CASE-BASED PLAYER SIMULATION FOR THE COMMERCIAL STRATEGY GAME DEFCON

Robin Baumgarten and Simon Colton
Combined Reasoning Group, Department of Computing, Imperial College
180 Queens Gate, London SW7 2RH, UK.
E-mail: robin.baumgarten06@doc.ic.ac.uk, sgc@doc.ic.ac.uk

KEYWORDS
AI-bot control, strategy games, planning, machine learning, decision trees, case-based reasoning, simulated annealing.

ABSTRACT

DEFCON is a nuclear war simulation strategy game released in 2006 by Introversion Software Ltd. We describe an approach to simulating human game-play using a variety of AI techniques, including simulated annealing, decision tree learning and case-based reasoning. We have implemented an AI-bot that uses a novel approach to planning fleet movements and attacks. This retrieves plans from a case base of previous games, then merges these using a method based on decision tree learning. In addition, we have implemented more sophisticated control over low-level actions such as firing missiles and guiding planes. In particular, we have written routines to enable the AI-bot to synchronise bombing runs, and enabled a simulated annealing approach to assigning bombing targets to planes and opponent cities to missiles. We describe how our AI-bot operates, and the experimentation we have performed in order to determine an optimal configuration for it. With this configuration, our AI-bot beat Introversion's finite state machine automated player in 76.7% of 150 matches played.

1. INTRODUCTION

DEFCON is a multi-player retro-style strategy game from Introversion Software Ltd. Players compete in a nuclear war simulation to destroy as many opponent cities as possible. Playing the game involves placing resources (nuclear silos, fleets of ships, airbases and radar stations) within an assigned territory, then guiding defensive missile launches, bombing runs, fighter attacks, fleet movements and missile strikes. There is an existing 1-player mode where a player can compete against the computer. This employs a finite state machine with five states which are carried out in sequence: (i) *placement* of resources (ii) *scouting* to uncover resources of the opponent (iii) an a*ssault* on the opponent via attacks from bombers (iv) a *strike* on the opponent when silos launch their missiles, and (v) a *final* state, where fleets of ships approach and attack random opponent positions.

We have implemented an alternative AI-bot to play DEFCON and compete against Introversion's finite state game player. The operation of this AI-bot is multifaceted and relies on a number of AI techniques, including simulated annealing, decision tree learning and case-based reasoning. In particular, the AI-bot maintains a case-base of previously played games to learn from, as described in section 2. Our AI-bot first uses a structure placement algorithm to determine where nuclear silos, airbases and radars should be deployed. To do this, the AI-bot retrieves games from the case-base, ranks them using a weighted sum of various attributes (including life span and effectiveness) of the silos, airbases and radars in the previous game, and then uses the ranking to determine placement of these resources in the game being played, as described in section 3.

Our AI-bot also controls naval resources, organized into fleets and meta-fleets. Because these move during the game, the AI-bot uses a high-level plan to dictate the initial placement and meta-fleet movement/attack strategies. To generate a bespoke plan to fit the game being played, the AI-bot again retrieves cases from the case-base, and produces a plan by extracting pertinant information from retrieved plans via decision tree learning, as described in section 4.

During the game, the AI-bot carries out the meta-fleet movement and attack plan using a movement desire model to take its context (including the targets assigned to ships and opponent threats) into account. The AI-bot also controls low-level actions at game-time, such as the launching of plane bombing runs, attempting to destroy incoming missiles, and launching missile attacks from fleets. As described in section 5, we implemented various more sophisticated controls over these low-level actions. In particular, we enabled our AI-bot to synchronise the timing of planes when they attack opponent silos. We also implemented a simulated annealing approach to solve the problem of assigning bombing targets to planes and opponent cities to missiles.

We have performed much experimentation to fine-tune our AI-bot in order to maximize the proportion of games it wins against Introversion's own player simulator. In particular, in order to determine the weights in the fitness function for the placement of silos and airbases, we have determined the correlation of various resource attributes with the final score of the matches. We have also experimented with the parameters of the simulated annealing search for assignments. Finally, we have experimented with the size of the case-base to determine if/when overfitting occurs. With the most favourable setup, in a session of 150 games, our AI-bot won 76.7% of the time. We describe our experimentation in section 6, and in sections 7 and 8, we conclude with related work and some indication of future work.

2. A CASE-BASE OF PLAYED GAMES

We treat the training of our AI-bot as a machine learning problem in the sense of (Mitchell, 1997), where an agent learns to perform better at a task through increased exposure to the task. To this end, the AI-bot is able to store previously played games in a case-base, and retrieve games in order to play its current match more effectively. After a game is played, the AI-bot records the following information as a XML data-point in the case-base:

- The starting positions of the airbases, radar stations, fleets and nuclear silos for both players.
- The meta-fleet movement and attack plan which was used (as described in section 4).
- Performance statistics for deployed resources. For nuclear silos, this includes the number of missiles attacked and destroyed, and planes shot down by each silo. For radar stations: the number of missiles identified. For airbases: the number of planes launched and the number of planes which were quickly lost.
- An abstraction of the opponent attacks which took place. We abstract these into waves, by clustering using time-frames of 500 seconds and a threshold of 5 missiles fired (these settings were determined empirically).
- Routes taken by opponent fleets.
- The final scores of the two players in the game.

Cases are retrieved from the case-base using the *starting configuration* of the game. There are 6 territories that players can be assigned to (North America, Europe, South Asia, etc.), hence there are ${}^{6}P_{2} = 30$ possible territory assignments in a two player game, which we call the starting configuration of the game. This has a large effect on the game, so only cases with the same starting configuration as the current game are retrieved. For the rest of the paper, we assume that a suitable case-base is available for our AI-bot to use before and during the game. How we populate this case-base is described in section 6.

3. PLACEMENT OF SILOS, AIRBASES AND RADARS

Airbases are structures from which bombing runs are launched; silos are structures which launch nuclear missiles at opponent cities and defend the player's own cities against opponent missile strikes and planes; and radar stations are able to identify the position of enemy planes, missiles and ships within a certain range. As such, all these structures are very important, and because they cannot be moved at game time, their initial placement by the AI-bot at the start of the game is key to a successful outcome. The AI-bot uses the previously played games to calculate airbase, silo and radar placement for the current game. To do this, it retrieves cases with the same starting configuration as the current game, as described above. For each retrieved game, it analyses the statistics of how each airbase, silo and radar performed.

Each silo is given an *effectiveness* score as a weighted sum of the normalized values for the following:

(a) the number of enemy missiles it shot at.
(b) the number of enemy missiles it shot down.
(c) the number of enemy planes it shot down.
(d) the time it survived before being destroyed.

With respect to the placement of silos, each case is ranked using the sum of the effectiveness of its silos. Silo placement from the most effective case is then copied for the current game. Similar calculations inform the placement of the radar stations and airbases, with details omitted here for brevity.

To find suitable weights in the weighted sum for effectiveness, we performed a correlation analysis for the retaining/losing of resources against the overall game score. This analysis was performed using 1500 games played randomly (see section 6 for a description of how randomly played games were generated). In figure 1, we present the Pearson product-moment correlation coefficient for each of the AI-bot's own resources.

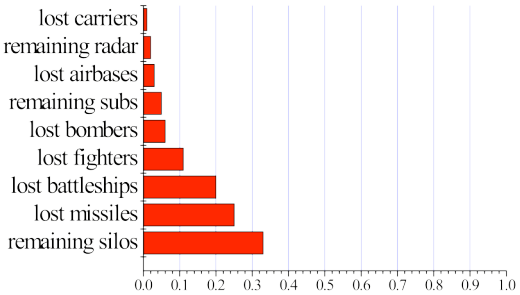

Figure 1. Pearson product-moment correlation coefficient for loss/retention of resources.

We note that – somewhat counter-intuitively – the *loss* of carriers, airbases, bombers, fighters, battleships and missiles is correlated with a winning game. This is explained by the fact that games where fewer of these resources were lost will have been games where the AI-bot didn't attack enough (when attacks are made, resources are inevitably lost). For our purposes, it is interesting that the retention of silos is highly correlated with winning games. This informed our choice of weights in the calculation of effectiveness for silo placement: we weighted value (d), namely the time a silo survived, higher than values (a), (b) and (c). In practice, for silos, we use $^{1}/_{10}$, $^{1}/_{3}$, $^{1}/_{6}$ and $^{2}/_{5}$ as weights for values (a), (b), (c) and (d) respectively. We used similar correlation analyses to determine how best to calculate the effectiveness of the placement of airbases and radar stations.

4. PLANNING SHIP MOVEMENTS

An important aspect of playing DEFCON is the careful control of naval resources (submarines, battleships and aircraft carriers). We describe here how our AI-bot generates a high-level plan for ship placement, movement and attacks at the start of the game, how it carries out such plans and how plans are automatically generated.

4.1 Plan Representation

It is useful to group resources into larger units so that their movements and attacks can be synchronized. DEFCON already allows collections of ships to be moved as a fleet, but players must target and fire each ship's missiles independently. To enhance this, we have introduced the notion of a *meta-fleet* which is a collection of a number of fleets of ships. Our AI-bot will typically have a small number of meta-fleets, (usually 1 or 2) with each one independently targeting an area of high opponent population. The meta-fleet movement and attack plans describe a strategy for each meta-fleet as a sub-plan, where the strategy consists of two large-scale movements of the meta-fleet. Each sub-plan specifies the following information:

(1) In what general area (sea territory) the ships in the meta-fleet should be initially placed, relative to the expected opponent fleet positions. (2) What the aim of the first large-scale movement should be, including where (if anywhere) the meta-fleet should move to, how it should move there and what general target area the ships should attack, if any. (3)

When the meta-fleet should switch to the second large-scale movement. (4) What the aim of the second large-scale movement should be, including the same details as for (2).

4.2 Carrying out Meta-fleet Movements at Game-time

Sea territories – assigned by DEFCON at the start of a game – are split into two oceans, and the plan dictates which one each meta-fleet should be placed in. The exact positions of the meta-fleet members are calculated at the start of the game using the case-base. That is, given the set of games retrieved, the AI-bot determines which sea territory contained on average most of the opponent's fleets. Within the chosen sea territory, the starting position depends on the aim of the first large-scale movement and estimated opponent fleet paths using the retrieved games. There are five aims for the large-scale movements, namely:

(a) to stay where they are and await the opponent's fleets in order to engage them later.
(b) to move in order to avoid the opponent's fleets.
(c) to move directly to the target of the attack.
(d) to move to the target avoiding the opponent's fleet.
(e) to move towards the opponent's fleet in order to intercept and engage them.

The aim determines whether the AI-bot places the fleets at positions with likely (cases (a) and (e)) or unlikely (cases (b) and (d)) opponent encounter probabilities, or as close to the attack spot as possible (case (c)). To determine a general area of the opponent's territory to attack (and hence to guide a meta-fleet towards), our AI-bot constructs an influence map (Tozour, 2001) built using opponent city population statistics. It uses the map to determine the highest centres of population density, and assigns these to the meta-fleets.

We implemented a central mechanism to determine both the best formation of a set of fleets into a meta-fleet, and the direction of travel of the meta-fleet given the aims of the large-scale movement currently being executed. During non-combative game-play, the central mechanism guides the meta-fleets towards the positions dictated in the plan, but this doesn't take into account the opponent's positions. Hence, we also implemented a movement desire model to take over from the default central mechanism when an attack on the meta-fleet is detected. This determines the direction for each ship in a fleet using (a) proximity to the ship's target if this has been specified (b) distance to any threatening opponent ships and (c) distance to any general opponent targets. A direction vector for each ship is calculated in light of the overall aim of the large-scale meta-fleet movement. For instance, if the aim is to engage the opponent, the ship will sail in the direction of the opponent's fleets.

The movement desire model relies on being able to predict where the opponent's fleets will be at certain times in the future. To estimate these positions, our AI-bot retrieves cases from the case-base at game-time, and looks at all the various positions the opponent's fleets were recorded at in the case. It then ranks these positions in terms of how close they are to the current positions of the opponent's fleets. To do this, it must assign each current opponent ship to one of the ships in the recorded game in such a way that the overall distance between the pairs of ships in the assignment is as low as possible. As this is a combinatorially expensive task, the AI-bot uses a simulated annealing approach to find a good solution, which is described in more detail in section 5. Once assignments have been made, the 5 cases with the closest assignments are examined and the fleet positions at specific times in the chosen retrieved games are projected onto the current game to predict the future position of the opponent's fleets. The 5 cases are treated as equally likely, thus fleets react to the closest predicted fleet positions according to the aim of their large-scale movement, e.g., approach or avoid it.

4.3 Automatically Generating Plans

As mentioned above, at the start of a game, the starting configuration is used to retrieve a set of cases. These are then used to generate a bespoke plan for the current game as follows. Firstly, each case contains the final score information of the game that was played. These are ranked in order of the AI-bot's score (which will be positive if it won, and negative if it lost). Within this ranking, the first half of the retrieved games are labeled as negative, and the second half are labeled as positive. [Hence, sometimes, winning games may be labeled negative and at other times, losing games may be labeled positive].

These positive and negative examples are then used to derive a decision tree which can predict whether a plan will lead to a positive or negative game. The attributes of the plan in the cases are used as attributes to split over in the decision tree, i.e., the number of meta-fleets, their starting sea territories, their first and second large-scale movement aims, etc. Our AI-bot uses the ID3 algorithm to learn the decision tree, and we portray the top nodes of an example tree in figure 2. We see that the most important factor for distinguishing positive and negative games is the starting sea territory for meta-fleet 1 (which can be in either low, mid or high enemy threat areas). Next, the decision tree uses the aim of the second large-scale meta-fleet movement, the attack time and the number of battleships in meta-fleet 1.

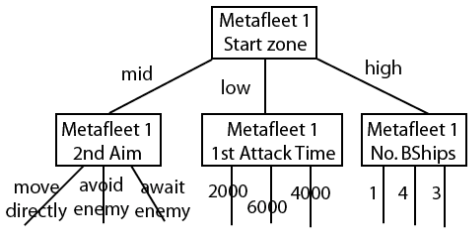

Figure 2. Top of a learned decision tree for plan outcomes.

Each branch from the top node to a leaf node in these decision trees represents a partial plan, as it will specify the values for some – but not necessarily all – of the attributes which make up a plan. The AI-bot chooses one of these branches by using an iterative fitness-proportionate method. That is, it chooses a path down the tree by looking at the subtree below each possible choice of value from the node it is currently looking at. Each subtree has a set of positive leaf nodes, and a set of negative leaf nodes, and the subtree with the highest proportion of positive leaf nodes is chosen (with a random choice between equally high-scoring sub-trees). This continues until a leaf node is reached. Having chosen a branch in this way, the AI-bot fills in the other attributes of the plan randomly.

5. SYNCHRONIZING ATTACKS

In order for players not to have to micro-manage the playing of the game, DEFCON automatically performs certain actions. For instance, air defence silos automatically target planes in attack range, and a battleship will automatically attack hostile ships and planes in its range. Players are expected to control where their planes attack, and where missiles are fired (from submarines, bombers and silos).

As mentioned above, the AI-bot uses an influence map to determine the most effective general radius for missile attacks from its silos, submarines and bombers. Within this radius it must assign a target to each missile. This is a combinatorially difficult problem, so we frame it as an instance of the *assignment problem* (Wilhelm, 1987), and our AI-bot searches for an injective mapping between the set of missiles and the set of cities using a simulated annealing heuristic search. To do this, it calculates the fitness of each mapping as the overall population of the cities mapped onto, and starts with a random mapping. Using two parameters, the starting temperature *S* and the cool-down rate *c*, a pair of missiles is chosen randomly, and the cities they are assigned to is swapped. Only if the fitness of the new mapping is smaller than *S*+1 multiplied by the current fitness is the new mapping kept. When each missile has been used in at least one swap, *S* is multiplied by *c* and the process continues until *S* reaches a cut-off value. For most of our testing, we used values *S=0.5*, *c=0.9* and a cut-off value of *0.04*, as these were found to be effective through some initial testing. We also experimented with these values, as described in section 6. Note that the mapping of planes to airbases for landing is a similar assignment problem, and we use a simulated annealing search process with the same *S* and *c* values.

Only silos can defend against missiles, and silos require a certain time to destroy. Thus attacks are more efficient when the time frame of missile strikes is kept small, so we enabled our AI-bot to organize planes to arrive at a target location at the same time. To achieve such a synchronized attack, our AI-bot makes individual planes take detours so that they arrive at the time that the furthest of them arrives without detour. Basic trigonometry gives two possible detour routes and our AI-bot uses the influence map to choose the route which avoids enemy territory the most.

6. EXPERIMENTATION

We tested the hypothesis that our AI-bot can learn to play DEFCON better by playing games randomly, then storing the games in the case-base for use as described above. To this end, for experiment 1, we generated 5 games per starting configuration (hence 150 games in total), by randomly choosing values for the plan attributes, then using the AI-bot to play against Introversion's own automated player. Moreover, whenever our AI-bot would ordinarily retrieve cases from the case-base, uninformed decisions were made for the fleet movement method, and the placement algorithm provided by Introversion was used. The five games were then stored in the case-base. Following this, we enabled the AI-bot to retrieve and use the cases to play against Introversion's player 150 times, and we recorded the percentage of games our AI-bot won. We then repeated the experiment with 10, rather than 5 randomly played games per starting configuration, then with 15, etc., up to 70 games.

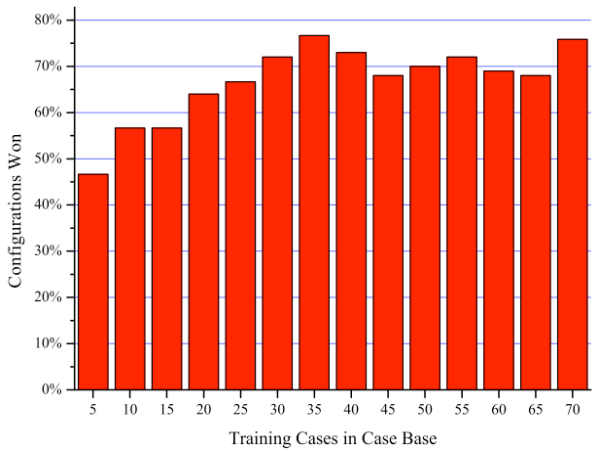

Figure 3. Performance vs. number of games in the case-base.

The results are portrayed in figure 3. We see that the optimal number of cases to use in the case-base is 35, and that our AI-bot was able to beat Introversion's player in 76.7% of the games. We analysed games with 35 cases and games with 40 cases to attempt to explain why performance degrades after this point, and we found that the decision tree learning process was more often using idiosyncracies from the cases in the larger case-base, hence overfitting (although this doesn't explain why a case-base size of 70 performed nearly as well as 35, which we are currently investigating). We describe some possible remedies for overfitting in section 8.

S	c	% Games won	Mean score differential
0	0	53.3	13.3
0.3	0.75	73.3	22.7
0.5	0.9	76.7	33.2
1.0	0.95	69.0	34.9
1.0	0.99	73.3	33.0

Table 1. Performance vs. simulated annealing parameters

Using the 35 cases optimum, we further experimented with the starting temperature and cool-down rate of the simulated annealing search for mappings. As described in section 5, our AI-bot uses the same annealing settings for all assignment problems, and we varied these from no annealing (*S=c=0)* to very high annealing (*S=1.0, c=0.99)*. As portrayed in table 1, we recorded both the proportion of wins in 150 games, and the score differential between the players. We see that our initial settings of *S=0.5*, *c=0.9* achieved the highest proportion of wins, whereas the setting *S=1.0, c=0.95* wins fewer games, but does so more convincingly.

In a final set of experiments, we tried various different meta-fleet planning setups to estimate the value of learning from played games. We found that with random generation of plans, our AI-bot won 50% of the time, and that by using hand-crafted plans developed using knowledge of playing DEFCON, we could increase this value to 63%. However, this was not as successful as our case-base learning approach, which – as previously mentioned – won 76.7% of games. This gives us confidence to try different learning methods, such as getting our AI-bot to play against itself, which we aim to experiment with in future work.

7. RELATED WORK

The use of case-based reasoning, planning and other AI techniques for board games is too extensive to cover, hence is beyond the scope of this paper. Cowling et. al have used DEFCON as a test-bed for AI techniques, in particular learning ship fleet formations (personal communication). Case-based reasoning has been used by (Fagan and Cunningham, 2003) for plan recognition in order to predict a player's actions when playing the Space Invaders game. They used a simplified planning language which didn't need preconditions and applied plan recognition to abstract state-action pairs. Using a plan library derived from games played by others, they achieved good predictive accuracy. In the real-time strategy game Wargus, a dynamic scripting approach was shown to outperform hand-crafted plans, as described in (Spronck, 2005). Hierarchical planning was tested by (Hoang et al., 2005) in an *Unreal Tournament* environment. They showed that this method had a clear advantage over finite state machines.

A comparison of artificial neural networks and evolutionary algorithms for optimally controlling a motocross bike in a video game was investigated by (Chaperot and Fyfe, 2005). Both methods were used to create riders which were compared with regards to their speed and originality. They found that the neural network found a faster solution but required hand crafted training data, while the evolutionary solution was slower, but found solutions that had not been found by humans previously. In commercial games, scripting and/or reactive behaviours have in general been sufficient to simulate planning, as full planning can be computationally expensive. However, the Dark Reign game from Activision uses a form of finite state machines that involves planning (Davis, 1999), and the first-person shooter game F.E.A.R employs goal-oriented action planning (Orkin, 2005).

8. CONCLUSIONS AND FUTURE WORK

We have implemented an AI-bot to play the commercial game DEFCON, and showed conclusively that it out-performs the existing automated player. In addition to fine-grained control over game actions, including the synchronization of attacks, intelligent assignment of targets via a simulated annealing search, and the use of influence maps, our AI-bot uses plans to determine large-scale fleet movements. It uses a case-base of randomly-planned previously played games to find similar games, some of which ended in success while others ended in failure. It then identifies the factors which best separate good and bad games by building a decision tree using ID3. The plan for the current game is then derived using a fitness-proportionate traversal of the decision tree to find a branch which acts as a partial plan, and the missing parts are filled in randomly. To carry out the fleet movements, ships are guided by a central mechanism, but this is superceded by a movement-desire model if a threat to the fleet is detected.

There are many ways in which we can further improve the performance of our AI-bot. In particular, we aim to lessen the impact of over-fitting when learning plans, by implementing different decision tree learning skills, filling in missing plan details in non-random ways, and by trying other logic-based machine learning methods, such as Inductive Logic Programming (Muggleton, 1995). We also hope to identify some markers for success during a game, in order to apply techniques based on reinforcement learning. There are also a number of improvements we intend to make to the control of low-level actions, such as more sophisticated detours that planes make to synchronise attacks.

We are also interested in designing an automated player which is able to *entertain* human players as well as (attempt to) beat them. Hence, we plan a series of experiments where human players describe what is unsatisfying about playing against the AI-bot, and we hope to implement methods which increase the enjoyment of players against the AI-bot. With improved skills to both beat and engage players, we will address the question of how to enable the AI-bot to play in a multi-player environment. This represents a significant challenge, as our AI-bot will need to collaborate with other players by forming alliances, which will require opponent modeling techniques. We aim to use DEFCON and similar video games to test various combinations of AI techniques, as we believe that integrating reasoning methods has great potential for building intelligent systems.

ACKNOWLEDGEMENTS

We are very grateful to Introversion Software for allowing us access to their DEFCON code, building an AI interface and for their continued input and enthusiasm for the project.

REFERENCES

Chaperot, B and Fyfe, C. 2005. *Motocross and aritifical neural networks*. Third International Conference on Game Design and Technology.

Davis, I. 1999. *Strategies for game AI.* In proceedings of the AAAI Spring Symposium on AI and computer games.

Fagan, M and Cunningham, P. 2003. *Case-based plan recognition in computer games.* In proceedings of ICCBR.

Mitchell, T. 1997. *Machine Learning*, McGraw Hill.

Muggleton, S. 1991. *Inductive Logic Programming.* New Generation Computing, 8(4) 295-318.

Hoang, H, Lee-Urban, S, and Muñoz-Avila, H. 2006. *Hierarchical Plan Representations for Encoding Strategic Game AI*. In Proceedings of AIIDE-05.

Orkin, J. 2005. *Agent architecture considerations for real-time planning in games.* In proceedings of AIIDE-05.

Spronck, P. 2005. *Adaptive Game AI*, Universitaire Pers Maastricht.

Tozour, P. 2001. *Influence mapping.* In Game Programming Gems, volume 2. Charles River Media.

Wilhelm, M. 1987. *Solving quadratic assignment problems by simulated annealing.* IIE transactions. 19(1):107-119.