depends mainly on move ordering. Alpha-beta explores at least approximately $2b^{d/2}$ nodes to find the minimax value, and consumes a memory space linear in d. Furthermore, in practice, the efficiency of alpha-beta depends on various enhancements. Transposition tables with Zobrist hashing [86] enables the tree search to reuse results when encountering a node already searched. Iterative deepening [68] iteratively searches at increasing depth enabling the program to approach an any time behaviour [39]. Minimal window search such as MTD(f) [56] uses the fact that many cuts are performed when the alpha-beta window is narrow. Principal variation search assumes that move ordering is right and that the moves of the principal variation can be searched with a minimal window [55]. The history heuristic gathers the results of moves obtained in previous searches and re-use them for dynamical move ordering [62].

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) has revolutionised Computer Go since it was introduced by Coulom [20], Chaslot et al. [19] and Kocsis and Szepesvári [36]. It combines game tree search with Monte Carlo sampling. As for minimax tree search, MCTS builds and searches a partial game tree in order to select high quality moves. However, rather than relying on a domain-dependent evaluation function, Monte Carlo simulations (rollouts) are performed starting from each nonterminal game position to the end of the game, and statistics summarising the outcomes are used to estimate the strength of each position. Another key part of its success is the use of UCT (Upper Confidence Bounds for Trees) to manage the exploration/exploitation trade-off in the search, often resulting in highly asymmetric partial trees, and a large increase in move quality for a given computational cost. The resulting algorithm is an any-time method, where move quality increases with the available time, and which, in its pure form, does not require any a priori domain knowledge. Researchers have been exploring its use in many games (with notable success in general game playing [6]), as well as in other domains.

The basic algorithm can be enhanced in various ways, to different effect in different applications. One common enhancement is the use of transposition tables, and a related

idea: Rapid Action Value Estimation (RAVE) [24], which uses the all-moves-as-first (AMAF) heuristic [13]. Another common enhancement is to use domain knowledge to bias rollouts [9, 25]. This must be used with care, as stronger moves in rollouts do not always lead to better quality move selections. A recent survey [12] provides an extremely thorough summary of the current state of the art in MCTS.

A* Search

A* [29] is one of the most popular search algorithms not only in games (e.g., pathfinding), but also in other single-agent search problems, such as AI planning. It explores a search graph in a best-first manner, growing an exploration area from the root node until a solution is found. Every exploration step expands a node by generating its successors. A* is guided using a heuristic function $h(n)$, which estimates the cost to go from a current node n to a (closest) goal node. Nodes scheduled for expansion, that are kept in a so-called *open list*, are ordered according to a function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from the root node to the current node n. Nodes with a smaller f value are seen as more promising, being considered more likely to belong to a solution with an optimal cost. When h is *admissible* (i.e., $h(n)$ never overestimates the true cost between n and the nearest goal, $h^*(n)$), A* returns optimal solutions.

A* remembers all visited nodes, to be able to prune duplicate states and to reconstruct a solution path to the goal node. Both the memory and the running time can be one important bottlenecks for A*, especially in large search problems. Iterative Deepening A* (IDA*) [38] reduces the memory needs down to an amount linear in the depth of the exploration path, at the price of repeatedly expanding parts of the state space by iterative deepening. Weighted A* [58] can provide a significant speed-up at the price of a bounded solution sub-optimality. There are many other search methods that build on A*, including methods for hierarchical pathfinding, and methods for optimal search on multiple CPU cores.

Multi-Player Search

Less research has been done on game-tree search in multi-player games than in two-player zero-sum games, partly because results are harder to obtain. The original tree-search algorithm for multi-player games, Max-n [47], is based on the game-theoretic backward induction algorithm [54] for computing a *subgame-perfect equilibrium* (in which each player maximizes its return under the assumption of game-theoretic rationality). To this, Max-n adds game-tree pruning and bounded-depth search. Prob-max-n is an extension of Max-n with probabilities [75]. The results on pruning in Max-n are very weak: shallow pruning occurs at depth one, but deeper pruning does not occur except in the last branch, which is not effective when the branching factor is high [40]. Greater pruning can be achieved by replacing the assumption of game-theoretic rationality with a "paranoid" assumption that all of the other players are trying to minimize one's own return. This assumption reduces the multi-player game to a two-player game, on which one can use a two-player game-tree search algorithm [74] to compute a maximin strategy for the multi-player game. In terms of game-playing performance, neither the paranoid algorithm nor the Max-n assumption of game-theoretic rationality have been especially successful in producing strong AI players, and more sophisticated ways are needed to model human interactions in multiplayer games [83]. As in two-player games in which MCTS surpassed alpha-beta, MCTS also surpassed Max-n in multi-player games [73] with enhancements or not [53] and has become the reference algorithm.

2.2 Abstraction

Hierarchical Task Networks (HTNs) and Behaviour Trees

HTN planning is a technique for generating sequences of actions to perform *tasks* (complex activities). In a game environment, these might be activities to be performed by an computer-controlled agent. For each kind of task there are *methods* that provide alternative ways to decompose the task into *subtasks* (simpler tasks). Usually each method contains preconditions or constraints that restrict when it can be used. Planning proceeds by decomposing tasks recursively into simpler and simpler subtasks, until actions are reached that the agent can perform directly. Any time a task t is produced for which none of the available methods is applicable, the planner backtracks to look for other ways to decompose the tasks above t.

Some HTN planners are custom-built for particular application domains. Others (e.g., SIPE-2 [82], O-PLAN [22, 77], and SHOP2 [51]) are *domain-configurable*, i.e., the planning engine is independent of any particular application domain, but the HTN methods in the planner's input are specific to the planning domain at hand. Most HTN planners are built to do *offline* planning, i.e., to generate the entire plan before beginning to execute it. But in video game environments where there are strict time constraints and where the outcomes of the planned actions may depend on many unpredictable factors (e.g., other agents), planning is often done *online* (concurrently with plan execution) (e.g., [44]). To create believable sequences of actions for virtual agents in Killzone 2, an HTN planner generates single-agent plans as if the world were static, and replans continually as the world changes [80, 18].

In the context of game programming, a *behavior tree* is a tree structure that interleaves calls to code executing some gaming behavior (e.g., harvest gold in an RTS game) and the conditions under which each of these code calls should be made. In this sense, they are closer to hierarchical FSMs [32] than to HTN planning. At each node there are lists of pairs $(cond, child)$ where *cond* is a condition to be evaluated in the current state of the game and *child* is either a code call or a pointer to another node. In current practice, behavior trees generally are constructed by hand. In principle, HTN planning could be used to generate behavior trees; but most existing HTN planners incorporate an assumption of a single-agent environment with actions that have deterministic outcomes (in which case the tree is simply a single path). A few HTN planners can reason explicitly about environments that are multi-agent and/or nondeterministic [41, 43, 42], including a well-known application of HTN planning to the game of bridge [52, 70]. Furthermore, the online HTN planning used in some video games can be viewed an on-the-fly generation of a path through a behavior tree.

Temporal Abstraction

The abstraction of time in real-time video games provides unique challenges for the use of approaches which search a directed graph, where the nodes represent states and the arcs represent transitions between states. Note that here we interpret "state" loosely – the states themselves will usually be rather abstract representations of the state of the game, and the transitions will usually only represent a small subset of possible transitions. If we introduce continuous time, then a naive state transition graph has infinite depth, since we may make a transition at any time. There are two primary methods for overcoming this problem: time-slicing and event-based approaches. When using time-slicing we divide time into even segments, and use algorithms which are guaranteed to respond within the given time, or consider applications where a non-response is not problematic. There are many examples in pathfinding (e.g. Björnsson [5]). Another time-slicing approach is given in Powley et al. [60] where macro-actions are used to plan large-scale actions which are then executed frame by

frame, where each transition corresponds to a macro-action executed over several frames. Event-based approaches consider an abstraction of the tree where branching is only possible following some significant event. Zagal and Mateas [85] has further discussion of abstract time in video games.

Level of Detail AI

One of the main challenges of creating high-fidelity game AI is the many levels of behaviour that are needed: low level actions, agent operations, tactics, unit coordination, high-level strategy etc. For example, real-time strategy games require multiple levels of detail. The AI needs to control tasks at different levels including creating and maintaining an economy, planning a high-level strategy, executing the strategy, dealing with contingencies, controlling hundreds of units, among many others. To deal with this problem, so-called managers are often used to take care of different gaming tasks [64]. Managers are programs specialized in controlling one of the tasks in the real-time strategy game such as the economy. In games such as role-playing games and first-person shooters, a lot of CPU and memory resources are spent in the area where the player-controlled character is located whereas little or no CPU time is spent in other areas [10]. This is often referred to as level-of-detail AI. One possible approach is to use AI techniques such as AI planning to generate high-level strategies while using traditional programming techniques such as FSMs to deal with low-level control.

Hierarchical Pathfinding

Pathfinding remains one of the most important search applications in games. Computation speed is crucial in pathfinding, as paths have to be computed in real-time, sometimes with scarce CPU and memory resources. Abstraction is a successful approach to speeding up more traditional methods, such as running A* on a flat, low-level graph representation of a map.

Hierarchical Pathfinding A* (HPA*) [7] decomposes a map into rectangular blocks called clusters. In an abstract path, a move traverses an entire cluster at a time. Abstract moves are refined into low-level moves on demand, with a search restricted to one cluster. Hierarchical Annotated A* (HAA*) [28] extends the idea to units with variable sizes and terrain traversal capabilities. Partial Refinement A* (PRA*) [76] builds a multi-level hierarchical graph by abstracting a clique at level k into a single node at level $k + 1$. After computing an abstract path at a given level, PRA* refines it level by level. A refinement search is restricted to a narrow corridor at the level at hand. Block A* [84] uses a database with all possible obstacle configurations in blocks of a fixed size. Each entry caches optimal traversal distances between any two points on the block boundary. Block A* processes an entire block at a time, instead of exploring the map node by node. Besides abstractions based on gridmaps, triangulation is another successful approach to building a search graph on a map [23].

Dynamic Scripting

Scripts, programs that use game-specific commands to control the game AI, are frequently used because it gives the game flexibility by decoupling the game engine from the program that controls the AI [4]. This allows gamers to modify the NPC's behaviour [59]. Carefully crafted scripts provide the potential for a compelling gaming experience. The flip side of this potential is that scripts provide little flexibility which reduces replayability. To deal with this issue, researchers have proposed dynamic scripting, which uses machine learning techniques to generate new scripts or modify existing ones [71]. Using a feedback mechanism such as

the one used in reinforcement learning, scripts are modified towards improving some signal from the environment (e.g., score of the game).

Learning Action Hierarchies

Various research directions have acknowledged that an hierarchical organization of actions can be highly beneficial for optimizing search, for more effective planning, and even for goal-directed behavioral control in robotics [8]. Psychological research points out that the brain is highly modularly organized, partitioning tasks and representations hierarchically and selectively combining representation on different levels for realizing effective behavioral control [16, 30, 57]. While thus the importance of hierarchies is without question, how such hierarchical representations may be learned robustly and generally is still a hard challenge although some research exists studying this problem [31].

"Black Box" Strategy Selection

An intuitive approach to applying adversarial search to video games is to start off with a collection of action scripts that are capable of playing entire games and then – while playing – select the one executed next by a look-ahead procedure. In [61], for instance, this idea has been applied to base-defense scenarios in which two players could choose among scripts ranging from concentrating forces to spreading out and attacking bases simultaneously. To choose the script to execute in the next time frame, the RTSplan algorithm simulates games for each script pair faster than real-time, fill a payoff matrix with the results, and then solves the simultaneous-move games- in which actions now refer to selecting scripts – using linear programming. Actions are then sampled according to the resulting action probabilities. Equipped with an efficient fast-forwarding script simulator and an opponent modelling module that monitors opponent actions to maintain a set of likely scripts the opponent is currently executing, RTSplan was able to defeat any individual script in its script collection.

3 Challenges

3.1 Search

- **Massive branching factor / depth.** MCTS has been successfully applied to trees with large branching factor and depth in games such as Go or Amazons. However, video games branching factor and depth in generally several order of magnitudes greater. Furthermore, the action space and the time can be continuous leading to much more complexity. MCTS has been studied in continuous action space leading to Hierarchical Optimistic Optimization applied to Trees (HOOT) [48]. HOO [14] is an extension of UCB addressing the case of a continuous set of actions. However, HOOT is a very general approach which is unlikely to work in complex video games.
  Often, the massive branching factor is caused by the high number of characters acting in the game. One character may have a continuous set of actions. But even with a reduced set of actions, i.e. north, west, south, east, wait, the number of characters yields an action space complexity that is finite but that is huge and not tractable with general tools such as HOOT. The problem of the huge action set in video games cannot be dealt with MCTS approach alone. A partial solution could be grouping sibling moves, but it will probably not be sufficient. Some abstraction scheme must be found to divide the

number of actions drastically. There is some promise for macro-action techniques such as [60] but to date these have been considered only in rather simple video games.

The length of simulated video games is the second obstacle. With many characters acting randomly, a mechanism must lead the game to its end. Assuming the game ends, it should do it quickly enough, and the game tree developed should go deeply enough to bring about relevant decisions at the top level. A first solution to encompass the huge depth or the time continuity, is to deal with abstract sequences grouping consecutive moves. A sequence of moves would correspond to a certain level of task with a domain-dependent meaning, e.g. a fight between two armies, or an army moving from a starting area to another. Tree search should consider these tasks as actions in the tree.

- **Automatically finding abstractions.** Learning abstractions has been a recurrent topic in the literature. Works include learning abstraction plans and hierarchical task networks from a collection of plan traces. Abstracted plans represent high-level steps that encompass several concrete steps from the input trace. In a similar vein, learning HTNs enables the automated acquisition of task hierarchies. These hierarchies can capture strategic knowledge to solve problems in the target domain.

  Despite some successes, there are a number of challenges that remain on this topic that are of particular interest in the context of games: existing work usually assumes a symbolic representation based on first-order logic. But RTS games frequently require reasoning with resources, hence the capability to abstract numerical information is needed. In addition, algorithms for automated abstraction need to incorporate spatial analysis whereby suitable abstraction from spatial elements is elicited. For example, if the game-play map includes choke points it will be natural to group activities by regions as separated by those choke points. Also, abstraction algorithms must deal with time considerations because well-laid out game playing strategies frequently consider timing issues. Last but not least, such abstractions need to be made dependent upon the actual behavioural capabilities available in the game. Choke points may thus not necessarily be spatially determined in an Euclidean sense, but may rather be behavior-dependent. For example, a flying agent does not care about bridges, but land units do. Thus, a significant challenge seems to make abstraction dependent on the capabilities of the agent considered. Learning such abstractions automatically clearly remains an open challenge at this point.
- **Capturing manually-designed abstraction.** Related to the previous point, representation mechanisms are needed to capture manually-created abstractions. Such representation mechanisms should enable the representation of game-playing strategies at different levels of granularity and incorporate knowledge about numerical information, spatial information and time. A challenge here is to avoid creating a cumbersome representation that is either very difficult to understand or for which adequate reasoning mechanisms are difficult to create.
- **1-player vs 2-player vs multiplayer.** One of the biggest reasons why game-tree search has worked well in two-player zero-sum games is that the game-theoretic assumption of a "rational agent" is a relatively good model of how human experts play such games, hence algorithms such as minimax game-tree search can produce reasonably accurate predictions of future play. In multiplayer games, the "rational agent" model is arguably less accurate. Each player's notion of what states are preferable may depend not only on his/her game score but on a variety of social and psychological factors: cameraderie with friends, rivalries with old enemies, loyalties to a team, the impetus to "gang up on the winner," and so forth. Consequently, the players' true utility values are likely to be

nonzero-sum, and evaluation functions based solely on the game score will not produce correct approximations of those utilities. Consequently, an important challenge is how to build and maintain accurate models of players' social preferences. Some preliminary work has been done on this topic [83], but much more remains to be done.

- **Hidden information / uncertainty.** Asymmetric access to information among players gives rise to rich gameplay possibilities such as bluffing, hiding, feinting, surprise moves, information gathering / hiding, etc. In many (indeed most) card and board games asymmetric hidden information is central to interesting gameplay. Information asymmetry is equally important for many video games, including the real-time strategy and first-person games which are the primary focus of this article. When played between human players, much of the gameplay interest arises through gathering / hiding information, or exploiting gaps in opponents' knowledge. Creating AI which deals with hidden information is more challenging than for perfect information, with the result that many AI players effectively "cheat" by having access to information which should be hidden. In some cases this is rather blatant, with computer players making anticipatory moves which would be impossible for a human player without perfect information, in a way which feels unsatisfactory to human opponents. In other cases, there is a simulation of limited access to information. Ideally computer characters should have access to similar levels of sensory information as their human counterparts. The problems of hidden information in discrete domains is well-studied in game theory (see e.g. Myerson [50]), where the hidden information is captured very neatly by the information set idea. Here we have a state-action graph as usual, but each player is generally not aware of the precise state of the game, but the player's partial information allows the player to know which subset of possible states he is in (which is an information set from his point of view). Searching trees of information sets causes a new combinatorial explosion to be handled. This is a challenging new area for research, with some promise shown in complex discrete-time domains by the Information Set Monte Carlo Tree Search approach of Cowling et al. [21].
- **Simulating the video game world forward in time.** Rapid forward models could provide a glimpse of the world's future via simulation. This would allow researchers to use techniques based on state space search (e.g. Monte Carlo Tree Search, HTN, classical planning) more effectively either for offline or online AIs. However, modern games often feature complex 3D worlds with rich physics and hence limited capability to speed-up a simulation (there are hardware limitations of CPUs and GPUs). Moreover, games are real-time environments that can feature non-deterministic mechanisms, including actions of opponents, which might not be always reliably simulated. Opponent modelling [79] and level-of-detail AI [11] approach may provide a partial solution to these issues. However, these features are not always present in current games, and the design and development of forward models provides an interesting research challenge.
- **Explainability.** Explanation is an important component in computer gaming environments for two reasons. First, it can help developers understand the reason behind decisions made by the AI, which is very important particularly during debugging. Second, it can be useful to introduce game elements to new players. This is particularly crucial in the context of complex strategic games such as the civilization game series.

  Generating such explanations must deal with two main challenges. First, explanations must be meaningful. That is, the explanation must be understandable by the player and/or developer and the context of the explanation must also be understood. Possible snapshots from the game GUI could help to produce such explanations, highlighting the explanatory context. Second, explanations must be timely in the sense that they must be

rapidly generated and be shown at the right time. The latter point is crucial particularly if explanations occur during game play. They should not increase the cognitive load of the player, since this might result in the player losing interest in the game.

It seems important to distinguish between preconditions and consequences when automatically generating explanations. Preconditions are the subset of relevant aspects of the game that must be in a certain state for the offered explanation (of what the bot has done) to hold. Consequences are those the bot expected to occur when it executes a particular behavior in a the specified context. In order to generate meaningful explanations, preconditions and consequences must thus be automatically identified and presented in an accessible format to ensure the generation of meaningful explanations.

- **Robustness.** Video games must continue to work whatever the state of the game world and action of the players. hence any algorithm to which uses search to enhance strategy or other aspects of gameplay must always yield acceptable results (as well as usually yielding better results than current approaches). Explainability (discussed in the previous paragraph) allows for better debugging and greatly improves the chances for robustness. Robustness may be achieved by techniques such as formal proofs of correctness and algorithm complexity, or by having simple methods working alongside more complex search methods so that acceptable results are always available.
- **Non-smooth trees.** Many games (chess is a prime example) have trees which are non-smooth in the sense that sibling nodes at significant depth in the tree (or leaf nodes) often have different game theoretic values. This behaviour makes tree search difficult, and is arguably a reason why MCTS, while effective at chess AI, cannot compete with minimax/alphabeta. It seems likely and possible that game trees for some video games will have this pathology. While this makes heuristic search approaches that do not explore all siblings perform unreliably, one possible source of comfort here is that for complex game trees made up of highly aggregated states and actions, smoothness is quite close to the property that makes such a game playable (since otherwise a game becomes too erratic to have coherent strategies for a human player). While we raise this as a challenge and a consideration, successful resolution of some of the other challenges in this section may give rise to relatively smooth trees.
- **Red teaming, Real-World Problems.** Red Teaming is a concept that originated in military planning, in which a "Red Team" is charged with putting itself in the role of the enemy, in order to test the plans of the friendly force – the "Blue Team". The term Computational Red Teaming (CRT) has been coined to describe the use of computational tools and models to assist with this planning process. A technology review on CRT was recently carried out for the Australian Department of Defence [26]. One application domain for CRT is tactical battle planning, in which battles are simulated to test the effectiveness of candidate strategies for each side, and a search is overlaid to search for strong Red and/or Blue strategies. The task has a great deal in common with the search for tactics and strategies in Real Time Strategy games, and shares many of the same challenges: the strategy space is huge, involving coordinated movements and actions of many agents; the domain is continuous, both spatially and temporally; the "fog of war" ensures that much information is hidden or uncertain. It may be that approaches that have been developed to address these issues in CRT will also be helpful in RTS and similar games, and vice-versa.
- **Memory / CPU constraints.** CPU and Memory consumption and management represents a key issue in respect to computer game speed and scalability, even against a background of increasing parallel processing capability. State of the art game development

is primarily focused on the quality of the graphical presentation of the game environment. As a result, game engines tend to consume and constrain the computational power available for AI computations [15]. Therefore, an AI subsystem responsible for hundreds of active agents has to scale well, due to the resource demand in question. Even proven generic techniques like A* often need to be tweaked in respect to the game engine to be used effectively in real-time [27]. This makes the use of computationally intensive techniques, like classical planning or simulating decision outcomes, very problematic.

- **General Purpose Methods: Transfer learning.** Many of the above techniques have the aim of being generally applicable in whole sets or classes of game environments. Once a particular behavioral strategy of an agent has been established in one environment, the same strategy might also be useful in others. One approach to tackle such tasks is to abstract from the concrete scenario, producing a more general scheme. If such abstractions are available, then transfer learning will be possible. However, clearly the hard challenge, which does not seem to be answerable in the general sense, is how to abstract. Abstractions in time for transferring timing techniques may be as valuable as abstractions in space, such as exploration patterns, or abstraction on the representational format, for example, generalizing from one object to another. General definitions for useful abstractions and formalizations for the utility of general purpose methods are missing at the moment. Even more so, the challenge of producing a successful implementation of a general purpose AI, which, for example, may benefit from playing one RTS game when then being tested in another, related RTS game is still wide open.

4 Promising Research Directions and Techniques

4.1 Abstract Strategy Trees

The RTSplan "black box" strategy selection algorithm described earlier requires domain knowledge implemented in form of action scripts. Moreover, its look-ahead is limited in the sense that its simulations assume that players stick to scripts chosen in the root position. An adhoc solution to this problem is to add decision points in each simulation, e.g.. driven by game events such as combat encounters, and apply RTSplan recursively. This, however, can slow down the search considerably. Another serious problem is that switching game-playing scripts in the course of a game simulation episode may be inadequate to improve local behaviours. As an example, consider two pairs of squads fighting in different areas of an RTS game map, requiring different combat strategies. Switching the global game playing script may change both local strategies, and we therefore will not be able to optimize them independently. To address this problem, we may want to consider a search hierarchy in which game units optimize their actions *independent* from peers at the same level of the command hierarchy. As an example, consider a commander in charge of two squads. The local tasks given to them are decided by the commander who needs to ensure that the objectives can be accomplished without outside interference. If this is the case, the squads are independent of each other and their actions can be optimized locally. This hierarchical organization can be mapped to a recursive search procedure that on different command levels considers multiple action sequences for friendly and enemy units and groups of units, all subject to spatial and temporal constraints. Because the independence of sibling groups speeds up the search considerably, we speculate that this kind of search approach could be made computationally efficient.

As a starting point we propose to devise a 2-level tactical search for mid-sized RTS combat scenarios with dozens of units that are positioned semi-randomly on a small map

region. The objective is to destroy all enemy units. The low-level search is concerned with small-scale combat in which effective targeting order and moving into weapon range is the major concern when executing the top-level commands given to it. The high-level search needs to consider unit group partitions and group actions such as movement and attacking other groups. As there are quite a few high-level moves to consider, it may be worthwhile investigating unit clustering algorithms and promising target locations for groups. Now the challenge is to intertwine the search at both levels and to speed it up so that it can be executed in real-time to generate game actions for all units. Once two-level search works, it will be conceptually easy to extend it to more than two levels and – hopefully – provide interesting gameplay which challenges human supremacy in multi-level adversarial planning.

4.2 Monte Carlo Tree Search

Video games have a massive branching factor and a very high depth due to a large number of game agents, each with many actions, and the requirement to make a decision each frame. Since abstraction is a promising technique for video games that replaces the raw states by abstract states, the concrete actions by abstract actions or sequences of actions, the challenge is to use these abstractions in a tree search algorithm. This will give an anticipation skill to the artificial player. Instead of developing a tree whose nodes are raw states and arcs are actions, MCTS might be adapted such that nodes would correspond to abstract states and arcs to a group of actions or a sequence of actions, or even to a sequence of groups of actions. The strength of MCTS is to develop trees whose nodes contain simple statistics such as the number of visits and the number of successes. Therefore nothing forbids MCTS to build these statistics on abstract entities. To this extent MCTS combined with abstractions is a very promising technique for designing artificial agents playing video games.

4.3 Game Theoretic Approaches

Designing artificial players using abstraction and tree search for video games raises the question of backing up the information from child nodes to parent nodes when the nodes and the arcs are abstract. An abstract arc may correspond to a joint sequence of actions between the players such as: let army A fight against army B until the outcome is known. Given a parent node with many kinds of sequences completing with different outcomes, the parent node could gather the information in a matrix of outcomes. This matrix could be processed by using game theoretic tools to find out a Nash equilibrium and to back up the set of best moves. The following question would be then how to integrate such a game theoretic approach within a minimax tree search or a MCTS approach.

4.4 Learning from Replays

Learning in real-time games features a mixture of strong potential advantages and practical barriers. Benefits include the possibility to adapt to a new environment or a new opponent, increasing the gaming experience of users. Learning can reduce the game development time, adding more automation to the process and creating intelligent AI bots more quickly. On the other hand, a system that learns often involves the existence of a large space of parameter combinations that gets explored during learning. A large number of parameter combinations makes a thorough game testing very difficult.

An important challenge is making industry developers more receptive to learning, identifying cases where potential disadvantages are kept within acceptable limits. Competitions can

facilitate the development and the promotion of learning in real-time games, as a stepping stone between the academia and the industry.

Learning from replays is particularly appealing because game traces are a natural source of input data which exists in high volumes from network data of human vs human games. There are many parts of a game that can benefit from learning, and multiple learning techniques to consider. Reinforcement learning has been used to learn team policies in first-person shooter games [69]. In domain-independent planning, sample plans have been used to learn structures such as macro-operators and HTNs which can greatly speed up a search.

4.5 Partitioning States

Partitioning states into regions, subareas, choke points, and other relevant clusters is often related to spatial and temporal constraints that are associated with these partitionings. In the reinforcement learning domain, state partitionings have been automatically identified to improve hierarchical reinforcement learning [66]. The main idea behind this approach is to focus state partitionings on particular states which separate different subareas, thus focusing on choke points. However, the automatic detection depends on the behavioral capabilities of the reinforcement learning agent, thus offering a more general purpose approach to the problem. With respect to factored Markov Decision Processes, Variable Influence Structure Analysis (VISA) has been proposed to develop hierarchical state decomposition by means of Bayesian networks [33]. With respect to skills and skill learning, hierarchical state partitionings have been successfully developed by automatically selecting skill-respective abstractions [37].

While it might not be straight-forward, these techniques stemming from the reinforcement learning and AI planning literature, seem ready to be employed in RTS games. The exact technical transfer, however, still needs to be determined. In the future, intelligent partitioning of states brings may bring closer the prospect of developing autonomously learning, self-developing agents.

4.6 Evolutionary Approaches

It is interesting to speculate as to whether evolutionary approaches which have been used successfully by authors such as Sipper [67] might be used in video games. Evolutionary algorithms have also been used in RTS-like domains with some success. For example, Stanley et al. [72] use real-time neuroevolution to evolve agents for simulated battle games, demonstrating that human-guided evolution can be successful at evolving complex behaviours. In another example, Louis and Miles [45] used a combination of case-based reasoning and genetic algorithms to evolve strategies, and later [49] combined this to co-evolve players for a tactical game by evolving influence maps, and combining these with A^* search. Genetic programming was used in [34] to evolve behavior trees of characters playing FPS game capturing basic reactive concepts of the tasks. In a light survey of evolution in games, Lucas and Kendall [46] discuss many applications of evolutionary search in games, including video games.

More recently, a number of researchers have been using evolutionary or co-evolutionary search in the context of Red Teaming (as mentioned in Subsection 3.1). In this approach, the evolutionary algorithm (or similar, such as a Particle Swarm Optimisation algorithm) is used to search over a space of possible strategies for a given scenario. These strategies are represented at a high level, for example as a set of paths to be followed by agents representing the resources available to each side, organised into squads. Strategies are then evaluated

using low-fidelity agent-based simulations, called *distillations*, in which these orders provide the overall goals of the agents, and simple rules determine their movements and other actions. The outcomes of these simulations provide fitness values to drive the evolutionary search.

The idea of organising agents into a heirarchical command structure is similar to the *Level of detail AI* abstraction discussed in Subsection 2.2. Another possible hybrid approach is to use temporal abstraction to subdivide the planning period, a coevolutionary search to simultaneously identify sets of strong strategies for each side at each decision point, and MCTS to select from these the strongest lines of play. An initial investigation of an idea on these lines, on a small example scenario, can be found in [2].

4.7 Competitions and Software Platforms

In recent years numerous competitions have emerged presenting researchers with a good opportunity to compare their AI approaches in specific games and scenarios – BotPrize, Starcraft, Simulated Car Racing and Demolition Derby, and ORTS to name only a few.

The drawback of some of these competitions is the narrow universe of the respective games. As a result, AI players are created specifically for the game and the competition's aim – e.g. AI in StarCraft exploits game mechanics for micromanagement to win unit on unit combat, but are too specialised for other RTSs let alone other game genres. Thus the challenge of generating a general purpose AI is lost.

To overcome this drawback, it is necessary to introduce a conceptual and technical layer between the designed AI and the used game. This could provide the capability to compare AI designs across various games and scenarios. Deeper understanding of the common problems and technical details of games (e.g. RTS, FPS) and AIs is necessary to produce platforms, tools, techniques, and methodologies for AI creation for games without limiting them to specific virtual environment. An interesting development here is the Atari 2600 simulator (see [3]) which allows precise simulation of 1980s arcade games at thousands of times normal speed due to increases in hardware performance.

The first possible approach is to actually create a layer providing game abstractions for all games of a certain genre and develop AIs on top of it, like the state of the art software platforms xAitment, Havok AI, AI-Implant, and Recast. The second approach is to create a configurable proto-game capable of reflecting the most important mechanics of all games in a respective field (e.g. ORTS aims to provide a description of game mechanics for RTSs), which would be build using the state of the art design patterns of the present games. The third possible approach is to create a multi-game competition utilizing multiple games of the same type where AIs cannot exploit a certain game. Finally, since solving open AI problems (e.g. pathfinding) is one of the main issues within the gaming industry, it would be beneficial to create competitions aimed at solving problems posed by the video games industry. This could draw the game industry's attention and help to establish a bridge between industry and academia.

5 Conclusion

Search algorithms are already integral to video game AI, with A* pathfinding used in a huge number of games. This paper has reflected upon the wider applications of search in games, particularly the use of search in operational, tactical and strategic decision making in order to provide a more interesting gameplay experience. It seems that there are many rich research directions here, as well as many opportunities for academics to work together with

those from the games industry and to build more interesting and repayable games in the future.

In this paper we have considered how we may combine advances in search algorithms (such as Monte Carlo Tree Search, minimax search and heuristically guided search) with effective tools for abstraction (such as Hierarchical Task Networks, dynamic scripting, "black box" strategy selection, player modelling and learning approaches) to yield a new generation of search-based AI approaches for video games. Continuing advances in CPU speed and memory capacity make computationally intensive search approaches increasingly available to video games that need to run in real time. We have discussed the challenges that new techniques must face, as well as promising directions to tackle some of these issues.

We expect to see many advances in search-based video game AI in the next few years. It is difficult to predict where the most important advances will come, although we hope that this paper may provide some insight into the current state of the art, and fruitful future research directions as foreseen by a range of leading researchers in AI for search and abstraction in games.

References

1 T. Anantharaman, M. Campbell, and F. Hsu. Singular extensions: adding selectivity to brute force searching. *Artificial Intelligence*, 43(1):99–109, 1989.

2 D. Beard, P. Hingston, and M. Masek. Using monte carlo tree search for replanning in a multistage simultaneous game. In *Evolutionary Computation, 2012 IEEE Congress on*, 2012.

3 M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *ArXiv e-prints*, July 2012.

4 L. Berger. *AI Game Programming Wisdom*, chapter Scripting: Overview and Code Generation. Charles River Media, 2002.

5 Yngvi Bjornsson, Vadim Bulitko, and Nathan Sturtevant. TBA*: time-bounded A*. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 431–436, 2009.

6 Yngvi Björnsson and Hilmar Finnsson. CadiaPlayer: A Simulation-Based General Game Player. *IEEE Trans. Comp. Intell. AI Games*, 1(1):4–15, 2009.

7 A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Dev.*, 1:7–28, 2004.

8 Matthew M. Botvinick, Yael Niv, and Andrew C. Barto. Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *Cognition*, 113(3):262–280, 2009.

9 Bruno Bouzy. Associating domain-dependent knowledge and Monte-Carlo approaches within a go playing program. *Information Sciences*, 175(4):247–257, 2005.

10 M. Brockington. *AI Game Programming Wisdom*, chapter Level-Of-Detail AI for a Large Role-Playing Game. Charles River Media, 2002.

11 Cyril Brom, T. Poch, and O. Sery. AI level of detail for really large worlds. In Adam Lake, editor, *Game Programming Gems 8*, pages 213–231. Cengage Learning, 2010.

12 C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo Tree Search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, March 2012.

13 B. Bruegmann. Monte Carlo Go. ftp://www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z, 1993.

14 Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and C. Szepesvari. Online optimization in x-armed bandits. In *Proceedings NIPS*, pages 201–208, 2008.

15 M. Buro and T. Furtak. RTS games and real-time AI research. In *Proceedings of the Behaviour Representation in Modeling and Simulation Conference*, pages 51–58, 2004.

16 Martin V. Butz, Olivier Sigaud, Giovanni Pezzulo, and Gianluca Baldassarre. Anticipations, brains, individual and social behavior: An introduction to anticipatory systems. In Martin V. Butz, Olivier Sigaud, Giovanni Pezzulo, and Gianluca Baldassarre, editors, *Anticipatory Behavior in Adaptive Learning Systems: From Brains to Individual and Social Behavior*, pages 1–18. 2007.

17 M. Campbell, A.J. Hoane, and F-H Hsu. Deep Blue. *Artificial Intelligence*, 134:57–83, 2002.

18 A Champandard, T Verweij, and R Straatman. The AI for Killzone 2's multiplayer bots. In *Proceedings of Game Developers Conference*, 2009.

19 Guillaume Maurice Jean-Bernard Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. Monte-Carlo Strategies for Computer Go. In *Proc. BeNeLux Conf. Artif. Intell.*, pages 83–91, Namur, Belgium, 2006.

20 Remi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. van den Herik, Paolo Ciancarini, and H. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin / Heidelberg, 2007.

21 Peter I. Cowling, Edward J. Powley, and Daniel Whitehouse. Information Set Monte Carlo Tree Search. *IEEE Trans. Comp. Intell. AI Games*, 4(2), 2012.

22 K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

23 Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Proceedings of the 21st national conference on Artificial intelligence – Volume 1*, AAAI'06, pages 942–947. AAAI Press, 2006.

24 Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

25 Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA, 2006.

26 Phillip Gowlett. Moving forward with computational red teaming. Technical Report DSTO-GD-0630, Joint Operations Division, Defence Science and Technology Organisation, Department of Defence, Fairbairn Business Park Department of Defence Canberra ACT 2600 Australia, March 2011.

27 Ross Graham, Hugh Mccabe, and Stephen Sheridan. Pathfinding in computer games. *ITB Journal Issue Number*, 8:57–81, 2003.

28 D. Harabor and A. Botea. Hierarchical Path Planning for Multi-Size Agents in Heterogeneous Environments. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games CIG-08*, pages 258–265, Perth, Australia, December 2008.

29 P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics*, SSC-4(2):100–107, 1968.

30 Masahiko Haruno, Daniel M. Wolpert, and Mitsuo Kawato. Hierarchical mosaic for movement generation. In T. Ono, G. Matsumoto, R.R. Llinas, A. Berthoz, R. Norgren, H. Nishijo, and R. Tamura, editors, *Excepta Medica International Coungress Series*, volume 1250, pages 575–590, Amsterdam, The Netherlands, 2003. Elsevier Science B.V.

31 Chad Hogg, Héctor Muñoz Avila, and Ugur Kuter. Htn-maker: learning htns with minimal additional knowledge engineering required. In *Proceedings of the 23rd National Conference on Artificial intelligence – Volume 2*, AAAI'08, pages 950–956. AAAI Press, 2008.

32 R. Houlette and D. Fu. *AI Game Programming Wisdom 2*, chapter The Ultimate Guide to FSMs in Games. Charles River Media, 2003.

33 Anders Jonsson and Andrew Barto. Causal graph based decomposition of factored mdps. *Journal of Machine Learning Research*, 7:2259–2301, December 2006.

34 Rudolf Kadlec. Evolution of intelligent agent behaviour in computer games. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague, 2008.

35 D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

36 Levente Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. In Johannes Furnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer Berlin / Heidelberg, 2006.

37 G. Konidaris and A. Barto. Efficient skill learning using abstraction selection. *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 1107–1112, 2009.

38 R. Korf. Depth-first Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 97:97–109, 1985.

39 R. Korf. Depth-first Iterative Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

40 Richard Korf. Multi-player alpha-beta pruning. *Artificial Intelligence*, 49(1):99–111, 1991.

41 Ugur Kuter and Dana S. Nau. Forward-chaining planning in nondeterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, pages 513–518, July 2004.

42 Ugur Kuter and Dana S. Nau. Using domain-configurable search control for probabilistic planning. In *National Conference on Artificial Intelligence (AAAI)*, pages 1169–1174, July 2005.

43 Ugur Kuter, Dana S. Nau, Marco Pistore, and Paolo Traverso. Task decomposition on abstract states, for planning under nondeterminism. *Artifical Intelligence*, 173:669–695, 2009.

44 H. Hoangand S. Lee-Urban and H. Munoz-Avila. Hierarchical plan representations for encoding strategic game ai. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, 2005.

45 S.J. Louis and C. Miles. Playing to learn: case-injected genetic algorithms for learning to play computer games. *Evolutionary Computation, IEEE Transactions on*, 9(6):669 – 681, dec. 2005.

46 S.M. Lucas and G. Kendall. Evolutionary computation and games. *Computational Intelligence Magazine, IEEE*, 1(1):10–18, Feb. 2006.

47 C. Luckhardt and K.B. Irani. An algorithmic solution for n-person games. In *5th national Conference on Artificial Intelligence (AAAI)*, volume 1, pages 158–162, 1986.

48 C. Mansley, A. Weinstein, and M. Littman. Sample-based planning for continuous action markov decision processes. In *Proceedings ICAPS*, 2011.

49 C. Miles and S.J. Louis. Towards the co-evolution of influence map tree based strategy game players. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 75–82, May 2006.

50 Roger B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, 1997.

51 Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, December 2003.

52 Dana S. Nau, S. Smith, and T. Throop. The Bridge Baron: A big win for AI planning. In *European Conference on Planning (ECP)*, September 1997.

53 Pim Nijssen and Mark Winands. Enhancements for multi-player monte-carlo tree search. In *ACG13 Conference*, Tilburg, 2011.

54 Martin J. Osborne. *An Introduction to Game Theory*. Oxford, 2004.

55 J. Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14:113–138, 1980.

56 A. Plaat, J. Schaeffer, W. Pils, and A. de Bruin. Best-first fixed depth minimax algorithms. *Artificial Intelligence*, 87:255–293, November 1996.

57 Tomaso Poggio and Emilio Bizzi. Generalization in vision and motor control. *Nature*, 431:768–774, 2004.

58 I. Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1(3):193–204, 1970.

59 F. Poiker. *AI Game Programming Wisdom*, chapter Creating Scripting Languages for Non-Programmers. Charles River Media, 2002.

60 Edward J Powley, Daniel Whitehouse, Graduate Student Member, and Peter I Cowling. Monte Carlo Tree Search with macro-actions and heuristic route planning for the Physical Travelling Salesman Problem. In *Proc. IEEE CIG, submitted*, 2012.

61 Franisek Sailer, Michael Buro, and Marc Lanctot. Adversarial planning through strategy simulation. In *Proceedings of the Symposium on Computational Intelligence and Games (CIG)*, pages 80–87, 2007.

62 J. Schaeffer. The history heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transcations on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, November 1989.

63 Jonathan Schaeffer, Yngvi Björnsson Neil Burch, Akihiro Kishimoto, Martin Müller, Rob Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007. Work named by Science Magazine as one of the 10 most important scientific achievements of 2007.

64 B. Scott. *AI Game Programming Wisdom*, chapter Architecting an RTS AI. Charles River Media, 2002.

65 C.E. Shannon. Programming a computer to play Chess. *Philosoph. Magazine*, 41:256–275, 1950.

66 Özgür Simsek and Andrew G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proc. ICML-2004*, pages 751–758, 2004.

67 Moshe Sipper. *Evolved to Win*. Lulu, 2011. Available at `http://www.lulu.com/`.

68 D.J. Slate and L.R. Atkin. Chess 4.5 – the northwestern university chess program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

69 Megan Smith, Stephen Lee-Urban, and Hector Muñoz-Avila. Retaliate: Learning winning policies in first-person shooter games. In *AAAI*, 2007.

70 Stephen J. J. Smith, Dana S. Nau, and Thomas Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

71 P. Spronck. *AI Game Programming Wisdom 3*, chapter Dynamic Scripting. Charles River Media, 2006.

72 K.O. Stanley, B.D. Bryant, and R. Miikkulainen. Real-time neuroevolution in the nero video game. *Evolutionary Computation, IEEE Transactions on*, 9(6):653–668, Dec. 2005.

73 Nathan Sturtevant. An analysis of uct in multi-player games. *International Computer Games Association*, 31(4):195–208, December 2011.

74 Nathan Sturtevant and Richard Korf. On pruning techniques for multi-player games. In *7th National Conference on Artificial Intelligence (AAAI)*, pages 201–207. MIT Press, 2000.

75 Nathan Sturtevant, Martin Zinkevich, and Michael Bowling. Prob-max-n: Playing n-player games with opponent models. In *National Conference on Artificial Intelligence (AAAI)*, 2006.

76 Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1392–1397, 2005.

77 A. Tate, B. Drabble, and R. Kirby. *O-Plan2: An Architecture for Command, Planning and Control*. Morgan-Kaufmann, 1994.

78 Julian Togelius, G Yannakakis, K Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, (99):1–1, 2011.

79 H. J. van den Herik, H. H. L. M. Donkers, and P. H. M. Spronck. Opponent Modelling and Commercial Games. In G. Kendall and S. Lucas, editors, *Proceedings of IEEE 2005 Symposium on Computational Intelligence and Games CIG'05*, pages 15–25, 2005.

80 Tim Verweij. A hierarchically-layered multiplayer bot system for a first-person shooter. Master's thesis, Vrije Universiteit of Amsterdam, 2007.

81 J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

82 David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

83 Brandon Wilson, Inon Zuckerman, and Dana S. Nau. Modeling social preferences in multi-player games. In *Tenth Internat. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 2011.

84 Peter Yap, Neil Burch, Robert C. Holte, and Jonathan Schaeffer. Block A*: Database-driven search with applications in any-angle path-planning. In *AAAI*, 2011.

85 J. P. Zagal and M. Mateas. Time in Video Games: A Survey and Analysis. *Simulation & Gaming*, 41(6):844–868, August 2010.

86 A. Zobrist. A new hashing method with application for game playing. *ICCA Journal*, 13(2):69–73, 1990.