UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Gabriel Synnaeve

Thèse dirigée par **Pierre Bessière**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale de Mathématiques, Sciences et Technologies de l'Information, Informatique**

Bayesian Programming and Learning for Multi-Player Video Games

Application to RTS AI

Thèse soutenue publiquement le **XXX**,
devant le jury composé de :

Mr Augustin Lux
Professeur à Grenoble Institut National Polytechnique, Président
Mr Stuart Russell
Professeur à l'Université de Californie à Berkeley, Rapporteur
Mr Philippe Leray
Professeur à l'Ecole Polytechnique de l'Université de Nantes, Rapporteur
Mr Marc Schoenauer
Directeur de Recherche à INRIA à Saclay, Examinateur
Mr Pierre Bessière
Directeur de Recherche au CNRS au Collège de France, Directeur de thèse

Remerciements

Résumé

TODO

Dans la première partie de cette thèse, nous détaillons les solutions actuelles aux problèmes qui se posent lors de la réalisation d'une IA de jeu multi-joueur, en donnant un aperçu des caractéristiques calculatoires et cognitives complexes des principaux types de jeux. En partant de ce constat, nous résumons les catégories transversales de problèmes, et nous introduisons comment elles peuvent être résolues par la modélisation bayésienne. Nous expliquons alors comment construire un programme bayésien en partant de connaissances et d'observations du domaine à travers un exemple simple de jeu de rôle. Dans la deuxième partie de la thèse, nous détaillons l'application de cette approche à l'IA de STR, ainsi que les modèles auxquels nous sommes parvenus. Pour le comportement réactif (micro-management), nous présentons un controleur multi-agent décentralisé et temps réel inspiré de la fusion sensori-motrice. Ensuite, nous accomplissons les adaptation dynamiques de nos stratégies et tactiques à celles de l'adversaire en le modélisant à l'aide de l'apprentissage artificiel (supervisé et non supervisé) depuis des traces de joueurs de haut niveau. Ces modèles probabilistes de joueurs peuvent être utilisés à la fois pour la prédiction des décisions/actions de l'adversaire, mais aussi à nous-même pour la prise de décision si on substitue les entrées par les notres. Enfin, nous expliquons l'architecture de notre joueur robotique de StarCraft, et nous précisions quelques détails techniques d'implémentation. Au delà des modèles et de leurs implémentations, il y a trois contributions principales: la reconnaissance de plan et la modélisation de l'adversaire par apprentissage artificiel, en tirant partie de la structure du jeu, la prise de décision multi-échelles en présence d'informations incertaines, et l'intégration des modèles bayésiens au contrôle temps réel d'un joueur artificiel.

TODO

Abstract

This thesis explores the use of Bayesian models in multi-player video games AI, particularly real-time strategy (RTS) games AI. Video games are an in-between of real world robotics and total simulations, as other players are not simulated, nor do we have control over the simulation. RTS games require having strategic (technological, economical), tactical (spatial, temporal) and reactive (units control) actions and decisions on the go. We used Bayesian modeling as an alternative to (boolean valued) logic, able to cope with incompleteness of information and (thus) uncertainty. Indeed, incomplete specification of the possible behaviors in scripting, or incomplete specification of the possible states in planning/search raise the need to deal with uncertainty. Machine learning helps reducing the complexity of fully specifying such models. We show that Bayesian programming can integrate all kinds of sources of uncertainty (hidden state, intention, stochasticity), through the realization of a fully robotic StarCraft player. Probability distributions are a mean to convey the full extent of the information we have and can represent by turns: constraints, partial knowledge, state space estimation and incompleteness in the model itself.

In the first part of this thesis, we review the current solutions to problems raised by multi-player game AI, by outlining the types of computational and cognitive complexities in the main gameplay types. From here, we sum up the transversal categories of problems, introducing how Bayesian modeling can deal with all of them. We then explain how to build a Bayesian program from domain knowledge and observations through a toy role-playing game example. In the second part of the thesis, we detail our application of this approach to RTS AI, and the models that we built up. For reactive behavior (micro-management), we present a real-time multi-agent decentralized controller inspired from sensory motor fusion. We then show how to perform strategic and tactical adaptation to a dynamic opponent through opponent modeling and machine learning (both supervised and unsupervised) from highly skilled players' traces. These probabilistic player-based models can be applied both to the opponent for prediction, or to ourselves for decision-making, through different inputs. Finally, we explain our StarCraft robotic player architecture and precise some technical implementation details.

Beyond models and their implementations, our contributions are threefolds: machine learning based plan recognition/opponent modeling by using the structure of the domain knowledge, multi-scale decision-making under uncertainty, and integration of Bayesian models with a real-

time control program.

Contents

Notations

Symbols

$\leftarrow$	assignment of value to the left hand operand
$\sim$	the right operand is the distribution of the left operand (random variable)
$\propto$	proportionality
$\approx$	approximation
$\#$	cardinal of a space or dimension of a variable

Variables

X and $Variable$	random variables (or logical propositions)
x and $value$	values
$\#X = \lvert X \rvert$	cardinal of variable X
$X_{1:n}$	the set of n random variables $X_1 \dots X_n$
$\{x \in \Omega \lvert Q(x)\}$	the set of elements from Ω which verify Q

Probabilities

$\mathrm{P}(X) = Distribution$	is equivalent to $X \sim Distribution$
$\mathrm{P}(x) = \mathrm{P}(X = x) = \mathrm{P}([X = x])$	Probability (distribution) that X takes the value x
$\mathrm{P}(X, Y) = \mathrm{P}(X \wedge Y)$	Probability (distribution) of the conjunction of X and Y
$\mathrm{P}(X \lvert Y)$	Probability (distribution) of X knowing Y

Conventions

TODO

Chapter 1

Introduction

1.1 Motivations

Video games AI research is yielding new approaches to a wide range of problems, for instance in RTS: pathfinding, multiple agents coordination, collaboration, prediction, planning and (multi-scale) reasoning under uncertainty. These problems are particularly interesting in the RTS framework because the solutions have to deal with many objects, imperfect information and micro-actions while running in real-time on desktop hardware.

- Multi-player video games Adversarial decision-making. Partial information. Real-time. Massive state spaces.
- RTS games Expert human play > AI. Data. Competitions.

1.2 Contributions

- A new approach to game AI with:
  - first-class uncertainty.
  - a tractable hierarchical decomposition of problems.
- Integration of learning in a decision-making model.
- An autonomous agent for StarCraft (BroodwarBotQ).

Finally, with this thesis, we hope to contribute a guide for industry practitioners who would like to have new tools for solving the ever increasing complexity of game AI, and more generally "huge state space" and multi-scale AI.

1.3 Reading Map

À la MacKay (Industry vs Research)

Chapter 2

Game AI

> *It is not that the games and mathematical problems are chosen because they are clear and simple; rather it is that they give us, for the smallest initial structures, the greatest complexity, so that one can engage some really formidable situations after a relatively minimal diversion into programming.*
>
> Marvin Minsky (Semantic Information Processing, 1968)

Is the primary goal of game AI to win the game? "Game AI" is simultaneously a research topic and an industry standard practice, for which the main metric is the fun the players are having. Its uses range from character animation, to behavior modeling, strategic play, and a true gameplay component. In this chapter, we will give our educated guess about the goals of game AI, and review what exists for a broad category of games: abstract strategy games, partial information and/or stochastic games, different genres of computer games. Let us then focus on gameplay (from a player point of view) characteristics of theses games so that we can enumerate game AI needs.

2.1 Goals of Game AI

NPC

Non-playing characters (NPC*), also called "mobs", represent a massive volume of game AI, as even a lot of multi-player games have NPC. They really represents players that are not conceived

to be played by humans, by opposition to "bots", which corresponds to human-playable characters controlled by an AI. NPC are an important part of ever more immersive single player adventures (The Elder Scrolls V: Skyrim), of cooperative gameplay (World of Warcraft, Left 4 Dead), or as helpers or trainers ("pets", strategy games). NPC can be a core part of the gameplay as in Creatures or Black and White or dull "quest giving poles" as in a lot of role-playing games. They are of interest for the game industry, but also for robotics, to study human cognition and for artificial intelligence in the large. So, the first goal of game AI is perhaps just to make the artificial world seem alive: a paint is not much fun to play in.

Win

During the last decade, the video games industry has seen the emergence of "e-sport". It is the professionalizing of specific competitive games at the higher levels, as in sports: with spectators, leagues, sponsors, fans and broadcasts. A list of major electronic sports games includes (but is not limited to): StarCraft: Brood War, Counter-Strike, Quake III, Warcraft III, Halo, StarCraft II. The first game to have had pro-gamers* was StarCraft: Brood War*, in Korea, with top players earning more than Korean top soccer players. Top players earn more than $400,000 a year but the professional average is below, around $50-60,000 a year [**?**], against the average South Korean salary at $16,300 in 2010. Currently, Brood War is being slowly phased out to StarCraft II. There are TV channels broadcasting Brood War (OnGameNet, previously also MBC Game) or StarCraft II (GOM TV, streaming) and for which it constitutes a major chunk of the air time. "E-sport" is important to the subject of game AI because it ensures competitiveness of the human players. It is less challenging to write a competitive AI for game played by few and without competitions than to write an AI for Chess, Go or StarCraft. E-sport, through the distribution of "replays*" also ensures a constant and heavy flow of human player data to mine and learn from. Finally, cognitive science researchers (like the Simon Fraser University Cognitive Science Lab) study the cognitive aspects (attention, learning) of high level RTS playing [**?**].

Good human players, through their ability to learn and adapt, and through high-level strategic reasoning, are still undefeated. Single players are often frustrated by the NPC behaviors in non-linear (not fully scripted) games. Nowadays, video games AI can be used as part of the gameplay as a challenge to the player. This is not the case in most of the games though, in decreasing order of resolution of the problem[1]: fast FPS* (first person shooters), team FPS, RPG* (role playing games), MMORPG* (Massively Multi-player Online RPG), RTS* (Real-Time Strategy). These games in which artificial intelligences do not beat top human players on equal footing requires increasingly more cheats to even be a challenge (not for long as they mostly do not adapt). AI cheats encompass (but are not limited to):

- RPG NPC often have at least 10 times more hit points (health points) than their human counterparts in equal numbers,
- FPS bots can see through walls and use perfect aiming,
- RTS bots see through the "fog of war*" and have free additional resources.

[1]Particularly considering games with possible free worlds and "non-linear" storytelling, current RPG and MMORPG are often limited *because* of the untracted "world interacting NPC" AI problem.

How do we build game robotic players ("bots", AI, NPC) which can provide some challenge, or be helpful without being frustrating, while staying fun?

Fun

The main purpose of gaming is entertainment. Of course, there are game genres like serious gaming, or the "gamification*" of learning, but the majority of people playing games are having fun. Cheating AI are not fun, and so the replayability* of single player games is very low. The vast majority of games which are still played after the single player mode are multi-player games, because humans are the most fun to play with. So how do we get game AI to be fun to play with? The answer seems to be 3-fold:

- For competitive and PvP* (players versus players) games: improve game AI so that it can play well *on equal footing with humans*,
- for cooperative and PvE* (players vs environment) games: optimize the AI for fun, "epic wins": the empowerment of playing your best and just barely winning,
- give the AI all the tools to adapt the game to the players: AI directors* (as in Left 4 Dead* and Dark Spore*), procedural content generation (e.g. automatic personalized Mario [**?**]).

In all cases, a good AI should be able to learn for the players' actions, recognize their behavior to deal with it in the most entertaining way. Examples for a few mainstream games: World of Warcraft instances or StarCraft II missions could be less predictable (less scripted) and always "just hard enough", Battlefield 3 or Call of Duty opponents could have a longer life expectancy (5 seconds in some cases), Skyrim's follower NPC could avoid blocking the player in doors, or going in front when she casts fireballs.

Programming

How do game developers want to deal with game AI programming? We have to understand the needs of industry game AI programmers:

- computational efficiency: most games are real-time systems, 3D graphics are computationally intensive, as a result the AI CPU budget is low,
- game designers often want to remain in control of the behaviors, so game AI programmers have to provide authoring tools,
- AI code has to scale with the state spaces while being debuggable,
- AI behaviors have to scale with the possible game states (which are not all predictable due to the presence of the human player),
- re-use accross games (game independant logic, at least libraries).

As a first approach, programmers can "hard code" the behaviors and their switches. For some structuring of such states and transitions, they can and do use state machines [**?**]. This solution does not scale well (exponential increase in the number of transitions), nor do they generate

autonomous behavior, and they can be cumbersome for the game designers to interact with. Hierarchical FSM is a partial answer to these problems: they scale better due to the sharing of transitions between macro-states and are more readable for game designers who can zoom-in on macro/englobing states. They still represent way too much programming work for complex behavior and are not more autonomous than classic FSM. **?** used an adaptive FSM mechanism inspired by evolutionary algorithms to play Quake III team games. Planning (using a search heuristic in the states space) efficiently gives autonomy to virtual characters. Planners like hierarchical task networks (HTN [**?**], Armed Assault, Killzone 2 [**??**]) or STRIPS ([**?**], F.E.A.R [**?**]) generate complex behaviors in the space of the combinations of specified states, and the logic can be re-used accross games. The drawbacks can be a large computational budget (for many agents and/or a complex world), the difficulty to specify reactive behavior, and less (or harder) control from the game designers. Behavior trees (Halo 2 [**?**], Spore) are a popular in-between HTN and HFSM technique providing scability through a tree-like hierarchy, control through tree editing and some autonomy through a search heuristic. A transversal technique for ease of use is to program game AI with a script (LUA, Python) or domain specific language (DSL*). From a programming or design point of view, it will have the drawbacks of the models it is based on. If everything is allowed (low-level inputs and outputs directly in the DSL), everything is possible at the cost of cumbersome programming, debugging and few re-use.

Even with scalable[2] architectures like behavior trees or the autonomy that planning provides, there are limitations (burdens on programmers/designers or CPU/GPU):

- complex worlds require either very long description of the state (in propositional logic) or high expressivity (higher order logics) to specify well-defined behaviors,
- the search space of possible actions increases exponentially with the volume and complexity of interactions with the world, thus requiring ever more efficient pruning techniques,
- once human players are in the loop (is it not the purpose of a game?), uncertainty has to be taken into account. Previous approaches can be "patched" to deal with uncertainty, at what cost?

Our thesis is that we can learn complex behaviors from exploration or observations (of human players) without the need to be explicitely programmed. Furthermore, the game designers can stay in control by choosing which demonstration to learn from and tuning parameters by hand if wanted. **?** showed it in the case of FPS AI (Unreal Tournament), with *inverse programming* to learn reactive behaviors from human demonstration. We extend it to tactical and even strategic behaviors.

2.2 Single Player Games

Single player games are not the main focus of our thesis, but they present a few interesting AI characteristics. They encompass all kinds of human cognitive abilities, from reflexes to higher level thinking.

[2]both computationally and in the number of lines of codes to write to produce a new behavior

Action games

Platform games (Mario, Sonic), time attack racing games (TrackMania), solo shoot-them-up ("schmups", Space Invaders, DodonPachi), sports games and rhythm games (Dance Dance Revolution, Guitar Hero) are games of reflexes, skill and familiarity with the environment. The main components of game AI in these genres is a quick path search heuristic, often with a dynamic environment. At the Computational Intelligence and Games conferences series, there have been Mario [**?**], PacMan [**?**] and racing competitions [**?**]: the winners often use (clever) heuristics coupled with a search algorithm (A* for instance). As there are no human opponents, reinforcement learning and genetic programming works well too. In action games, the artificial player most often has a big advantage on its human counterpart as reaction time is one of the key characteristics.

Puzzles

Point and click (Monkey Island, Kyrandia, Day of the Tentacle), graphic adventure (Myst, Heavy Rain), (tile) puzzles (Minesweeper, Tetris) games are games of logical thinking and puzzle solving. The main components of game AI in these genres is an inference engine with sufficient domain knowledge (an ontology). AI research is not particularly active in the genre of puzzle games, perhaps because solving them has more to do with writing down the ontology than with using new AI techniques. A classic well-studied logic-based, combinatorial puzzle is Sudoku, which has been formulated as a SAT-solving [**?**] and constraint satisfaction problem [**?**].

2.3 Abstract Strategy Games

Tic-tac-toe, Minimax

Tic-tac-toe (noughts and crosses) is a solved game*, meaning that it can be played optimally from each possible position. How did it came to get solved? Each and every possible positions (26,830) have been analyzed by a Minimax (or its variant Negamax) algorithm. Minimax is an algorithm which can be used to determine the optimal score a player can get for a move in a zero-sum game*. The Minimax theorem states:

Theorem. *For every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that (a) Given player 2's strategy, the best payoff possible for player 1 is V, and (b) Given player 1's strategy, the best payoff possible for player 2 is -V.*

Applying this theorem to Tic-tac-toe, we can say that winning is +1 point for the player and losing is -1, while draw is 0. The exhaustive search algorithm which takes this property into account is described in Algorithm 1. The result of applying this algorithm to the Tic-tac-toe situation of Fig. 2.1 is exhaustively represented in Fig. 2.2. For zero-sum games (as abstract strategy games discussed here), there is a (simpler) Minimax variant called Negamax, shown in Algorithm 5 in Appendix A.

Algorithm 1 Minimax algorithm

```
function MINI(depth)
    if depth ≤ 0 then
        return −value()
    end if
    min ← +∞
    for all possible moves do
        score ← maxi(depth − 1)
        if score < min then
            min ← score
        end if
    end for
    return min
end function
function MAXI(depth)
    if depth ≤ 0 then
        return value()
    end if
    max ← −∞
    for all possible moves do
        score ← mini(depth − 1)
        if score > min then
            max ← score
        end if
    end for
    return max
end function
```

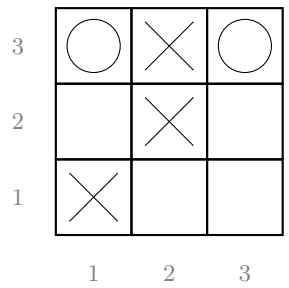


Figure 2.1: A Tic-tac-toe board position, in which it is the "circles" player's turn to play. The labeling explains the indexing (left to right, bottom to top, starting at 1) of the grid.

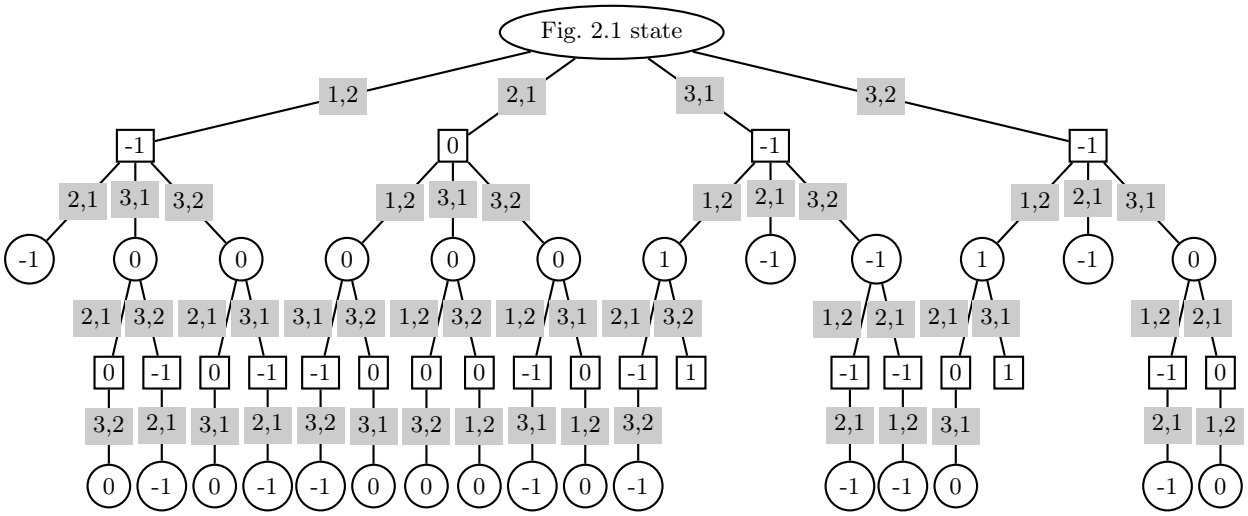


Figure 2.2: Minimax tree with initial position at Fig. 2.1 state, **nodes** are states and **edges** are transitions, labeled with the move. Leafs are end-game states: 1 point for the win and -1 for the loss. Player is "circles" and plays first (first edges are player's moves).

Checkers, Alpha-beta

Checkers, Chess and Go are also zero sum, perfect-information*, partisan*, deterministic strategy game. Theoretically, they all can be solved by exhaustive Minimax. In practice though, it is often intractable: their bounded versions are at least in PSPACE and their unbounded versions are EXPTIME-hard [**?**]. We can see the complexity of Minimax as $O(b^d)$ with b the average branching factor of the tree (to search) and d the average length (depth) of the game. For Checkers $b \approx 8$, but taking pieces is mandatory, resulting in a mean adjusted branching factor of ≈ 4, while the mean game length is 70 resulting in a game tree complexity of $\approx 10^{31}$ [**?**]. It is already too large to have been solved by Minimax alone (on current hardware). From 1989 to 2007, there were artificial intelligences competitions on Checkers, all using at least alpha-beta pruning: a technique to make efficient cuts in the Minimax search tree while not losing optimality. The state space complexity of Checkers is the smallest of the 3 above-mentioned games with $\approx 5.10^{20}$ legal possible positions (conformations of pieces which can happen in games). As a matter of fact, Checkers have been (weakly) solved, which means it was solved for perfect perfect play on both sides (and always ends in a draw) [**?**]. Not all positions resulting from imperfect play have been analyzed.

Alpha-beta pruning (see Algorithm 2) is a branch-and-bound algorithm which can reduce Minimax search down to a $O(b^{d/2}) = O(\sqrt{b^d})$ complexity if the best nodes are searched first ($O(b^{3d/4})$ for a random ordering of nodes). α is the maximum score than we (the maximizing player) are assured to get given what we already evaluated, while β is the minimum score than the minimizing player is assured to get. When evaluating more and more nodes, we can only get a better estimate and so α can only increase while β can only decrease. If we find a node for which β becomes less than α, it means that this position results from sub-optimal play. When it is because of an update of β, the sub-optimal play is on the side of the maximizing player (his α is not high enough to be optimal and/or the minimizing player has a winning move faster in

Algorithm 2 Alpha-beta algorithm

```
function ALPHABETA(node,depth,α,β,player)
    if depth ≤ 0 then
        return value(player)
    end if
    if player == us then
        for all possible moves do
            α ← max(α, alphabeta(child, depth − 1, α, β, opponent))
            if β ≤ α then
                break
            end if
        end for
        return α
    else
        for all possible moves do
            β ← min(β, alphabeta(child, depth − 1, α, β, us))
            if β ≤ α then
                break
            end if
        end for
        return β
    end if
end function
```

the current sub-tree) and this situation is called an α cut-off. On the contrary, when the cut results from an update of α, it is called a β cut-off and means that the minimizing player would have to play sub-optimally to get into this sub-tree. When Starting the game, α is initialized to $-\infty$ and β to $+\infty$. A worked example is given on Figure 2.3. Alpha-beta is going to be helpful to search much deeper than Minimax in the same allowed time. The best Checkers program (since the 90s), which is also the project which solved Checkers [**?**], Chinook, has openings and end-game (for lest than eight pieces of fewer) books, and for the mid-game (when there are more possible moves) relies on a deep search algorithm. So, appart for the beginning and the ending of the game, for which it plays by looking up a database, it used a search algorithm. As Minimax and Alpha-beta are depth first search heuristics, all programs which have to answer in a fixed limit of time use *iterative deepening*, a technique which consists in fixing limited depth which will be considered maximal and evaluating this position. As it does not relies in winning moves at the bottom, because the search space is too big in $branching^{depth}$, we need moves evaluation heuristics. We then iterate on growing the maximal depth for which we consider moves, but we are at least sure to have a move to play in a short time (at least the greedy depth 1 found move).

Chess, Heuristics

With a branching factor of ≈ 35 and an average game length of 80 moves [**?**], the average game-tree complexity of chess is $35^{80} \approx 3.10^{123}$. **?** also estimated the number of possible (legal) positions to be of the order of 10^{43}, which is called the Shannon number. Chess AI needed a little more than just Alpha-beta to win against top human players (not that Checkers could not benefit

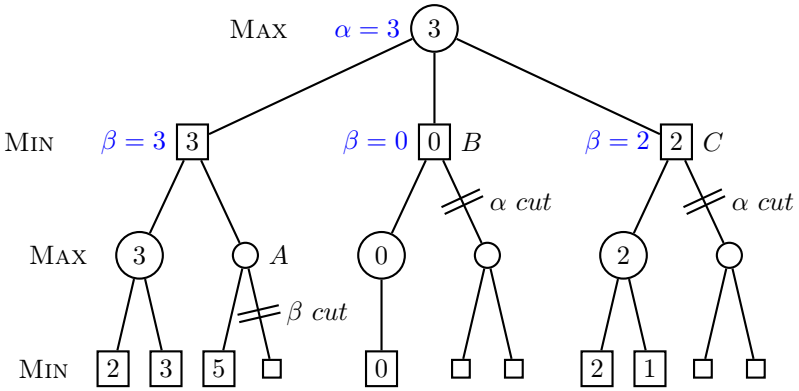


Figure 2.3: Alpha-beta cuts on a Minimax tree, **nodes** are states and **edges** are transitions, labeled with the values of positions at the bottom (max depth). Here is the trace of the algorithm: **1.** descend leftmost first and evaluated 2 and 3, **2.** percolate max(2,3) higher up to set $\beta = 3$, **3.** β-cut in A because its value is at least 5 (which is superior to $\beta = 3$), **4.** Update of $\alpha = 3$ at the top node, **5.** α-cut in B because it is at most 0 (which is inferior to $\alpha = 3$), **6.** α-cut in C because it is at most 2, **7.** conclude the best value for the top node is 3.

it), particularly on 1996 hardware (first win of a computer against a reigning world champion, Deep Blue vs. Garry Kasparov). Once an AI has openings and ending books (databases to look-up for classical moves), how can we search deeper during the game, or how can we evaluate better a situation? In iterative deepening Alpha-beta (or other search algorithms like Negascout [**?**] or MTD-f[**?**]), one needs to know the value of a move at the maximal depth. If it does not correspond to the end of the game, there is a need for a evaluation heuristic. Some may be straight forward, like the resulting value of an exchange in pieces points. But some strategies sacrifice a queen in favor of a more advantageous tactical position or a checkmate, so evaluation heuristics need to take tactical positions into account. In Deep Blue, the evaluation function had 8000 cases, with 4000 positions in the openings book, all learned from 700,000 grandmaster games [**?**]. Nowadays, Chess programs are better than deep blue and generally also search less positions. For instance, Pocket Fritz (HIARCS engine) beats current grandmasters [**??**] while evaluating 20,000 positions per second (740 MIPS on a smartphone) against Deep Blue's (11.38 GFlops) 200 millions per second.

Go, Monte-Carlo Tree Search

With an estimated number of legal 19x19 Go positions of $2.081681994 * 10^{170}$ [**?**] (1.196% of possible positions), and an average branching factor above Chess for gobans* from 9x9 and above, Go sets another limit for AI. For 19x19 gobans, the game tree complexity is up to 10^{360} [**?**]. The branching factor varies greatly, from ≈ 30 to 300 (361 cases at first), while the mean depth (number of plies in a game) is between 150 to 200. Approaches other than systematic exploration of the game tree are required. One of them is Monte Carlo Tree Search (MCTS*). Its principle is to randomly sample which nodes to expand and which to exploit in the search tree, instead of systematically expanding the build tree as in Minimax. For a given *node* in the search tree, we note $Q(node)$ the sum of the simulations rewards on all the runs through *node*, and $N(node)$ the visits count of *node*. Algorithm 3 details the MCTS algorithm and Fig. 2.4

explains the principle.

Algorithm 3 Monte-Carlo Tree Search algorithm. ExpandFrom(*node*) is the tree (growing) policy function on how to select where to search from situation *node* (exploration or exploitation?) and how to expand the game tree (deep-first, breadth-first, heuristics?) in case of untried actions. Evaluate(*tree*) may have 2 behaviors: **1.** if *tree* is complete (terminal), it gives an evaluation according to games rules, **2.** if *tree* is incomplete, it has to give an estimation, either through simulation (for instance play at random) or an heuristic. BestChild picks the action that leads to the better value/reward from *node*. Merge(*node*, *tree*) changes the existing tree (with *node*) to take all the $Q(\nu)\forall\nu \in tree$ (new) values into account. If *tree* contains new nodes (there were some exploration), they are added to *node* at the right positions.

function MCTS(*node*)
  while computational time left **do**
    $tree \leftarrow$ ExpandFrom(*node*)
    $tree.values \leftarrow$ Evaluate(*tree*)
    Merge(*node*, *tree*)
  end while
  return BestChild(*node*)
end function

The MoGo team [**??**] introduced Upper Confidence Bounds for Trees (UCT*) for MCTS* in Go AI. MoGo became the best 9x9 and 13x13 Go program, and the first to win against a pro on 9x9. UCT specializes MCTS in that it specifies ExpandFrom (as in Algorithm. 4) tree policy with a specific exploration-exploitation trade-off. UCB1 [**?**] views the tree policy as a multi-armed bandit problem and so ExpandFrom(*node*) UCB1 chooses the arms with the best upper confidence bound:

$$\underset{c \in node.children}{\arg\max} \frac{Q(c)}{N(c)} + k\sqrt{\frac{\ln N(node)}{N(c)}}$$

in which k fixes the exploration-exploitation trade-off: $\frac{Q(c)}{N(c)}$ is simply the average reward when going through c so we have exploitation only for $k = 0$ and exploration only for $k = \infty$.

? showed that the probability of selecting sub-optimal actions converges to zero and so that UCT MCTS converges to the minimax tree and so is optimal. Empirically, they found several convergences rates of UCT to be in $O(b^{d/2})$, as fast as Alpha-beta tree search, and able to deal with larger problems (with some error). For a broader survey on MCTS methods, see [**?**].

Algorithm 4 UCB1 ExpandFrom

function ExpandFrom(*node*)
  if *node* is terminal **then**
    return *node* `// terminal`
  end if
  if $\exists c \in node.children\ s.t.\ N(c) = 0$ **then**
    return c `// grow`
  end if
  return ExpandFrom($\arg\max_{c \in node.children)} \frac{Q(c)}{N(c)} + k\sqrt{\frac{\ln N(node)}{N(c)}}$) `// select`
end function

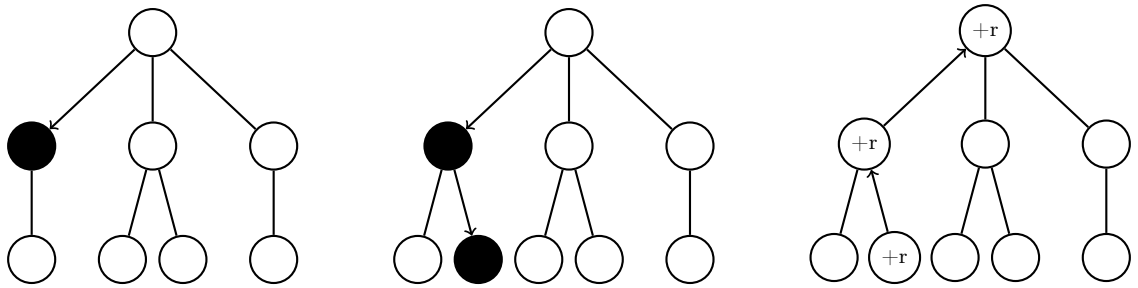


Figure 2.4: An iteration of the **while** loop in MCTS, from left to right: ExpandFrom select, ExpandFrom grow, Evaluate & Merge.

With Go, we see clearly that humans do not play abstract strategy games using the same approach. Top Go players can reason about their opponent's move, but they seem to be able to do it in a qualitative manner, at another scale. So, while tree search algorithms help a lot for tactical play in Go, particularly by integrating openings/ending knowledge, pattern macthing algorithms are not yet at the strategical level of human players. When a MCTS algorithm learns something, it stays at the level of possible actions (even considering positional hashing*), while the human player seems to be able to generalize, and re-use heuristics learned at another level.

2.4 Games with Uncertainty

An exhaustive list of games or even of games genres is beyond the scope/range of this thesis. All uncertainty boils down to incompleteness of information, being it the physics of the dice being thrown or the inability to measure what is happening in the opponent's brain. However, we will speak of 2 types (sources) of uncertainty: extensional uncertainty, which is due to incompleteness in direct, measurable information, and intentional uncertainty, which is related to randomness in the game or in (the opponent's) decisions. As extreme illustrations of this: an agent acting without sending is under full extensional uncertainty, while an agent whose acts are the results of a perfect random generator is under full intentional uncertainty. The uncertainty coming from the opponent's mind/cognition lies in between, depending on the simplicity to model the game as an optimization procedure. The harder the game is to model, the harder it is to model the trains of thoughts our opponents can follow.

Monopoly

In Monopoly, there is few hidden information (*Chance* and *Community Chest* cards only), but there is randomness in the throwing of dice[3], and a substantial influence of the player's knowledge of the game. A very basic playing strategy would be to just look at the ruturn on investment (ROI) with regard to prices, rents and frequencies, choosing only based on the money you have and the possible actions of buying or not. A less naive way to play should evaluate the questions of buying with regard to what we already own, what others own, our cash and advancement

[3]Note that the sum of two uniform distributions is not a uniform but a Irwin-Hall, $\forall n > 1, \mathrm{P}([\sum_{i=1}^{n}(X_i \in U(0,1))] = x) \propto \frac{1}{(n-1)!}\sum_{k=0}^{n}(-1)^k\binom{n}{k}\max\left((x-k)^{n-1}, 0\right)$, converging towards a Gaussian (central limit theorem).

in the game. The complete state space is huge (places for each players × their money × their possessions), but according to **?**, we can model the game for one player (as he has no influence on the dice rolls and decisions of others) as a Markov process on 120 ordered pairs: 40 board spaces × possible number of doubles rolled so far in this turn (0, 1, 2). With this model, it is possible to compute more than simple ROI and derive applicable and interesting strategies. So, even in monopoly, which is not lottery playing or simple dice throwing, a simple probabilistic modeling yields a robust strategy. Additionally, **?** used genetic algorithms to generate the most efficient strategies for portfolio management.

Monopoly is an example of a game in which we have complete direct information about the state of the game, intentional uncertainty due to the roll of the dice (randomness) can be dealt with thanks to probabilistic modeling (Markov processes here). The opponent's actions are relatively easy to model due to the fact that the goal is to maximize cash and that there are not many different efficient strategies (not many Nash equilibrium if it were a stricter game) to attain it. In general, the presence of chance does not invalidates previous (game theoretic / game trees) approaches but transforms exact computational techniques into stochastic ones: finite states machines become probabilistic Bayesian networks for instance.

Battleship

Battleship (also called "naval combat") is a guessing game generally played with two 10×10 grids for each players: one is the player's ships grid, and one is to remember/infer the opponent's ships positions. The goal is to guess where the enemy ships are and sink them by firing shots (torpedoes). There is **incompleteness** of information but no randomness. Incompleteness can be dealt with with probabilistic reasoning. The classic setup of the game consist in two ships of length 3 and one ship of each lengths of 2, 4 and 5; in this setup, there are 1,925,751,392 possible arrangements for the ships. The way to take advantage of all possible information is to update the probability that there is a ship for all the squares each time we have additional information. So for the 10×10 grid we have a 10×10 matrix $O_{1:10,1:10}$ with $O_{i,j} \in \{true, false\}$ being the ith row and jth column random variable of the case being occupied. With *ships* being unsunk ships, we always have:

$$\sum_{i,j} \mathrm{P}(O_{i,j} = true) = \frac{\sum_{k \in ships} length(k)}{10 \times 10}$$

For instance for a ship of size 3 alone at the beginning we can have prior distributions on $O_{1:10,1:10}$ by looking at combinations of its placements (see Fig. 2.5). We can also have priors on where the opponent likes to place her ships. Then each round we will either hit or miss in i, j. When we hit, we know $\mathrm{P}(O_{i,j} = true) = 1.0$ and will have to revise the probabilities of surrounding areas, and everywhere if we learned the size of the ship, with possible placement of ships. If we did not sunk a ship, the probabilities of uncertain (not 0.0 or 1.0) positions around i, j will be highered according to the sizes of remaining ships. If we miss, we know $\mathrm{P}([O_{i,j} = false]) = 1.0$ and can also revise (lower) the surrounding probabilities, an example of that effect is shown in Fig. 2.5.

Battleship is a game with few intensional uncertainty (no randomness), particularly because the goal quite strongly conditions the action (sink ships as fast as possible) but a large part of extensional uncertainty (incompleteness of direct information), which goes down rather quickly

once we act, if we update a probabilistic model of the map/grid. If we compare Battleship to a variant in which we could see the adversary board, playing would be straightforward (just hit ships we know the position on the board), now in real Battleship we have to model our uncertainty due to the incompleteness of information, without even beginning to take into account the psychology of the opponent in placement as a prior. The cost of solving an imperfect information game increases greatly from its perfect information variant: it seems to be easier to model stochasticity (chance, a source of randomness) than to model a hidden (complex) system for which we only observe (indirect) effects.

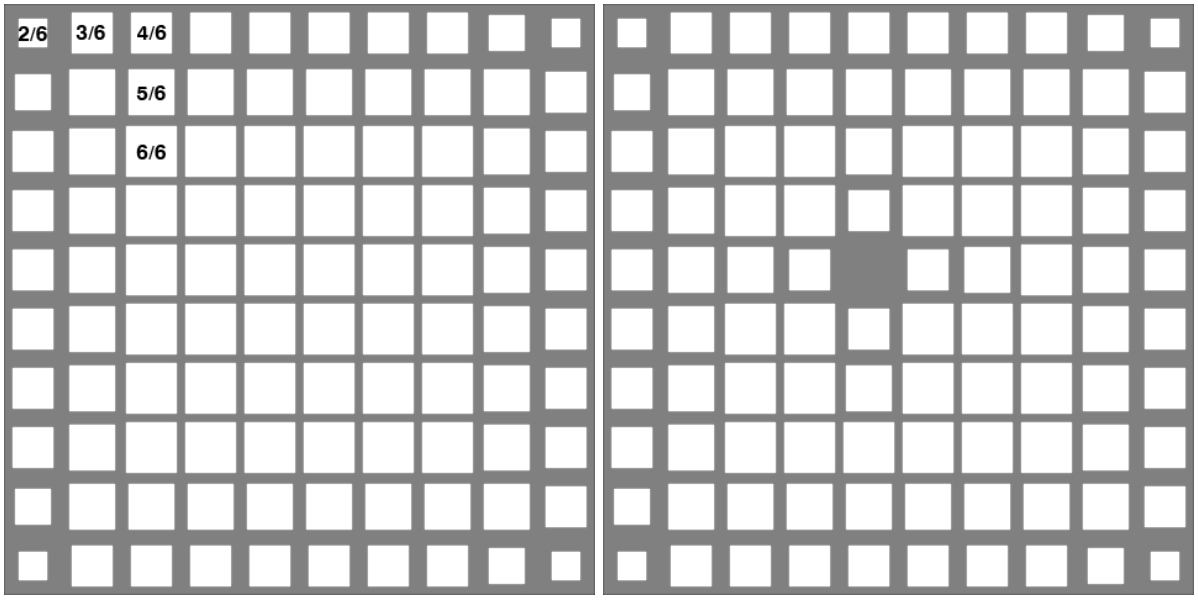


Figure 2.5: Left: visualization of probabilities for squares to contain a ship of size 3 ($\mathrm{P}(O_{i,j}) = \mathit{true}$) initially, assuming uniform distribution of this type of ship. Annotations correspond to the *number of combinations* (on six, the maximum number of conformations), Right: same probability after a miss in (5, 5). The larger the white area, the higher the probability.

Poker

Poker[4] is a zero-sum (without the house's cut), imperfect information and stochastic betting game. Poker "AI" is as old as game theory [**?**], but the research effort for human-level Poker AI started in the end of the 90s. The interest for Poker AI is such that there are annual AAAI computer Poker competitions[5]. **?** defend Poker as an interesting game for decision-making research, because the task of building a good/high level Poker AI (player) entails to take decisions with incomplete information about the state of the game, incomplete information about the opponents' intentions, and model their thoughts process to be able to bluff efficiently. A Bayesian network can combine these uncertainties and represent the player's hand, the opponents' hands and their playing behavior conditioned upon the hand, as in [**?**]. A simple "risk evaluating" AI (folding and raising according to the outcomes of its hands) will not prevail against good human

[4]We deal mainly with the *Limit Hold'em* variant of Poker.
[5]`http://www.computerpokercompetition.org/`

players. Bluffing, as described by **?** "to create uncertainty in the opponent's mind", is an element of Poker which needs its own modeling. **?** also give a Bayesian treatment to Poker, separating the uncertainty resulting from the game (draw of cards) and from the opponents' strategies, and focusing on bluff. From a game theoretic point of view, Poker is a Bayesian game*. In a Bayesian game, the normal form representation requires describing the strategy spaces, type spaces, payoff and belief functions for each player. It maps to all the possible game trees along with the agents' information state representing the probabilities of individual moves, called the extensive form. Both these forms scale very poorly (exponentially). **?** used the sequence form transformation, the set of realization weights of the sequences of moves[6], to search over the space of randomized strategies for Bayesian games automatically. Unfortunately, strict game theoretic optimal strategies for full-scale Poker are still not tractable this way, two players Texas Hold'em having a state space $\approx O(10^{18})$. **?** approximated the game-theoretic optimal strategies through abstraction and are able to beat strong human players (not world-class opponents).

Poker is a game with both extensional and intentional uncertainty, from the fact that the opponents' hands are hidden, the chance in the draw of the cards, the opponents' model about the game state and their model about our mental state(s) (leading our decision(s)). While the iterated reasoning ("if she does A, I can do B") is (theoretically) finite in Chess due to perfect information, it is not the case in Poker ("I think she thinks I think..."). The combination of different sources of uncertainty (as in Poker) makes it complex to deal with it (somehow, the sources of uncertainty must be separated), and we will see that both these sources of uncertainties arise (at different levels) in video games.

2.5 FPS

Gameplay and AI

First person shooters gameplay* consist in controlling an agent in first person view, centered on the weapon, a gun for instance. The firsts FPS popular enough to bring the genre its name were Wolfeinstein 3D and Doom, by iD Software. Other classic FPS (series) include Duke Nukem 3D, Quake, Half-Life, Team Fortress, Counter-Strike, Unreal Tournament, Tribes, Halo, Medal of Honor, Call of Duty, Battlefield, etc. The distinction between "fast FPS" (e.g. Quake series, Unreal Tournament series) and others is made on the speed at which the player moves. In "fast FPS", the player is always jumping, running much faster and playing more in 3 dimensions than on discretely separate 2D ground planes. Game types include (but are not limited to):

- single-player missions, depending of the game design.
- capture-the-flag (CTF), in which a player has to take the flag inside the enemy camp and bring it back in her own base.
- free-for-all (FFA), in which there are no alliances.
- team deathmatch (TD), in which two (or more) teams fight on score.
- various gather and escort (including hostage or payload modes), in which one team has to find and escort something/somebody to another location.

[6] for a given player, a sequence is a path down the game tree isolating only moves under their control

- duel/tournament/deathmatch, 1 vs 1 matches (mainly "fast FPS").

From these various game types, the player has to maximize its damage (or positive actions) output while staying alive. For that, she will navigate her avatar in an uncertain environment (partial information and other players intentions) and shoot (or not) at targets with specific weapons.

Some games allow for instant (or delayed, but asynchronous to other players) respawn (recreation/rebirth of a player), most likely in the "fast FPS" (Quake-like) games, while in others, the player has to wait for the end of the round to respawn. In some games, weapons, ammunitions, health, armor and items can be picked on the ground (mainly "fast FPS"), in others, they are fixed at the start or can be bought in game (with points). The map design can make the gameplay vary a lot, between indoors, outdoors, arena-like or linear maps. According to maps and gameplay styles, combat may be well-prepared with ambushes, sniping, indirect (zone damages), or close proximity (even to fist weapons). Most often, there are strong tactical positions and effective ways to attack them.

While "skill" (speed of the movements, accuracy of the shots) is easy to emulate for an AI, team-play is much harder for bots and it is always a key ability. Team-play is the combination of distributed evaluation of the situation, planning and distribution of specialized tasks. Very high skill also requires integrating over enemy's tactical plans and positions to be able to take indirect shots (grenades for instance) or better positioning (coming from their back), which is hard for AI too. An example of that is that very good human players consistently beat the best bots (nearing 100% accuracy) in Quake III (which is an almost pure skill "fast FPS"), because they take advantage of being seen just when their weapons reload or come from their back. Finally, bots which equal the humans by a higher accuracy are less fun to play with: it is a frustrating experience to be shot across the map, by a bot which was stuck in the door because it was pushed out of its trail.

State of the art

FPS AI consists in controlling an agent in a complex world: it can have to walk, run, crouch, jump, swim, interact with the environment and tools, and sometimes even fly. Additionally, it has to shoot at moving, coordinated and dangerous, targets. On a higher level, it may have to gather weapons, items or power-ups (health, armor, etc.), find interesting tactical locations, attack, chase, retreat...

The Quake III AI is a standard for Quake-like games [**?**]. It consists in a layered architecture (hierarchical) FSM. At the sensory-motor level, it has an Area Awareness System (AAS) which detects collisions, accessibility and computes paths. The level above provides intelligence for chat, goals (locations to go to), goals and weapons selection through fuzzy logic. Higher up, there are the behavior FSM ("seek goals", chase, retreat, fight, stand...) and production rules (if-then-else) for squad/team AI and orders. A team of bots always behaves following the orders of one of the bots. Bots have different natures and tempers, which can be accessed/felt by human players, specified by fuzzy relations on "how much the bot wants to do, have, or use something". A genetic algorithm was used to optimize the fuzzy logic parameters for specific purposes (like performance). This bot is fun to play against and is considered a success, however Quake-like games makes it easy to have high level bots because very good players have high accuracy (no

fire spreading), so they do not feel cheated if bots have a high accuracy too. Also, the game is mainly indoors, which facilitates tactics and terrain reasoning. Finally, cooperative behaviors are not very evolved and consist in acting together towards a goal, not with specialized behaviors for each agent.

The more recent FPS games have dealt with these limitations by using combinations of STRIPS planning (F.E.A.R. [?]), hierarchical task networks (HTN) (Killzone 2 [?] and ArmA [?]), Behavior trees (Halo 2 [?]). Left4Dead (a cooperative PvE FPS) uses a global supervisor of the AI to set the pace of the threat to be the most enjoyable for the player [?].

In research, ? focused on learning rules for opponent modeling, planning and reactive planning (on Quake), so that the robot builds plan by anticipating the opponent's actions. ?? used robotics inspired Bayesian models to quickly learn the parameters from human players (on Unreal Tournament). ? and ? applied evolutionary neural networks to optimize Quake III bots. Predicting opponents positions is a central task to believability and has been solved successfully using particle filters [?] and other models (like Hidden Semi-Markov Models) [?]. Multi-objective neuro-evolution [??] combines learning from human traces with evolutionary learning for the structure of the artificial neural network, and leads to realistic (human-like) behaviors, in the context of the BotPrize challenge (judged by humans) [?].

Challenges

Single-player FPS campaigns immersion could benefit from more realistic bots and clever squad tactics. Multi-player FPS are competitive games, and a better game AI should focus on:

- **believability** requires the AI to take decisions with the same informations than the human player (fairness) and to be able to deal with unknown situation.
- **surprise** and unpredictability is key for both performance and the long-term interest in the human player in the AI.
- **performance**, to give a challenge to human players, can be achieved through cooperative, planned and specialized behaviors.

2.6 (MMO)RPG

Gameplay and AI

Inspired directly by tabletop and live action role-playing games (Dungeon & Dragons) as new tools for the game masters, it is quite natural for the RPG to have ended up on computers. The firsts digital RPG were text (Wumpus) or ASCII-art (Rogue, NetHack) based. The gameplay evolved considerably with the technique. Nowadays, what we will call a role playing game (RPG) consist in the incarnation by the human player of an avatar (or a team of avatars) with a class: warrior, wizard, rogue, priest, etc., having different skills, spells, items, health points, stamina/energy/mana (magic energy) points. Generally, the story brings the player to solve puzzles and fight. In a fight, the player has to take decisions about what to do but plays a lesser role in performing the action than in a FPS game. In a FPS, she has to move the character (egocentrically) and aim to shoot; in a RPG, she has to position itself (often way less precisely

and continually) and just decide which ability to use on which target (or a little more for "action RPG"). Classic RPG include: Fallout, The Elders Scrolls (from Arena to Skyrim), Secret of Mana, Zelda, Final Fantasy, Diablo, Baldur's Gate. A MMORPG (e.g. World of Warcraft, AION or EVE Online) consist in a role-playing game in a persistent, multi-player world. There usually are players-run factions fighting each others' (PvP) and players versus environment (PvE) situations.
glosPvE may be a cooperative task in which human players fight together against different NPC, and in which the cooperation is at the center of the gameplay. PvP is also a cooperative task, but more policy and reactions-based than a trained and learned choregraphy as for PvE. We can distinguish three types (or modality) of NPC which have different game AI needs:

- world/neutral/civilian NPC: gives quests, takes part in the world's or game's story, talks,
- "mob"/hostile NPC that the player will fight,
- "pets"/allied NPC: acts by the players' sides.
- persistent character AI could maintain the players' avatars in the world when they are disconnected.

NPC acting strangely are sometimes worse for the player's immersion than immobile and dull ones. However, it is more fun for the player to battle with hostile NPC which are not too dumb or predictable. Players really expect allied NPC to at least not hinder them, and it is even better when they adapt to what the player is doing.

State of the art

Methods used in FPS* are also used in RPG*. The needs are sometimes a little different for RPG games, for instance RPG need interruptible behaviors, behaviors that can be stopped and resumed (dialogs, quests, fights...) that is. RPG also require stronger story-telling capabilities than other gameplay genres, for which they use (logical) story representations (ontologies) [**??**]. Behavior multi-queues [**?**] resolve the problems of having resumable, collaborative, real-time and parallel behavior, and tested their approach on Neverwinter Nights. Basically they use prioritized behavior queues which can be inserted (for interruption and resumption) in each others. AI directors are control programs tuning the difficulty and pace of the game session in real-time from player's metrics. **?** used an AI director to adapt the difficulty of Dark Spore to the performance (interactions and score) of the player in real-time.

Challenges

There are mainly two axes for RPG games to bring more fun: interest in the game play(s), and immersion. For both these topics, we think game AI can bring a lot:

- **believability** of the agents will come from AI approaches than can deal with new situations, being it because they were not dealt with during game development (because the "possible situations" space is too big) or because they were brought by the players' unforeseeable actions. Scripts and strict policies approaches will be in difficulty here.

- **interest** (as opposed to boredom) for the human players in the game style of the AI will come from approaches which can generate different behaviors in a given situation. Expectable AI particularly affects replayability negatively.
- **performance** relative to the gameplay will come from AI approaches than can fully deal with cooperative behavior. One solution is to design mobs to be orders of magnitude stronger (in term of hit points and damages) than players characters, or more numerous. Another, arguably more entertaining, solution is to bring the mobs behavior to a point where they are a challenge for the team of human players.

Both believability and performance require to deal with uncertainty of the game environment. RPG AI problem spaces are not tractable for a frontal (low-level) search approach nor are there few enough situations to consider to just write a bunch of script and puppeteer artificial agents at any time.

2.7 RTS

As RTS are the central focus on this thesis, we will discuss specific problems and solution more in depth in their dedicated chapters, simply brushing here the underlying major research problems. Major RTS include the Command&Conquer, Warcraft, StarCraft, Age of Empires and Total Annihilation series.

Gameplay and AI

RTS gameplay consist in gathering resources, building up an economic and military power through growth and technology, to defeat your opponent by destroying his base, army and economy. It requires dealing with strategy, tactics, and units management (often called micro-management) in real-time. Strategy consist in what will be done in the long term as well as predicting what the enemy is doing. It particularly deals with the economy/army trade-off estimation, army composition, long-term planning. The three aggregate indicators for strategy are aggression, production, and technology. The tactical aspect of the gameplay is dominated by military moves: when, where (with regard to topography and weak points), how to attack or defend. This implies dealing with *extensional* (what the invisible units under "fog of war*" are doing) and *intentional* (what will the visible enemy units do) uncertainty. Finally, at the actions/motor level, micro-management is the art of maximizing the effectiveness of the units *i.e.* the damages given/damages received ratio. For instance: retreat and save a wounded unit so that the enemy units would have to chase it either boosts your firepower or weakens the opponent's. Both [**?**] and **?** propose that RTS AI is one of the most challenging genres, because all levels in the hierarchy of decisions are of importance.

More will be said about RTS in the dedicated chapter **??**.

State of the art & Challenges

? called for AI research in RTS games and identified the technical challenges as:

- **adversarial planning under uncertainty**, and for that abstractions have to be found both allowing to deal with partial observations and to plan in real-time.

- **learning and opponent modeling**: adaptivity plays a key role in the strength of human players.

- **spatial and temporal reasoning**: tactics using the terrain features and good timings are essential for higher level play.

To these challenges, we would like to add the difficulty of inter-connecting all special cases resolutions of these problems: both for the collaborative (economical and logistical) management of the resources, and for the sharing of uncertainty quantization in the decision-making processes. Collaborative management of the resources require arbitrage between sub-models on resources repartition. By sharing information (and its uncertainty) between sub-models, decisions can be made that account better for the whole knowledge of the AI system. This will be extended further in the next chapters as RTS are the main focus of this thesis.

2.8 Games Characteristics

All the types of video games that we saw before require to deal with imperfect information and sometimes with randomness, while elaborating a strategy (possibly from underlying policies). From a game theoretic point of view, these video games are close to what is called a Bayesian game* [**?**]. However, solving Bayesian games is non-trivial, there are no generic and efficient approaches, and so it has not been done formally for card games with more than a few cards. **?** approximated a game theoretic solution for Poker through abstraction heuristics, it leads to believe than game theory can be applied at the higher (strategic) abstracted levels of video games.

We do not pretend to do a complete taxonomy of video games and AI (e.g. [**?**]), but we wish to provide all the major informations to differentiate game genres (gameplays). To grasp the challenges they pose, we will provide abstract measures of complexity.

Combinatory

"How does the state of possible actions grow?" To measure this, we used a measure from perfect information zero-sum games (as Checkers, Chess and Go): the branching factor b and the depth d of a typical game. The complexity of a game (for taking a decision) is proportional to b^d. The average branching factor for a board game is easy to compute: it is the average number of possible moves for a given player. For Poker, we set $b = 3$ for *fold, check* and *raise*. d should then be defined over some time, the average number of events (decisions) per hour in Poker is between 20 to 240. For video-games, we defined b to be the average number of possible moves at each decision, so for "continuous" or "real-time" games it is some kind of function of the useful discretization of the virtual world at hand. d has to be defined as a frequency at which a player (artificial or not) has to take decisions to be competitive in the game, so we will give it in $d/time\_unit$. For instance, for a car (plane) racing game, $b \approx 50 - 500$ because b is a combination of throttle ($\ddot{x}$) and direction (θ) sampled values that are relevant for the game world, with d/min at least 60: a player needs to correct her trajectory *at least* once a second. In RTS games, $b \approx 200$ is a lower bound (in StarCraft we may have between 50 to 400 units to

control), and very good amateurs and professional players perform more than 300 actions per minute.

The sheer size of b and d in video games make it seem intractable, but humans are able to play, and to play well. To explain this phenomenon, we introduce "vertical" and "horizontal" continuities in decision making. Fig. 2.6 shows how one can view the decision-making process in a video game: at different time scales, the player has to choose between strategies to follow, that can be realized with the help of different tactics. Finally, at the action/output/motor level, these tactics have to be implemented one way or another. So, matching Fig. 2.6, we could design a Bayesian model:

- $S^{t,t-1} \in Attack, Defend, Collect, Hide$, the strategy variable
- $T^{t,t-1} \in Front, Back, Hit-and-run, Infiltrate$, the tactics variable
- $A^{t,t-1} \in low\_level\_actions$, the actions variable
- $O^t_{1:n} \in \{observations\}$, the set of observations variables

$$\mathrm{P}(S^{t,t-1}, T^{t,t-1}, A^{t,t-1}, O^t_{1:n}) = \mathrm{P}(S^{t-1}).\mathrm{P}(T^{t-1}).\mathrm{P}(A^{t-1})$$
$$.\mathrm{P}(O^t_{1:n}).\mathrm{P}(S^t|S^{t-1}, O^t_{1:n}).\mathrm{P}(T^t|S^t, T^{t-1}, O^t_{1:n}).\mathrm{P}(A^t|T^t, A^{t-1}, O^t_{1:n})$$

- $\mathrm{P}(S^t|S^{t-1}, O^t_{1:n})$ should be read as "the probability distribution on the strategies at time t is determined/influenced by the strategy at time $t-1$ and the observations at time t".
- $\mathrm{P}(T^t|S^t, T^{t-1}, O^t_{1:n})$ should be read as "the probability distribution on the tactics at time t is determined by the strategy at time t, tactics at time $t-1$ and the observations at time t".
- $\mathrm{P}(A^t|T^t, A^{t-1}, O^t_{1:n})$ should be read as "the probability distribution on the actions at time t is determined by tactics at time t, the actions at time $t-1$ and the observations at time t".

Vertical continuity

In the decision-making process, vertical continuity describes when taking a higher-level decision implies a strong conditioning on lower-levels decisions. As seen on Figure 2.7: higher level abstractions have a strong conditioning on lower levels. For instance, and in non-probabilistic terms, if the choice of a strategy s (in the domain of S) entails a strong reduction in the size of the domain of T, we consider that there is a vertical continuity between S and T. There is vertical continuity between S and T if $\forall s \in \{S\}, \{\mathrm{P}(T^t|[S^t = s], T^{t-1}, O^t_{1:n}) > \epsilon\}$ is sparse in $\{\mathrm{P}(T^t|T^{t-1}, O^t_{1:n}) > \epsilon\}$. There can be local vertical continuity, for which the previous statement holds only for one (or a few) $s \in \{S\}$, which are harder to exploit. Recognizing vertical continuity allows us to explore the state space efficiently, filtering out absurd considerations with regard to the higher decision level(s).

Examples? To edit, to choose with Pierre, ideas (+ point out example "filtering out"):
navigation: S=always go north, T -> limited
whatever 2 player game: S=imitate, T -> limited

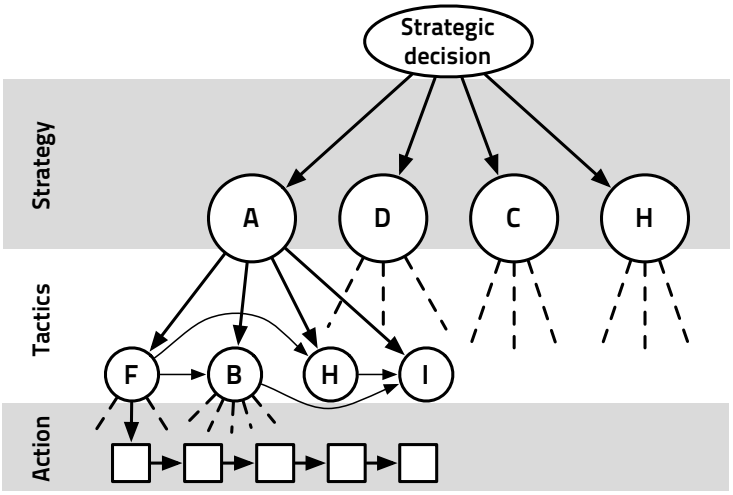


Figure 2.6: Abstract decision hierarchy in a video game. It is segmented in abstraction levels: at the strategical level, a decision is taken in *Attack*, $Defend$, *Collect* and $Hide$; at the tactical level, it is decided between $Front$, *Back*, $Hit-and-run$, $Infiltrate$; and at the actions level between the player's possible interactions with the world.

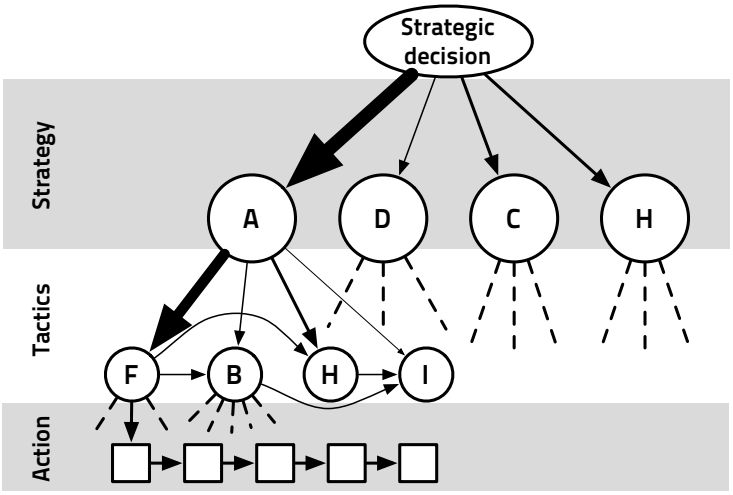


Figure 2.7: Vertical continuity in decision-making in a video game. There is a high vertical continuity between strategy and tactics as $\mathrm{P}([T^t = Front]|[S^t = Attack], T^{t-1}, O^t_{1:n})$ is much higher than other values for $\mathrm{P}(T^t|S^t, T^{t-1}, O^t_{1:n})$. The thicker the arrow, the higher the transition probability.

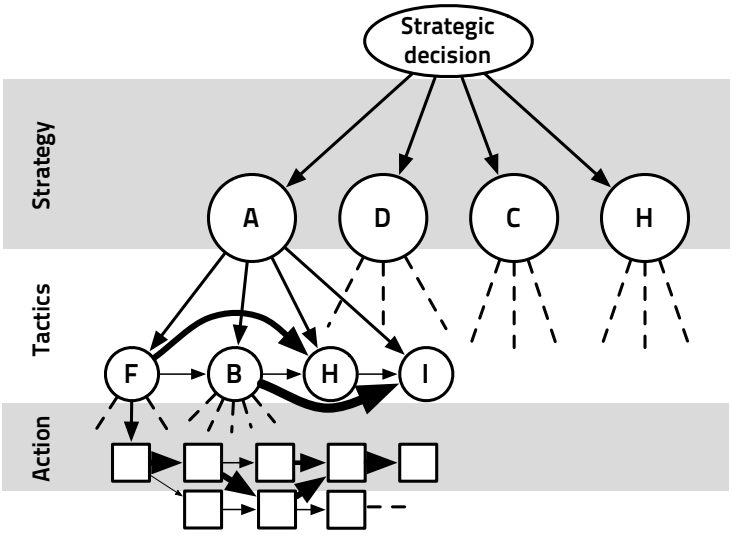


Figure 2.8: Horizontal continuity in decision-making in a video game. The thicker the arrow, the higher the transition probability.

RTS: S=short term win by all-in, T -> limited
simpler examples?

Horizontal continuity

Horizontal continuity also helps out cutting the search in the state space to only relevant states. At a given abstraction level, it describes when taking a decision implies a strong conditioning on the next time-step decision (for this given level). As seen on Figure 2.8: previous decisions on a given level have a strong conditioning on the next ones. For instance, and in non-probabilistic terms, if the choice of a tactic t (in the domain of T^t) entails a strong reduction in the size of the domain of T^{t+1}, we consider that T has horizontal continuity. There is horizontal continuity between T^{t-1} and T^t if $\forall t \in \{T\}, \{\mathrm{P}(T^t|S^t, [T^{t-1} = t], O^t_{1:n}) > \epsilon\}$ is sparse in $\{\mathrm{P}(T^t|S^t, O^t_{1:n}) > \epsilon\}$. There can be local horizontal continuity, for which the previous state holds only for one (or a few) $t \in \{T\}$. Recognizing horizontal continuity boils down to recognizing sequences of (frequent) actions from decisions/actions dynamics and use that to reduce the search space of subsequent decisions. Of course, at a sufficiently small time-step, most continuous games have high horizontal continuity: the size of the time-step is strongly related to the design of the abstraction levels for vertical continuity.

Examples? To edit, to choose with Pierre, ideas (+ point out example "filtering out"):
navigation: T^t=go round the block, T^{t+1} -> limited by future position
RTS: cannot switch between Front and Back attacks with the same army at close time-steps
simpler examples?

Randomness

Randomness can be inherent to the gameplay. In board games and table top role-playing, randomness often comes from throwing dice(s) to decide the outcome of actions. In decision-making theory, this induced stochasticity is dealt with in the framework of a Markov decision process (MDP)*. A MDP is a tuple of (S, A, T, R) with:

- S a finite set of states
- A a finite set of actions
- $T_a(s, s') = \mathrm{P}([S^{t+1} = s']|[S^t = s], [A^t = a])$ the probability that action a in state s at time t will lead to state s' at time $t + 1$
- $R_a(s, s')$ the immediate reward for going from state s to state s' with action a.

MDP can be solved through dynamic programming or the Bellman value iteration algorithm [**?**]. In video games, the sheer size of S and A make it intractable to use MDP directly on the whole AI task, but they are used (in research) either locally or at abstracted levels of decision. Randomness inherent to the process is one of the sources of intentional uncertainty, and we can consider player's intentions in this stochastic framework. Modeling this source of uncertainty is part of the challenge of writing game AI models.

Partial observations

Partial information is another source of randomness, which is found in shi-fu-mi, poker, RTS* and FPS* games, to name a few. We will not go down to the fact that the throwing of the dice seemed random because we only have partial information about its physics, or of the seed of the deterministic random generator that produced its outcome. Here, partial observations refer to the part of the game state which is deliberatively hidden between players (from a gameplay point of view): hidden hands in poker or hidden units in RTS* games due to the fog of war*. In decision-making theory, partial observations are dealt with in the framework of a partially observable Markov decision process (POMDP)* [**?**]. A POMDP is a tuple (S, A, O, T, Ω, R) with S, A, T, R as in a MDP and:

- O a finite set of observations
- Ω conditional observations probabilities specifying: $\mathrm{P}(O^{t+1}|S^{t+1}, A^t)$

In a POMDP, one cannot know exactly which state they are in and thus must reason with a probability distribution on S. Ω is used to update the distribution on S (the belief) uppon taking the action a and observing o, we have: $\mathrm{P}([S^{t+1} = s']) \propto \Omega(o|s', a).\sum_{s \in S} T_a(s, s').\mathrm{P}([S^t = s])$. In game AI, POMDP are computationally intractable for most problems, and thus we need to model more carefully (without full Ω and T) the dynamics and the information value.

Time Constant(s)

For novice to video games, we give some orders of magnitudes of the time constants involved. Indeed, we present here only real-time video games and time constants are central to comprehension of the challenges at hand. In all games, the player is taking actions continuously sampled

at the minimum of the human interface device (mouse, keyboard, pad) refreshment rate and the game engine loop: at *least* 30Hz. In most racing games, there is a quite high continuity in the input which is constrained by the dynamics of movements. In other games, there are big discontinuities, even if fast FPS* control resembles racing games a lot. RTS* professional gamers are giving inputs at ≈ 300 actions per minute (APM*).

There are also different time constants for a strategic switch to take effect. In RTS games, it may vary between the build duration of at least one building and one unit (1-2 minutes) to a lot more (5 minutes). In an RPG or a team FPS, it may even be longer (up to one full round or one game by requiring to change the composition of the team and/or the spells) or shorter by depending on the game mode. For tactical moves, we consider that the time for a decision to have effects is proportional to the mean time for a squad to go from the middle of the map (arena) to anywhere else. In RTS games, this is usually between 20 seconds to 2 minutes. Maps variability between RPG* titles and FPS* titles is high, but we can give an estimate of tactical moves to use between 10 seconds (fast FPS) to 5 minutes (some FPS, RPG).

Recap

We present the main qualitative results for big classes of gameplays (with examples) in a table page 36.

2.9 Player Characteristics

In all these games, knowledge and learning plays a key role. Humans compensate their lack of (conscious) computational power with pattern matching, abstract thinking and efficient memory structures. Particularly, we can classify required abilities for players to perform well in different gameplays by:

- Virtuosity: the technical skill or ease of the player with given game's actions and interactions. At high level of skills, there is a strong parallel with playing a music instrument. In racing games, one has to drive precisely and react quickly. In FPS* games, players have to aim and move, while fast FPS motions resemble driving. In RPG* games, players have to use skill timely as well as have good avatar placement. Finally, in RTS* games, players have to control several units (in "god's view"), from tens to hundreds and even sometimes thousands. They can use control groups, but it does not cancel the fact that they have to control both their economy and their armies.

- Deductive reasoning: the capacity to follow logical arguments, forward inference: $[A \Rightarrow B] \wedge A \Rightarrow B$. It is particularly important in Chess and Go to infer the outcomes of moves. It is important for FPS games to deduce where the enemy can be from what one has seen. In RPG games, players deduce quests solutions as well as skills/spells combinations effects. In RTS games, there is a lot of strategy and tactics that have to be infered from partial information.

- Inductive reasoning: the capacity for abstraction, generalization, going from simple observations to a more general concept. A simple example of generalization is: [Q of the sample has attribute A] $\Rightarrow$ [Q of the population has attribute A]. In all games, it is important

quantization in increasing order: no, negligible, few, some, moderate, much

Game	Combinatory	Partial information	Randomness	Vertical continuity	Horizontal continuity
Checkers	$b \approx 4-8; d \approx 70$	no	no	few	some
Chess	$b \approx 35; d \approx 80$	no	no	some	few
Go	$b \approx 30-300; d \approx 150-200$	no	no	some	moderate
Limit Poker	$b \approx 3^a$; $d/hour \in [20 \ldots 240]^b$	much	much	moderate	few
Time Racing (TrackMania)	$b \approx 50-1,000^c; d/min \approx 60+$	no	no	much	much
Team FPS (Counter-Strike) (Team Fortress 2)	$b \approx 100-2,000^d$; $d/min \approx 100^e$	some	some	some	moderate
FFPS duel (Quake III)	$b \approx 200-5,000^d$; $d/min \approx 100^e$	some	negligible	some	much
MMORPG (WoW, DAoC)	$b \approx 50-100^f$; $d/min \approx 60^g$	few	moderate	moderate	much
RTS (StarCraft)	$b \approx 200^h; d/min = APM \approx 300^i$	much	negligible	moderate	some

[a] fold,check,raise

[b] number of decisions taken per hour

[c] $\{\ddot{x} \times \theta \times \phi\}$ sampling×50Hz

[d] $\{X \times Y \times Z\}$ sampling×50Hz + firing

[e] 60 "continuous move actions"+ 40 (mean) fire actions per sec

[f] in RPGs, there are usually more abilities than in FPS games, but the player does not have to aim to hit a target, and thus positioning plays a lesser role than in FPS.

[g] move and use abilities/cast spells

[h] atomic dir/unit × # units + constructions + productions

[i] for pro-gamers, counting group actions as only one action

to be able to induce the overall strategy of the opponent's from action-level observations. In games which benefit from a really abstract level of reasoning (Chess, Go, RTS), it is particularly important as it allows to reduce the complexity of the problem at hand, by abstracting it and reasoning in abstractions.

- Decision-making: the capacity to take decisions, possibly under uncertainty and stress. Selecting a course of actions to follow while not being sure of their effect is a key ability in games, particularly in (very) partial information games as Poker (in which randomness even adds to the problem) and RTS games. It is important in Chess and Go too as reasoning about abstractions (as for RTS) brings some uncertainty about the effects of moves too.

- Knowledge: we distinguish two kinds of knowledge in games: knowledge of the game and knowledge of particular situations ("knowledge of the map"). The duality doesn't exist in Chess, Go and Poker. However, for instance for racing games, knowledge of the map is most often more important than knowledge of the game (game mechanics and game rules, which are quite simple). In FPS games, this is true too as good shortcuts, ambushes and efficient movements comes from good knowledge of the map, while rules are quite simple. In RPG, rules (skills and spells effects, durations, costs...) are much more complex to grasp, so knowlege of the game is perhaps more important. Finally, for RTS games, there are some map-specific strategies and tactics, but the game rules and overall strategies are also already complex to grasp. The longer are the rules of the game to explain, the higher the complexity of the game and thus the benefit from knowledge of the game itself.

- Psychology: the capacity to know the opponent's move by knowing them, their style, and reasoning about what they think. It is particularly important in competitive games for which there are multiple possible styles, which is not really the case for Chess, Go and racing games. As players have multiple possible valid moves, reasoning about their decision-making process and effectively predicting what they will do is what differentiate good players from the best ones.

Finally, we made a quick survey among gamers and regrouped the 108 answers in table 2.9 page 39. The goal was a to get a grasp on which skills are correlated to which gameplays. The questions that we asked can be found in Appendix A.2. Answers mostly come from good to highly competitive (on a national level) amateurs. To test the "psychology" component of the gameplay, we asked players if they could predict the next moves of their opponents by knowing them personally or by knowing what the best moves (best play) was for them. As expected, Poker has the strongest psychological dimension, followed by FPS and RTS games.

Game	Virtuosity[a] (sensory-motor) [0-2]	Deduction [b] (analysis) [0-2]	Induction [c] (abstraction) [0-2]	Decision-Making[d] (taking action) [0-2]	Opponent prediction[e] -1: subjectivity 1: objectivity	Advantageous knowledge[f] -1: map 1: game
Chess (n: 35)	X	1.794	1.486	1.657	0.371	X
Go (n: 16)	X	1.688	1.625	1.625	0.562	X
Poker (n: 22)	X	1.136	1.591	1.545	-0.318	X
Racing (n: 19)	1.842	0.263	0.211	1.000	0.500	-0.316
Team FPS (n: 40)	1.700	1.026	1.075	1.256	0.150	-0.105
Fast FPS (n: 22)	2.000	1.095	1.095	1.190	0.000	0.150
MMORPG (n: 29)	1.069	1.069	1.103	1.241	0.379	0.759
RTS (n: 86)	1.791	1.849	1.687	1.814	0.221	0.453

[a]"skill", dexterity of the player related to the game, as for a musician with their instrument.
[b]capacity for deductive reasoning; from general principle to a specific conclusion.
[c]capacity for inductive reasoning, abstraction, generalization; from specific examples to a general conclusion.
[d]capacity to select a course of actions in the presence of several alternatives, possibly under uncertainty and stress.
[e]what is the biggest indicator to predict the opponent's moves: their psychology (subjective) or the rules and logic of the game (objective)?
[f]Is knowledge of the game in general, or of the map specifically, preferable in order to play well?

Chapter 3

Bayesian Modeling of Multi-player Games

On voit, par cet Essai, que la théorie des probabilités n'est, au fond, que le bon sens réduit au calcul; elle fait apprécier avec exactitude ce que les esprits justes sentent par une sorte d'instinct, sans qu'ils puissent souvent s'en rendre compte.

One sees, from this Essay, that the theory of probabilities is basically just common sense reduced to calculus; it makes one appreciate with exactness that which accurate minds feel with a sort of instinct, often without being able to account for it.

Pierre-Simon de Laplace (Essai philosophique sur les probabilités, 1814)

HERE, we now present the use of probability theory as an alternative to logic, transforming incompleteness into uncertainty. Bayesian models can deal with both intentional and extensional uncertainty, that is: uncertainty coming from intentions of the players or the stochasticity of the game, as well as uncertainty coming from imperfect information and the model's limits. We will first show how these problems are addressed by probabilities and how to structure a full Bayesian program. We illustrate the approach with a model evaluated on a simulated MMORPG* situation.

3.1 Problems in Game AI

Partial information

If the AI has perfect information, behaving realistically is the problem. Cheating bots are not fun, so either AI should just use information available to the players, or it should fake having only partial information. In probabilistic modeling, sensor models allow for the computation of the state S from observations O by asking $\mathrm{P}(S|O)$. One can easily specify or learn $\mathrm{P}(O|S)$: either the game designers specify it, or the bot uses perfect information to get S and learns

(counts) $\mathrm{P}(O|S)$. Then infer $\mathrm{P}(S|O) = \frac{\mathrm{P}(O|S).\mathrm{P}(S)}{\mathrm{P}(O)}$ (Bayes rule). Incompleteness of information is just uncertainty about the full state.

Randomness

Some games have inherent stochasticity (variable random action effects) part of the gameplay. In any case, an AI can't assume optimal play from the opponent due to the complexity of video games. In the context of state estimation and control, dealing with this randomness require ad-hoc methods for scripting of boolean logic, while it is dealt with natively through probabilistic modeling. Where a program would have to test for an effect/value V to be in a given interval to decide on a given course of action A, a probabilistic models just computes the distribution on A given V: $\mathrm{P}(A|V) = \frac{\mathrm{P}(V|A).\mathrm{P}(A)}{\mathrm{P}(V)}$.

Vertical continuity

Abstracting higher level cognitive functions (strategy and tactics for instance) is an efficient way to break the complexity barrier of writing game AI. Exploiting the vertical continuity, i.e. the conditioning of higher level thinking on actions, is totally possible in a hierarchical Bayesian model. With strategies as values of S and tactics as values of T, $\mathrm{P}(T|S)$ gives the conditioning of T on S and thus enables us to evaluate only those T values that are possible with given S values.

Horizontal continuity

Real-time games may use discrete time-steps (24Hz for instance for StarCraft), it does not prevent temporal continuity in strategies, tactics and, most strongly, actions. There are several Bayesian models able to deal with sequences, filter models, from which Hidden Markov Models (HMM*) [?], Kalman filters [?] and particle filters [?] are specializations of. With states S and observations O, filter models under the Markov assumption represent the joint $\mathrm{P}(S^0).\mathrm{P}(O^0|S^0).\prod_{t=1}^{T}[\mathrm{P}(S^t|S^{t-1}).\mathrm{P}(O^t|S^t)]$.

Autonomy and Programming

Autonomy is the ability to deal with new states, the challenge of autonomous characters arises from state spaces too big to be fully specified (in scripts / FSM). Again, probabilistic modeling enables one to recognize unspecified states as soft mixtures of known states. It also allows to deal with unknown state as an incomplete version of a known state which subsumes it. Autonomy can also be reached through learning, being it through online or offline learning. Offline learning is used to learn parameters that does not have to be specified by the programmers and/or game designers. One can use data or experiments with known/wanted situations (supervised learning, reinforcement learning), or explore data (unsupervised learning) or game states (evolutionary algorithms). Online learning can provide adaptivity of the AI to the player and/or its own competency playing the game.

Figure 3.1: An advisor introducing his student to Bayesian modeling, adapted from Land of Lisp with the kind permission of **?**.

3.2 The Bayesian Programming Methodology

The reverend Thomas Bayes is the first credited to have worked on inversing probabilities: by knowing something about the probability of A given B, how can one draw conclusions on the probability of B given A? Bayes theorem states:

$$\mathrm{P}(A|B) = \frac{\mathrm{P}(B|A)\mathrm{P}(A)}{\mathrm{P}(B)}$$

? rediscovered this theorem and published the work of inductive reasoning by using probabilities. Later, by extending logic to *plausible reasoning*, **?** arrived at the same properties than **?** probability theory. Plausible reasoning originates from logic, whose statements have degrees of plausibility represented by real numbers. Adding consistency: a) all the possible ways to reach a conclusion leads to the same result, b) information cannot be ignored, c) two equal states of knowledge have the same plausibilities; Cox derived the "product-rule" ($\mathrm{P}(AB) = \mathrm{P}(A \cap B) = \mathrm{P}(A|B)\mathrm{P}(B)$) and the "sum-rule" ($\mathrm{P}(A + B) = \mathrm{P}(A) + \mathrm{P}(B) - \mathrm{P}(AB)$ or $\mathrm{P}(A \cup B) = \mathrm{P}(A) + \mathrm{P}(B) - \mathrm{P}(A \cap B)$) of probabilities. As for rules of inferences, the links between logic and plausible reasoning are direct, with $C = [A \Rightarrow B]$:

- modus ponens

  $$\frac{[A \Rightarrow B] \wedge [A = true]}{B = true}$$

  translates to

  $$\mathrm{P}(B|AC) = \frac{\mathrm{P}(AB|C)}{\mathrm{P}(A|C)}$$

- modus tollens

  $$\frac{[A \Rightarrow B] \wedge [B = false]}{A = false}$$

  translates to

  $$\mathrm{P}(A|C\neg B) = \frac{\mathrm{P}(A\neg B|C)}{\mathrm{P}(\neg B|C)}$$

Also, additionally to the two strong logic syllogisms above, plausible reasoning gets two weak syllogisms, from:

- $$\mathrm{P}(A|BC) = \mathrm{P}(A|C)\frac{\mathrm{P}(B|AC)}{\mathrm{P}(B|C)}$$

  we get

  $$\frac{[A \Rightarrow B] \wedge [B = true]}{A \text{ becomes more plausible}}$$

- $$\mathrm{P}(B|C\neg A) = \mathrm{P}(B|C)\frac{\mathrm{P}(\neg A|BC)}{\mathrm{P}(\neg A|C)}$$

  we get

  $$\frac{[A \Rightarrow B] \wedge [A = false]}{B \text{ becomes less plausible}}$$

Indeed, this was proved by **?**, producing Cox's theorem (also named Cox-Jayne's theorem):

Theorem. *A system for reasoning which satisfies:*

- *divisibility and comparability, the plausibility of a statement is a real number,*
- *common sense, in presence of total information, reasoning is isomorphic to Boolean logic,*
- *consistency, two identical mental states should have the same degress of plausibility,*

is isomorphic to probability theory.

So the degrees of belief of any consistent induction mechanism, verify Kolmogorov's axioms. **?** showed that if reasoning is made in a system which is not isomorphic to probability theory, then it is always possible to find a *Dutch book* (a set of bets which guarantees a profit regardless of the outcomes).

Inspired by plausible reasoning, we present Bayesian programming, a formalism that can be used to describe entirely any kind of Bayesian model. It subsumes Bayesian networks and Bayesian maps, as it is equivalent to probabilistic factor graphs **?**. There are mainly two parts in a Bayesian program (BP)*, the **description** of how to compute the joint distribution, and the **question(s)** that it will be asked.

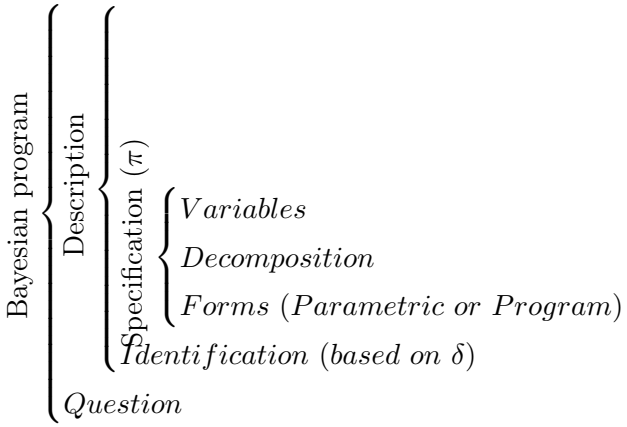
The description consists in exhibiting the relevant *variables* $\{X^1, \ldots, X^n\}$ and explain their dependencies by *decomposing* the joint distribution $\mathrm{P}(X^1 \ldots X^n|\delta, \pi)$ with existing preliminary (*prior*) knowledge π and data δ. The *forms* of each term of the product specify how to compute their distributions: either parametric forms (laws or probability tables, with free parameters that can be learned from data δ) or recursive questions to other Bayesian programs.

Answering a question is computing the distribution $\mathrm{P}(Searched|Known)$, with $Searched$ and $Known$ two disjoint subsets of the variables.

$$\begin{aligned} & \mathrm{P}(Searched|Known) && (3.1)\\ = & \frac{\sum_{Free}\mathrm{P}(Searched,\ Free,\ Known)}{\mathrm{P}(Known)} && (3.2)\\ = & \frac{1}{Z} \times \sum_{Free}\mathrm{P}(Searched,\ Free,\ Known) && (3.3) \end{aligned}$$

General Bayesian inference is practically intractable, but conditional independence hypotheses and constraints (stated in the description) often simplify the model. There are efficient ways

to calculate the joint distribution like message passing and junction tree algorithms [?????]. Also, there are different well-known approximation techniques: either by sampling with Monte-Carlo (and Monte-Carlo Markov Chains) methods [??], or by solving a simpler model which approximate the full joint as with variational Bayes [?].



For the use of Bayesian programming in sensory-motor systems, see [?]. For its use in cognitive modeling, see [?]. For its first use in video games (first person shooter gameplay, Unreal Tournament), see [?].

To sum-up, by viewing probabilities as an extension of logic, the method by which to build Bayesian models gets clearer: there is a strong parallel between writing a Bayesian program and logic or declarative programming. In order:

1. Isolate the variables of the problem: it is the first prior that the programmer puts into the system. The variables can be anything, from existing input or output values of the problem to abstract/aggregative values or parameters of the model. Discovering which variable to use for a given problem is one of the most complicated form of machine learning.

2. Suppose and describe the influences and dependencies between these variables. This is another prior that the programmer can have on the problem, and learning the structure between these variables is the second most complicated form of learning [??].

3. Fill the priors and conditional probabilities parameters. The programmer needs to be an expert of the problem to put relevant parameters, although this is the easiest to learn from data once variables and structure are specified. Learning the structure can be seen as learning the parameters of a fully connected model (and then removing dependencies where are the less influent parameters) but most of these kinds of model are totally untractable.

In the following, we will show this method applied to a simulated MMORPG* fight situation.

3.3 Modeling of a Bayesian MMORPG player

We will now present a model of a MMORPG* player with the Bayesian programming framework [?]. A role-playing game (RPG) consist in the incarnation by the human player of an

avatar with a class (warrior, wizard, rogue, priest...) having different skills, spells, items, health points, stamina/energy/mana (magic energy) points. Massively Multi-Player Online Role-Playing Games (e.g. World of Warcraft, AION, EVE Online, Star Wars: The Old Republic) consist in controlling a single character, which can earn experience, as well as equip always better items to increase its health points, damage per second and other skills (heals, enhancements...). Players are immersed in a unique and persistent (thus the "massively multi-player") world, with its history and epic battles. Players have to combat each others (PvP*) due to the factions they belong to. They also have to team-up and battle against the environment (PvE*) to aquire better items and level-up. This is a cooperative task in which human players fight together against different NPC.

More specifically here, we modeled the "druid" class, which is complex because it can cast spells to deal damages or other negative effects as well as to heal allies or enhance their capacities ("buff" them). The model described here deals only with a sub-task of a global AI for autonomous NPC. The problem that we try to solve is: how do we choose which skill to use and on which target in a PvE battle? Possible targets are all our allies and foes. Possible skills are all that we know, we will just try and get a distribution over target and skills and pick the most probable combination that is yet possible to achieve (enough energy/mana, no cooldown/reload time). For that, we first choose what should be the target given all surrounding variables: is an ally near death that he should be healed, which foe should we focus our attacks on? Once we have the distribution over possible targets, we search the distribution about our skills, pondered by the one on targets. However, some variables can be things that humans subconsciously interpolate from perceptions.

Variables

A simple set of variables is as follows:

- Target: $T^{t-1,t} \in \{t_1 \dots t_n\}$ at t and $t-1$ (previous action). T^t is also abusively noted T in the rest of the model. We have the n characters as possible targets; each of them has their own properties variables (the indicing in the following variables).

- Hit Points: $HP_1 \dots HP_n$ s.t. $HP_i \in [0 \dots 9]$. Health/Hit points (HP) are discretized in 10 levels, from the lower to the higher.

- Distance: $D_1 \dots D_n$ s.t. $D_i \in \{Contact, Close, Far, VeryFar\}$. Distance ($D$) is discretized in 4 zones around the robot character: *contact* where it can attack with a contact weapon, *close*, *far* and (*very far*) to the further for which we have to move even for shooting the longest distance weapon/spell.

- Ally: $A_1 \dots A_n$ s.t. $A_i \in \{true, false\}$. Ally (A) is a Boolean variable mentioning if character i is allied with the robot character.

- Derivative Hit Points: $\Delta HP_1 \dots \Delta HP_n$ s.t. $\Delta HP_i \in \{-, 0, +\}$. Delta hit points ($\Delta HP$) is a 3-valued interpolated value from the previous few seconds of fight that informs about the ith character getting wounded or healed (or nothing).

- Imminent Death: $ID_1 \dots ID_n$ s.t. $ID_i \in \{true, false\}$. Imminent death (ID) is also an interpolated value that encodes HP, ΔHP and incoming attacks/attackers. This is a Boolean variable saying if the ith character if going to die anytime soon. This is an example of what we consider that an experienced human player will infer automatically from the screen and notifications.

- Class: $C_1 \dots C_n$ s.t. $C_i \in \{Tank, Contact, Ranged, Healer\}$. Class ($C$), simplified over 4 values, gives the type of the ith character: a Tank can take a lot of damages and taunt enemies, a Contact damager can deal huge amounts of damage with contact weapons (rogue, barbarian...), Ranged stands for the class that deals damages from far away (hunters, mages...) and Healers are classes that can heal (in considerable amounts).

- Resists: $R_1 \dots R_n$ s.t. $R_i \in \{Nothing, Fire, Ice, Nature, FireIce, IceNat, FireNat, All\}$. The Resist variable is the combination of binary variables of resistance to certain types of (magical) damages into one variable. With 3 possible resistances we get (2^3) 8 possible values. For instance "$R_i = FireNat$" means that the ith character resists fire and nature-based damages. Armor (physical damages) could have been included, and the variables could have been separated.

- Skill: $S \in \{Skill_1 \dots Skill_m\}$. The skill variable takes here all the possible skills for the given character, and not only the available ones to cast at the moment to be able to have reusable probability tables (i.e. learnable tables).

Decomposition

Target selection

The joint distribution is simplified (by conditional independence of variables) as:

$$\mathrm{P}(T, T^{t-1}, HP_{1:n}, D_{1:n}, A_{1:n}, \Delta HP_{1:n}, ID_{1:n}, C_{1:n}) = \tag{3.4}$$

$$\mathrm{P}(T^{t-1}).\mathrm{P}(T|T^{t-1}).\prod_{i=1}^{n}[\mathrm{P}(HP_i|A_i, C_i, T).\mathrm{P}(D_i|A_i, t).\mathrm{P}(A_i|T) \tag{3.5}$$

$$\mathrm{P}(\Delta HP_i|A_i, C_i, T).\mathrm{P}(C_i|A_i, T).\mathrm{P}(ID_i|T)] \tag{3.6}$$

We want to compute the probability distribution on the variable Target (T), so we have to consider the joint distribution with all variables on which Target (T) is conditionally dependant : the previous value of Target (T^{t-1}), and all the variables on each character (except for Resists). The probability of a given target depends on the previous one (it encodes the previous decision and so all previous states). The health (or hit) points (HP_i) depends on the facts that the ith character is an ally (A_i), on his class (C_i), and if he is a target (T). Such a conditional probability table should be learned, but we can already foresee that a targeted ally with a tanking class ($C = tank$) would have a high probability of having low hit points (HP) because taking it for target means that we intend to heal him. The distance of the unit i (D_i) is more probably far if unit i is an enemy ($A_i = false$) and we target it ($T = i$) as our kind of druid attacks with ranged spells and does not fare well in the middle of the battlefield. The probability of the ith character being an ally depends on if we target allies of foes more often: that is $\mathrm{P}(A_i|T)$ encodes our propensity to target allies (with heals and enhancements) or foes (with damaging

or crippling abilities). The probability that the ith units is being damaged (losing hit points: ΔHP_i = "$-$") is higher for foes ($A_i = false$) with vulnerable classes ($C_i = healer$) that are more succeptible to be targeted ($T = i$), and also for allies ($A_i = true$) whose role is to take most of the damages ($C_i = tank$). As for A_i, the probability of ID_i is driven by our inclination of targeting characters near death. The probability of C_i is driven by the distribution of foes and allies population, tuned with a soft evidence of which classes our druid human player will target more frequently.

Skill selection

For skill selection, we need additional variables. The joint distribution of the updated model is:

$$\mathrm{P}(S, T^{t-1,t}, HP_{1:n}, D_{1:n}, A_{1:n}, \Delta HP_{1:n}, ID_{1:n}, C_{1:n}, R_{1:n}) = \tag{3.7}$$

$$\mathrm{P}(S).\mathrm{P}(T^t|T^{t-1}).\mathrm{P}(T^{t-1}).\prod_{i=1}^{n}[\mathrm{P}(HP_i|A_i, C_i, S, T).\mathrm{P}(D_i|A_i, S, T).\mathrm{P}(A_i|S, T) \tag{3.8}$$

$$\mathrm{P}(\Delta HP_i|A_i, S, T).\mathrm{P}(R_i|C_i, S, T).\mathrm{P}(C_i|A_i, S, T)] \tag{3.9}$$

As previously for targets, we are interested in the conditional probabilities of each character's state variables given other state variables and given our target (T) and the skill that we use (S). If we target the ith unit ($T = i$), happening to be an allied ($A_i = true$) tank ($C_i = tank$) with a "big heal" ($S = big\_heal$), the probability that they will have low hit points ($HP_i = 0$ or 1 (very low)) is very high. Some skills have optimal ranges to be used at and so $\mathrm{P}(D_i)$ will be affected. As we use heals $s_h \in \{heals\}$ exclusively on allies, $\mathrm{P}(A_i = true|S = s_h) = 1.0$. Conversely, as we use attacks ($s_a \in \{damage\}$) exclusively on foes, $\mathrm{P}(A_i = false|S = s_a) = 1.0$. The probability that the ith units is being damaged (ΔHP_i = "$-$") will top when we use a heal ($S = heal$) on an ally, as we want to maintain everyone alive. The one of the resist to "nature based effects" ($R_i = nature$) for skills involving "nature" damage ($s \in \{nature\_damage\}$) will be very low as it will be useless to use spells that the receiver is protected against. The probability that the ith character will die soon (ID_i) will be high if i is targeted ($T = i$) with a big heal or big instant damage ($S = big\_heal$ or $S = big\_damage$, depending on whether i is an ally or not).

Parameters

- $\mathrm{P}(T^{t-1})$ is unknown and unspecified (uniform). In the question, we always know what was the previous target, except when there was not one.
- $\mathrm{P}(T|T^{t-1})$ is a probability corresponding to the propensity to switch targets. It can be learned, or uniform if there is no previous target (it the prior on the target then).
- $\mathrm{P}(S)$ is unknown and so unspecified, it could be a prior.
- $\mathrm{P}(HP_i|A_i, C_i, S, T)$ is a $2 \times 4 \times m \times 2$ probability table, indexed on if the ith character is an ally or not, on its class, on the skills ($\#S = m$) and on where it is the target or not. It can be learned (and/or parametrized), for instance $\mathrm{P}(HP_i = x|a_i, c_i, s, t) = \frac{1+\mathrm{count}(x,a_i,c_i,s,t)}{10+\mathrm{count}(a_i,c_i,s,t)}$.
- $\mathrm{P}(D_i|A_i, S, T)$ is a $4 \times 2 \times m \times 2$ (possibly learned) probability table.
- $\mathrm{P}(A_i|S, T)$ is a $2 \times m \times 2$ (possibly learned) probability table.

- P($\Delta HP_i|A_i, S, T$) is a $3 \times 2 \times m \times 2$ (possibly learned) probability table.
- P($R_i|C_i, S, T$) is a $8 \times 4 \times m \times 2$ (possibly learned) probability table.
- P($C_i|A_i, S, T$) is a $4 \times 2 \times m \times 2$ (possibly learned) probability table.

Identification

If there were only perceived variables, learning the right conditional probability tables would just be counting and averaging. However, some variables encode combinations of perceptions and passed states. We could learn such parameters through the EM algorithm but we propose something simpler for the moment as our "not directly observed variables" are not complex to compute, we compute them from perceptions as the same time as we learn. For instance we map in-game values to their discrete values for each variables online and only store the resulting state "compression". In the following Results part however, we did not apply learning but instead manually specified the probability tables to show the effects of gamers' *common sense* rules and how it/they can be correctly encoded in this model.

Questions

In any case, we ask our (updated) model:

$$P(S, T|t^{t-1}, hp_{1:n}, d_{1:n}, a_{1:n}, \Delta hp_{1:n}, id_{1:n}, c_{1:n}, r_{1:n}) \tag{3.10}$$

Which means that we want to know the distribution on S and T knowing all the state variables. We then choose to do the highest scoring combination of $S \wedge T$ that is available (skills may have cooldowns or cost more mana/energy that we have available).

As (Bayes rule) $P(S, T) = P(S|T).P(T)$, to decompose this question, we can ask:

$$P(T|t^{t-1}, hp_{1:n}, d_{1:n}, a_{1:n}, \Delta hp_{1:n}, id_{1:n}, c_{1:n})$$

Which means that we want to know the distribution on T knowing all the relevant state variables, followed by (with the newly computed distribution on T):

$$P(S|T, hp_{1:n}, d_{1:n}, a_{1:n}, \Delta hp_{1:n}, id_{1:n}, c_{1:n}, r_{1:n})$$

in which we use this distribution on T to compute the distribution on S with:

$$P(S = skill_1|\dots) = \sum_T P(S = skill_1|T, \dots).P(T)$$

We here choose to sum over all possible values of T. Note that we did not ask:
$P(S|T = most\_probable, \dots)$ but computed instead

$$\sum_T P(S|T, hp_{1:n}, d_{1:n}, a_{1:n}, \Delta hp_{1:n}, id_{1:n}, c_{1:n}, r_{1:n})$$

In presence of several possible targets and skills, this computation may have a high complexity, so we could choose not to do the sum and use and instantiate "most probable values", for instance of Target, but there we would make a choice earlier and so lose information. There are possibly

good combinations of S and T for a value of T that is not the most probable one. This downside may be so hard that we may want to reduce the complexity of computation by simplifying our model or its computation to be able to sum. We propose a solution in the discussion.

Here is the full Bayesian program corresponding to this model:

$$
BP \begin{cases} Desc. \begin{cases} Spec.(\pi) \begin{cases} Variables \\ S, T^{t-1,t}, HP_{1:n}, D_{1:n}, A_{1:n}, \Delta HP_{1:n}, ID_{1:n}, C_{1:n}, R_{1:n} \\ Decomposition \\ \mathrm{P}(S, T^{t-1,t}, HP_{1:n}, D_{1:n}, A_{1:n}, \Delta HP_{1:n}, ID_{1:n}, C_{1:n}, R_{1:n}) = \\ \mathrm{P}(S).\mathrm{P}(T|T^{t-1}).\mathrm{P}(T^{t-1}).\prod_{i=1}^{n}[\mathrm{P}(HP_i|A_i, C_i, S, T).\mathrm{P}(D_i|A_i, S, T).\mathrm{P}(A_i|S, T) \\ \mathrm{P}(\Delta HP_i|A_i, S, T).\mathrm{P}(R_i|C_i, S, T).\mathrm{P}(C_i|A_i, S, T)] \\ Forms \\ \text{probability tables parametrized on wether } i \text{ is targeted} \end{cases} \\ Identification\ (using\ \delta) \\ \text{learning (e.g. Laplace succession law) or manually specified} \end{cases} \\ Question \\ \mathrm{P}(S, T|T^{t-1}, HP_{1:n}, D_{1:n}, A_{1:n}, \Delta HP_{1:n}, ID_{1:n}, C_{1:n}, R_{1:n}) \end{cases}
$$

Example

Figure 3.2: Example setup A (left) and B (right). There are 2 foes and 2 players taking damages ("MT" and "tank"). Players with stars can heal allies, players with dotted lines will soon die ($ID = true$). Our model controls the blue character, green players are allies, while red characters are foes. The larger the surrounding ellipse is, the more health points the characters have.

This model has been applied to a simulated situation with 2 foes and 4 allies while our robot took the part of a "druid", a versatile class that can cast spells to do direct damages, damages over time, buff (enhancements), debuff, crowd-control, heal and heal over time. We display a schema of this situation in Fig. 3.2 The arrows indicate foes attacks on allies. MT stands for "main tank", Add for "additional foe". We worked with the skills corresponding to a Druid. HOT stands for heal over time, DOT for damage over time, "abol" for abolition and "regen" for regeneration, a "buff" is an enhancement and a "dd" is a direct damage. "Root" is a spell which disables the target to move for a short period of time, useful to flee or to put some distance

between the enemy and the druid to cast attack spells. "Small" spells are usually faster to cast than "big" spells. The difference between setup A and setup B is simply to test the concurrency between healing and dealing damage and the changes in behavior if the player can lower the menace (damage dealer).

$$Skills \in \{small\_heal, big\_heal, HOT, poison\_abol, malediction\_abol,$$
$$buff\_armor, regen\_mana, small\_dd, big\_dd, DOT, debuff\_armor, root\}$$

We did not do the "Identification" part, which consists in learning the probability tables from observations. To keep things simple and because we wanted to analyze the answers of the model, we worked with manually defined probability tables, which we will change to watch their effects on the full behavior of the model. In the experiments, we will try different values for the "soft evidence that a selected target will soon die", that is $\mathrm{P}(ID_i|T = i)$, and see how the behavior of the agent changes. We set the probability to target the same target as before ($\mathrm{P}(T^t = i|T^{t-1} = i)$) to 0.4 and the previous target to "Lich" so that the prior probability for all other 6 targets is 0.1 (4 times more chances to target the Lich than any other character). We set the probability to target an enemy (instead of an ally) $\mathrm{P}(A_i = false|T = i)$ to 0.6. This means that our robotic druid is mainly a damage dealer and just a backup healer. For the "target selection" sub-model, we can see on Fig. 3.3 (left) that the evolution from selecting the main foe "Lich" to selecting the ally "Tank" is driven by the increase of "soft evidence that a selected target will soon die" and our robot eventually moves on targeting their "Tank" ally (to heal them). We can see on Fig. 3.3 (right) that, at some point, the robotic Druid prefers to kill the dying "add" (additional foe) to save their ally Tank instead of healing them. Note that there is no variable showing the relation between "Add" and "Tank" (the first is attacking the second, who is taking damages from the first), but this could be taken into account in a more complete model.

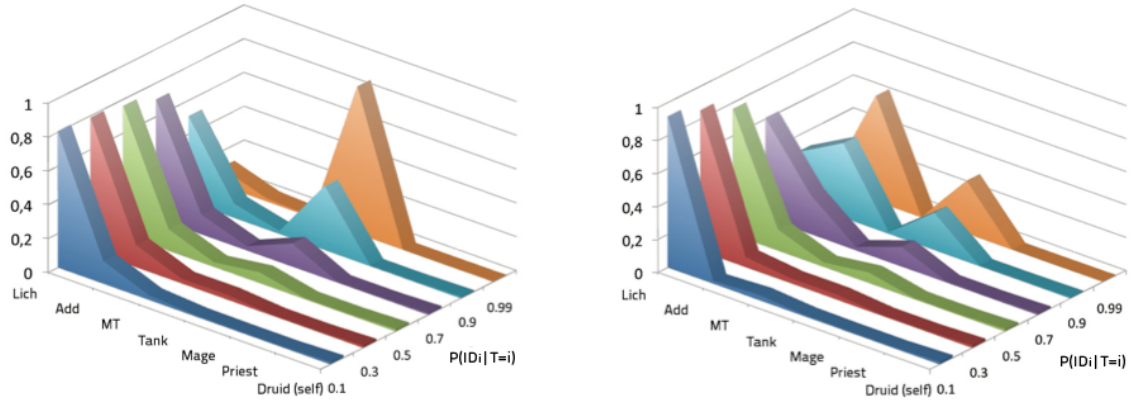


Figure 3.3: Left: probabilities of targets depending on the soft evidence that a target is dying ($\mathrm{P}(ID_i = true|T = i)$) with setup A (no ally is risking death). Right: same, with setup B (the "tank" ally is risking death).

For the "skill selection" model, we can see on Fig. 3.4 the influence of ID_i on Skill which is coherent with the Target distribution: either, in setup A (left), we evolve with the increase

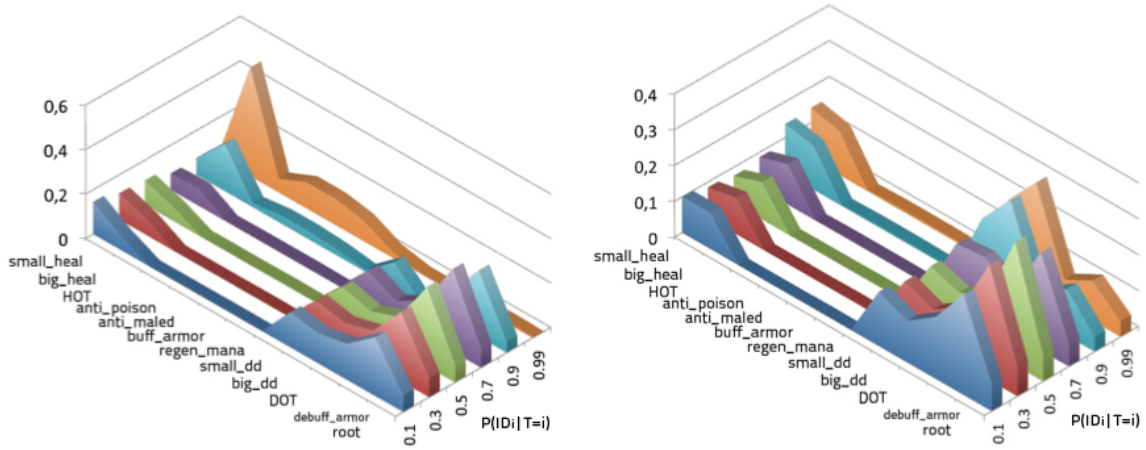


Figure 3.4: Left: Probabilities of skills depending on the soft evidence that a target is dying ($P(ID_i = true|T = i)$) with setup A (no ally is risking death). Right: same, with setup B (the "tank" ally is risking death).

of $P(ID_i = true|Target = i)$ to choose to heal our ally or, in setup B (right), to deal direct damage (and hopefully, kill) the foe attacking him. As you can see here, when we have the highest probability to attack the main enemy ("Lich", when $P(ID_i = true|Target = i)$ is low), who is a $C = tank$, we get a high probability for the Skill $debuff\_armor$. We only cast this skill if the debuff is not already present, so perhaps that we will cast $small\_dd$ instead. To conclude this example, Fig. 3.5 shows the distribution on $P(T, S|all\_status\_variables)$ with setup A and the probability to target the previous target (set to "Lich" here) only ≈ 2 times greater than any other character (so that we focus less on the same character), soft evidences $P(ID_i = true|Target = i) = 0.9$ and $P(A_i = false|Target = i) = 0.6$. On the decision-making side of this model, a simple approach would be a greedy one: if the first couple (T, S) is already done or not available, we perform the second, and so on.

Discussion

This limited model served the purpose of presenting Bayesian programming in practice. While it was used in a simulation, it showcases the approach one can take to break down the problem of autonomous control of NPC*. The choice of the skill or ability to use and the target on which to use it puts hard constraints on every others decisions the autonomous agent has to take to perform its ability action. Thus, such a model shows that:

- cooperative behavior is not too hard to incorporate in a decision (instead of being hard-coded),
- it can be learned, either from observations of a human player or by reinforcement (exploration),
- it is computationally tractable (for use in all games), the inference is just a series of "probabilistic *if*s",

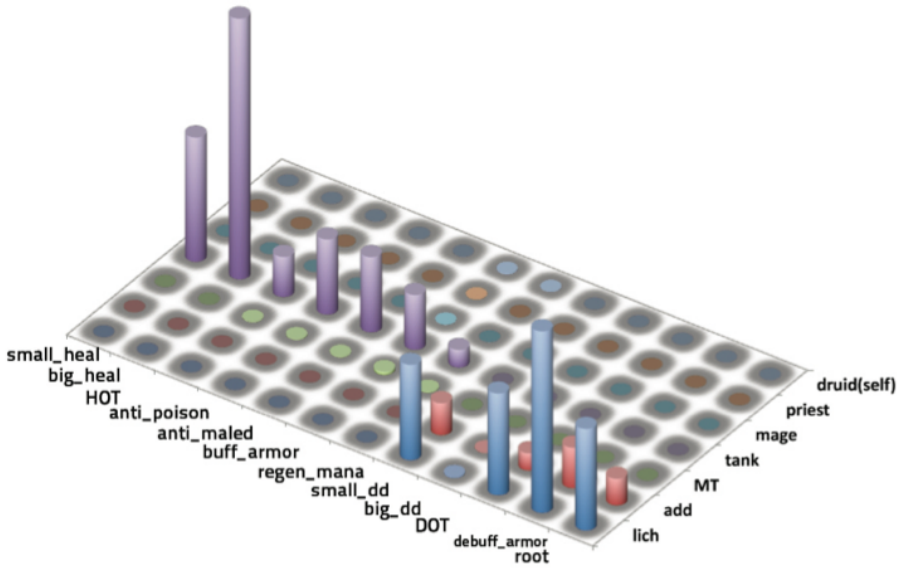


Figure 3.5: Log-probabilities of Target and Skill with setup A, and $\mathrm{P}(ID_i = true|T = i) = 0.9, \mathrm{P}(A_i = false|T = i) = 0.6$. This plot corresponds to the answer to the full question on which decision-making has to be done.

Moreover, using this model on another agent than the one controlled by the AI can give a prediction on what it will do, resulting in human-like, adaptive, playing style.

We did not kept at the research track of Bayesian modeling MMORPG games due to the difficulty to work on these types of games: the studios have too much to lose to "farmer" bots to accept any automated access to the game. Also, there are no sharing format of data (like replays) and the invariants of the game situations are fewer than in RTS games. Finally, RTS games have international AI competitions which were a good motivation to compare our approach with other game AI researchers.

Chapter 4

RTS AI: *StarCraft: Broodwar*

We think of life as a journey with a destination (success, heaven). But we missed the point. It was a musical thing, we were supposed to sing and dance while music was being played.

Alan Watts

THIS chapter explains the basics of RTS gameplay, particularly StarCraft. We then list the (computational) challenges brought by RTS gameplay. We present a transversal decomposition of the RTS AI domain in levels of abstractions (strategy, tactics, micro-management*), which we will use for the rest of the dissertation.

4.1 How does the game work

RTS Gameplay

We first introduce the basic components of a real-time strategy (RTS) game. The player is usually referred as the "commander" and perceives the world in an allocentric "God's view", performing mouse and keyboard actions to give orders to units (or squads of units) within a circumvented area (the "map"). In a RTS, players need to gather resources to build military units and defeat their opponents. To that end, they often have *worker units* (or extraction structures) than can gather resources needed to build *workers*, *buildings*, *military units* and *research upgrades*. Workers are often also builders (as in StarCraft) and are weak in fights compared to military units. Resources may have different uses, for instance in StarCraft: minerals are used for everything, whereas gas is only required for advanced buildings or military units, and technology upgrades. Buildings and research upgrades define technology trees (directed acyclic graphs) and each state of a tech tree* (or build tree*) allow for different unit type production abilities and unit spells/abilities. The military units can be of different types, any combinations of ranged,

casters, contact attack, zone attacks, big, small, slow, fast, invisible, flying... In the end, a central part of the gameplay is that units can have attacks and defenses that counter each others as in a soft rock-paper-scissors. Also, from a player point of view, most RTS games are only partially observable due to the *fog of war** which hides units and new buildings which are not in sight range of the player's units.

In chronological order, RTS include (but are not limited to): Ancient Art of War, Herzog Zwei, Dune II, Warcraft, Command & Conquer, Warcraft II, C&C: Red Alert, Total Annihilation, Age of Empires, StarCraft, Age of Empires II, Tzar, Cossacks, Homeworld, Battle Realms, Ground Control, Spring Engine games, Warcraft III, Total War, Warhammer 40k, Sins of a Solar Empire, Supreme Commander, StarCraft II. The differences in gameplay are in the order of number, nature and gathering methods of resources; along with construction, research and production mechanics. The duration of games vary from 15 minutes for the fastest to (1-3) hours for the ones with the biggest maps and longest gameplays. We will now focus on StarCraft, on which our robotic player is implemented.

A StarCraft Game

StarCraft is a science-fiction RTS game released by Blizzard Entertainment^{TM} in March 1998. It was quickly expanded into *StarCraft: Brood War* (SC: BW) in November 1998. In the following, when referring to StarCraft, we mean StarCraft with the Brood War expansion. StarCraft is a canonical RTS game in the sense that it helped define the genre and most gameplay mechanics seen in other RTS games are present in StarCraft. It is as much based on strategy than tactics, by opposition to the Age of Empires and Total Annihilation series in which strategy is prevalent. In the following of the thesis, we will focus on duel mode, also known as 1 vs. 1 (1v1). Team-play (2v2 and higher) and "free for all" are very interesting but were not studied in the framework of this research. These game modes particularly add a layer of coordination and bluff respectively.

StarCraft sold 9.5 millions licenses worldwide, 4.5 millions in South Korea alone [**?**], and reigned on competitive RTS tournaments for more than a decade. Numerous international competitions (World Cyber Games, Electronic Sports World Cup, BlizzCon, OnGameNet StarLeague, MBCGame StarLeague) and professional gaming (mainly in South Korea [**?**]) produced a massive amount of data of highly skilled human players. In South Korea, there are two TV channels dedicated to broadcasting competitive video games, particularly StarCraft. The average salary of a pro-gamer* there was up to 4 times the average South Korean salary [**?**] (up to $200,000/year on contract for NaDa). Professional gamers perform about 300 actions (mouse and keyboard clicks) per minute while following and adapting their strategies, while their hearts reach 160 beats per minute (BPM are displayed live in some tournaments). StarCraft II is currently (2012) taking over StarCraft in competitive gaming but a) there is still a strong pool of highly skilled StarCraft players and b) StarCraft II has a really similar gameplay.

StarCraft (like most RTS) has a *replay** mechanism, which enables to record every player's actions such that the state of the game can be deterministically re-simulated. The only piece of stochasticity comes from "attack miss rates" ($\approx 47\%$) when a unit is on a lower ground than its target. These randomness generator seed is saved along with the actions in the replay. All high level players use this feature heavily either to improve their play or study opponents' styles. Observing replays allows player to see what happened under the *fog of war**, so that they can

understand timing of technologies and attacks, and find clues/evidences leading to infer the strategy as well as weak points.

In StarCraft, there are three factions with very different units and technology trees:

- *Terran*: humans with strong defensive capabilities and balanced, averagely priced biological and mechanical units.
- *Protoss*: advanced psionic aliens with expensive, slow to produce but resistant units.
- *Zerg*: insectoid alien race with cheap, quick to produce but weak units.

All factions use workers to gather resources, and all other characteristics are different: from military units to "tech trees*", gameplay styles. Races are so different that highly skilled players focus on playing with a single race (but against all three others). There are two types of resources, often located close together, minerals and gas. From minerals, one can build basic buildings and units, which opens the path to more advanced buildings, technologies and units, which will in turn all require gas to be produced. While minerals can be gathered at an increasing rate (bounded asymptotically) the more workers are put at work, the gas gathering rate is quickly limited to 3 workers per gas geyser (i.e. per base). There is another third type of "special" resource (called *supply**), which is the current limit of population a player can control. It can be upgraded by building special buildings (Protoss Pylon, Terran Supply Depot) or units (Zerg Overlord), giving 9 to 10 additional supply, up to a hard limit of 200. Some units cost more supply than others (from 0.5 for a Zergling to 8 for a Protoss Carrier or Terran Battlecruiser). In Figure 4.1, we show the very basics of Protoss economy and buildings.

To reach a competitive amateur level, players have to study *openings** and hone their *build orders**. An opening corresponds to the first strategic moves of a game, as in other abstract strategy games (Chess, Go). They are classified/labeled with names from high level players. A build-order is a formally described and accurately timed sequence of buildings to construct in the beginning. As there is a bijection between optimal population and time (in the beginning, before any fight), build orders are indexed on the total population of the player as in the example for Protoss in table 4.1. At the highest levels of play, StarCraft games usually last between 6 (shortest games, with a successful rush from one of the player) to 30 minutes (long economically and technologically developed game).

Supply	Build	Note
8	Pylon	"supply/psi/control/population" building
10	Gateway	units producing structure
12	Assimilator	constructed on a gas geyser, to gather gas
14	Cybernetics Core	technological building
16	Pylon	"supply/psi/control/population" building
16	Range	it is a research/tech
16	Dragoon	first military unit

Table 4.1: An example of the beginning of a "2 Gates Goon Range" Protoss build order which focus on building dragoons and their attack range upgrade quickly.

Figure 4.1: A StarCraft screenshot of a Protoss base, with annotations. The interface (heads up display at the bottom) shows the *mini-map**. The center of the interface (bottom) shows the selected unit (or group of units), here the Protoss Nexus (main economical building, producing workers and to which resources are brought) which is in the center of the screen and circled in green. The bottom right part shows the possible actions (here build a Protoss Probe or set a rally point). The top right of the screen shows the minerals (442), gas (208) and supply* (25 total on 33 current maximum). The dotted lines demarcate economical parts with active workers: red for minerals mining and green for gas gathering. The plain cut outs of buildings show: a Pylon (white), a Forge (orange, for upgrades and access to static defense), a Cybernetics Core (yellow, for technological upgrades and expanding the tech tree), a Gateway (pink, producing ground units), a Photon Cannon (blue, static defense).

We now present the evolution of a game (Figures 4.2 and 4.3) during which we tried to follow the build order presented in table 4.1 ("2 Gates Goon Range"[1]) but we had to adapt to the fact that the opponent was Zerg (possibility for a faster rush) by building a Gateway at 9 supply and building a Zealot (ground contact unit) before the first Dragoon. The first screenshot (image 1 in Figure 4.2) is at the very start of the game: 4 Probes (Protoss workers), and one Nexus (Protoss main building, producing workers and depot point for resources). At this point players have to think about what openings* they will use. Should we be aggressive early on or opt for a more defensive opening? Should we specialize our army for focused tactics, to the detriment of being able to handle situations (tactics and armies compositions) from the opponent that we did not foresee? In picture 2 in the same Figure (4.2), the first gate is being built. At this moment, players often send a worker to "scout" the enemy's base, as they cannot have military units yet, it is a safe way to discover where they are and inquiry about what they are doing. In

[1]On Teamliquid: `http://wiki.teamliquid.net/starcraft/2_Gate_Goon_Range_(vs._Terran)`

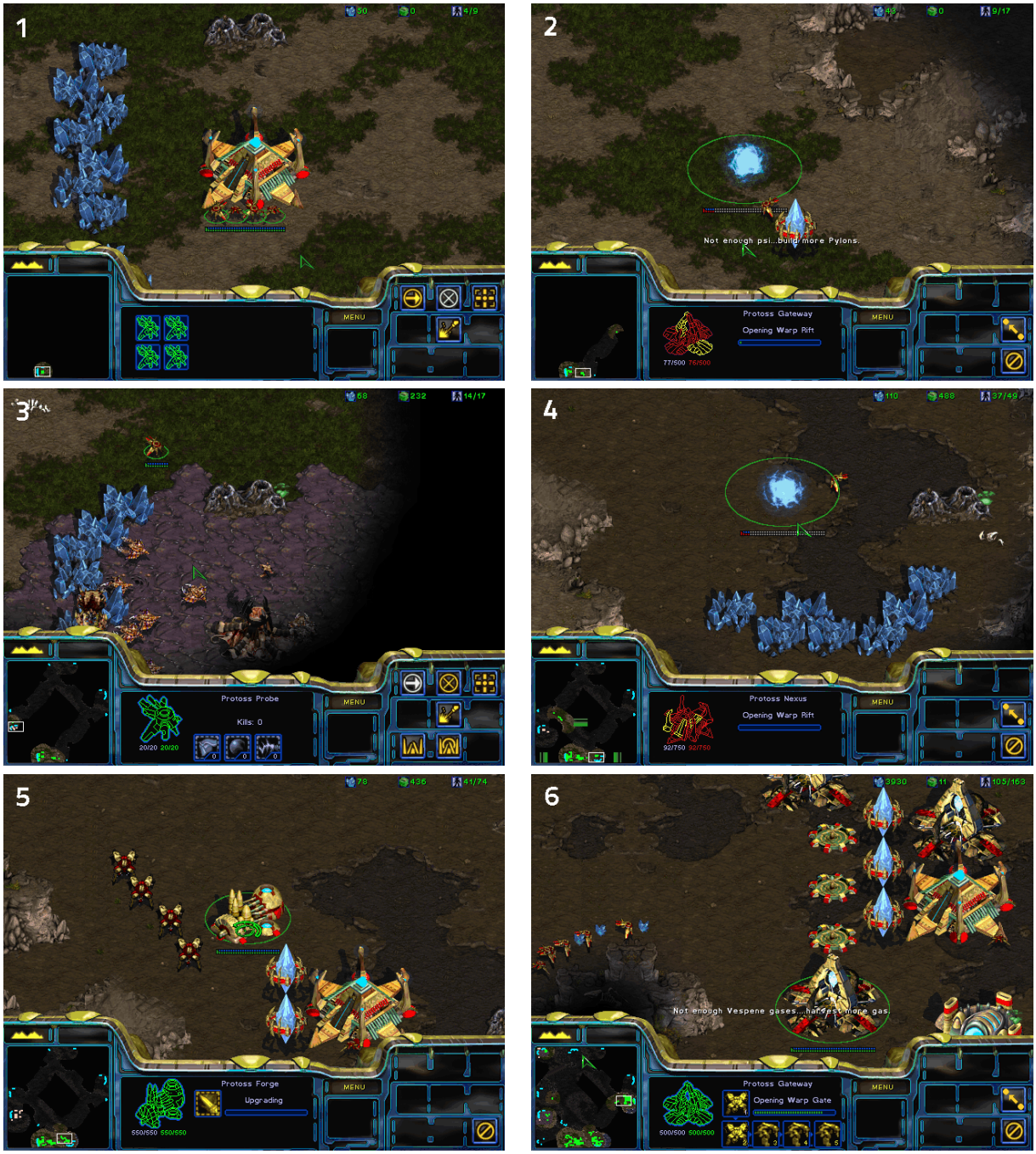


Figure 4.2: Start and economical parts of a StarCraft game. The order of the screenshots goes from left to right and from top to bottom.

picture 3, the player scouts their opponent, thus gathering the first bits of information about their early tech tree. Thus, knowing to be safe from an early attack ("rush"), the player decides to go for a defensive and economical strategy for now. Picture 4 shows the player "expanding", which is the act of making another base at a new resource location, for economical purposes. In picture 5, we can see the upgrading of the ground weapons along with 4 ranged military units, staying in defense at the expansion. This is a technological decision of losing a little of potential

Figure 4.3: Military moves from a StarCraft (PvT) game. The order of the screenshots goes from left to right and from top to bottom.

"quick military power" (from military units which could have been produced for this minerals and gas) in exchange of global upgrade for all units (alive and to be produced), for the whole game. This is an investment in both resources and time. Picture 6 showcases the production queue of a Gateway as well as workers transferring to a second expansion (third base). Being safe and having expanded his tech tree* to a point where the army composition is well-rounded, the player opted for a strategy to win by profiting from an economical lead. Figure 4.3 shows the aggressive moves of the same player: in picture 7, we can see a flying transport with artillery and area of effect "casters" in it. The goal of such an attack is not to win the game right away but to weaken the enemy's economy: each lost worker has to be produced, and, mainly, the missing gathering time adds up quite quickly. In picture 8, the transport in unloaded directly inside the enemy's base, causing huge damages to their economy (killing workers, Zerg Drones). This is the use of a specialized tactics, which can change the course of a game. At the same time, it involves only few units (a flying transport and its cargo), allowing for the main army to stay in defense at base. It capitalizes on the maneuverability of the flying Protoss Shuttle, the technological advancements allowing area of effects attacks and the large zone that the opponent has to cover to defend against it. In picture 9, an invisible attacking unit (circled in white) is harassing the oblivious enemy's army. This is an example of how technology advance on the opponent can be

game changing (the opponent does not have detection in range, thus is vulnerable to cloaked units). Finally, picture 10 shows the final attack, with a full ground army marching on the enemy's base.

4.2 RTS AI Challenges

In combinatorial game theory terms, competitive StarCraft (1 versus 1) is a zero sum, partial-information, deterministic[2] strategy game. StarCraft subsumes Wargus (Warcraft II open source clone), which has an estimated mean branching factor $1.5.10^3$ [**?**] (Chess: ≈ 35, Go: < 360): **?** finds a branching factor greater than 1.10^6 for StarCraft. In a given game (with maximum map size) the number of possible positions is roughly of 10^{11500}, versus the Shannon number for Chess (10^{43}) [**?**]. Also, we believe strategies are much more balanced in StarCraft than in most other games. Otherwise, how could it be that more than a decade of professional gaming on StarCraft did not converge to a finite set of fixed (imbalanced) strategies?

Humans deal with this complexity by abstracting away strategy from low level actions: there are some highly restrictive constraints on where it is efficient ("optimal") to place economical main buildings (Protoss Nexus, Terran Command Center, Zerg Hatchery/Lair/Hive) close to minerals spots and gas geysers. Low-level micro-management* decisions have high *horizontal continuity* and humans see these tasks more like playing a musical instrument skillfully. Finally, the continuum of strategies is analyzed by players and some distinct strategies are identified as some kinds of "invariants", or "entry points" or "principal components". From there, strategic decisions impose some (sometimes hard) constraints on the possible tactics, and the complexity is broken by considering only the interesting state space due to this high *vertical continuity*.

Most challenges of RTS games have been identified by [**??**]. We will try to anchor them in the human reasoning that goes on while playing a StarCraft game.

- *Adversarial planning under uncertainty* (along with *resource management*): it diverges from traditional planning under uncertainty in the fact that the player also has to plan *against* the opponent's strategy, tactics, and actions. From the human point of view, he has to plan for which buildings to build to follow a chosen opening* and subsequent strategies, under a restricted resources (and time) budget. At a lower level, and less obvious for humans, are the plans they make to coordinate units movements.

- *Learning and opponent modeling*, Buro accurately points out that human players need very few ("a couple of") games to spot their opponents' weaknesses. Somehow, human opponent modeling is related to the "induction scandal", as Russell called it: how do humans learn so much so fast? We learn from our mistakes, we learn the "play style" of our opponent's, we are quickly able to ask ourselves: "what should I do now given what I have seen *and* the fact that I am playing against player XYZ?". "Could they make the same attack as last time?". To this end, we use high level representations of the states and the actions with compressed invariants, causes and effects.

- *Spatial and temporal reasoning* (along with *decision making under uncertainty*), this is related to planning under uncertainty but focuses on the special relations between what

[2]The only stochasticity is in attacks failures (miss rate) from lower grounds to higher grounds, which is easily averaged.

can and what cannot be done in a given time. Sadly, it was true in 2004 but is still true today: "RTS game AI [...] falls victim to simple common-sense reasoning". Humans learn sequences of actions, and reason only about actions coherent with common sense. For AI of complex systems, this common-sense is hard to encode and to use.

- Collaboration (teamplay), which we will not deal with here: a lot has to do with efficient communication and "teammate modeling".

As we have seen previously more generally for multi-player video games, all these challenges can be handled by Bayesian modeling. While acknowledging these challenges for RTS AI, we now propose a different task decomposition, in which different tasks will solve some parts of these problems at their respective levels.

4.3 Tasks decomposition and linking

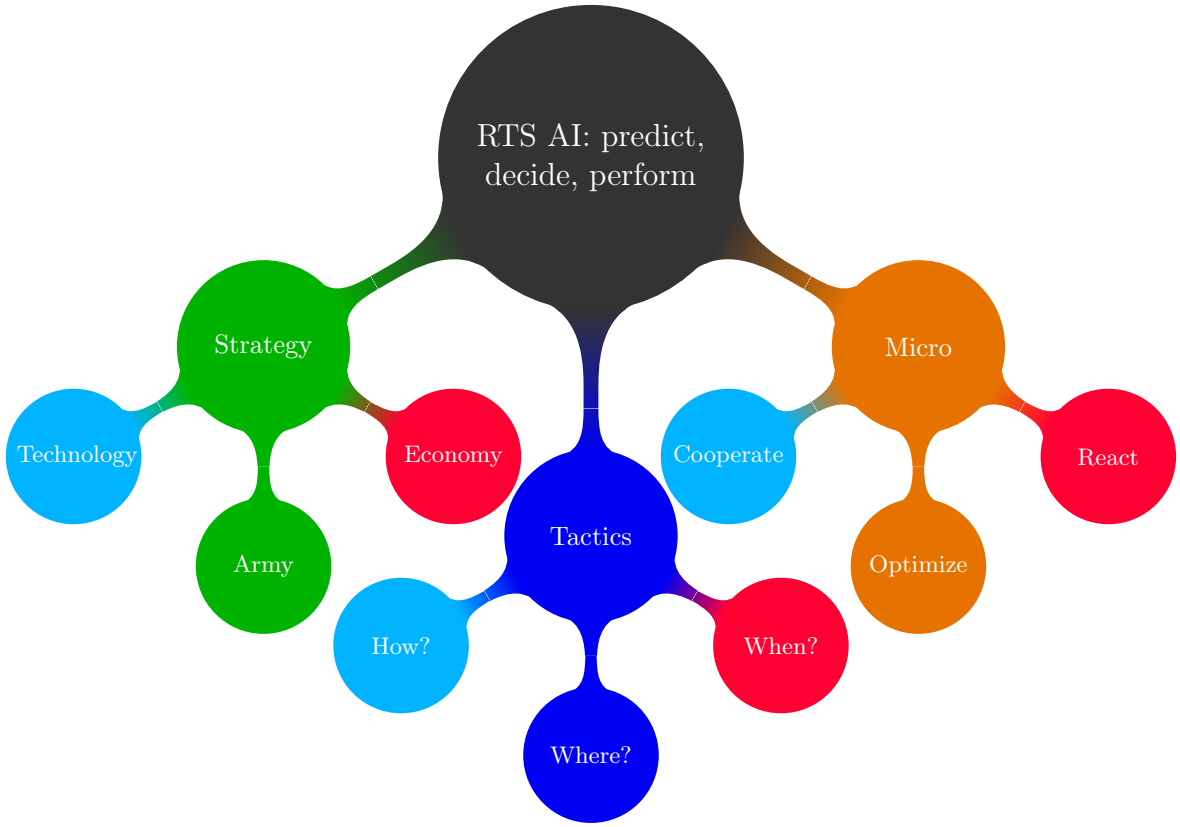


Figure 4.4: A mind-map of RTS AI. This is the tasks decomposition that we will use for the rest of the thesis.

We decided to decompose RTS AI in the three levels which are used by the gamers to describe the games: *strategy*, *tactics*, *micro-management*. We remind the reader that parts of the map not in the sight range of the player's units are under *fog of war**, so the player has only partial information about the enemy buildings and army. The way by which we expand the tech tree, the specific units composing the army, and the general stance (aggressive or defensive)

form what we call *strategy*. At the lower level, the actions performed by the player (human or not) to optimize the effectiveness of its units is called *micro-management**. In between lies *tactics*: where to attack, and how. A good human player takes much data in consideration when choosing: are there flaws in the defense? Which spot is more worthy to attack? How much am I vulnerable for attacking here? Is the terrain (height, chokes) to my advantage? The concept of strategy is a little more abstract: at the beginning of the game, it is closely tied to the build order and the intention of the first few moves and is called the *opening*, as in Chess. Then, the long term strategy can be partially summed up by three signals/indicators:

- aggression: how much is the player aggressive or defensive?
- initiative: how much is the player adapting to the opponent's strategy vs. how much is the player being original?
- technology/production/economy (tech/army/eco) distribution of resources: how much is the player spending (relatively) in these three domains?

At high levels of play, the tech/army/eco balance is putting a hard constraint on the aggression and initiative directions: if a player invested heavily in their army production, they should attack soon to leverage this investment. To the contrary, all other things being equal, when a player expands*, they are being weak until the expansion repaid itself, so they should play defensively. Finally, there are some technologies (researches or upgrades) which unlocks an "attack timing" or "attack window", for instance when a player unlocks an invisible technology (or unit) before that the opponent has detection technology (detectors). On the other hand, while "teching" (researching technologies or expanding the tech tree*), particularly when there are many intermediate technological steps, the player is vulnerable because of the immediate investment they made, which did not pay off yet.

From this RTS domain tasks decomposition, we can draw the mind map given in Figure 4.4. We also characterized these levels of abstractions by:

- the conditioning that the decisions on one abstraction level have on the other, as discussed above: strategy conditions tactics which conditions low-level actions (that does not mean than there cannot be some feedback going up the hierarchy).
- the quantity of direct information that a player can hope to get on the choices of their opponent. While a player can see directly the micro-management of their enemy (movements of units on the battlefront), they cannot observe all the tactics, and a big part of tactics gameplay is to hide or fake them well. Moreover, key technological buildings are sometimes hidden, and the player has to form beliefs about the long term strategy of the opponent (without being in their head).
- the time which is required to switch behaviors of a given level. For instance a change of strategy will require to either *a)* (technology) build at least two buildings or a building and a research/upgrade, *b)* (army) build a few units, *c)* take a new expansion (new base). A change of tactics corresponds to a large repositioning of the army(ies), while a change in micro-management is very quick (like moving units out of an area-of-effect damage zone).

This is shown in Figure 4.5. We will discuss the state of the art for the each of these subproblems (and the challenges listed above) in their respective parts.

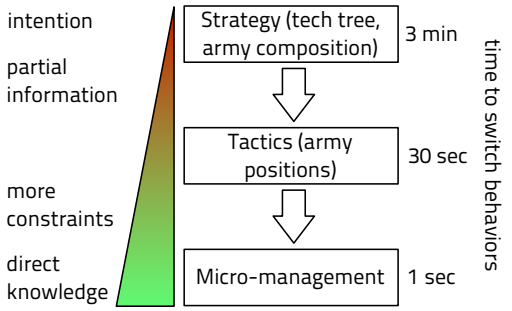


Figure 4.5: Gameplay levels of abstraction for RTS games, compared with their level of direct (and complete) information and orders of magnitudes of time to chance their policies.

To conclude, we will now present our works on these domains (strategy, tactics, micro-management) in separate chapters. In Figure 4.6, we present the flow of informations between the different inference and decision-making parts of the bot architecture. One can also view this problem as having a good model of one's strategy, one's opponent strategy, and taking decisions. The software architecture that we propose is to have services building and maintaining the model of the enemy as well as our state, and decision-making modules using all this information to give orders to actuators (filled in gray in Fig. 4.6).

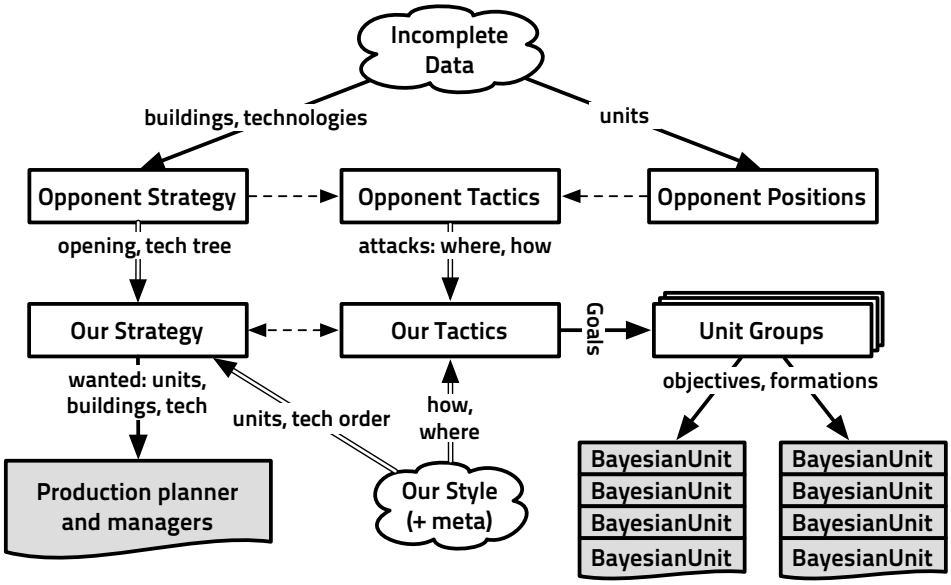


Figure 4.6: Information-centric view of the architecture of the major components of the bot. Arrows are labeled with the information or orders they convey: dotted arrows are conveying constraints, double lined arrows convey distributions, plain and simple arrows convey direct information or orders. The gray parts perform game actions (as the physical actions of the player on the keyboard and mouse).

Chapter 5

Micro-management

La vie est la somme de tous vos choix.

Life is the sum of all your choices.

Albert Camus

We present a Bayesian sensory-motor model for multi-agent (decentralized) units control in an adversarial setting. Orders, coming from higher up in the decision hierarchy, are integrated as another sensory input. We evaluated our model on classic StarCraft micro-management* tasks. This work was published in [**?**].

- Problem: optimal control of units in a (real-time) huge adversarial actions space (collisions, accelerations, terrain, damages, areas of effects, foes, goals...).
- Problem that we solve: efficient coordinated control of units incorporating all low level actions and inputs, plus higher level orders and representations.
- Type: it is a problem of *multi-agent control in an adversarial environment*[1].
- Complexity: PSPACE-complete [**??**]. Our solutions are real-time on a laptop.

[1]Strictly, it can be modeled as a POMDP* for each unit independently with S the states of all the other units (enemies and allied altogether) which are known through observations O by conditional observations probabilities Ω, with A the set of actions for the given unit, T transition probabilities between states and depending on actions, and the reward function R based on goal execution, unit survivability and so on... It can also be viewed as a (gigantic) POMDP* solving the problem for all (controlled units) at once, the advantage is that all states S for allied units is known, the disadvantage is that the combinatorics of T and A make it intractable for useful problems.

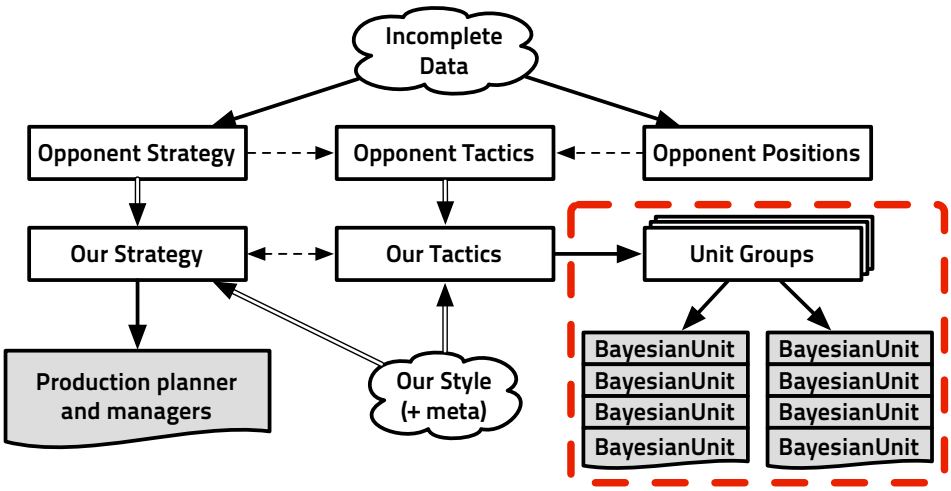


Figure 5.1: Information-centric view of the architecture of the bot, the part concerning this chapter (micro-management*) is in the dotted rectangle

5.1 Units Management

In this part, we focus on micro-management*, which is the lower-level in our hierarchy of decision-making levels (see Fig. **??**) and directly affect units control/inputs. Micro-management consists in maximizing the effectiveness of the units *i.e.* the damages given/damages received ratio. These has to be performed simultaneously for units of different types, in complex battles, and possibly on heterogeneous terrain. For instance: retreat and save a wounded unit so that the enemy units would have to chase it either boosts your firepower (if you save the unit) or weakens the opponent's (if they chase). In the field of units control, the dimension of the set of possible actions each micro-turn (for instance: 1/24th of a second in StarCraft) constrains reasoning about the state of the game to be hierarchical, with different levels of granularity. In most RTS games, a unit can go (at least) in its 24 surrounding tiles (see Figure 5.2, combination of N, S, E, W up to the 2nd order), stay where it is, attack, and sometimes cast different spells: which yields more than 26 possible actions each turn. Even if we consider only 8 possible directions, stay, and attack, with N units, there are 10^N possible combinations each turn (all units make a move each turn). As large battles in StarCraft account for *at least* 20 units on each side, optimal units control hides in too big a search space to be fully explored in real-time (sub-second reaction at least) on normal hardware, even if we take only one decision per unit per second.

Our distributed sensory-motor model for micro-management* is able to handle both the complexity of unit control and the need of hierarchy (see Figure 5.1, this chapter focuses on the part inside the dotted line). We treat the units independently, thus reducing the complexity (no communication between our "Bayesian units"), and allows to take higher-level orders into account along with local situation handling. For instance: the tactical planner may decide to retreat, or go through a choke under enemy fire, each Bayesian unit will have the higher-level order as a sensory input, along with topography, foes and allies positions. From its perception, our Bayesian robot **?** can compute the distribution over its motor control. The sensory inputs

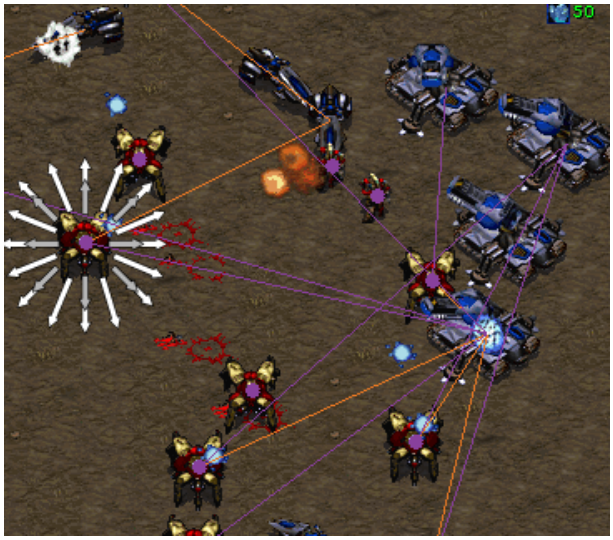


Figure 5.2: Screen capture of a fight in which our bot controls the bottom-left units in StarCraft. The 24 possible directions are represented for a unit with white and grey arrows.

given to a "Bayesian unit" controls its objective(s) or goal(s) and the parametrization of his probabilistic model controls its behavior and degree of freedom. As an illustration (only), two of the extreme cases are $\mathrm{P}(Direction = x|Objective = x) = 1$: no freedom, $\mathrm{P}(Direction = x|Objective = y) = \mathrm{P}(Direction = x)$: no influence of the objective. The performances of our models are evaluated against the original StarCraft AI and a reference AI: they have proved excellent in this benchmark setup.

5.2 Related Works

Technical solutions include finite states machines (FSM) [**?**], genetic algorithms (GA) [**???**], reinforcement learning (RL) [**??**], case-based reasoning (CBR) [**??**], continuous action models [**?**], reactive planning [**?**], upper confidence bounds tree (UCT) [**?**], potential fields [**?**], influence maps[**?**], and cognitive human-inspired models [**?**].

FSM are well-known and widely used for control tasks due to their efficiency and implementation simplicity. However, they don't allow for state sharing, which increases the number of transitions to manage, and state storing, which makes collaborative behavior hard to code [**?**]. Hierarchical FSM (HFSM) solve some of this problems (state sharing) and evolved into behavior trees (BT, hybrids HFSM) [**?**] and behavior multi-queues (resumable, better for animation) [**?**] that conserved high performances. However, adaptivity of behavior by parameters learning is not the main focus of these models, and unit control is a task that would require a huge amount of hand tuning of the behaviors to be really efficient. Also, these architectures does not allow reasoning under uncertainty, which helps dealing with local enemy and even allied units. Our agents see local enemy (and allied) units but do not know what action they are going to do. They could have perfect information about the allied units intentions, but this would need extensive communication between all the units.

Some interesting uses of reinforcement learning (RL)* [**?**] to RTS research are concurrent

hierarchical (units Q-functions are combined higher up) RL* [?] to efficiently control units in a multi-effector system fashion. ? advocate the use of prior domain knowledge to allow faster RL* learning and applied their work on a large scale (while being not as large as StarCraft) turn-based strategy game. In real game setups, RL* models have to deal with the fact that the state spaces to explore is enormous, so learning will be slow or shallow. It also requires the structure of the game to be described in a partial program (or often a partial Markov decision process) and a shape function [?]. RL can be seen as a transversal technique to learn parameters of an underlying model, and this underlying behavioral model matters. The same problems arise with evolutionary learning techniques. ? used evolutionary learning techniques, but face the same problem of dimensionality.

Case-based reasoning (CBR) allows for learning against dynamic opponents [?] and has been applied successfully to strategic and tactical planning down to execution through behavior reasoning rules [?]. CBR limitations (as well as RL) include the necessary approximation of the world and the difficulty to work with multi-scale goals and plans. These problems led respectively to continuous action models [?], an integrated RL/CBR algorithm using continuous models, and reactive planning [?], a planning decomposition similar to hierarchical task networks [?] in that sub-plans can be changed at different granularity levels. Reactive planning allows for multi-scale (hierarchical) goals/actions integration and has been reported working on StarCraft, the main drawback is that it does not address uncertainty and so can not simply deal with hidden information (both extensional and intentional). Fully integrated FSM, BT, RL and CBR models all need vertical integration of goals, which is not very flexible (except in reactive planning).

Monte-Carlo planning [?] and upper Upper confidence bounds tree (UCT) planning (coming from Go AI) [?] samples through the (rigorously intractable) plans space by incrementally building the actions tree through Monte-Carlo sampling. UCT for tactical assault planning [?] in RTS does not require to encode human knowledge (by opposition to Monte-Carlo planning) but it is too costly, both in learning and running time, to go down to units control on RTS problems. Our model subsumes potential fields [?], which are powerful and used in new generation RTS AI to handle threat, as some of our Bayesian unit sensory inputs are potential damages and tactical goodness (height for the moment) of positions. ? presented a multi-agent potential fields based bot able to deal with fog of war in the Tankbattle game. ? and ? co-evolved influence map trees for spatial reasoning in RTS games. ? used influence maps to achieve intelligent squad movement to flank the opponent in a RTS game. A drawback for potential field-based techniques is the large number of parameters that has to be tuned in order to achieve the desired behavior. Our model provides flocking and local (subjective to the unit) influences on the pathfinding as in [?], which uses self-organizing-maps (SOM). In their paper, Preuss *et al.* are driven by the same quest for a more natural and efficient behavior for units in RTS. We would like to note that potential fields and influence maps are reactive control techniques, and as such, they do not perform any form of lookahead. In their raw form (without specific adaptation to deal with it), they can lead units to be stuck in local optimums (potential wells).

Pathfinding is used differently in planning-based approaches and reactive approaches. ? is an example of the permeable interface between pathfinding and reactive control with influence maps augmented tactical pathfinding and flocking. As we used pathfinding as the mean to get a sensory input towards the objective, we were free to use a *low resolution* and *static* pathfinding for which A* was enough. Our approach is closer to the one of [?]: combining a simple path for

the group with flocking behavior. In large problems and/or when the goal is to deal with multiple units pathfinding taking collisions (and sometimes other tactical features), more efficient, incremental and adaptable approaches are required. Even if specialized algorithms, such as D*-Lite **?** exist, it is most common to use A* combined with a map simplification technique that generates a simpler navigation graph to be used for pathfinding. An example of such technique is Triangulation Reduction A*, that computes polygonal triangulations on a grid-based map **?**. In recent commercial RTS games like Starcraft 2 or Supreme Commander 2, flocking-like behaviors are inspired of continuum crowds ("flow field") **?**. A comprehensive review about (grid-based) pathfinding was recently done by Sturtevant **?**.

Finally, there are some cognitive approaches to RTS AI [**?**], and we particularly agree with Wintermute *et al.* analysis of RTS AI problems. Our model has some similarities: separate and different agents for different levels of abstraction/reasoning and also a perception-action approach (see Figure 5.1).

5.3 A Bayesian Model for Units Control

A Simple Top-Down Solution

How do we set the reward or value function for micro-management*: is staying alive better than killing an enemy unit? Even if we could compute to the end of the fight and/or apply the same approach that we have for board games, how do we infer the best "set of next moves" for the enemy when the space of possible moves is so huge and the number of possible reasoning methods (sacrifices and influences of other parts of the game for instance) is bigger than for Chess? As complete search through the min/max tree, if there exists such thing in a RTS, is intractable, we propose a greedy target selection heuristic leading the movements of units to benchmark our Bayesian model against. In this solution, each unit can be viewed as an effector, part of a multi-body (multi-effector) agent. Let $\mathbf{U}$ be the set of the m units to control, $\mathbf{A} = \mathbf{D} \cup \mathbf{S}$ be the set of possible actions (all n possible **D**irections, standing ground included, and **S**kills, firing included), and $\mathbf{E}$ the set of enemies. As $|\mathbf{U}| = m$, we have $|\mathbf{A}|^m$ possible combinations each turn, and the enemy has $|\mathbf{A}|^{|\mathbf{E}|}$.

The idea behind the heuristic used for target selection is that units need to focus fire (less incoming damages if enemy units die faster) on units that do the most damages, have the less hit points, and take the most damages from their attack type. This can be achieved by using a data structure, shared by all our units engaged in the battle, that stores the damages corresponding to future allied attacks for each enemy units. Whenever a unit will fire on a enemy unit, it registers there the future damages on the enemy unit. We also need a set of priority targets for each of our unit types that can be drawn from expert knowledge or learned from reinforcement learning battling all unit types. A unit select its target among the most focus fired units with positive future hit point (current hit points minus registered damages), while prioritizing units from the priority set of its type. The units group can also impose its own priorities on enemy units (for instance to achieve a goal). The target selection heuristics is fully depicted in 6 in the appendices.

The only degenerated case would be if all our units register their targets at once (and all the enemy units have the same priority) and it never happens (plus, units fire rates have a

randomness factor). Indeed, our Bayesian model uses this target selection heuristic (see 6), but that is all both models have in common. From there, units are controlled with a very simple FSM: fire when possible (weapon reloaded and target in range), move towards target when out of range.

Our Model: a Bayesian Bottom-Up Solution

We use Bayesian programming as an alternative to logic, transforming incompleteness of knowledge about the world into uncertainty. In the case of units management, we have mainly *intensional* uncertainty. Instead of asking questions like: where are other units going to be next frame? 10 frames later? Our model is based on rough estimations that are not taken as ground facts. Knowing the answer to these questions would require for our own (allied) units to communicate a lot and to stick to their plan (which does not allow for quick reaction nor adaptation). For enemy units, it would require exploring the tree of possible plans (intractable) whose we can only draw samples from [**?**]. Even so, taking enemy minimax (to which depth?) moves for facts would assume that the enemy is also playing minimax (to the same depth) following exactly the same valuation rules as ours. Clearly, RTS micro-management* is more inclined to reactive planning than board games reasoning. That does not exclude having higher level (strategic and tactic) goals. In our model, they are fed to the unit as sensory inputs, that will have an influence on its behavior depending on the situation/state the unit is in.

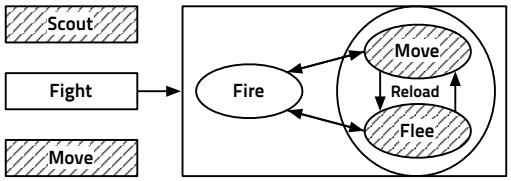


Figure 5.3: Bayesian unit modal FSM, detail on the fight mode. Stripped modes are Bayesian.

We propose to model units as sensory-motor robots described within the Bayesian robot programming framework [**?**]. A Bayesian model uses and reasons on distributions instead of predicates, which deals directly with uncertainty. Our Bayesian units are simple hierarchical finite states machines (states can be seen as modes) that can scout, fight and move (see Figure 5.3). Each unit type has a reload rate and attack duration, so their fight mode will be like:

if $canFire \wedge t = selectTarget() \wedge inRange(t)$ **then**
  $attack(t)$
else if $needFlee()$ **then**
  $flee()$
else
  $fightMove()$
end if

The unit needs to determine where to go when fleeing and moving during a fight, optimally with regard to its target and the attacking enemies, while avoiding collisions (which results in

blocked units and time lost) as much as possible. $flee()$ and $fightMove()$ call the Bayesian model (expressed in Bayesian programming, see section 3.) that follows:

Variables

- $Dir_{i \in [\![0 \dots n]\!]} \in \{True, False\}$: at least one variable for each atomic direction the unit can go to. $\mathrm{P}(Dir_i = True) = 1$ (also noted $\mathrm{P}(Dir_i) = 1$) means that the unit will certainly go in direction i ($\Leftrightarrow \mathbf{D}[i]$). For example, in StarCraft we use the 24 atomic directions (48 for the smallest and fast units as we use a proportional scale) plus the current unit position (stay where it is) as shown in Figure 5.2. We could use one variable with 24 directions, the approach would be the same.

- $Obj_{i \in [\![0 \dots n]\!]} \in \{True, False\}$: direction of the objective (given by a higher rank model). $\mathrm{P}(Obj_i) = 1$ means that the direction i is totally in the direction of the objective (move, retreat or offensive position computed by the strategic or tactical manager). In our StarCraft AI, we use the scalar product between the direction i and the objective vector (output of the pathfinding) with a minimum value of 0.01 so that the probability to go in a given direction is proportional to its alignment with the objective. Note that some situation have a null objective (the unit is free to move).

- $Dmg_{i \in [\![0 \dots n]\!]} \in [DamageValues]$ for instance, with $ubhp$ standing as unit base hit points, $Dmg_i \in \left\{0, [\![0 \dots \frac{ubhp}{2}]\!],]\!]\frac{ubhp}{2} \dots ubhp[\!], [\![ubhp \cdots + \inf[\!]\right\}$. This will act as subjective potential fields [?] in which the (repulsive) influence of the potential damages map depends on the unit type. In our StarCraft AI, this is directly drawn from two constantly updated potential damage maps (air, ground). For instance, it allows our scouting units to avoid potential attacks as much as possible.

- $A_{i \in [\![0 \dots n]\!]} \in \{None, Small, Big\}$: occupation of the direction i by a allied unit. The model can effectively use many values (other than "occupied/free") because directions may be multi-scale (for instance we indexed the scale on the size of the unit) and, in the end, small and/or fast units have a much smaller footprint, collision wise, than big and/or slow. In our AI, instead of direct positions of allied units, we used their (linear) interpolation $\frac{|unit, \mathbf{D}[i]|}{unit\_speed}$ frames later (to avoid squeezing/expansion).

- $E_{i \in [\![0 \dots n]\!]} \in \{None, Small, Big\}$: occupation of the direction i by a enemy unit. As above.

- $Occ_{i \in [\![0 \dots n]\!]} \in \{None, Building, StaticTerrain\}$ (this could have been 2 variables or we could omit static terrain but we stay as general as possible): repulsive effect of buildings and terrain (cliffs, water, walls).

There is basically one set of (sensory) variables per effect in addition to the Dir_i values. In general, if one decides to cover a lot of space with directions (*i.e.* have more than just atomic directions, *i.e.* use this model for planning), one needs to consider directions whose paths collide with each others. For instance, a $\mathbf{D}[i]$ far from the unit can force the unit to go through a wall of allied units ($A_j = Big$) or potential damages.

Decomposition

The joint distribution (JD) over these variables is a specific kind of fusion called inverse programming [**?**]. The sensory variables are considered independent knowing the actions, contrary to standard naive Bayesian fusion, in which the sensory variables are considered independent knowing the phenomenon.

$$\begin{aligned} & \mathrm{P}(Dir_{1:n}, Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n}) && (5.1)\\ = \quad & JD = \prod_{i=1}^{n} \mathrm{P}(Dir_i)\mathrm{P}(Obj_i|Dir_i) && (5.2)\\ & \mathrm{P}(Dmg_i|Dir_i)\mathrm{P}(A_i|Dir_i) && (5.3)\\ & \mathrm{P}(E_i|Dir_i)\mathrm{P}(Occ_i|Dir_i) && (5.4) \end{aligned}$$

We assume that the i directions are independent depending on the action because dependency is already encoded in (all) sensory inputs. We do not have $\mathrm{P}(Obj_i) = 1, \mathrm{P}(Obj_{j\neq i}) = 0$ but a "continuous" function on i for instance.

Forms

- $\mathrm{P}(Dir_i)$ prior on directions, unknown, so unspecified/uniform over all i. $\mathrm{P}(Dir_i) = 0.5$.
- $\mathrm{P}(Obj_i|Dir_i)$ for instance, "probability that this direction is the objective knowing that we go there" $\mathrm{P}(Obj_i = T|Dir_i = T)$ is very high (close to one) when rushing towards an objective, whereas it is far less important when fleeing. Probability table: $\mathrm{P}(Obj_i|Dir_i) = table[obj, dir]$
- $\mathrm{P}(Dmg_i|Dir_i)$ probability of damages values in some direction knowing this is the unit direction. $\mathrm{P}(Dmg_i \in [ubhp, +\inf[|Dir_i = T)$ has to be small in many cases. Probability table.
- $\mathrm{P}(A_i|Dir_i)$ probability table that there is an ally in some direction knowing this is the unit direction. Used to avoid collisions.
- $\mathrm{P}(E_i|Dir_i)$ probability table, same as above with enemy units, different parameters as we may want to be stucking enemy units, or avoid them.
- $\mathrm{P}(Occ_i|Dir_i)$ probability table that there is a blocking building or terrain element is some direction, knowing this is the unit direction, $\mathrm{P}(Occ_i = Static|Dir_i = T)$ will be very low (0), whereas $\mathrm{P}(Occ_i = Building|Dir_i = T)$ will also be very low but triggers building attack (and destruction) when there are no other issues.

Additional variables

There are additional variables for specific modes/behaviors:

- $Prio_{i\in[\![0\ldots n]\!]} \in \{True, False\}$: combined effect of the priority targets that attract the unit while in fight ($fightMove()$). The JD is modified as $JD \times \Pi_{i=1}^{n} \mathrm{P}(Prio_i|Dir_i)$, where $\mathrm{P}(Prio_i|Dir_i)$ is a probability table, that corresponds to the attraction of a priority (maybe

out of range) target in this direction. This is efficient to be able to target casters or long range units for instance.

- $Att_{i\in[\![0...n]\!],j\in[\![0...m]\!]}$: allied units attractions and repulsions to produce a *flocking* behavior while moving. Different than A_i, the JD would become $JD \times \Pi_{i=1}^{n}\Pi_{j=1}^{m}\mathrm{P}(Att_{i,j}|Dir_i)$, where $\mathrm{P}(Att_{i,j}|Dir_i)$ is a probability table for flocking: a too close unit j will repel the Bayesian unit ($\mathrm{P}(Att_{i,j}|Dir_i) < mean$) whereas another unit j will attract depending on its distance (and possibly, leadership).

- $Dir^{t-1}_{i\in[\![0...n]\!]} \in \{True, False\}$: the previous selected direction, $Dir_i^{t-1} = T$ iff the unit went to the direction i, else $False$ for a steering (smooth) behavior. The JD would then be $JD \times \Pi_{i=1}^{n}\mathrm{P}(Dir_i^{t-1}|Dir_i)$, with $\mathrm{P}(Dir_i^{t-1}|Dir_i)$ the influence of the last direction, either a table or a parametrized Bell shape over all the i.

- One can have a distribution over a n valued variable $Dir \in \{D\}$. (Each set of boolean random variable can be seen as a $|D|$ valued variable.) The JD would then be $JD \times \mathrm{P}(Dir).\Pi_{i=1}^{n}\mathrm{P}(Dir_i|Dir)$.

Identification

Parameters and probability tables can be learned through reinforcement learning **[??]** by setting up different and pertinent scenarii and search for the set of parameters that maximizes a reward function. In our current implementation, the parameters and probability table values are mainly hand specified.

Question

When in $fightMove()$, the unit asks:

$$\mathrm{P}(Dir_{1:n}|Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n}, Prio_{1:n})$$

When in $flee()$ or while moving or scouting (different balance/parameters), the unit asks:

$$\mathrm{P}(Dir_{1:n}|Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n})$$

When flocking, the unit asks:

$$\mathrm{P}(Dir_{1:n}|Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n}, Att_{1:n,1:m})$$

From there, the unit can either go in the most probable Dir_i or sample through them. We describe the effect of this choice in the next section (and in Fig. 5.5). A simple Bayesian fusion from 3 sensory inputs is shown in Figure 5.4, in which the final distribution on Dir peaks at places avoiding damages and collisions while being towards the goal. Here follows the full Bayesian program of the model (5.3):

5.4 Results on StarCraft

StarCraft micro-management involves ground, flying, ranged, contact, static, moving (at different speeds), small and big units (see Figure 5.2). Units may also have splash damage, spells,

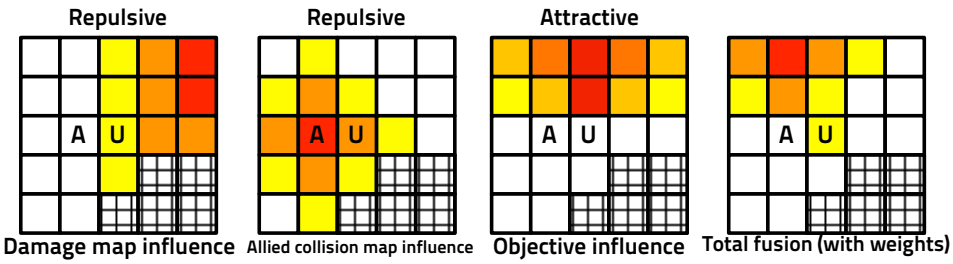


Figure 5.4: Simple example of Bayesian fusion from 3 sensory inputs (damages and collisions avoidance, goal attraction). The grid pattern represents statically occupied terrain, the unit we control is in `U`, an allied unit is in `A`. The result is displayed on the rightmost image.

$$
\text{Bayesian program}
\begin{cases}
\text{Description}
\begin{cases}
\text{Specification } (\pi)
\begin{cases}
Variables \\
Dir_{1:n}, Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n} \\
Decomposition \\
\mathrm{P}(Dir_{1:n}, Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n}) = \prod_{i=1}^{n} \left[\mathrm{P}(Dir_i)\right. \\
\left.\mathrm{P}(Obj_i|Dir_i)\mathrm{P}(Dmg_i|Dir_i)\mathrm{P}(A_i|Dir_i)\mathrm{P}(E_i|Dir_i)\mathrm{P}(Occ_i|Dir_i)\right] \\
Forms \\
\mathrm{P}(Dir_i) : \text{prior on directions (crossing policy)} \\
\mathrm{P}(XYZ_i|Dir_i) : \text{ probability tables}
\end{cases} \\
Identification\ (using\ \delta) \\
\text{reinforcement learning or hand specified (or distribs. parameters optimization)}
\end{cases} \\
Question \\
\text{fightmoving : } \mathrm{P}(Dir_{1:n}|Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n}, Prio_{1:n}) \\
\text{fleeing : } \mathrm{P}(Dir_{1:n}|Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n}) \\
\text{flocking : } \mathrm{P}(Dir_{1:n}|Obj_{1:n}, Dmg_{1:n}, A_{1:n}, E_{1:n}, Occ_{1:n}, Att_{1:n,1:m})
\end{cases}
$$

and different types of damages whose amount will depend on the target size. It yields a rich states space and needs control to be very fast: human progamers can perform up to 400 "actions per minute" in intense fights. The problem for them is to know which actions are effective and the most rewarding to spend their actions efficiently. A robot does not have such physical limitations, but yet, badly chosen actions have negative influence on the issue of fights.

Our Robot Architecture

Our full robot has separate agents types for separate tasks (strategy, tactics, economy, army, as well as enemy estimations and predictions): the part that interests us here, the unit control, is managed by Bayesian units directly. Their objectives are set by military goal-wise atomic units group, themselves spawned to achieve tactical goals (see Fig. 5.1). Units groups tune their Bayesian units modes (scout, fight, move) and give them Obj_i as sensory inputs. The Bayesian unit is the smallest entity and controls individual units as sensory-motor robots according to

the model described above. The only inter Bayesian units communication about attack targets is handled by a structure shared at the units group level.

Experiments

Our implementation[2] (BSD licensed) uses BWAPI[3] to get information from and to control StarCraft. We produced three different AI to run experiments with, along with the original AI (OAI) from StarCraft:

- Heuristic only AI (HOAI), section 4.1: this AI shares the target selection heuristic with our other AI and will be used as a dummy reference (in addition to StarCraft original AI) to avoid bias due to the target selection heuristic.
- Bayesian AI picking best (BAIPB): this AI follows the model of section 4.2 and selects the most probable Dir_i as movement.
- Bayesian AI sampling (BAIS): this AI follows the model of section 4.2 and samples through Dir_i according to their probability ($\Leftrightarrow$ according to Dir distribution).

The experiments consisted in having the AIs fight against each others on a micro-management scenario with mirror matches of 12 and 36 ranged ground units (Dragoons). In the 12 units setup, the unit movements during the battle is easier (less collision probability) than in the 36 units setup. We instantiate only the army manager (no economy in this special maps), one units group manager and as many Bayesian units as there are units provided to us in the scenario. The results are presented in Table 5.1.

12 units \ 36 units	OAI	HOAI	BAIPB	BAIS
OAI	(50%)	64%	9%	3%
HOAI	59%	(50%)	11%	6%
BAIPB	93%	97%	(50%)	3%
BAIS	93%	95%	76%	(50%)

Table 5.1: Win ratios over at least 200 battles of OAI, HOAI, BAIPB and BAIS in two mirror setups: 12 and 36 ranged units. Read line vs column: for instance HOAI won 59% of its matches against OAI in the 12 units setup. Note: The average amount of units left at the end of battles is grossly proportional to the percentage of wins.

These results show that our heuristic (HAOI) is comparable to the original AI (OAI), perhaps a little better, but induces more collisions. For Bayesian units however, the "pick best" (BAIPB) direction policy is very effective when battling with few units (and few movements because of static enemy units) as proved against OAI and HOAI, but its effectiveness decreases when the number of units increases: all units are competing for the best directions (to $flee()$ or $fightMove()$ in) and they collide. The sampling policy (BAIS) has way better results in large armies, and significantly better results in the 12 units vs BAIPB, supposedly because BAIPB moves a lot (to chase wounded units) and collide with BAIS units. Sampling entails that the

[2] BroodwarBotQ, code and releases: `http://github.com/SnippyHolloW/BroodwarBotQ`
[3] BWAPI: `http://code.google.com/p/bwapi/`

competition for the best directions is distributed among all the "bests to good" wells of well-being, from the units point of view. We also ran tests in setups with flying units in which BAIPB fared as good as BAIS (no collision for flying units) and way better than OAI.

Uses and extensions

This model is currently at the core of the micro-management of our StarCraft bot. We use it mainly with four modes corresponding to four behaviors (four sets of parameters):

- Scout: in this mode, the (often quick and low hit points) unit avoids danger by modifying locally its pathfinding-based, objectives oriented route to avoid damages according to $\mathrm{P}(Dmg_i|Dir_i)$.
- In position: in this mode, the unit try to keep its ground but can be "pushed" by other units wanting to pass through with $\mathrm{P}(A_i|Dir_i)$. This is useful at a tactical level to do a wall of units that our units can traverse but the opponent's cannot. Basically, there is an attraction to the position of the unit and a stronger repulsion of the interpolation of movements of allied units.
- Flock: in this mode, our unit moves influenced by other allied units through $\mathrm{P}(Att_{i\in[\![0...n]\!],j\in[\![0...m]\!]})$ that repulse or attract it depending on its distance to the interpolation of the allied unit j. It allows our units to move more efficiently by not splitting around obstacles and colliding less.
- Fight: in this mode, our unit will follow the damages gradient to smart positions, for instance close to tanks (they cannot fire too close to their position) or far from too much contact units if our unit can attack with range. Our unit moves are also influenced by its priority targets, its goal (go through a choke, flee, etc.) and other units.

This model can be used to specify the behavior of units in RTS games. Instead of relying on a "units push each other" physics model for handling dynamic collision of units, this model makes the units react themselves to collision in a more realistic fashion (a marine cannot push a tank, the tank will move). The realism of units movements can also be augmented with a simple-to-set $\mathrm{P}(Dir^{t-1}|Dir^t)$ steering parameter, although we do not use it in the competitive setup.

5.5 Discussion

If we learn the parameters of such a model to mimic existing data (data mining) or to maximize a reward function (reinforcement learning), we can interpret the parameters that will be obtained more easily than parameters of an artificial neural network for instance. Parameters learned in one setup can be reused in another if they are understood. We claim that specifying or changing the behavior of this model is much easier than changing the behavior generated by a FSM, and game developers can have a fine control over it. Dynamic switches of behavior (as we do between the scout/flock/inposition/fight modes) are just one probability tables switch away. In fact, probability tables for each sensory input (or group of sensory inputs) can be linked to sliders in a "behavior editor" and game makers can specify the behavior of their units by specifying

the degree of effect of each perception (sensory input) on the behavior of the unit and see the effect in real time. This is not restricted to RTS and could be applied to RPG* and even FPS* gameplays*.

Future work could consist in using reinforcement learning [?] or evolutionary algorithms [?] to learn the probability tables. It should enhance the performance of our Bayesian units in specific setups. It implies making up challenging scenarii and dealing with huge sampling spaces [?]. Also, we could use multi-modality [?] and inverse programming [?] to get rid of the remaining (small: fire-retreat-move) FSM. Also, there are yet many collision cases that remain unsolved (particularly visible with contact units like Zealots and Zerglings), so we could also try:

- adding local priority rules to solve collisions (for instance through an asymmetrical $\mathrm{P}(Dir_i^{t-1}|Dir_i)$) that would entails units crossing lines with a preferred side (some kind of "social rule"),
- use a units group level supervision using Bayesian units' distributions over Dir as preferences or constraints (for a solver),
- use $\mathrm{P}(Dir)$ as an input to another Bayesian model at the units group level of reasoning.

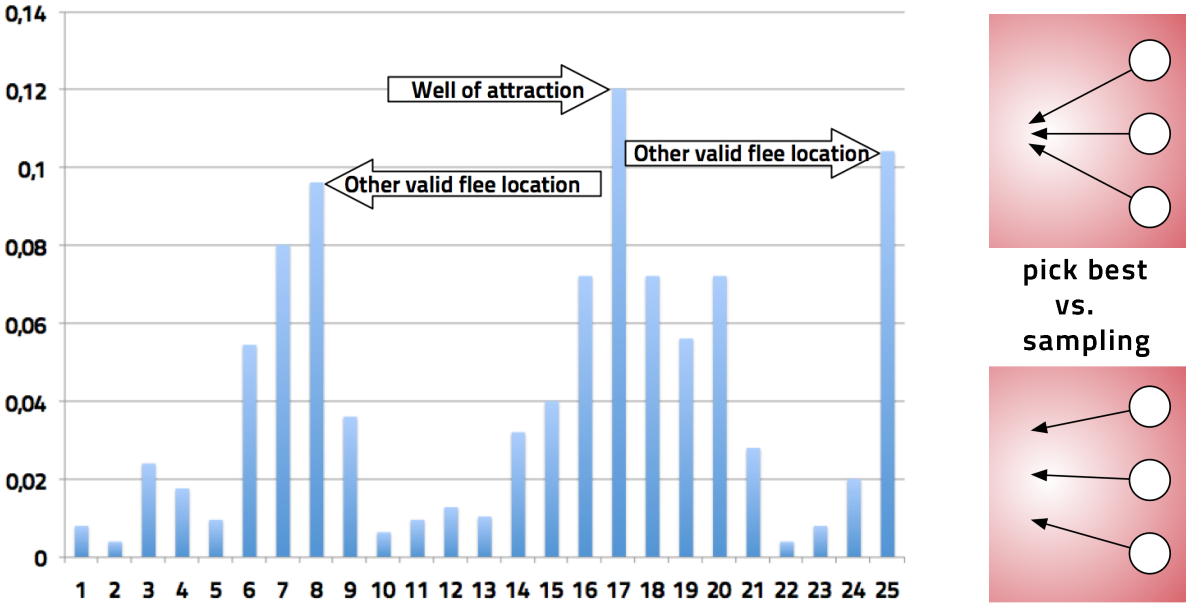


Figure 5.5: Example of $\mathrm{P}(Dir)$ when fleeing, showing why sampling (BAIS, bottom graphic on the right) may be a better instead of picking the most best direction (BAIPB, here $Dir = 17$ in the plot, top graphic on the right) and triggering units collisions.

Finally, there are still two points on which our reactive *decentralized* model can be challenged with "optimality":

- local optimum which could stuck units: concave (strong) repulsors (static terrain, very high damage field) could trap our reactive, small look-ahead unit. A pragmatic solution to that is to remember that the Obj sensory inputs come from the pathfinder and have

its influence to grow when in difficult situations (not moved for a long time, concave shape detected...). Another solution is inspired by ants: simply release a repulsive field (a repulsive "pheromone") behind the unit and it will be repulsed by places it already visited instead of oscillating around the local optima (see Fig. B.2).

- collision due to concurrency for "best" positions: as seen in Figure. 5.5, units may compete for a well of potential. The solution that we use is to sample in the Dir distribution, which gives better results than picking the most probable direction as soon as there are many units. Another solution, inspired by [**?**], would be for the Bayesian units to communicate their Dir distribution to the units group which would give orders that optimize either the sum of probabilities, or the minimal discrepancy in dissatisfaction, or the survival of costly units (as shown in Fig. B.3)...

We have implemented this model in StarCraft, and it outperforms the original AI as well as other bots (we had a tie with the winner of AIIDE 2010 StarCraft micro-management competition, winning with ranged units and losing with contact units). Our approach does not require vertical integration of higher level goals, as opposed to CBR and reactive planning [**??**], it can have a completely different model above feeding sensory inputs like Obj_i. It scales well with the number of units to control thanks to the absence of communication at the unit level, and is more robust and maintainable than a FSM [**?**].

Chapter 6

Tactics

Probability theory can tell us how our hypothesis fares relative to the alternatives that we have specified; it does not have the creative imagination to invent new hypotheses for us.

E.T. **?**

We present a Bayesian model for tactical decision-making in real-time strategy games. The main idea is to adapt the model to inputs from (possibly) biased heuristics. We evaluated the model in prediction of the enemy tactics on professional gamers data. This work was accepted for publication in [**?**].

- Problem: make the most efficient tactical decisions (attacks and defenses) in the absolute (knowing everything: armies positions, players intentions, effects of each possible actions).
- Problem that we solve: make the most efficient tactical decisions (in average) knowing what we saw from the opponent and our model of the game.
- Type: prediction is problem of *inference* or *plan recognition* from *partial observations*; adaptation given what we know is a problem of *decision-making under uncertainty.*
- Complexity: at least PSPACE-complete[1]. Our solutions are real-time on a laptop.

[1] the space of tactics can be seen as a finite combination of possible military actions and geographical positions, and thus tactics can be modeled as a POMDP*

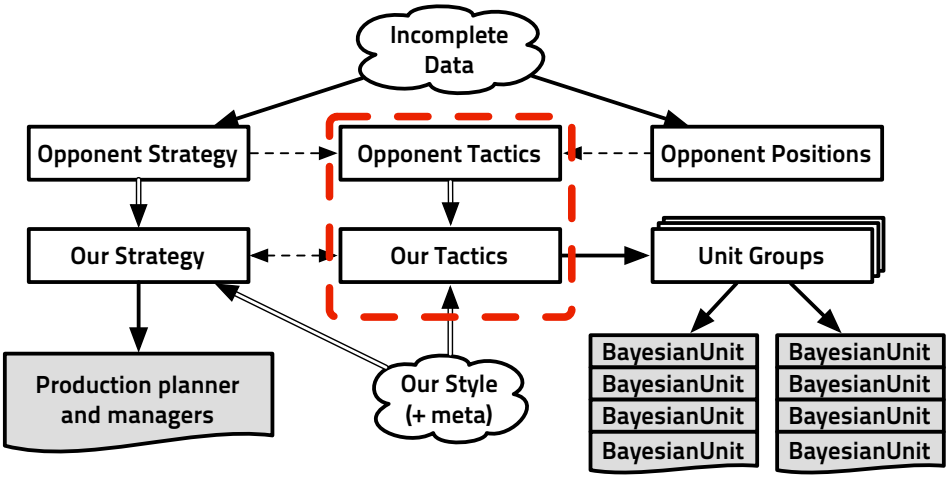


Figure 6.1: Information-centric view of the architecture of the bot, the part concerning this chapter (tactics) is in the dotted rectangle. Dotted arrows represent constraints on what is possible, plain simple arrows represent simple (real) values, either from data or decisions, and double arrows represent probability distributions on possible values. The grayed surfaces are the components actuators (passing orders to the game).

6.1 What are Tactics?

In their study on human like characteristics in RTS games, Hagelbäck and Johansson **?** found out that "tactics was one of the most successful indicators of whether the player was human or not". Tactics are in between strategy (high-level) and micro-management (lower-level), as seen in Fig. 4.5. We propose a model which can either predict enemy attacks or give us a distribution on where and how we should attack the opponent. Information from the higher-level strategy constrains what types of attacks are possible. As shown in Fig. 6, information from units positions (or possibly an enemy units particle filter as in **?** or Chapter 9) constrains where the armies can possibly be in the future. In the context of our StarCraft bot, once we have a decision: we generate a goal (attack order) passed to units groups. A Bayesian model for micro-management **?**, in which units are attracted or repulsed by dynamic (goal, units, damages) and static (terrain) influence maps, actually moves the units in StarCraft. Other works on strategy prediction **??**, which will be presented further (Chapter 7) allows us to infer the enemy tech tree and strategies from incomplete information (due to the fog of war).

Units have different abilities, which leads to different possible tactics. Each faction has invisible (temporarily or permanently) units, flying transport units, flying attack units and ground units. Some units can only attack ground or air units, some others have splash damage attacks, immobilizing or illusion abilities. Fast and mobile units are not cost-effective in head-to-head fights against slower bulky units. We used the gamers' vocabulary to qualify different types of tactics: *ground* attacks (raids or pushes) are the most normal kind of attacks, carried by basic units which cannot fly. Then comes *air* attacks (air raids), which use flying units mobility to quickly deal damage to undefended spots. *Invisible* attacks exploit the weaknesses (being them positional or technological) in detectors of the enemy to deal damage without retaliation.

Finally, *drops* are attacks using ground units transported by air, combining flying units mobility with cost-effectiveness of ground units, at the expense of vulnerability during transit.

6.2 Related Works

Aha et al. **?** used case-based reasoning (CBR) to perform dynamic tactical plan retrieval (matching) extracted from domain knowledge in Wargus. Ontañó et al. **?** based their real-time case-based planning (CBP) system on a plan dependency graph which is learned from human demonstration in Wargus. A case based behavior generator spawn missing goals which are missing from the current state and plan according to the recognized state. In **??**, they used a knowledge-based approach to perform situation assessment to use the right plan, performing runtime adaptation by monitoring its performance. Sharma et al. **?** combined CBR and reinforcement learning to enable reuse of tactical plan components. **?** used richly parametrized CBR for strategic and tactical AI in Spring (Total Annihilation open source clone). Cadena and Garrido **?** used fuzzy CBR (fuzzy case matching) for strategic and tactical planning. Chung et al. **?** adapted Monte-Carlo tree search (MCTS) to planning in RTS games and applied it to a capture-the-flag mod of Open RTS. Balla and Fern **?** applied upper confidence bounds on trees (UCT: a MCTS algorithm) to tactical assault planning in Wargus.

In Starcraft, Weber et al. **??** produced tactical goals through reactive planning and goal-driven autonomy, finding the more relevant goal(s) to follow in unforeseen situations. Kabanza et al. **?** performs plan and intent recognition to find tactical opportunities. On spatial and temporal reasoning, Forbus et al. **?** presented a tactical qualitative description of terrain for wargames through geometric and pathfinding analysis. Perkins **?** automatically extracted choke points and regions of StarCraft maps from a pruned Voronoi diagram, which we used for our regions representations. Wintermute et al. **?** used a cognitive approach mimicking human attention for tactics and units control. Ponsen et al. **?** developed an evolutionary state-based tactics generator for Wargus. Finally, Avery et al. **?** and Smith et al. **?** co-evolved influence map trees for spatial (tactical) reasoning in RTS games.

Our approach (and bot architecture, depicted in Fig. 6) can be seen as goal-driven autonomy **?** dealing with multi-level reasoning by passing distributions (without any assumption about how they were obtained) on the module input. Using distributions as messages between specialized modules makes dealing with uncertainty first class, this way a given model do not care if the uncertainty comes from incompleteness in the data, a complex and biased heuristic, or another probabilistic model. We then take a decision by sampling or taking the most probable value in the output distribution. Another particularity of our model is that it allows for prediction of the enemy tactics using the same model with different inputs. Finally, our approach is not exclusive to most of the techniques presented above, and it could be interesting to combine it with UCT **?** and more complex/precise tactics generated through planning.

6.3 A Bayesian Tactical Model

Dataset

We downloaded more than 8000 replays to keep 7649 uncorrupted, 1v1 replays of very high level StarCraft games (pro-gamers leagues and international tournaments) from specialized websites[2][3][4], we then ran them using BWAPI[5] and dumped units positions, pathfinding and regions, resources, orders, vision events, for attacks (we trigger an attack tracking heuristic when one unit dies and there are at least two military units around): types, positions, outcomes. Basically, every BWAPI event was recorded, the dataset and its source code are freely available[6].

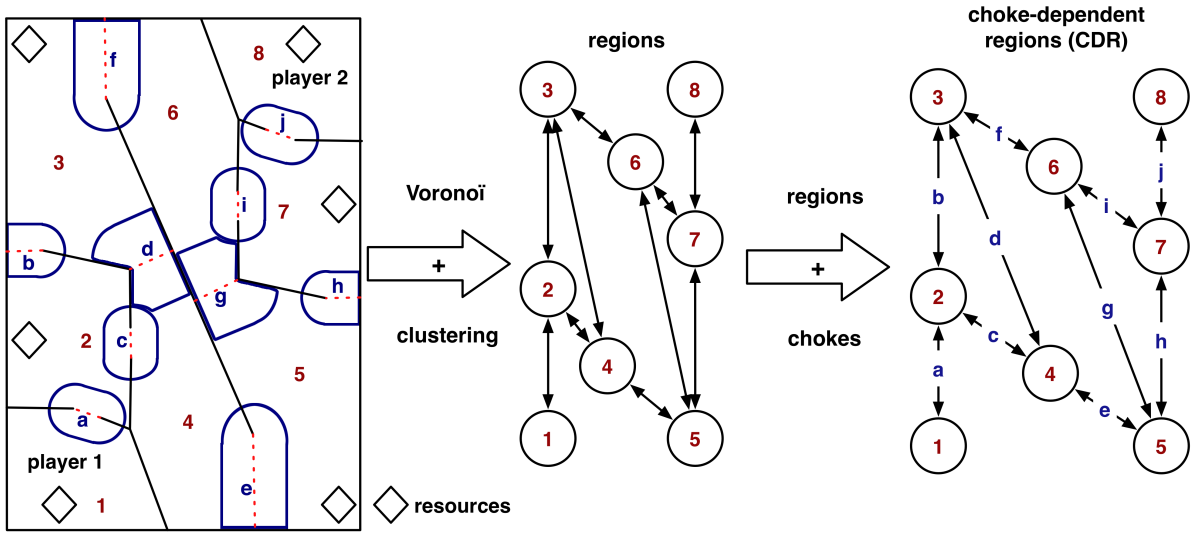


Figure 6.2: A very simple map on the left, which is transformed into regions (between chokes in dotted red lines) by Voronoi tesselation and clustering. These plain regions (numbers in red) are then augmented with choke-dependent regions (letters in blue)

We used two kinds of regions: BroodWar Terrain Analyser (BWTA) regions and choke-dependent (choke-centered) regions. BWTA regions are obtained from a pruned Voronoi diagram on walkable terrain [**?**] and give regions for which chokes are the boundaries. As battles often happens at chokes, choke-dependent regions are created by doing an additional (distance limited) Voronoi tesselation spawned at chokes, its regions set is $(regions \setminus chokes) \cup chokes$. Figure 6.2 illustrate regions and choke-dependant regions (CDR). Results for choke-dependent regions are not fully detailed.

Tactical Model

The idea is to have (most probably biased) lower-level heuristics from units observations which produce information exploitable at the tactical level, and take some advantage of strategic

[2] `http://www.teamliquid.net`
[3] `http://www.gosugamers.net`
[4] `http://www.iccup.com`
[5] `http://code.google.com/p/bwapi/`
[6] `http://snippyhollow.github.com/bwrepdump/`

inference too. The advantages are that 1) learning will de-skew the model output from biased heuristic inputs 2) the model is agnostic to where input variables' values come from 3) the updating process is the same for supervised learning and for reinforcement learning.

We note $s^{a \text{ or } d}_{unit\ type}(r)$ for the balanced score of units from attacker or defender (^{a} or ^{b}) of a given type in region r. The balanced score of units is just the sum on all units of each unit score ($= minerals\_value + \frac{4}{3} gas\_value + 50 supply\_value$). The heuristics we used in our benchmarks (which we could change) are:

$$economical\_score^d(r) = \frac{s^d_{workers}(r)}{\sum_{i \in regions} s^d_{workers}(i)}$$

$$tactical\_score^d(r) = \sum_{i \in regions} s^d_{army}(i) \times dist(i,r)^{-1.5}$$

We used $^{-1.5}$ such that the tactical value of a region in between two halves of an army, each at distance 2, would be higher than the tactical value of a region at distance 4 of the full (same) army. For flying units, *dist* is the Euclidean distance, while for ground units it takes pathfinding into account.

$$ground\_defense^d(r) = \frac{s^d_{can\_attack\_ground}(r)}{s^a_{ground\_units}(r)}$$

$$air\_defense^d(r) = \frac{s^d_{can\_attack\_air}(r)}{s^a_{air\_units}(r)}$$

$$invis\_defense^d(r) = number^d_{detectors}$$

We preferred to discretize continuous values to enable quick complete computations. An other strategy would keep more values and use Monte Carlo sampling for computation. We think that discretization is not a concern because 1) heuristics are simple and biased already 2) we often reason about imperfect information and this uncertainty tops discretization fittings.

Variables

With n regions, we have:

- $A_{1:n} \in \{true, false\}$, A_i: attack in region i or not?
- $E_{1:n} \in \{no, low, high\}$, E_i is the discretized economical value of the region i for the defender. We choose 3 values: *no* workers in the regions, *low*: a small amount of workers (less than half the total) and *high*: more than half the total of workers in this region i.
- $T_{1:n} \in discrete\ levels$, T_i is the tactical value of the region i for the defender, see above for an explanation of the heuristic. Basically, T is proportional to the proximity to the defender's army. In benchmarks, discretization steps are $0, 0.05, 0.1, 0.2, 0.4, 0.8$ (log_2 scale).
- $TA_{1:n} \in discrete\ levels$, TA_i is the tactical value of the region i for the attacker (see above).

- $B_{1:n} \in \{true, false\}$, B_i tells if the region belongs (or not) to the defender. $\mathrm{P}(B_i = true) = 1$ if the defender has a base in region i and $\mathrm{P}(B_i = false) = 1$ if the attacker has one. Influence zones of the defender can be measured (with uncertainty) by $\mathrm{P}(B_i = true) \geq 0.5$ and vice versa.

- $H_{1:n} \in \{ground, air, invisible, drop\}$, H_i: in predictive mode: how we will be attacked, in decision-making: how to attack, in region i.

- $GD_{1:n} \in \{no, low, med, high\}$: ground defense (relative to the attacker power) in region i, result from a heuristic. *no* defense if the defender's army is $\geq 1/10th$ of the attacker's, *low* defense above that and under half the attacker's army, *medium* defense above that and under comparable sizes, *high* if the defender's army is bigger than the attacker.

- $AD_{1:n} \in \{no, low, med, high\}$: same for air defense.

- $ID_{1:n} \in \{no\ detector, one\ detector, several\}$: invisible defense, equating to numbers of detectors.

- $TT \in [\emptyset, building_1, building_2, building_1 \wedge building_2, techtrees, \dots]$: all the possible technological trees for the given race. For instance $\{pylon, gate\}$ and $\{pylon, gate, core\}$ are two different Tech Trees, see chapter **??**.

- $HP \in \{ground, ground \wedge air, ground \wedge invis, ground \wedge air \wedge invis, ground \wedge drop, ground \wedge air \wedge drop, ground \wedge invis \wedge drop, ground \wedge air \wedge invis \wedge drop\}$: **h**ow **p**ossible types of attacks, directly mapped from TT information. In prediction, with this variable, we make use of what we can infer on the opponent's strategy **??**, in decision-making, we know our own possibilities (we know our tech tree as well as the units we own).

Finally, for some variables, we take uncertainty into account with "soft evidences": for instance for a region in which no player has a base, we have a soft evidence that it belongs more probably to the player established closer. In this case, for a given region, we introduce the soft evidence variable(s) B' and the coherence variable λ_B and impose $\mathrm{P}(\lambda_B = 1|B, B') = 1.0\ iff\ B = B'$, else $\mathrm{P}(\lambda_B = 1|B, B') = 0.0$; while $\mathrm{P}(\lambda_B|B, B')\mathrm{P}(B')$ is a new factor in the joint distribution. This allows to sum over $\mathrm{P}(B')$ distribution (soft evidence).

Decomposition

The joint distribution of our model contains soft evidence variables for all input family variables $(E, T, TA, B, GD, AD, ID, HP)$ to be as general as possible, *i.e.* to be able to cope with all possible uncertainty (from incomplete information) that may come up in a game. To avoid being too verbose, we explain the decomposition only with the soft evidence for the family of variables

B, the principle holds for all other soft evidences. For the n considered regions, we have:

$$\begin{aligned}
& \mathrm{P}(A_{1:n}, E_{1:n}, T_{1:n}, TA_{1:n}, B_{1:n}, B'_{1:n}, \lambda_{B,1:n}, && (6.1)\\
& H_{1:n}, GD_{1:n}, AD_{1:n}, ID_{1:n}, HP, TT) && (6.2)\\
= & \prod_{i=1}^{n} [\mathrm{P}(A_i)\mathrm{P}(E_i, T_i, TA_i, B_i|A_i) && (6.3)\\
& \mathrm{P}(\lambda_{B,i}|B_{1:n}, B'_{1:n})\mathrm{P}(B'_{1:n}) && (6.4)\\
& \mathrm{P}(AD_i, GD_i, ID_i|H_i)\mathrm{P}(H_i|HP)]\,\mathrm{P}(HP|TT)\mathrm{P}(TT) && (6.5)
\end{aligned}$$

Forms and Learning

We will explain the forms for a given/fixed i region number:

- $\mathrm{P}(A)$ is the prior on the fact that the player attacks in this region, in our evaluation we set it to $n_{battles}/(n_{battles}+n_{not\ battles})$. In a given match it should be initialized to uniform and progressively learn the preferred attack regions of the opponent for predictions, learn the regions in which our attacks fail or succeed for decision-making.

- $\mathrm{P}(E, T, TA, B|A)$ is a covariance table of the economical, tactical (both for the defender and the attacker), belonging scores where an attacks happen. We just use Laplace succession law ("add one" smoothing) **?** and count the co-occurrences, thus almost performing maximum likelihood learning of the table.

- $\mathrm{P}(\lambda_B|B, B') = 1.0\ iff\ B = B'$ is just a coherence constraint.

- $\mathrm{P}(AD, GD, ID|H)$ is a covariance table of the air, ground, invisible defense values depending on how the attack happens. As for $\mathrm{P}(E, T, TA, B|A)$, we use a Laplace's law of succession to learn it.

- $\mathrm{P}(H|HP)$ is the distribution on how the attack happens depending on what is possible. Trivially $\mathrm{P}(H = ground|HP = ground) = 1.0$, for more complex possibilities we have different maximum likelihood multinomial distributions on H values depending on HP.

- $\mathrm{P}(HP|TT)$ is the direct mapping of what the tech tree allows as possible attack types: $\mathrm{P}(HP = hp|TT) = 1$ is a function of TT (all $\mathrm{P}(HP \neq hp|TT) = 0$).

- $\mathrm{P}(TT)$: if we are sure of the tech tree (prediction without fog of war, or in decision-making mode), $\mathrm{P}(TT = k) = 1$ and $\mathrm{P}(TT \neq k) = 0$; otherwise, it allows us to take uncertainty about the opponent's tech tree and balance $\mathrm{P}(HP|TT)$. We obtain a distribution on what is possible ($\mathrm{P}(HP)$) for the opponent's attack types.

There are two approaches to fill up these probability tables, either by observing games (supervised learning), as we did in the evaluation section, or by acting (reinforcement learning). In match situation against a given opponent, for inputs that we can unequivocally attribute to their intention (style and general strategy), we also refine these probability tables (with Laplace's rule of succession). To keep things simple, we just refine $\sum_{E,T,TA} \mathrm{P}(E, T, TA, B|A)$ corresponding to their aggressiveness (aggro) or our successes and failures, and equivalently for $\mathrm{P}(H|HP)$. Indeed, if we sum over E, T and TA, we consider the inclination of our opponent to venture into

enemy territory or the interest that we have to do so by counting our successes with aggressive or defensive parameters. In $\mathrm{P}(H|HP)$, we are learning the opponent's inclination for particular types of tactics according to what is available to their, or for us the effectiveness of our attack types choices.

The model is highly modular, and some parts are more important than others. We can separate three main parts: $\mathrm{P}(E, T, TA, B|A)$, $\mathrm{P}(AD, GD, ID|H)$ and $\mathrm{P}(H|HP)$. In prediction, $\mathrm{P}(E, T, TA, B|A)$ uses the inferred (uncertain) economic (E), tactical (T) and belonging (B) scores of the opponent while knowing our own tactical position fully (TA). In decision-making, we know E, T, B (for us) and estimate TA. In our prediction benchmarks, $\mathrm{P}(AD, GD, ID|H)$ has the lesser impact on the results of the three main parts, either because the uncertainty from the attacker on AD, GD, ID is too high or because our heuristics are too simple, though it still contributes positively to the score. In decision-making, it allows for reinforcement learning to have pivoting tuple values for AD, GD, ID at which to switch attack types. In prediction, $\mathrm{P}(H|HP)$ is used to take $\mathrm{P}(TT)$ (coming from strategy prediction **?**, chapter 7) into account and constraints H to what is possible. For the use of $\mathrm{P}(H|HP)\mathrm{P}(HP|TT)\mathrm{P}(TT)$ in decision-making, see the Results sections.

Questions

For a given region i, we can ask the probability to attack here,

$$\mathrm{P}(A_i = a_i|e_i, t_i, ta_i, \lambda_{B,i} = 1) \tag{6.6}$$

$$= \frac{\sum_{B_i, B_i'} \mathrm{P}(e_i, t_i, ta_i, B_i|a_i)\mathrm{P}(a_i)\mathrm{P}(B_i').P(\lambda_{B,i}|B_i, B_i')}{\sum_{A_i, B_i, B_i'} \mathrm{P}(e_i, t_i, ta_i, B_i|A_i)\mathrm{P}(A_i)\mathrm{P}(B_i')\mathrm{P}(\lambda_{B,i}|B_i, B_i')} \tag{6.7}$$

$$\propto \sum_{B_i, B_i'} \mathrm{P}(e_i, t_i, ta_i, B_i|a_i)\mathrm{P}(a_i)\mathrm{P}(B_i')\mathrm{P}(\lambda_{B,i}|B_i, B_i') \tag{6.8}$$

and the mean by which we should attack,

$$\mathrm{P}(H_i = h_i|ad_i, gd_i, id_i) \tag{6.9}$$

$$\propto \sum_{TT,P} [\mathrm{P}(ad_i, gd_i, id_i|h_i)\mathrm{P}(h_i|HP)\mathrm{P}(HP|TT)\mathrm{P}(TT)] \tag{6.10}$$

For clarity, we omitted some variables couples on which we have to sum (to take uncertainty into account) as for B (and B') above. We always sum over estimated, inferred variables, while we know the one we observe fully. In prediction mode, we sum over TA, B, TT, P; in decision-making, we sum over E, T, B, AD, GD, ID. The complete question that we ask our model is $\mathrm{P}(A, H|FullyObserved)$. The maximum of $\mathrm{P}(A, H)$ may not be the same as the maximum of $\mathrm{P}(A)$ or $\mathrm{P}(H)$, for instance think of a very important economic zone that is very well defended, it may be the maximum of $\mathrm{P}(A)$, but not once we take $\mathrm{P}(H)$ into account. Inversely, some regions are not defended against anything at all but present little or no interest. Our joint distribution 6.3 can be rewritten: $\mathrm{P}(Searched, FullyObserved, Estimated)$, so we ask:

$$\mathrm{P}(A_{1:n}, H_{1:n}|FullyObserved) \tag{6.11}$$

$$\propto \sum_{Estimated} \mathrm{P}(A_{1:n}, H_{1:n}, Estimated, FullyObserved) \tag{6.12}$$

The full Bayesian program of the model is as follows (6.13):

$$
\begin{cases}
\text{Bayesian program} \\
\begin{cases}
\text{Description} \\
\begin{cases}
\text{Specification } (\pi) \\
\begin{cases}
Variables \\
A_{1:n}, E_{1:n}, T_{1:n}, TA_{1:n}, B_{1:n}, B'_{1:n}, \lambda_{B,1:n}, \\
H_{1:n}, GD_{1:n}, AD_{1:n}, ID_{1:n}, HP, TT \\
Decomposition \\
\mathrm{P}(A_{1:n}, E_{1:n}, T_{1:n}, TA_{1:n}, B_{1:n}, B'_{1:n}, \lambda_{B,1:n}, \\
H_{1:n}, GD_{1:n}, AD_{1:n}, ID_{1:n}, HP, TT) \\
= \prod_{i=1}^{n} \left[\mathrm{P}(A_i) \mathrm{P}(E_i, T_i, TA_i, B_i | A_i) \right. \\
\mathrm{P}(\lambda_{B,i} | B_{1:n}, B'_{1:n}) \mathrm{P}(B'_{1:n}) \\
\left. \mathrm{P}(AD_i, GD_i, ID_i | H_i) \mathrm{P}(H_i | HP) \right] \mathrm{P}(HP | TT) \mathrm{P}(TT) \\
Forms \\
\mathrm{P}(A_r) \text{ prior on attack in region } i \\
\mathrm{P}(E, T, TA, B | A) \text{ covariance/probability table} \\
\mathrm{P}(\lambda_B | B, B') = 1.0 \; iff \; B = B', \; else \; \mathrm{P}(\lambda_B | B, B') = 0.0 \; (functional \; Dirac) \\
\mathrm{P}(AD, GD, ID | H) \text{ covariance/probability table} \\
\mathrm{P}(H | HP) = Categorical(4, HP) \\
\mathrm{P}(HP = hp | TT) = 1.0 \; iff \; TT \to hp, \; else \; \mathrm{P}(HP | TT) = 0.0 \\
\mathrm{P}(TT) \text{ comes from a strategic model}
\end{cases} \\
Identification \; (using \; \delta) \\
\mathrm{P}(A_r = true) = \frac{n_{battles}}{n_{battles} + n_{not \; battles}} = \frac{\mu_{battles/game}}{\mu_{regions/map}} \text{ (probability to attack a region)} \\
\text{it could be learned online (preference of the opponent) :} \\
\mathrm{P}(A_r = true) = \frac{1 + n_{battles}(r)}{2 + \sum_{i \in regions} n_{battles}(i)} \text{ (online for each game)} \\
\mathrm{P}(E = e, T = t, TA = ta, B = b | A = True) = \frac{1 + n_{battles}(e,t,ta,b)}{|E| \times |T| \times |TA| \times |B| + \sum_{E,T,TA,B} n_{battles}(E,T,TA,B)} \\
\mathrm{P}(AD = ad, GD = gd, ID = id | H = h) = \frac{1 + n_{battles}(ad,gd,id,h)}{|AD| \times |GD| \times |ID| + \sum_{AD,GD,ID} n_{battles}(AD,GD,ID,h)} \\
\mathrm{P}(H = h | HP = hp) = \frac{1 + n_{battles}(h,hp)}{|H| + \sum_{H} n_{battles}(H,hp)}
\end{cases} \\
Questions \\
\forall i \in regions \mathrm{P}(A_i | e_i, t_i, ta_i, \lambda_{B,i} = 1) \\
\forall i \in regions \mathrm{P}(H_i | ad_i, gd_i, id_i) \\
\mathrm{P}(A, H | FullyObserved)
\end{cases}
\end{cases}
$$

6.4 Results on StarCraft

Learning

To measure fairly the prediction performance of such a model, we applied "leave-100-out" cross-validation from our dataset: as we had many games (see Table. 6.1), we set aside 100 games of each match-up for testing (with more than 1 battle per match: rather ≈ 15 battles/match) and train our model on the rest. We write match-ups XvY with X and Y the first letters of the factions

involved (Protoss, Terran, Zerg). Note that mirror match-ups (PvP, TvT, ZvZ) have less games but twice as many attacks from a given faction. Learning was performed as explained in III.B.3: for each battle in r we had one observation for: $\mathrm{P}(e_r, t_r, ta_r, b_r | A = true)$, and $\#regions - 1$ observations for the i regions which were not attacked: $\mathrm{P}(e_{i\neq r}, t_{i\neq r}, ta_{i\neq r}, b_{i\neq r} | A = false)$. For each battle of type t we had one observation for $P(ad, gd, id | H = t)$ and $P(H = t|p)$. By learning with a Laplace's law of succession **?**, we allow for unseen event to have a non-null probability.

An exhaustive presentation of the learned tables is out of the scope of this chapter, but we displayed interesting cases in which the learned probability tables meet concur with human expertise in Figures 6.3,6.4,6.5. In Fig. 6.3, we see that air raids/attacks are quite risk averse and it is two times more likely to attack a region with less than 1/10th of the flying force in anti-aircraft warfare than to attack a region with up to one half of our force. We can also notice than drops are to be preferred either when it is safe to land (no anti-aircraft defense) or when there is a large defense (harassment tactics). In Fig. 6.4 we can see that, in general, there are as many ground attacks at the sum of other types. The two top graphs show cases in which the tech of the attacker was very specialized, and, in such cases, the specificity seems to be used. In particular, the top right graphic may be corresponding to a "fast Dark Templars rush". Finally, Fig. 6.5 shows the transition between two types of encounters: tactics aimed at engaging the enemy army (a higher T value entails a higher $\mathrm{P}(A)$) and tactics aimed at damaging the enemy economy (at high E, we look for opportunities to attack with a small army where T is lower).

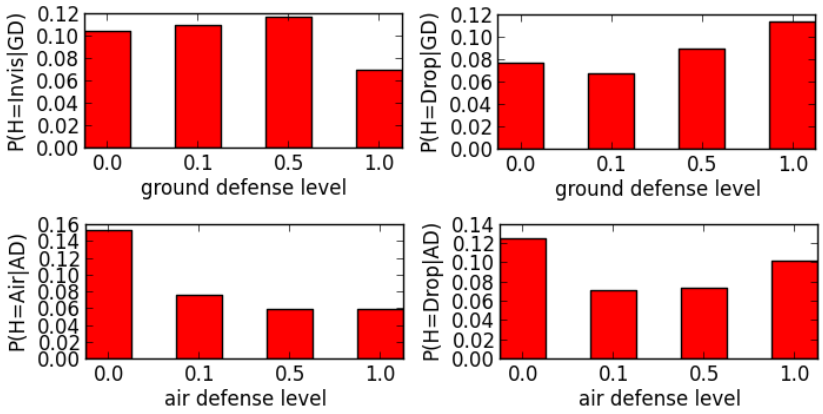


Figure 6.3: (top) $\mathrm{P}(H = invis)$ and $\mathrm{P}(H = drop)$ for varying values of GD (summed on other variables); (bottom) $\mathrm{P}(H = air)$ and $\mathrm{P}(H = drop)$ for varying values of AD (summed on other variables), for Terran in TvP. We can see that it is far more likely that invisible ("sneaky") attacks happen where there is low ground presence (top left plot). For drops, we understand that the high value for $\mathrm{P}(H = drop|GD = 1.0)$ is caused by the fact that drop armies are small and this value corresponds to drops which are being expected by the defender. Drops at lower values of GD correspond to unexpected (surprise) drops. As ground units are more cost efficient than flying units in a static battle, we see that both $\mathrm{P}(H = invis|AD = 0.0)$ and $\mathrm{P}(H = drop|AD = 0.0)$ are much more probable than situations with air defenses.

Prediction Performance

We learned and tested one model for each race and each match-up. As we want to predict *where* ($\mathrm{P}(A_{1:n})$) and *how* ($\mathrm{P}(H_{battle})$) the next attack will happen to us, we used inferred enemy TT

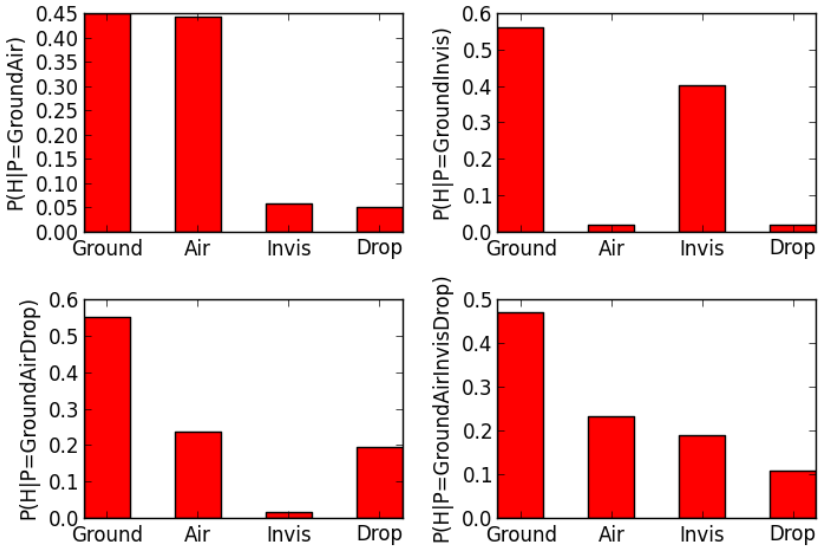


Figure 6.4: P($H|HP$) for varying values of H and for different values of P (derived from inferred TT), for Protoss in PvT. Conditioning on what is possible given the *tech tree* gives a lot of information about what attack types are possible or not. More interestingly, it clusters the game phases in different tech levels and allows for learning the relative distributions of attack types with regard to each phase. For instance, the last (bottom right) plot shows the distribution on attack types at the end of a technologically complete game.

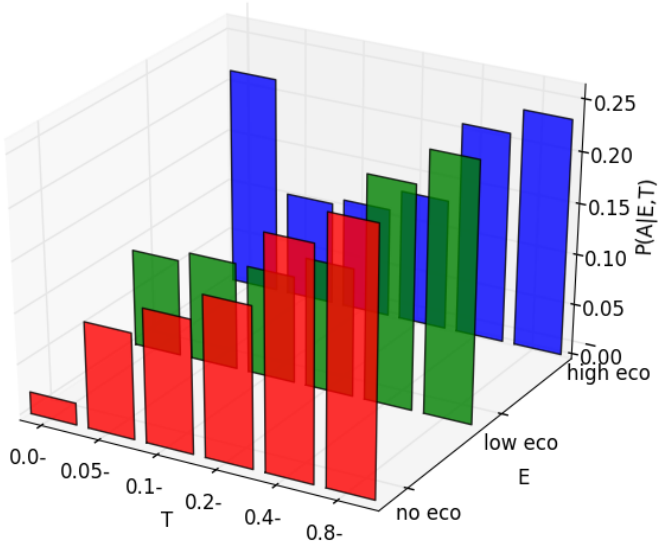


Figure 6.5: P(A) for varying values of E and T, summed on the other variables, for Terran in TvT. Higher economical values is strongly correlated with surprise attacks with small tactical squads and no defenses, which almost never happens in open fields ("no eco") as this would lead to very unbalanced battles (in terms of army sizes): it would not benefit the smaller party, which can flee and avoid confrontation, as opposed to when defending their base.

(to produce P) and TA, our scores being fully known: E, T, B, ID. We consider GD, AD to be fully known even though they depend on the attacker force, we should have some uncertainty

on them, but we tested that they accounted (being known instead of fully unknown) for 1 to 2% of P(H) accuracy (in prediction) once P was known. We should point that pro-gamers scout very well and so it allows for a highly accurate TT estimation with **?**. Training requires to recreate battle states (all units positions) and count parameters for 5,000 to 30,000 battles. Once that is done, inference is very quick: a look-up in a probability table for known values and $\#F$ look-ups for free variables F on which we sum. We chose to try and predict the next battle 30 seconds before it happens, 30 seconds being an approximation of the time needed to go from the middle of a map (where the entropy on "next battle position" is maximum as the army's average distance to all other regions is minimal) to any region by ground, so that the prediction is useful for the defender (they can position their army).

The model code[7] (for learning and testing) as well as the datasets (see above) are freely avaible. Raw results of predictions of positions and types of attacks 30 seconds before they happen are presented in Table. 6.1: for instance the bold number (38.0) corresponds to the percentage of good positions (regions) predictions (30 sec before event) which were ranked 1st in the probabilities on $A_{1:n}$ for Protoss attacks against Terran (PvT). The measures on *where* corresponds to the percentage of good prediction and the mean probability for given ranks in P($A_{1:n}$) (to give a sense of the shape of the distribution). As the most probable The measures on *how* corresponds to the percentage of good predictions for the most probable P(H_{battle}) and the number of such battles seen in the test set for given attack types. We particularly predict well ground attacks (trivial in the early game, less in the end game) and, interestingly, Terran and Zerg drop attacks. The *where & how* row corresponds to the percentage of good predictions for the maximal probability in the joint P($A_{1:n}, H_{1:n}$): considering only the most probable attack (more information is in the rest of the distribution, as shown for *where*!) according to our model, we can predict *where* **and** *how* an attack will occur in the next 30 seconds $\approx 1/4$th of the time. Finally, note that scores are not ridiculous 60 seconds before the attack neither (obviously, TT, and thus P, are not so different, nor are B and E): PvT *where* top 4 ranks are 35.6, 8.5, 7.7, 7.0% good versus 38.0, 16.3, 8.9, 6.7% 30 seconds before; *how* total precision 60 seconds before is 70.0% vs. 72.4%, *where & how* maximum probability precision is 19.9% vs. 23%.

When we are mistaken, the mean ground distance (pathfinding wise) of the most probable predicted region to the good one (where the attack happens) is 1223 pixels (38 build tiles, or 2 screens in StarCraft's resolution), while the mean max distance on the map is 5506 (172 build tiles). Also, the mean number of regions by map is 19, so a random *where* (attack destination) picking policy would have a correctness of 1/19 (5.23%). For choke-centered regions, the numbers of good *where* predictions are lower (between 24% and 32% correct for the most probable) but the mean number of regions by map is 42. For *where & how*, a random policy would have a precision of 1/(19*4), and even a random policy taking the high frequency of ground attacks into account would at most be $\approx$ 1/(19*2) correct. For the location only (*where* question), we also counted the mean number of different regions which were attacked in a given game (between 3.97 and 4.86 for regions, depending on the match-up, and between 5.13 and 6.23 for choke-dependent regions). The ratio over these means would give the best prediction rate we could expect from a *baseline heuristic* based solely on the location data. These are attacks that actually happened, so the number of regions a player have to be worried about is at least this one (or more, for regions which were not attacked during a game but were potential targets). This

[7] `https://github.com/SnippyHolloW/AnalyzeBWData`

baseline heuristic would yield (depending on the match-up) prediction rates between 20.5 and 25.2% for regions, versus our 32.8 to 40.9%, and between 16.1% and 19.5% for choke-dependent regions, versus our 24% to 32%. Note that our current model consider a uniform prior on regions (no bias towards past battlefields) and that we do not incorporate any derivative of the armies' movements. There is no player modeling at all: learning and fitting the mean player's tactics is not optimal, so we should specialize the probability tables for each player. Also, we use all types of battles in our training and testing. Short experiments showed that if we used only attacks on bases, the probability of good *where* predictions for the maximum of $\mathrm{P}(A_{1:n})$ goes above 50% (which is not a surprise, there are far less bases than regions in which attacks happen). To conclude on tactics positions prediction: if we sum the 2 most probable regions for the attack, we are right at least half the time; if we sum the 4 most probable (for our robotic player, it means it prepares against attacks in 4 regions as opposed to 19), we are right $\approx 70\%$ of the time.

Mistakes on the type of the attack are high for invisible attacks: while these tactics can definitely win a game, the counter is strategic (it is to have detectors technology deployed) more than positional. Also, if the maximum of $\mathrm{P}(H_{battle})$ is wrong, it doesn't mean than $\mathrm{P}(H_{battle} = good) = 0.0$ at all! The result needing improvements the most is for air tactics, because countering them really is positional, see our discussion in the conclusion.

In Game Decision-Making

In a StarCraft game, our bot has to make decisions about where and how to attack or defend, it does so by reasoning about opponent's tactics, bases, its priors, and under strategic constraints (Fig. **??**). Once a decision is taken, the output of the tactical model is an offensive or defensive goal. There are different military goal types (base defense, ground attacks, air attacks, drops...), and each type of goal has pre-requisites (for instance: a drop goal needs to have the control of a dropship and military units to become active). The spawned goal then autonomously sets objectives for Bayesian units **?**, sometimes procedurally creating intermediate objectives or canceling itself in the worst cases.

The destinations of goals are from $\mathrm{P}(A)$, while the type of the goal comes from $\mathrm{P}(H)$. In input, we fully know tactical scores of the regions according to our military units placement TA (we are the attacker), what is possible for us to do P (according to units available) and we estimate E, T, B, ID, GD, AD from past (partial) observations. Estimating T is the most tricky of all because it may be changing fast, for that we use a units filter which just decays probability mass of seen units. An improvement would be to use a particle filter **?**, with a learned motion model. From the joint 6.11 $\mathrm{P}(A_{1:n}, H_{1:n}|ta, p, tt)$ may arise a couple i, H_i more probable than the most probables $\mathrm{P}(A_i)$ and $\mathrm{P}(H_j)$ taken separately (the case of an heavily defended main base and a small unprotected expand for instance). Fig. 6.6 displays the mean $\mathrm{P}(A, H)$ for Terran (in TvZ) attacks decision-making for the most 32 probable type/region tactical couples. It is in this kind of landscape (though more steep because Fig. 6.6 is a mean) that we sample (or pick the most probable couple) to take a decision. Also, we may spawn defensive goals countering the attacks that we predict from the opponent.

Finally, we can steer our technological growth towards the opponent's weaknesses. A question that we can ask our model (at time t) is $\mathrm{P}(TT)$, or, in two parts: we first find i, h_i which maximize

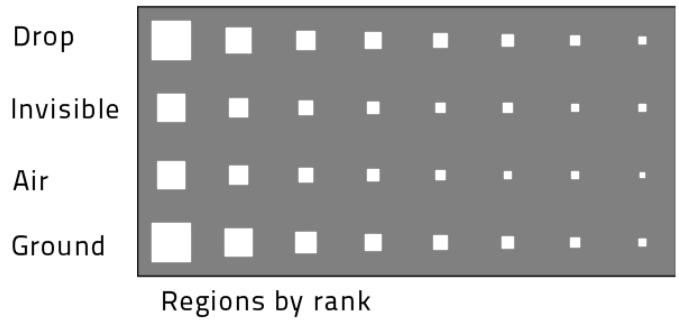


Figure 6.6: Mean $\mathrm{P}(A, H)$ for all H values and the top 8 $\mathrm{P}(A_i, H_i)$ values, for Terran in TvZ. The larger the white square area, the higher $\mathrm{P}(A_i, H_i)$. A simple way of taking a tactical decision according to this model, and the learned parameters, is by sampling in this distribution.

$\mathrm{P}(A, H)$ at time $t + 1$, and then ask a more directive:

$$\mathrm{P}(TT|h_i) \propto \sum_{P} \mathrm{P}(h_i|HP)\mathrm{P}(HP|TT)\mathrm{P}(TT)$$

so that it gives us a distribution on the tech trees (TT) needed to be able to perform the wanted attack type. To take a decision on our technology direction, we can consider the distances between our current tt^t and all the probable values of TT^{t+1}.

6.5 Discussion

There are three main research directions for possible improvements:

- improving the underlying heuristics: the heuristics presented here are quite simple but they may be changed, and even removed or added, for another RTS or FPS, or for more performance. In particular, our "defense against invisible" heuristic could take detector positioning/coverage into account. Our heuristic on tactical values can also be reworked to take terrain tactical values into account (chokes and elevation in StarCraft). For the estimated position of enemy units, we could use a particle filter **?** with a motion model (at least one for ground units and one for flying units).

- improving the dynamic of the model: there is room to improve the dynamics of the model: considering the prior probabilities to attack in regions given past attacks and/or considering evolutions of the T,TA,B,E values (derivatives) in time. The discretizations that we used may show their limits, though if we want to use continuous values, we need to setup a more complicated learning and inference process (MCMC sampling).

- improving the model itself: finally, one of the strongest assumptions (which is a drawback particularly for prediction) of our model is that the attacking player is always considered to attack in this most probable regions. While this would be true if the model was complete (with finer army positions inputs and a model of what the player thinks), we believe such an assumption of completeness is far fetched. Instead we should express that incompleteness in the model itself and have a "player decision" variable $D \sim Multinomial(\mathrm{P}(A_{1:n}, H_{1:n}), player)$.

We have presented a Bayesian tactical model for RTS AI, which allows both for opposing tactics prediction and autonomous tactical decision-making. Being a probabilistic model, it deals with uncertainty easily, and its design allows easy integration into multi-granularity (multi-scale) AI systems as needed in RTS AI. Without any temporal dynamics, the position prediction is above a baseline heuristic ([32.8-40.9%] vs [20.5-25.2%]). Moreover, its exact prediction rate of the joint position and tactical type is in [23-32.8]% (depending on the match-up), and considering the 4 most probable regions it goes up to $\approx 70\%$. More importantly, it allows for tactical decision-making under (technological) constraints and (state) uncertainty. It can be used in production thanks to its low CPU and memory footprint. The dataset, its documentation[8], as well as our model implementation[9] (and other data-exploration tools) are free software and can be found online. We plan to use this model in our StarCraft AI competition entry bot as it gives our bot tactical autonomy and a way to adapt to our opponent.

[8] `http://snippyhollow.github.com/bwrepdump/`
[9] `https://github.com/SnippyHolloW/AnalyzeBWData`

Table 6.1: Results summary for multiple metrics at 30 seconds before attack. The number in bold (38.0) is read as "38% of the time, the region i with probability of rank 1 in $P(A_i)$ is the one in which the attack happened 30 seconds later".

%: good predictions Pr: mean probability		Protoss						Terran						Zerg					
		P		T		Z		P		T		Z		P		T		Z	
total # games		1336		7225		6082		7225		1384		6322		6082		6322		598	
measure	rank	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr
where	1	40.9	.334	**38.0**	.329	34.5	.304	35.3	.299	34.4	.295	39.0	0.358	32.8	.31	39.8	.331	37.2	.324
	2	14.6	.157	16.3	.149	13.0	.152	14.3	.148	14.7	.0147	17.8	.174	15.4	.166	16.6	.148	16.9	.157
	3	7.8	.089	8.9	.085	6.9	.092	9.8	.09	8.4	.087	10.0	.096	11.3	.099	7.6	.084	10.7	.100
	4	7.6	.062	6.7	.059	7.9	.064	8.6	.071	6.9	.063	7.0	.062	8.9	.07	7.7	.064	8.6	.07
measure	type	%	N	%	N	%	N	%	N	%	N	%	N	%	N	%	N	%	N
how	G	97.5	1016	98.1	1458	98.4	568	100	691	99.9	3218	76.7	695	86.6	612	99.8	567	67.2	607
	A	44.4	81	34.5	415	46.8	190	40	5	13.3	444	47.1	402	14.2	155	15.8	19	74.2	586
	I	22.7	225	49.6	337	12.9	132	NA	NA	NA	NA	36.8	326	32.6	227	NA	NA	NA	NA
	D	55.9	340	42.2	464	45.2	93	93.5	107	86	1183	62.8	739	67.7	535	81.4	86	63.6	588
total		76.3	1662	72.4	2674	71.9	983	98.4	806	88.5	4850	60.4	2162	64.6	1529	94.7	674	67.6	1802
where & how (%)		32.8		23		23.8		27.1		23.6		30.2		23.3		30.9		26.4	

Chapter 7

Strategy

Strategy without tactics is the slowest route to victory. Tactics without strategy is the noise before defeat.
All men can see these tactics whereby I conquer, but what none can see is the strategy out of which victory is evolved.

Sun Tzu (The Art of War, 476-221 BC)

We NTRO

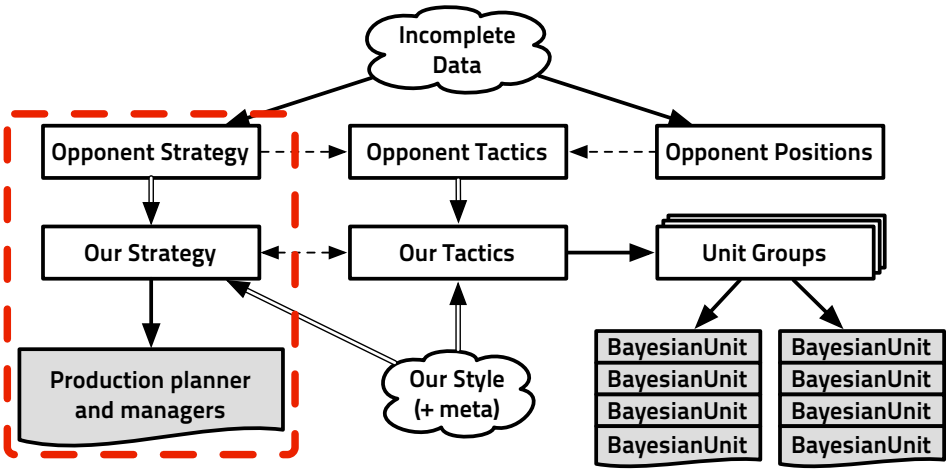


Figure 7.1: Information-centric view of the architecture of the bot, the part concerning this chapter is in the dotted rectangle

- Problem: take the winning strategy in the absolute (knowing everything: complete observations, players intentions, effects of each possible actions).
- Problem that we solve: take the winning strategy (in average) knowing what we saw from the opponent and our model of the game.
- Type: prediction is problem of *inference* or *plan recognition* from *incomplete informations*; adaptation given what we know is a problem of *planning under constraints*.
- Complexity: simple StarCraft decision problems are NP-hard [**?**], basic StarCraft strategy with full information (remember that StarCraft is partially observable) is mappable to the Generalized Geography problem and thus is PSPACE-hard [**?**], Chess [**?**] and Go (with Japanese ko rules) are EXPTIME-complete [**?**]. Our solutions are real-time on a laptop.

7.1 What is a Strategy?

From last chapter, we recall the *tech tree** is a directed acyclic graph which contains the whole technological (buildings and upgrades) development of a player. Also, each unit and building has a *sight range* that provides the player with a view of the map. Parts of the map not in the sight range of the player's units are under *fog of war* and the player ignores what is and happens there. In RTS games jargon, an *opening** denotes the same thing than in Chess: an early game plan for which the player has to make choices. In Chess because one can not move many pieces at once (each turn), in RTS games because during the development phase, one is economically limited and has to choose between economic and military priorities and can only open so many tech paths at once. The opening* corresponds to the first military (tactical) moves that will be performed and, in StarCraft, it corresponds to the 5 (early rushes) to 15 minutes (advanced technology / late push) timespan.

Players have to find out what opening their opponents are doing to be able to effectively deal with the strategy (army composition) and tactics (military moves: where and when) thrown at them. For that, players scout each other and reason about the incomplete information they can bring together about army and buildings composition. This paper presents a probabilistic model able to predict the *opening** of the enemy that is used in a StarCraft AI competition entry bot (see Figure **??**). Instead of hard-coding strategies or even considering plans as a whole, we consider the long term evolution of our tech tree* and the evolution of our army composition separately (but steering and constraining each others), as shown in Fig. **??**. With this model, our bot asks "what units should I produce?" (assessing the whole situation), being able to revise and adapt its production plans.

Later in the game, as the possibilities of strategies "diverge" (as in Chess), there are no longer fixed foundations that we can speak of as for openings. Instead, what is interesting is to know the technologies available to the enemy as well as have a sense of the composition of their army. The players have to adapt to each other technologies and armies compositions either to be able to defend or to attack. Some units are more cost-efficient than others against particular compositions. Some combinations of units play well with each others (for instance biological units with "medics" or a good ratio of strong contact units backed with fragile ranged or artillery units). Finally, some units can be game changing by themselves like cloaking units, detectors, massive area of effects units...

7.2 Related Works

[?]

There are related works in the domains of opponent modeling [**???**]. The main methods used to these ends are case-based reasoning (CBR) and planning or plan recognition [**?????**]. There are precedent works of Bayesian plan recognition [**?**], even in games with Albrecht *et al.* [**?**] using dynamic Bayesian networks to recognize a user's plan in a multi-player dungeon adventure.

Aha *et al.* [**?**] used CBR to perform dynamic plan retrieval extracted from domain knowledge in Wargus (Warcraft II clone). Ontañón *et al.* [**?**] base their real-time case-based planning (CBP) system on a plan dependency graph which is learned from human demonstration. In [**??**], they use CBR and expert demonstrations on Wargus. They improve the speed of CPB by using a decision tree to select relevant features. Hsieh and Sun [**?**] based their work on Aha *et al.*'s CBR [**?**] and used StarCraft replays to construct states and building sequences. Strategies are choices of building construction order in their model.

Schadd *et al.* [**?**] describe opponent modeling through hierarchically structured models of the opponent behaviour and they applied their work to the Spring RTS (Total Annihilation clone). Hoang *et al.* [**?**] use hierarchical task networks (HTN) to model strategies in a first person shooter with the goal to use HTN planners. Kabanza *et al.* [**?**] improve the probabilistic hostile agent task tracker (PHATT [**?**], a simulated HMM for plan recognition) by encoding strategies as HTN.

Dereszynski *et al.* [**?**] used an HMM which states are extracted from (unsupervised) maximum likelihood on the dataset. The HMM parameters are learned from unit counts (both buildings and military units) every 30 seconds and Viterbi inference is used to predict the most likely next states from partial observations.

The work described in this section can be classified as probabilistic plan recognition. Strictly speaking, we present model-based machine learning used for prediction of plans, while our model is not limited to prediction. It performs two levels of plan recognition, both are learned from the replays: tech tree prediction (unsupervised) and opening prediction (semi-supervised or supervised depending on the labeling method).

7.3 Strategy prediction

> *What is of supreme importance in war is to attack the enemy's strategy.*
> Sun Tzu

Replays Labeling

We used Weber and Mateas [**?**] dataset of labeled replays. It is composed of 9316 StarCraft: Broodwar game logs, between $\approx$ 500 and 1300 per *match-up*. A match-up is a set of the two opponents races, Protoss versus Terran (PvT) is a match-up, PvZ is another one. They are distinguished because strategies distribution are very different across match-ups (see Table 7.2). Weber and Mateas used logic rules on building sequences to put their labels, concerning only tier 2 strategies (no tier 1 rushes).

Openings are closely related to *build orders* (BO) but different BO can lead to the same opening and some BO are shared by different openings. Particularly, if we do not take the time

at which the buildings are constructed, we may be wrong too often. For that reason, we tried to label replays with the statistical appearance of key features with a semi-supervised method (see Figure 7.2). Indeed, the purpose of our opening prediction model is to help our StarCraft playing bot to deal with rushes and special tactics. This was not the main focus of Weber and Mateas' labels, which follow more the development of the tech tree. We used the key components of openings that we want to be aware of as features for our labeling algorithm as show in Table 7.1.

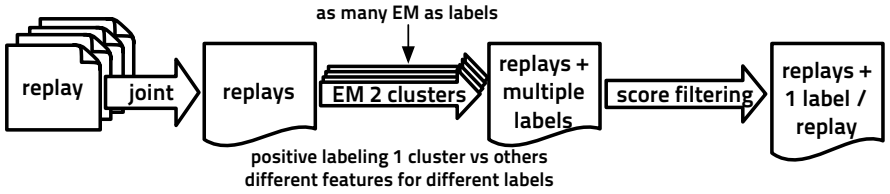


Figure 7.2: Data centric view of our semi-supervised labeling of replays

The selection of the features along with the opening labels is the supervised part of our labeling method. The knowledge of the features and openings comes from expert play and the StarCraft liquipedia[1]. They are all presented in Table 7.1. For instance, if we want to find out which replays correspond to the "fast Dark Templar" (DT, Protoss invisible unit) opening, we put the time at which the first Dark Templar is constructed as a feature and perform clustering on replays with it. This is what is needed for our playing bot: to be able to know when he has to fear "fast DT" opening and build a detector unit quickly to be able to deal with invisibility.

For the clustering part, we tried k-means, expectation-maximization (EM) with equal shape (bivariate normal distribution with proportional covariances matrices) and EM with the normal distribution shapes and volumes chosen with a Bayesian information criterion (BIC). Best BIC models were almost always the most agreeing with expert knowledge (15/17 labels). We used the R package Mclust [??] to perform full EM clustering. We produce "2 bins clustering" for each set of features (corresponding to each opening), and label the replays belonging to the cluster with the lower norm of features' appearances (that is exactly the purpose of our features). Figures 7.5 and 7.6 show the clusters out of EM with the features of the corresponding openings. We thought of clustering because there are two cases in which you build a specific military unit of research a specific upgrade: either it is part of your opening, or it is part of your longer term game plan or even in reaction to the opponent. So the distribution over the time at which a feature appears is bimodal, with one (sharp) mode corresponding to "opening with it" and the other for the rest of the games, as can be seen in Figure 7.3.

Finally, some replays are labeled two or three times with different labels (due to the different time of effect of different openings), so we apply a filtering to transform multiple label replays into unique label ones (see Figure 7.2). For that we choose the openings labels that were happening the earliest (as they are a closer threat to the bot in a game setup) if and only if they were also the most probable or at 10% of probability of the most probable label (to exclude transition boundaries of clusters) for this replay. We find the earliest by comparing the norms of the clusters means in competition. All replays without a label or with multiple labels (*i.e.* which did not had a unique solution in filtering) after the filtering were labeled as *unknown*. We

[1] `http://wiki.teamliquid.net/starcraft/`

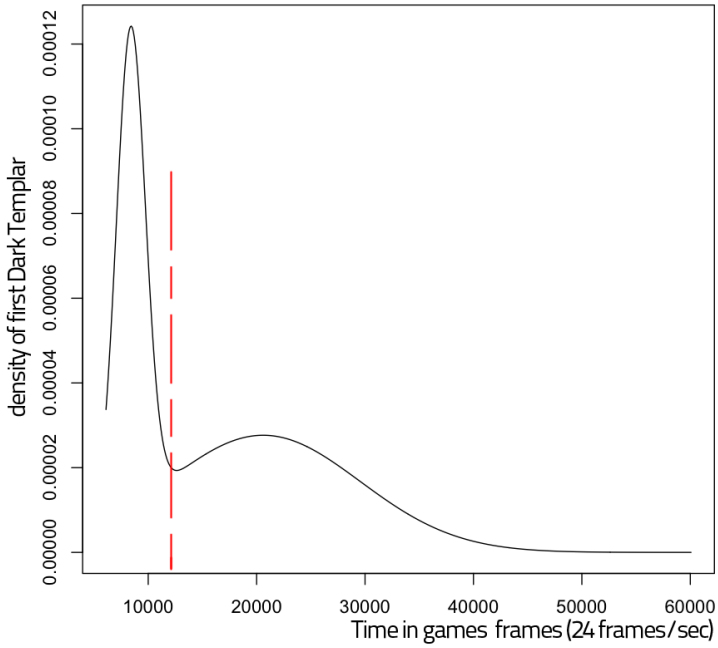


Figure 7.3: Protoss vs Terran distribution of first appearance of Dark Templars (Protoss invisible unit).

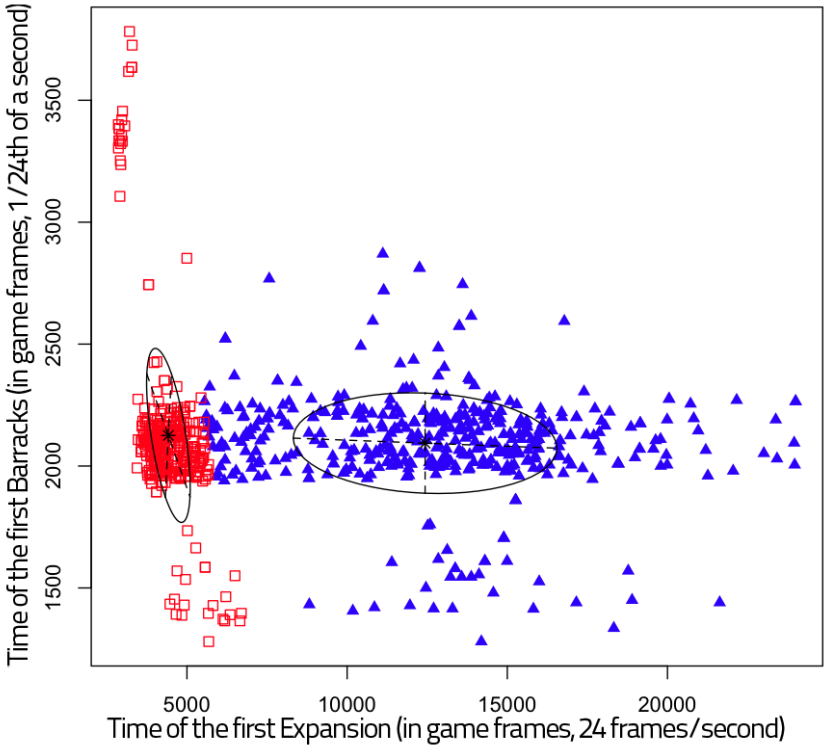


Figure 7.4: Terran vs Zerg Barracks and first Expansion timings (Terran). The bottom left cluster (squares) is the one labeled as *fast_ exp*.

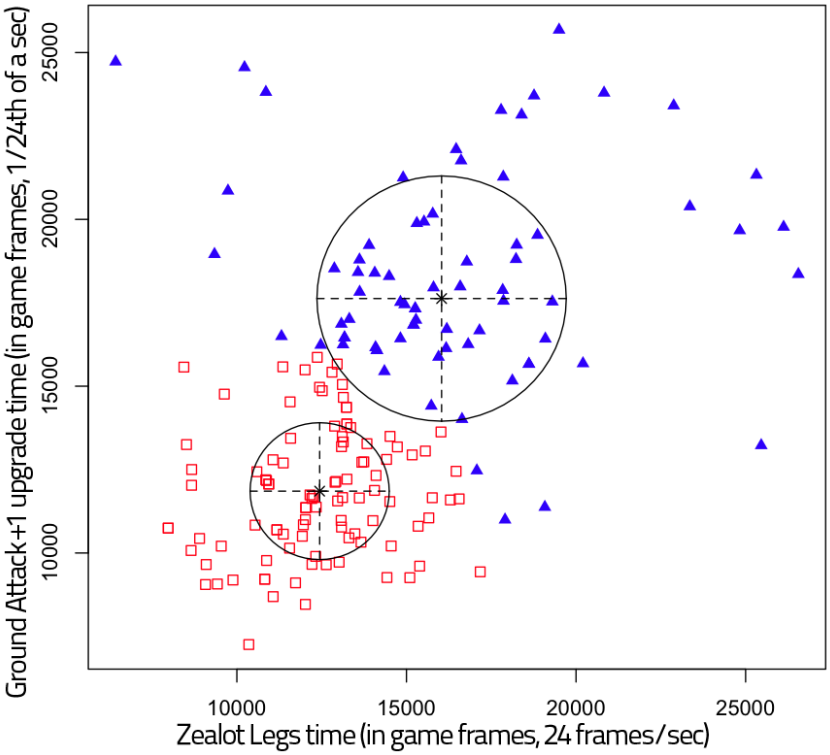


Figure 7.5: Protoss vs Protoss Ground Attack +1 and Zealot Legs upgrades timings. The bottom left cluster (squares) is the one labeled as *speedzeal*.

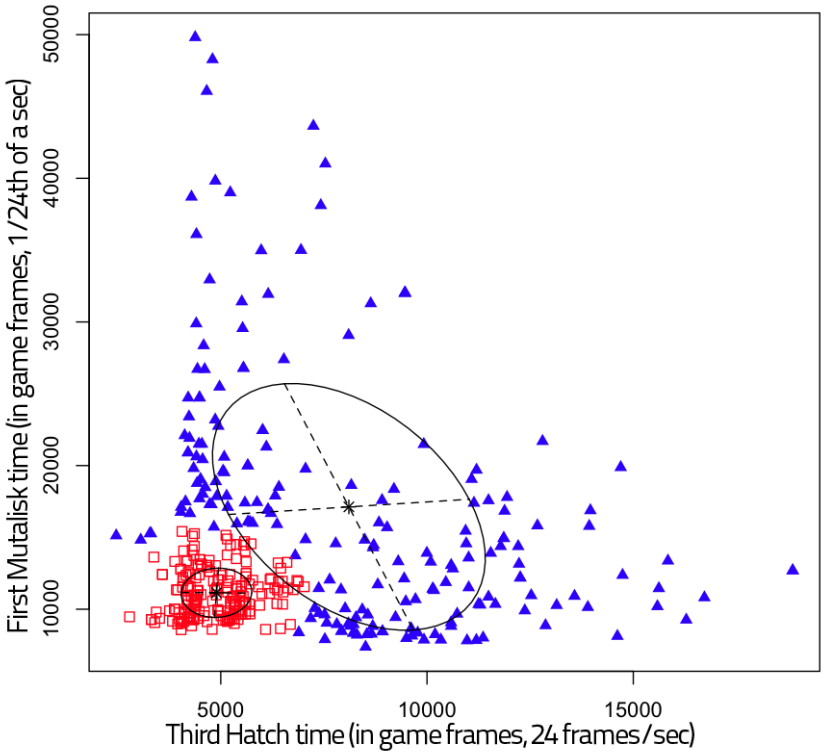


Figure 7.6: Zerg vs Protoss time of the third Hatch and first appearance of Mutalisks. The bottom left cluster (squares) is the one labeled as *mutas*.

Table 7.1: Opening/Strategies labels of the replays (Weber's and ours are not always corresponding)

Race	Weber's labels	Our labels	Features	Note (what we fear)
Protoss	FastLegs	speedzeal	Legs, GroundWeapons+1	quick speed+upgrade attack
	FastDT	fast_dt	DarkTemplar	invisible units
	FastObs	nony	Goon, Range	quick long ranged attack
	ReaverDrop	reaver_drop	Reaver, Shuttle	tactical attack zone damages
	Carrier	corsair	Corsair	flying units
	FastExpand	templar	Storm, Templar	powerful zone attack
		two_gates	2ndGateway, Gateway, Zealot	aggressive rush
	Unknown	unknown	(no clear label)	
Terran	Bio	bio	3rdBarracks, 2ndBarracks, Barracks	aggressive rush
	TwoFactory	two_facto	2ndFactory	strong push (long range)
	VultureHarass	vultures	Mines, Vulture	aggressive harass, invisible
	SiegeExpand	fast_exp	Expansion, Barracks	economical advantage
	Standard			
	FastDropship	drop	DropShip	tactical attack
	Unknown	unknown	(no clear label)	
Zerg	TwoHatchMuta	fast_mutas	Mutalisk, Gas	early air raid
	ThreeHatchMuta	mutas	3rdHatch, Mutalisk	massive air raid
	HydraRush	hydras	Hydra, HydraSpeed, HydraRange	quick ranged attack
	Standard	(speedlings)	(ZerglingSpeed, Zergling)	(removed, quick attacks/mobility)
	HydraMass			
	Lurker	lurkers	Lurker	invisible and zone damages
	Unknown	unknown	(no clear label)	

Table 7.2: Openings distributions for Terran in all the match-ups

	vs Protoss		vs Terran		vs Zerg	
Opening	Nb	Percentage	Nb	Percentage	Nb	Percentage
bio	62	6.2	25	4.4	197	22.6
fast_exp	438	43.5	377	65.4	392	44.9
two_facto	240	23.8	127	22.0	116	13.3
vultures	122	12.1	3	0.6	3	0.3
drop	52	5.2	10	1.7	121	13.9
unknown	93	9.3	34	5.9	43	5.0

then used this labeled dataset as well as Weber and Mateas' labels in the testing of our Bayesian model for opening prediction.

Opening Prediction Model

Our predictive model is a Bayesian program, it can be seen as the "Bayesian network" represented in Figure 7.7. It is a generative model and this is of great help to deal with the parts of the observations' space where we do not have too much data (RTS games tend to diverge from one another as the number of possible actions grow exponentially). Indeed, we can model our uncertainty by putting a large standard deviation on too rare observations and generative models tend to converge with fewer observations than discriminative ones [**?**]. Here is the description of our Bayesian program:

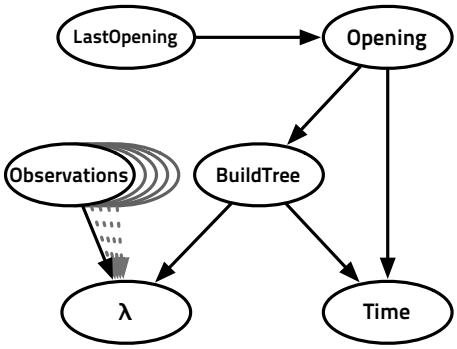


Figure 7.7: Graph representation of the opening (and tech tree) prediction model

Variables

- $BuildTree \in [\emptyset, building_1, building_2, building_1 \wedge building_2, techtrees, \dots]$: all the possible building trees for the given race. For instance $\{pylon, gate\}$ and $\{pylon, gate, core\}$ are two different $BuildTrees$.

- N *Observations*: $O_{i \in [\![1 \dots N]\!]} \in \{0, 1\}$, O_k is 1 ($true$) if we have seen (observed) the kth building (it can have been destroyed, it will stay "seen").

- *Opening*: $Op^t \in [opening_1 \dots opening_M]$ take the various opening values (depending on the race).

- *LastOpening*: $Op^{t-1} \in [opening_1 \dots opening_M]$, Opening value of the previous time step (allows filtering, taking previous inference into account).

- $\lambda \in \{0, 1\}$: coherence variable (restraining $BuildTree$ to possible values with regard to $O_{[\![}1 \dots N]\!]$)

- $Time$: $T \in [\![1 \dots P]\!]$, time in the game (1 second resolution).

At first, we generated all the possible (according to the game rules) $BuildTree$ values (between ≈ 500 and 1600 depending on the race). We observed that a lot of possible $BuildTree$ values are too absurd to be performed in a competitive match and were never seen during the learning. So, we restricted $BuildTree$ to have its value in all the build trees encountered in our replays dataset. There are 810 build trees for Terran, 346 for Protoss and 261 for Zerg (≈ 3000 replays/race), all learned from the (unlabeled) replays.

Decomposition

The joint distribution of our model is the following:

$$\begin{aligned} & \mathrm{P}(T, BuildTree, O_1 \dots O_N, Op^t, Op^{t-1}, \lambda) && (7.1)\\ = \; & \mathrm{P}(Op^t|Op^{t-1}) && (7.2)\\ & \mathrm{P}(Op^{t-1}) && (7.3)\\ & \mathrm{P}(BuildTree|Op^t) && (7.4)\\ & \mathrm{P}(O_{[\![1\dots N]\!]}) && (7.5)\\ & \mathrm{P}(\lambda|BuildTree, O_{[\![1\dots N]\!]}) && (7.6)\\ & \mathrm{P}(T|BuildTree, Op^t) && (7.7) \end{aligned}$$

This can also be see as Figure 7.7.

Forms

- $\mathrm{P}(Op^t|Op^{t-1})$ is optional, we use it as a filter so that the previous inference impacts the current one. We use a functional Dirac:

  $$\begin{aligned} & \mathrm{P}(Op^t|Op^{t-1}) \;\; (Dirac)\\ = \; & 1 \text{ if } Op^t = Op^{t-1}\\ = \; & 0 \text{ else} \end{aligned}$$

  This does not prevent our model to switch predictions, it just uses previous inference posterior $\mathrm{P}(Op^{t-1}$ to average $\mathrm{P}(Op^t)$.

- $\mathrm{P}(Op^{t-1})$ copied from one inference to another (mutated from $\mathrm{P}(Op^t)$). The first $\mathrm{P}(Op^{t-1})$ is bootstrapped with the uniform distribution, we could also use a prior on openings in the given match-up.

- $\mathrm{P}(BuildTree|Op^t)$ is learned from the labeled replays. $\mathrm{P}(BuildTree|Op^t)$ are $\mathrm{card}(\{openings\})$ different histogram over the values of $BuildTree$.

- $\mathrm{P}(O_{[\![1\dots N]\!]})$ is unspecified, we put the uniform distribution.

- $\mathrm{P}(\lambda|BuildTree, O_{[\![1\dots N]\!]})$ is a functional Dirac that restricts $BuildTree$ values to the ones than can co-exist with the observations.

  $$\begin{aligned} & \mathrm{P}(\lambda = 1|buildTree, o_{[\![1\dots N]\!]})\\ = \; & 1 \text{ if } buildTree \text{ can exist with } o_{[\![1\dots N]\!]}\\ = \; & 0 \text{ else} \end{aligned}$$

  A $BuildTree$ value ($buildTree$) is compatible with the observations if it covers them fully. For instance, $BuildTree = \{pylon, gate, core\}$ is compatible with $o_{\#core} = 1$ but it is not compatible with $o_{\#forge} = 1$. In other words, $buildTree$ is incompatible with $o_{[\![1\dots N]\!]}$ *iff* $\{o_{[\![1\dots N]\!]} \backslash \{o_{[\![1\dots N]\!]} \wedge buildTree\}\} \neq \emptyset$.

- $\mathrm{P}(T|BuildTree, Op^t)$ are "bell shape" distributions (discretized normal distributions). There is one bell shape per couple $(opening, buildTree)$. The parameters of these discrete Gaussian distributions are learned from the labeled replays.

Identification (learning)

The learning of the $\mathrm{P}(BuildTree|Op^t)$ histogram is straight forward counting of occurrences from the labeled replays with "add-one smoothing" (Laplace's law of succession [**?**]). The learning of the $\mathrm{P}(T|BuildTree, Op^t)$ bell shapes parameters takes into account the uncertainty of the couples $(buildTree, opening)$ for which we have few observations. Indeed, the normal distribution $\mathrm{P}(T|buildTree, opening)$ begins with a high σ^2, and **not** a Dirac with μ on the seen T value and $sigma = 0$. This accounts for the fact that the first observation may have been an outlier. This learning process is independent on the order of the stream of examples, seeing point A and then B or B and then A in the learning phase produces the same result.

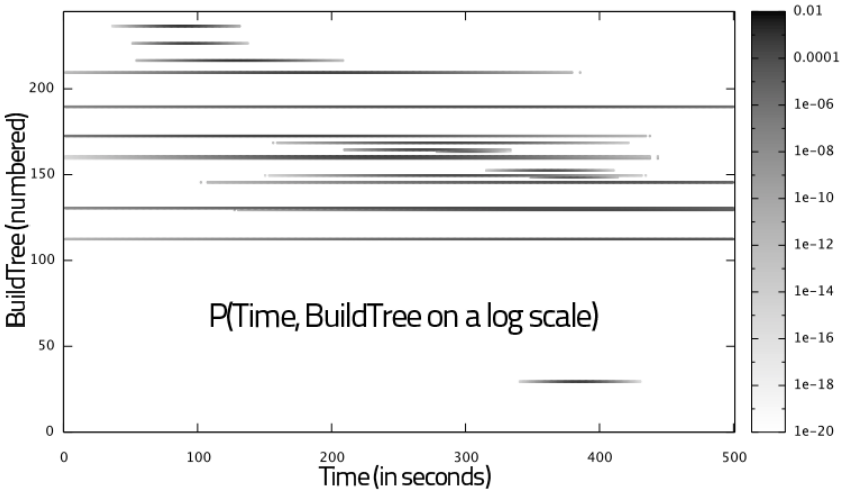


Figure 7.8: $\mathrm{P}(Time, BuildTree|Opening^t = ReaverDrop)$

Questions

The question that we will ask in all the benchmarks is:

$$\mathrm{P}(Op|T = t, O_{[\![1...N]\!]} = o_{[\![1...N]\!]}, \lambda = 1) \tag{7.8}$$
$$\propto \mathrm{P}(Op).\mathrm{P}(o_{[\![1...N]\!]}) \tag{7.9}$$
$$\times \textstyle\sum_{BuildTree} \mathrm{P}(\lambda|BuildTree, o_{[\![1...N]\!]}) \tag{7.10}$$
$$.\mathrm{P}(BuildTree|Op).\mathrm{P}(t|BuildTree, Op) \tag{7.11}$$

Note that if we see $\mathrm{P}(BuildTree, Time)$ as a plan, asking $\mathrm{P}(BuildTree|Opening, Time)$ boils down to use our "plan recognition" mode as a planning algorithm, which could provide good approximations of the optimal goal set [**?**]. This gives us a distribution on the build trees to follow (build orders) to achieve a given opening.

To sum-up, the full Bayesian program is:

$$
\text{Bayesian program}
\begin{cases}
\text{Description}
\begin{cases}
\text{Specification } (\pi)
\begin{cases}
Variables \\
T, BuildTree, O_1 \dots O_N, Op^t, Op^{t-1}, \lambda \\
Decomposition \\
\mathrm{P}(T, BuildTree, O_1 \dots O_N, Op^t, Op^{t-1}, \lambda) \\
= \mathrm{P}(Op^t | Op^{t-1}) \mathrm{P}(Op^{t-1}) \mathrm{P}(BuildTree | Op^t) \\
\mathrm{P}(O_{[\![1 \dots N]\!]}) \mathrm{P}(\lambda | BuildTree, O_{[\![1 \dots N]\!]}) \mathrm{P}(T | BuildTree, Op^t) \\
Forms \\
\mathrm{P}(Op^t | Op^{t-1}) = functional\ Dirac \text{ (filtering)} \\
\mathrm{P}(BuildTree | Op^t) = Categorical(\theta) \\
\mathrm{P}(\lambda | BuildTree, O_{[\![1 \dots N]\!]}) = functional\ Dirac \text{ (coherence)} \\
\mathrm{P}(T | BuildTree = bt, Op^t = op) = Discrete\mathcal{N}(\mu_{bt,op}, \sigma^2_{bt,op})
\end{cases} \\
Identification \\
\mathrm{P}(BuildTree = bt | Op^t = op) = \theta_{bt,op} = \frac{1 + \mathrm{count}(bt,op)}{\#BuildTree + \mathrm{count}(op)} \\
(\mu_{bt,op}, \sigma_{bt,op})_{\mathrm{ML}} = \arg\max_{\mu,\sigma} \mathrm{P}(T | BuildTree = bt, Op^t = op; \mu, \sigma)
\end{cases} \\
Question \\
\mathrm{P}(Op | T = t, O_{[\![1 \dots N]\!]} = o_{[\![1 \dots N]\!]}, \lambda = 1)
\end{cases}
$$

Tech Tree Prediction Model

Results on StarCraft

Prediction

For each match-up, we ran cross-validation testing with 9/10th of the dataset used for learning and the remaining 1/10th of the dataset used for testing. We ran tests finishing at 5, 10 and 15 minutes to capture all kinds of openings (early to late ones). To measure the predictive capability of our model, we used 3 metrics:

- the *final* prediction, which is the opening that is predicted at the end of the test,
- the *online twice* (OT), which counts the openings that have emerged as most probable twice a test (so that their predominance is not due to noise),
- the *online once* > 3 (OO3), which counts the openings that have emerged as most probable openings after 3 minutes (so that these predictions are based on really meaningful information).

After 3 minutes, a Terran player will have or be building his first supply depot, barracks, refinery (gas), and at least factory or expansion. A Zerg player would have his first overlord, zergling pool, extractor (gas) and most of the time his expansion and lair tech. A Protoss player would have his first pylon, gateway, assimilator (gas), cybernectics core, and sometimes his robotics center, or forge and expansion.

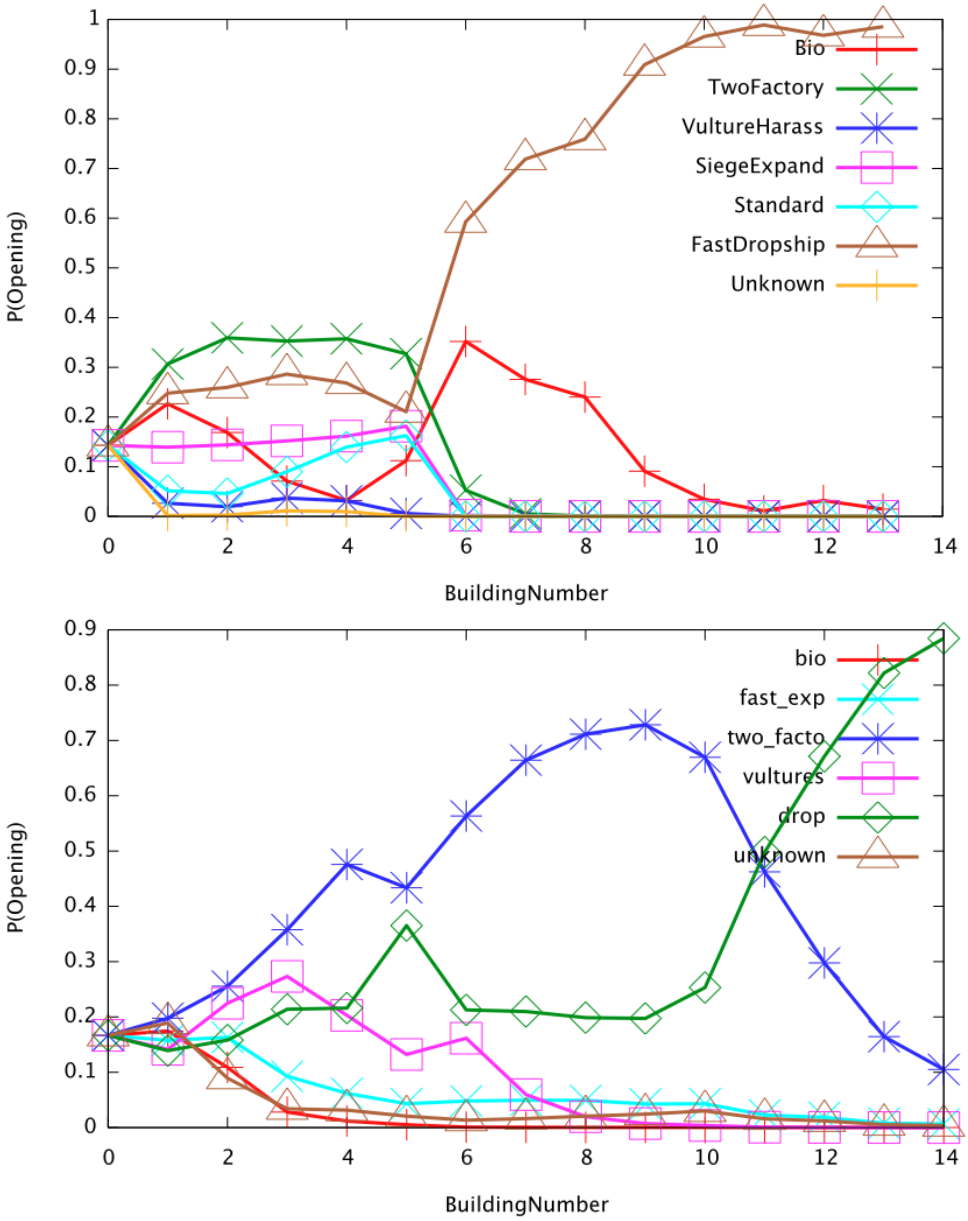


Figure 7.9: Evolution of P(*Opening*) with increasing observations in a TvP match-up, with Weber's labeling on top, our labeling on the bottom. The x-axis corresponds to the construction of buildings.

Table 7.6 sums up all the prediction probabilities (scores) of our model in all the match-ups with both labeling of the game logs. Please note that when an opening is mispredicted, the distribution on openings is often not $\mathrm{P}(badopening) = 1, \mathrm{P}(others) = 0$ and that we can extract some value out of these distributions. Also, we observed that $\mathrm{P}(Opening = unknown) > \mathrm{P}(others)$ is often a case of misprediction: our bot would use the next prediction in this case. Figure 7.9 shows the evolution of the distribution $\mathrm{P}(Opening)$ during a replay for Weber's and our labelings. Figure 7.10 shows the resistance of our model to noise. We randomly removed some observations (buildings, attributes), from 1 to 15, knowing that for Protoss and Terran we use 16 buildings observations and 17 for Zerg. We think that our model copes well with noise because it backtracks unseen observations: for instance if we have only the *core* observation, it will work with build trees containing *core* that will passively infer unseen *pylon* and *gate*. Also, uncertainty is handled natively.

Performances

The first iteration of this model was not making use of the structure imposed by the game in the form of "possible build trees" and was at best very slow, at worst intractable without sampling. With the model presented here, the performances are ready for production as shown in Table 7.4. The memory footprint is around 3.5Mb on a 64bits machine. Learning computation time is linear in the number of games logs events ($O(N)$ with N observations), which are bounded, so it is linear in the number of game logs. It can be serialized and done only once when the dataset changes. The prediction computation corresponds to the sum in the question (III.B.5) and so its computational complexity is in $O(N \cdot M)$ with N build trees and M possible observations, as $M << N$, we can consider it linear in the number of build trees (values of $BuildTree$).

Openings strengths and weaknesses

Discussion

Possible Uses

Developing beforehand a RTS game AI that specifically deals with whatever strategies the players will come up is very hard. And even if game developers were willing to patch their AI afterwards, it would require a really modular design and a lot of work to treat each strategy. With our model, the AI can adapt to the evolutions in play by learning its parameters from the replay, and it can dynamically adapt during the games by using the reverse question $\mathrm{P}(BuildTree|Opening, Time)$, or even $\mathrm{P}(TechTree|Opening, Time)$ if we use a $TechTree$ variable encoding buildings and technology upgrades. This question would give the distribution over technology trees knowing the opening we want to perform at which time. This would allow for the bot to dynamically choose/change build orders.

We could also use this model in a commentary assistant AI. In the StarCraft and StarCraft 2 communities, there are a lot of pro-gamers* tournaments that are commented and we could provide a tool for commentators to estimate the probabilities or different openings or technology paths. As in commented poker matches, where the probabilities of different hands are drawn on screen for the spectators, we could display the probabilities of openings. In such a setup

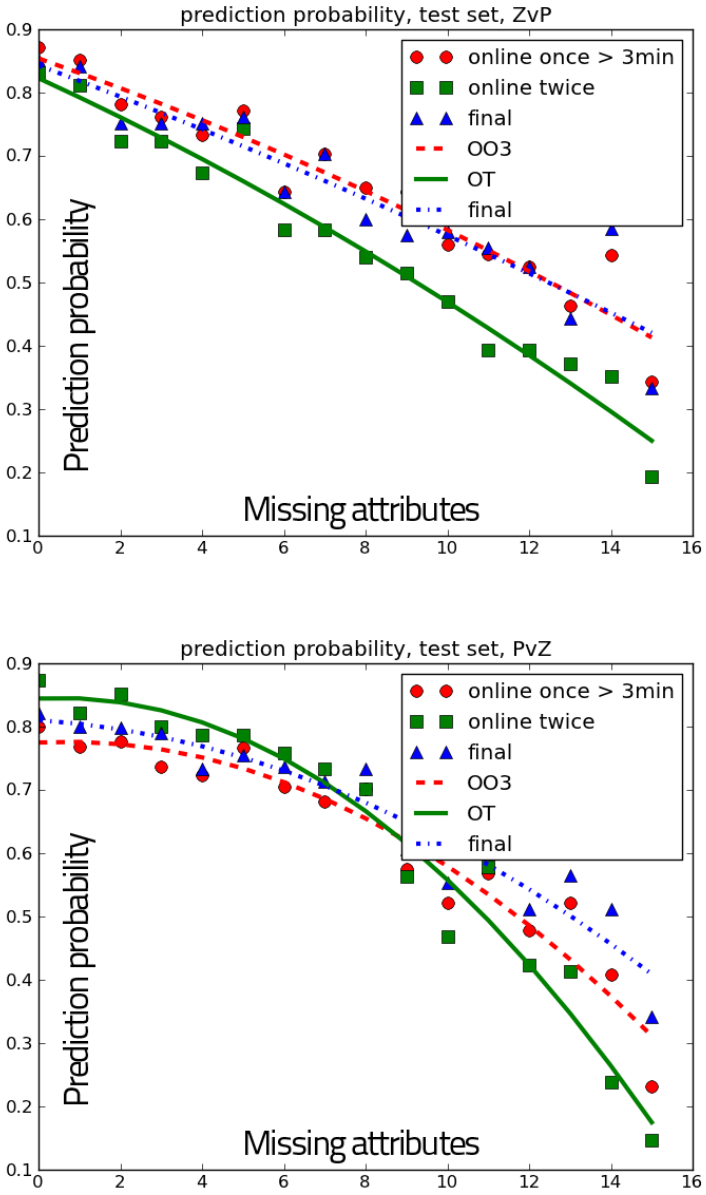


Figure 7.10: Two extreme evolutions of the 3 probabilities of opening recognition with increasing noise (15 missing attributes/observations/buildings correspond to 93.75% missing information for Protoss and Terran openings prediction and 88.23% of missing attributes for Zerg openings prediction). Zerg opening prediction probabilitly on top, Protoss bottom.

we could use more features as the observers and commentators can see everything that happens (upgrades, units) and we limited ourselves to "key" buildings in the work presented in this paper.

Improvements

First, our prediction model can be upgraded to have a higher recognition rate: we could reason about $t+1$ explicitly before computing the distribution over possible openings at t and thus compute the distribution over technology trees at $t+1$. Perhaps it would increase the results of $\mathrm{P}(Opening|Observations)$, but it almost surely would increase $\mathrm{P}(BuildTree^{t+1}|Observations)$ which is important for late game predictions. We could also make use of more features as we currently only use at most 20 features (only buildings), and never all at once. Perhaps also that incorporating priors per match-up would lead to better results.

Then, we could feed it with *more* replays during the learning by scrapping more progamers level replays websites. Also, we could learn from replays of bot vs bot matches. For the learning part, the labeling of replays is very important, and our labeling methods can be improved. We could explore auto-supervised learning [**?**]. Clearly, some match-ups are handled better, either in the replays labeling part and/or in the prediction part. Replays could be labeled by humans and we would do supervised learning then. Or they could be labeled by a combination of rules (as in [**?**]) and statistical analysis (as the method presented here). Finally, the replays could be labeled by match-up dependent openings (instead of race dependent openings currently) and could contain either the two parts of the opening or the game time at which the label is the most relevant, as openings are often bimodal ("fast expand into mutas", "corsairs into reaver", etc.).

Finally, a hard problem is detecting the "fake" builds of very highly skilled players. Indeed, some progamers have build orders which purpose are to fool the opponent into thinking that they are performing opening A while they are doing B. For instance, they could leadthe opponent to think they are going to *tech* and perform an early rush instead. We think that this can be handled by our model by changing $\mathrm{P}(Opening|LastOpening)$ by $\mathrm{P}(Opening|LastOpening, LastObservations)$ and adapting the influence of the last prediction with regard to the last observations (i.e., we think we can learn some "fake" label on replays).

7.4 Adaptation

Army Adaptation Model

We downloaded more than 9000 replays to keep 7708 uncorrupted, 1v1 replays of very high level StarCraft games (pro-gamers leagues and international tournaments) from specialized websites[2]. Then, we ran them using BWAPI[3] and dumped units positions, pathfinding and regions, resources, orders, vision events, and for attacks (we trigger a robust attack tracking heuristic when a unit dies and there are at least two military units around): types, positions, units engaged on each side, and outcomes. Basically, every BWAPI event was recorded, the dataset, its source code, and this model implementation are freely available[4].

[2] `http://www.teamliquid.net`, `http://www.gosugamers.net`, `http://www.iccup.com`

[3] `http://code.google.com/p/bwapi/`

[4] `http://snippyhollow.github.com/bwrepdump/`

In this model, we assume that we have some incomplete knowledge of the opponent's tech tree and a quite complete knowledge of his army composition. We want to know what units we should build now, to adapt our army to their, while staying coherent with our own tech tree and tactics constraints. To that end, we reduce armies to clusters (or mixtures of clusters) that could have generated a given composition. In this lower dimension (usually between 5 to 15 mixture components), we can reason on which mixture of clusters the opponent is probably going to have, according to his current mixture and his tech tree. As we learned how to pair compositions strengths and weaknesses, we can adapt to this "future mixture". This Bayesian program can be seen as the Bayesian network represented in Figure 7.11.

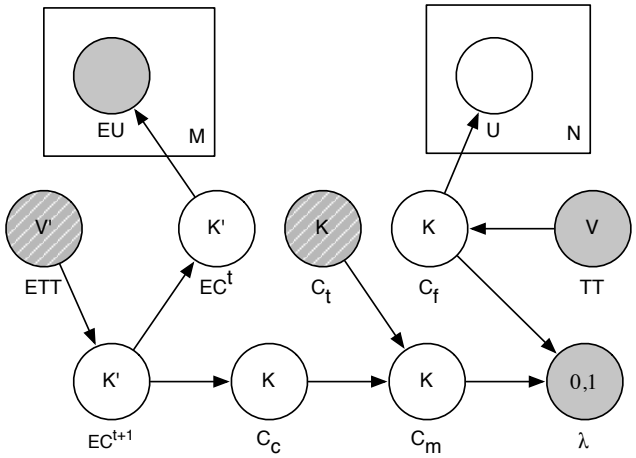


Figure 7.11: Graphical (plate notation) representation of the army composition Bayesian model, gray variables are known while gray hatched variables are given by distributions on their values. C_t can also be known (if a decision was taken on a subsequent model).

Variables

- TT^t, ETT^t are two $TechTree$ variables, one for us (TT) and one for the enemy (ETT), at time t. $TechTree \in \{\emptyset, \{building_1\}, \{building_2\}, \{building_1 \wedge building_2\}, \dots\}$: all the possible building trees for the given race (strictly speaking, we here only use build trees). For instance $\{pylon, gate\}$ and $\{pylon, gate, core\}$ are two different $TechTrees$. TT has V possible values and ETT has V' possible values (depending on the faction).

- $C^{t+1}_{t,c,m,f}, EC^{t,t+1} \in \{clusters\}$ (see further), our army cluster (C), both wanted (C_t for $tactics$,C_c for $counter$, C_m for $merge$) and decided (C_f for $final$) at $t+1$, and the enemy's cluster (EC) estimated at t from their units that we saw, and estimated at $t+1$ from their (estimated distribution on) tech trees and previous EC^t. C_t, C_c, C_m, C_f have K values (K units clusters for us) and EC^t, EC^{t+1} have K' values. C_t is a distribution of the clusters we want for tactics (another model) or just a value (for instance we just need a *dropship/shuttle*). C_c is the counter to what we infer the enemy army will be (EC^{t+1}). C_m merge our tactics desiderata with our adaptivity, while C_f take the tech tree constraints into account. This can seem tedious but these variables are all to be seen as different

states of the two variables C and EC, representing army compositions, respectively for us and for the enemy.

- $U^{t+1}_{1:N}, EU^t_{1:M} \in [0 \dots 1]$, our N unit types (U) proportions (indice on type) at time $t+1$, and the M enemy units types (EU) at time t. For instance, an army with equal numbers of *zealots* and *dragoons* (and nothing else) is represented as $\{U_{zealot} = 0.5, U_{dragoon} = 0.5, \forall ut \neq zealot|dragoon\ U_{ut} = 0.0\}$.

- $\lambda \in \{0, 1\}$ is a coherence variable unifying C^{t+1}_m and C^{t+1}_f to possible values with regard to TT^t.

For tech trees (TT and ETT) values, it would be absurd to generate all the possible combinations (for instance no player is going to build four barracks before building the first supply depot), we used all the values that were encountered in our games (replays) dataset. The bases of these trees were buildings, with some types being counted for repetition, for instance a tech tree of $\{pylon, gateway\}$ is not the same as $\{pylon, gateway, gateway\}$: we effectively count the gateway twice. This way, we counted 273 probable tech tree values for Protoss, 211 for Zerg, and 517 for Terran (the ordering of add-on buildings is multiplying tech trees for Terran). Should it happen, we can deal with unseen tech tree either by using the closest one (in set distance) or using an additional value of no knowledge.

Decomposition

The joint distribution of our model is the following:

$$
\begin{aligned}
& \mathrm{P}(TT^t, C^{t+1}_t, C^{t+1}_c, C^{t+1}_m, C^{t+1}_f, && (7.12)\\
& ETT^t, EC^t, EC^{t+1}, U^{t+1}_{1:N}, EU^t_{1:M}) && (7.13)\\
= & \mathrm{P}(EU^t_{1:M}|EC^t)\mathrm{P}(EC^t|EC^{t+1}) && (7.14)\\
\times & \mathrm{P}(EC^{t+1}|ETT^t)\mathrm{P}(ETT^t) && (7.15)\\
\times & \mathrm{P}(C^{t+1}_c|EC^{t+1})\mathrm{P}(C^{t+1}_t)\mathrm{P}(TT^t)\mathrm{P}(C^{t+1}_f|TT^t) && (7.16)\\
\times & \mathrm{P}(\lambda|C^{t+1}_f, C^{t+1}_m)\mathrm{P}(C^{t+1}_m|C^{t+1}_t, C^{t+1}_c) && (7.17)\\
\times & \mathrm{P}(U^{t+1}_{1:N}|C^{t+1}_f) && (7.18)
\end{aligned}
$$

This can also be see as Figure 7.11.

Forms

- Both $\mathrm{P}(U^{t+1}_{1:N}|C^{t+1}_f)$ and $\mathrm{P}(EU^t_{1:M}|EC^t)$ are results of the clustering of armies compositions ($U_{1:N}$, $EU_{1:M}$) into EC and C, and so depend from the clustering model. They are both learned from the data through expectation maximization. In the best model that we tried, they come from a Gaussian mixtures model (GMM) and so are from $\mathcal{N}(\mu_{z_i}, \sigma^2_{z_i})$ with $z_i \sim Categorial(K)$, respectively $Categorical(K')$ **?**.

- $\mathrm{P}(EC^t|EC^{t+1})$ is a probability table of dimensions $K' \times K'$ resulting of the temporal dynamic between clusters, that is learned from the dataset with a Laplace succession law ("add one" smoothing) **?**.

- $\mathrm{P}(ETT^t)$ is a categorical distribution on V' values, i.e. an histogram distribution on the enemy's tech trees. It comes from the strategy prediction model explained in **?**. For us, $TT \sim Categorical(V)$ too, and we know our own tech tree (TT) exactly.

- $\mathrm{P}(EC^{t+1}|ETT^t)$ is a probability table of dimensions $K' \times V'$ resulting of the inter-dependency between some tech trees and clusters/mixtures. It is learned from the dataset with a Laplace succession law.

- $\mathrm{P}(C_c^{t+1}|EC^{t+1})$ is a probability table of dimensions $K \times K'$, which is learned from battles with a Laplace succession law on victories: $\mathrm{P}(c_c|ec) = \frac{n_{victories}(c_c,ec)+1}{n_{battles}(c_c,ec)+K}$.

- $\mathrm{P}(C_t^{t+1})$ is a $Categorical(K)$ (histogram on the K clusters values) coming from a tactical sub-model. Note that it can be degenerate: $\mathrm{P}(C_t^{t+1} = shuttle) = 1.0$, it serves the purpose of merging tactical needs with strategic ones.

- $\mathrm{P}(C_f^{t+1}|TT)$ is either a table, as for $P(EC^{t+1}|ETT^t)$, or a functional Dirac distribution which tells which C_f values (c_f) are compatible with the current tech tree $TT = tt$. It is compatible if this tech tree (tt) allows for building all units present in c_m. For instance, $tt = \{pylon, gate, core\}$ is compatible with a $c_m = \{\mu_{U_{zealot}} = 0.5, \mu_{U_{dragoon}} = 0.5, \forall ut \neq zealot|dragoon\ \mu_{U_{ut}} = 0.0\}$, but it is not compatible with $c'_m = \{\mu_{U_{arbiter}} = 0.1, \dots\}$.

- $\mathrm{P}(\lambda|C_f^{t+1}, C_m^{t+1})$ is a functional Dirac that restricts C_m values to the ones that can co-exist with our tech tree (C_f).

$$\begin{aligned} & \mathrm{P}(\lambda = 1|C_f, c_m) \\ = \ & 1 \text{ if } \mathrm{P}(C_f = c_m) \neq 0 \\ = \ & 0 \text{ else} \end{aligned}$$

  This can be (and was) simply implemented as a function. This is not strictly necessary (as one could have $\mathrm{P}(C_f|TT, C_m)$ which would do the same for $\mathrm{P}(C_f) = 0.0$) but it allows us to have the same table form for $\mathrm{P}(C_f|TT)$ than for $\mathrm{P}(EC|ETT)$, should we wish to use the learned co-occurrences tables (with respect to the good race).

Identification (learning)

The first approach to clusterize units was a simple fusion $\mathrm{P}(U_{1:N}, C) \propto \prod_i \mathrm{P}(U_i|C)$ with rectangular $\mathrm{P}(U_i|C)$ distributions above thresholds for basic and special (casters) units, and their compositions. So we fixed $\approx 8 + 4 + 4 * 8 = 44$ clusters (for each race) and set their parameters. The problem with this approach is that it requires a high game strategy expertize and understanding of the multiple influences of parameters, while it cannot take into account singular combinations of 3 (or more) unit types. We alternatively learned Gaussian mixture models (GMM) with the expectation-maximization (EM) algorithm on 5 to 15 mixtures with sperical, tied, diagonal and full co-variance matrices **?**. We kept the best scoring models according to the Bayesian information criterion **?**: $-2 \ln \max Likelihood + k \ln(n)$, with k the number of parameters and n the number of observations.

For the categorical probability tables, we used Laplace rule of succession ("add-one smmoothing") **?**: $\mathrm{P}(A = a|B = b) = \frac{n(a,b)+1}{n(a,b)+n(\neg a,b)+|A|}$.

Questions

The question that we ask to know which units to produce is:

$$
\begin{aligned}
& \mathrm{P}(U_{1:N}^{t+1}|eu_{1:M}^{t}, tt^{t}, \lambda = 1) && (7.19)\\
\propto & \sum_{ETT^{t}, EC^{t}, EC^{t+1}} [\mathrm{P}(EC^{t+1}|ETT^{t})\mathrm{P}(ETT^{t}) && (7.20)\\
\times & \ \mathrm{P}(eu_{1:M}^{t}|EC^{t}) && (7.21)\\
\times & \sum_{C_f^{t+1}, C_m^{t+1}, C_c^{t+1}, C_t^{t+1}} [\mathrm{P}(C_c^{t+1}|EC^{t+1}) && (7.22)\\
\times & \ \mathrm{P}(C_t^{t+1})\mathrm{P}(C_f^{t+1}|tt^{t}) && (7.23)\\
\times & \ \mathrm{P}(\lambda|C_f^{t+1}, C_m^{t+1}) && (7.24)\\
\times & \ \mathrm{P}(U_{1:N}^{t+1}|C_f^{t+1})]] && (7.25)
\end{aligned}
$$

or we can sample $U_{1:N}^{t+1}$ from $P(C_f^{t+1})$ or even from c_f^{t+1} (a realization of $C_f^{t=1}$) if we ask $\mathrm{P}(C_f^{t+1}|eu_{1:M}^{t}, tt^{t}, \lambda = 1)$. Note that here we do not know fully neither the value of ETT nor of C_t so we take the most information that we have into account. The evaluation of the question is proportional to $|ETT| \times |EC|^2 \times |C|^4 = V' \times K'^2 \times K^4$. But we do not have to sum on the 4 types of C in practice: $\mathrm{P}(C_m|C_t, C_c)$ is a simple linear combination of vectors and $\mathrm{P}(\lambda = 1|C_f, C_m)$ is a linear filter function, so we just have to sum on C_c and practical complexity is proportional to $V' \times K'^2 \times K$. As we have most often ≈ 10 Gaussian components in our GMM, K or K' are in the order of 10 (5 to 12 in practice), while V and V' are between 211 and 517 as noted above.

To benchmark our clustering, we use a slightly different model in which $\mathrm{P}(EC^{t+1}) = \mathrm{P}(EC^{t})$ (we want to know which army counters the other in a given battle). The question that we will ask is $\mathrm{P}(C_c^{t+1}|eu_{1:M}^{t})$.

All the results presented in this section represent the nine match-ups (races combinations) in 1 versus 1 (duel) of StarCraft. We worked with a data-set of 7708 replays of highly skilled human players. For each match-up, we set aside 100 test matches, and use the remaining of the dataset for learning. Performance wise, for the biggest dataset (PvT) the learning part takes around 100 second on a 2.8 Ghz Core 2 Duo CPU (and it is easily serializable) for 2408 games (57 seconds to fit and select the GMM, and 42 seconds to fill the probability tables). We preferred robustness to precision and thus we did not remove outliers: better scores can easily be achieved by considering only stereotypical armies/battles, or more amateur players (which styles are less pronounced).

To benchmark our clustering methods (home-made, K-means, GMM), we reduced battle observations to clusters for both parties and used the learned $P(C_c|EC)$ to estimate the outcome of the battle. For that, we used battles with *disparities* (the maximum strength ratio of one army over the other) of 1.1 to 1.5. There is an average of 5 battles per game at a 1.3 disparity threshold (more above). A summary of the main metrics is shown in Table 7.6: we can see that predicting battle outcomes (even with a high disparity) with "just probabilities" of $P(C_c|EC)$ (without taking the forces into account) gives relevant results. Note that this is a very high level (abstract) view of a battle, we do not consider tactical positions, nor players' attention, actions, etc. We can somehow view the "just prob." results as the military efficiency improvement we can (at least) expect from having the right army composition. Also, without explicitly labeling clusters,

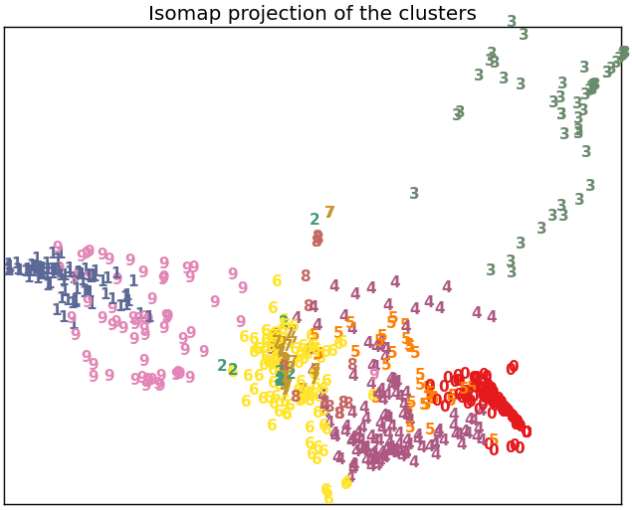


Figure 7.12: 2 dimensional Isomap projection of a small dataset of battles for Zerg (vs Protoss) with most probable Gaussian mixture model components as labels.

one can apply thresholding to special units (observers, arbiters, science vessels...) to generate more specific clusters: we did not include these results here but they sometimes drastically increase prediction scores.

We also plotted the generated clusters in all kind of way, Figure 7.4 shows a 2D Isomap projection of the battles on a small dataset. Figure 7.13 simply lays out the concept of *army composition* and expert gamers recognized the clusters clearly identified by the clustering (GMM). Figure 7.14 showcases the dynamics of clusters components: $P(EC^t|EC^{t+1}$, for Zerg (vs Protoss) for Δt of 2 minutes. As shown in Figure **??**, this model is part of the strategy module of our next AIIDE competition bot.

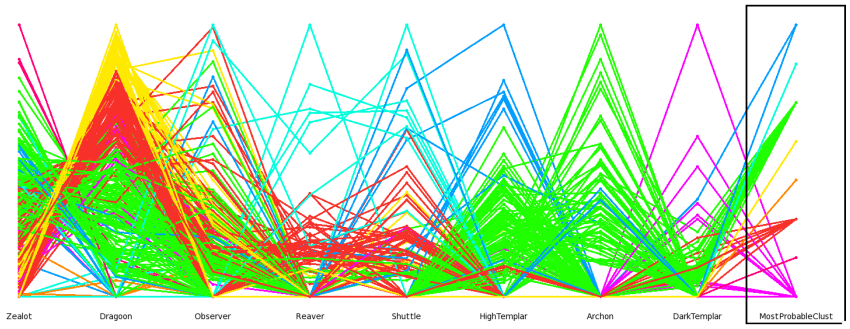


Figure 7.13: Parallel plot of a small dataset of Protoss (vs Protoss, i.e. in the PvP match-up) army clusters on most important unit types (for the match-up). Each normalized vertical axis represents the percentage of the units of the given unit type in the army composition (we didn't remove outliers, so most top (tip) vertices represent 100%), except for the rightmost (framed) which links to the most probable GMM component. Note that several traces can (and do) go through the same edge.

Here, we focused on asking $\mathrm{P}(U_{1:N}^{t+1}|eu1 : M^t, tt^t, \lambda = 1)$, and evaluated (in the absence of ground truth for full armies compositions) the two key components that are $\mathrm{P}(U_{1:N}|C)$ (or

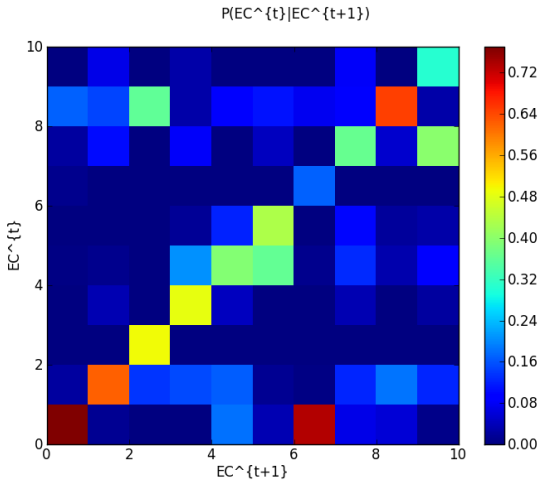


Figure 7.14: Dynamics of clusters: P($EC^t|EC^{t+1}$) for Zerg

P($EC_{1:M}|EC$)) and P($C_c|EC$). Many other questions can be asked: P($TT^t|eu^t_{1:M}$) can help us adapt our tech tree development to the opponent's army. If we know the opponent's army composition only partially, we can benefit of knowledge about ETT to know what is possible, but also probable, by asking $P(EC^t|Observations)$.

We presented a probabilistic model inferring the best army composition given what was previously seen (from replays, or previous games), integrating adaptation to the opponent with other constraints (tactics). One of the main advantages of this approach is to be able to deal natively with incomplete information, due to player's intentions, and to the fog of war in RTS. The army composition dimensionality reduction (clustering: unsupervised) can be applied to any game and coupled with other techniques, for instance for situation assessment in case-based planning. The results in battle outcome prediction (from few information) shows its situation assessment potential. Finally, we will use this model in our StarCraft AI competition entry bot as it facilitates adaptive decision-making on the production plan. The main research direction is now to find a way to evaluate the full model integrated to a bot, both for performance of the bot and for the fun of a human player.

Results on StarCraft

> *However beautiful the strategy, you should occasionally look at the results.* Winston Churchill

Discussion

Table 7.3: Prediction probabilities for all the match-ups

	Weber and Mateas' labels									Our labels								
	5 minutes			10 minutes			15 minutes			5 minutes			10 minutes			15 minutes		
match-up	final	OT	OO3	final	OT	OO3	final	OT	OO3	final	OT	OO3	final	OT	OO3	final	OT	OO3
PvP	0.65	0.53	0.59	0.69	0.69	0.71	0.65	0.67	0.73	0.78	0.74	0.68	0.83	0.83	0.83	0.85	0.83	0.83
PvT	0.75	0.64	0.71	0.78	0.86	0.83	0.81	0.88	0.84	0.62	0.69	0.69	0.62	0.73	0.72	0.6	0.79	0.76
PvZ	0.73	0.71	0.66	0.8	0.86	0.8	0.82	0.87	0.8	0.61	0.6	0.62	0.66	0.66	0.69	0.61	0.62	0.62
TvP	0.69	0.63	0.76	0.6	0.75	0.77	0.55	0.73	0.75	0.50	0.47	0.54	0.5	0.6	0.69	0.42	0.62	0.65
TvT	0.57	0.55	0.65	0.5	0.55	0.62	0.4	0.52	0.58	0.72	0.75	0.77	0.68	0.89	0.84	0.7	0.88	0.8
TvZ	0.84	0.82	0.81	0.88	0.91	0.93	0.89	0.91	0.93	0.71	0.78	0.77	0.72	0.88	0.86	0.68	0.82	0.81
ZvP	0.63	0.59	0.64	0.87	0.82	0.89	0.85	0.83	0.87	0.39	0.56	0.52	0.35	0.6	0.57	0.41	0.61	0.62
ZvT	0.59	0.51	0.59	0.68	0.69	0.72	0.57	0.68	0.7	0.54	0.63	0.61	0.52	0.67	0.62	0.55	0.73	0.66
ZvZ	0.69	0.64	0.67	0.73	0.74	0.77	0.7	0.73	0.73	0.83	0.85	0.85	0.81	0.89	0.94	0.81	0.88	0.94
overall	0.68	0.62	0.68	0.73	0.76	0.78	0.69	0.76	0.77	0.63	0.67	0.67	0.63	0.75	0.75	0.63	0.75	0.74

Table 7.4: Extremes of computation time values (in seconds, Core 2 Duo 2.8Ghz)

Race	Nb Games	Learning time	Inference μ	Inference σ^2
T (max)	1036	0.197844	0.0360234	0.00892601
T (Terran)	567	0.110019	0.030129	0.00738386
P (Protoss)	1021	0.13513	0.0164457	0.00370478
P (Protoss)	542	0.056275	0.00940027	0.00188217
Z (Zerg)	1028	0.143851	0.0150968	0.00334057
Z (Zerg)	896	0.089014	0.00796715	0.00123551

Zerg | Protoss	two gates	fast dt	reaver drop	corsair	nony
speedlings	0.417	0.75	NED	NED	0.5
lurkers	NED	0.493	NED	0.445	0.533
fast mutas	NED	0.506	0.5	0.526	0.532

Terran | Protoss	fast dt	reaver drop	corsair	nony
two facto	0.552	0.477	NED	0.578
rax fe	0.579	0.478	0.364	0.584

Table 7.5: Opening/Strategies labels counted for victories against each others for the PvZ (top, on 1408 games) and PvT (bottom, on 1657 games) match-ups. NED stands for Not Enough Data to conclude a preference/discrepancy towards one opening. The results should be read as win rates of columns openings vs lines openings.

Table 7.6: Summary of the clustering evaluation (for GMM), main metrics

forces	scores	PvP		PvT		PvZ		TvT		TvZ		ZvZ	
disparity	in %	m	ws	m	ws	m	ws	m	ws	m	ws	m	ws
1.1	heuristic	**63**	63	58	58	58	58	**65**	**65**	70	70	56	56
	just prob.	54	58	68	*72*	60	61	55	56	69	69	62	63
	prob×heuristic	61	**63**	**69**	**72**	**59**	**61**	62	64	**70**	**73**	**66**	**69**
1.3	heuristic	**73**	73	66	66	**69**	**69**	**75**	72	72	**72**	70	70
	just prob.	56	57	65	*66*	54	55	56	57	62	61	63	61
	prob×heuristic	72	**73**	**70**	**70**	66	66	71	**72**	**72**	70	**75**	**75**
1.5	heuristic	75	75	73	73	**75**	**75**	78	**80**	**76**	76	75	75
	just prob.	52	55	61	61	54	54	55	56	61	*63*	56	60
	prob×heuristic	**75**	**76**	**74**	**75**	72	72	**78**	78	73	**76**	**77**	**80**

Table 7.7: Winner prediction scores (in %) for 3 main metrics. For the left columns ("m"), we considered only military units. For the right columns ("ws") we also considered static defense and workers. The "heuristic" metric is a baseline heuristic for battle winner prediction for comparison using army values, while "just prob." only considers $P(C_c|EC)$ to predict the winner, and "prob×heuristic" balances the heuristic's predictions with $\sum_{C_c,EC} P(C_c|EC)P(EC)$.

Chapter 8

BroodwarBotQ: putting it all together

Dealing with failure is easy: Work hard to improve. Success is also easy to handled: You've solved the wrong problem. Work hard to improve.

Alan J. Perlis (1982)

In this chapter, we present some of the engineering that went in the robotic player's (bot) implementation. We will also present the different flows of informations and how decisions are made during a game. Finally we will present the results of the full robotic player to various bots competitions.

8.1 Code Architecture

mapping schéma code <-> schéma info-flow.

8.2 A Game Walkthrough

The tree of decisions.

8.3 Results

AIIDE 2010,2011, Ladder.

#	Bot	Race	ELO rating	Total games	Wins	Losses	Draws	Crashes	Wins w/o crashed games	Losses w/o crashed games	BWAPI rev	Description	Status
1	Skynet_v2_0_1		2156	1493	1300 (87.07%)	189	4	0 (0%)	1175 (85.89%)	189	4025	-	enabled
2	UAlbertaBot_bwapi_4025		2130	153	135 (88.24%)	17	1	1 (0.65%)	129 (88.36%)	16	4025	-	enabled
3	UAlbertaBot_aiide_2011		2105	1890	1434 (75.87%)	404	52	109 (5.77%)	1402 (80.16%)	295	3769	-	disabledNewerVersion
4	Skynet_aiide_2011		2005	844	690 (81.75%)	132	22	20 (2.37%)	603 (81.82%)	112	3769	-	disabledNewerVersion
5	Aiur_bwapi_4000		1944	1389	939 (67.6%)	409	41	9 (0.65%)	886 (66.77%)	400	4025	-	enabled
6	Aiur_aiide_2011		1931	1912	1337 (69.93%)	512	63	11 (0.58%)	1242 (68.77%)	501	3769	-	enabled
7	ItayUndermind_aiide_2011		1895	1742	1056 (60.62%)	671	15	0 (0%)	968 (58.52%)	671	3769	-	enabled
8	krasi0_2_15		1881	873	588 (67.35%)	272	13	0 (0%)	502 (63.79%)	272	4025	-	enabled
9	Machine_0_11		1863	990	646 (65.25%)	306	38	36 (3.64%)	605 (66.27%)	270	4025	-	enabled
10	krasi0_2_14		1841	831	589 (70.88%)	214	28	1 (0.12%)	483 (66.71%)	213	4025	-	disabledNewerVersion
11	Overmind_aiide_2010		1796	1295	872 (67.34%)	386	37	1 (0.08%)	839 (66.53%)	385	2423	-	enabled
12	BroodwarBotQ_bwapi_4000		1777	1569	790 (50.35%)	639	140	31 (1.98%)	726 (49.25%)	608	4025	-	enabled
13	Machine_0_1		1767	90	58 (64.44%)	31	1	6 (6.67%)	55 (67.9%)	25	4025	-	disabledNewerVersion
14	EISBot_aiide_2011		1745	1774	960 (54.11%)	750	64	0 (0%)	907 (52.7%)	750	3769	-	enabled
15	Undermind_aiide_2011		1705	1605	826 (51.46%)	633	146	0 (0%)	685 (46.79%)	633	3769	-	enabled
16	SPAR_aiide_2011		1602	1780	731 (41.07%)	1003	46	397 (22.3%)	687 (51.31%)	606	3769	-	enabled
17	Nova_aiide_2011		1599	1840	705 (38.32%)	1029	106	101 (5.49%)	658 (38.89%)	928	3769	-	enabled
18	BroodwarBotQ_aiide_2011		1528	1475	484 (32.81%)	811	180	11 (0.75%)	445 (31.23%)	800	3769	-	disabledNewerVersion
19	BTHAI_Zerg_2_7		1424	117	25 (21.37%)	90	2	2 (1.71%)	25 (21.74%)	88	4025	-	disabledTemporarily
20	Dreadbot_bwapi_4000		1338	1359	391 (28.77%)	921	47	230 (16.92%)	316 (29.98%)	691	4025	-	enabled
21	Cromulent_aiide_2011		1331	1887	480 (25.44%)	1311	96	14 (0.74%)	441 (24.05%)	1297	3769	-	enabled
22	Nevermind_2011_sscai		1278	108	21 (19.44%)	85	2	0 (0%)	16 (15.53%)	85	4025	-	disabledTemporarily
23	BTHAI_aiide_2011		1235	1092	251 (22.99%)	805	36	178 (16.3%)	197 (22.91%)	627	3769	-	disabledNewerVersion
24	BTHAI_Protoss_aiide_2011		1223	104	13 (12.5%)	88	3	19 (18.27%)	5 (6.49%)	69	3769	-	disabledNewerVersion
25	bigbrother_aiide_2011		1173	1975	416 (21.06%)	1467	92	156 (7.9%)	299 (17.57%)	1311	3769	-	enabled
26	BTHAI_Terran_aiide_2011		1141	93	9 (9.68%)	82	2	17 (18.28%)	6 (8.22%)	65	3769	-	disabledNewerVersion
27	BTHAI_Terran_2_7		1117	802	158 (19.7%)	631	13	114 (14.21%)	120 (18.46%)	517	4025	-	enabled
28	BTHAI_Protoss_2_7		1059	835	143 (17.13%)	679	13	9 (1.08%)	90 (11.64%)	670	4025	-	enabled
29	Quorum_aiide_2011		886	1851	139 (7.51%)	1619	93	116 (6.27%)	85 (5.06%)	1503	3769	-	enabled

Figure 8.1: Bots ladder on February 12th, 2012. With BroodwarBotQ using a Bayesian model for opponent's strategy prediction as well as for micro-management*.

Chapter 9

Perspectives

Man's last mind paused before fusion, looking over a space that included nothing but the dregs of one last dark star and nothing besides but incredibly thin matter, agitated randomly by the tag ends of heat wearing out, asymptotically, to the absolute zero.
Man said, "AC, is this the end? Can this chaos not be reversed into the Universe once more? Can that not be done?"
AC said, "THERE IS AS YET INSUFFICIENT DATA FOR A MEANINGFUL ANSWER."

Isaac Asimov (The Last Question, 1956)

INTRO

9.1 Units Control (Micro-management)

Reinforcement learning, [**?**]
Evolving control policies, [**?**]

9.2 Strategy and Tactics

Bandits (contextual bands)
Planning [**?**]
Probabilistic planning?

9.3 Inter-game Adaptation (Meta-game)

Other cases of learning not dealt with in previous chapters: [**?**]

Laplace smoothing $P(ETT = ett|Player = p) = \frac{1+nbgames(ett,p)}{\#ETT+nbgames(p)}$. Same for $EClusters$ and $ETactics$.

The full (reinforcement) learning of the bot can be seen as learning degrees of liberty one by one with a hierarchy from strategies to actions by tactics. C.f R-IAC Baranes & Oudeyer 2009 and 2012; PI^2-CMA (ES).

Chapter 10

Conclusion

10.1 Contrib

Résumer les contributions

Approaches

- A tractable decomposition of game AI in a hierarchy of predictions, decisions and actions.
- Integration of learning in a decision-making model.
- An autonomous agent for StarCraft (BroodwarBotQ).

Results

Recall what works, what should be extended.

10.2 Perspectives: Not a solved problem yet

Computers don't beat good (experts) humans (higher level strategic thinking: common sense, plus vision/interpolation for efficient micro). They are not so fun (do not adapt that much, our bot is the most adaptive ATM). Competition results. Tout ce qui peut se faire en recherche et ce qui est directement applicable par l'industrie.

Appendix A

Game AI

A.1 Algorithms

Algorithm 5 Negamax algorithm

```
function NEGAMAX(depth)
    if depth ≤ 0 then
        return value()
    end if
    α = −∞
    for all possible moves do
        α = max(α, −negamax(depth − 1))
    end for
    return α
end function
```

A.2 "Gamers' survey" in section 2.9 page 39

Questions

How good are you?

- Very good
- Good

How important is the virtuosity? (to win the game) reflexes, accuracy, speed, "mechanics"

- 0 (not at all, or irrelevant)
- 1 (counts, can be game changing for people on equal level at other answers)
- 2 (counts a lot, can make a player win even if he is a little worse on lower importance gameplay features)

How important is deductive thinking? (to win the game) "If I do A he can do B but not C" or "I see E so he has done D", also called analysis, forward inference."

- 0 (not at all, or irrelevant)
- 1 (counts, can be game changing for people on equal level at other answers)
- 2 (counts a lot, can make a player win even if he is a little worse on lower importance gameplay features)

How important is inductive thinking? (to win the game) "He does A so he may be thinking B" or "I see D and E so (in general) he should be going for F (learned)", also called generalization, "abstraction".

- 0 (not at all, or irrelevant)
- 1 (counts, can be game changing for people on equal level at other answers)
- 2 (counts a lot, can make a player win even if he is a little worse on lower importance gameplay features)

How hard is decision-making? (to win the game) "I have options A, B and C, with regard to everything I know about this game, I will play B (to win)", selection of a course of actions.

- 0 (not at all, or irrelevant)
- 1 (counts, can be game changing for people on equal level at other answers)
- 2 (counts a lot, can make a player win even if he is a little worse on lower importance gameplay features)

You can predict the next move of your opponent: (is knowledge of the game or of the opponent more important)

- 1 by knowing what the best moves are / the best play for him?
- -1 by knowing him personally (psychology)?
- 0 both equal

What is more important:

- 1 knowledge of the game (general strategies, tactics, timings)
- -1 knowledge of the map (specific strategies, tactics, timings)
- 0 both equal

Results

Game	Virtuosity (sensory-motor) [0-2]			Deduction (analysis) [0-2]			Induction (abstraction) [0-2]			Decision-Making (acting) [0-2]			Opponent -1: subjectivity 1: objectivity			Knowledge -1: map 1: game		
	top	rest	n	top	rest	n	top	rest	n	top	rest	n	top	rest	n	top	rest	n
Chess	X	X	X	1.714	1.815	34	1.714	1.429	35	1.714	1.643	35	0.714	0.286	35	X	X	X
Go	X	X	X	2.000	1.667	16	2.000	1.600	16	2.000	1.600	16	1.000	0.533	16	X	X	X
Poker	X	X	X	1.667	0.938	22	1.667	1.562	22	1.333	1.625	22	-0.167	-0.375	22	X	X	X
Racing	2.000	1.750	19	0.286	0.250	19	0.571	0.000	19	1.286	0.833	19	0.571	0.455	18	-0.286	-0.333	19
TeamFPS	1.917	1.607	40	1.083	1.000	39	1.417	0.929	40	1.417	1.185	39	0.000	0.214	40	-0.083	-0.115	38
FFPS	2.000	2.000	22	1.250	1.000	21	1.125	1.077	21	1.125	1.231	21	0.250	-0.154	21	0.250	0.083	20
MMORPG	1.118	1.000	29	1.235	0.833	29	1.176	1.000	29	1.235	1.250	29	0.471	0.250	29	0.706	0.833	29
RTS	1.941	1.692	86	1.912	1.808	86	1.706	1.673	83	1.882	1.769	86	0.118	0.288	86	0.412	0.481	86

Table A.1: Results of the survey (means) for "top" players and the "rest" of answers, along with the total number of answers (n).

Appendix B

StarCraft AI

B.1 Datasets

B.2 Algorithms

Algorithm 6 (Simplified) Target selection heuristic, efficiently implemented with a enemy units $\leftrightarrow$ damages bidirectional map.

```
function ON_DEATH(unit)
    remove_incoming_damages(unit.target, damages(unit, unit.target))
end function
function REGISTER_TARGET(unit, target)
    add_incoming_damages(target, damages(unit, target))
    unit.target ← target
end function
function SELECT_TARGET(unit)
    for all eunit ∈ focus_fire_order(enemy_units) do
        if eunit.type ∈ priority_targets(unit.type) then
            if in_range(unit, eunit) then
                register_target(unit, eunit)
            else if unit.prio_target == NULL then
                unit.prio_target ← eunit
            end if
        end if
    end for
    if unit.target == NULL then
        for all eunit ∈ focus_fire_order(enemy_units) do
            if in_range(unit, eunit) then
                register_target(unit, eunit)
            end if
        end for
    end if
    if unit.target == NULL and unit.prio_target == NULL then unit.prio_target ←
closer(unit, enemy_units)
    end if
end function
```

Algorithm 7 Flow algorithm making sure there are no convex closures

function INIT($region$)
  $\{dir_{i,j} \leftarrow list() | (i,j) \in region\}$
  $updated \leftarrow \{(i,j) | (i,j) \in entrance(region)\}$
  $\{dir_{i,j} \leftarrow dir\_towards(region) | (i,j) \in updated\}$
  while $\exists dir_{i,j} == list()$ **do**
    $(sources, new\_updated) \leftarrow neighbours\_couples(updated)$
    for all $((x,y),(i,j)) \in (sources, new\_updated)$ **do**
      if $(x-i, y-j) \notin dir_{x,y}$ **then**
        $dir_{i,j}.append((i-x, j-y))$
      end if
    end for
    $updated \leftarrow new\_updated$
  end while
end function
function BUILD(i,j)
  for all $(x,y) \in neighbours(i,j)$ **do**
    $dir_{x,y}.remove((x-i, y-j))$
    if $dir_{x,y}.empty()$ **then**
      $build(x,y)$
    end if
  end for
end function

Algorithm 8 Production/Construction Planning

function INIT
end function

B.3 Figures

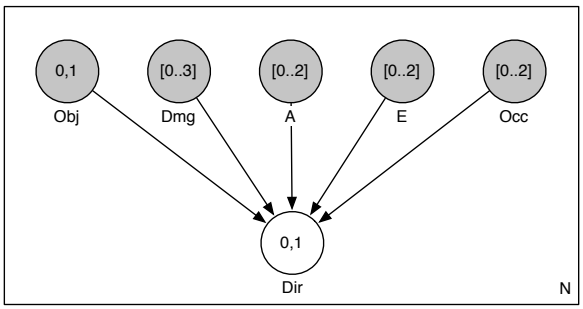


Figure B.1: Plate diagram of a Bayesian unit with N possible directions.

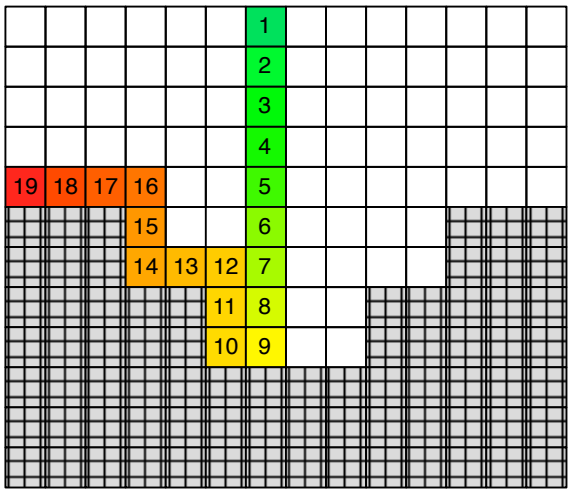


Figure B.2: Example of trailing repulsive charges (repulsive "pheromones") at already visited positions for Bayesian units to avoid being blocked by local optimums. The trajectory is indicated by the increasing numbers (most recent unit position in 19) and the (decaying) strength of the trailing repulsion is weaker in green and stronger in red. Related to 5.5.

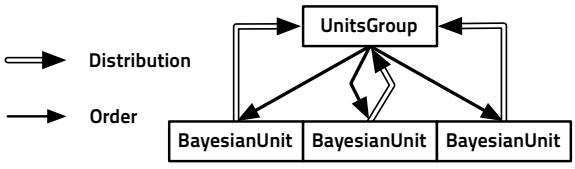


Figure B.3: Example of the decision taken at the units group level from "compressed" information in the form of the distribution on Dir for each Bayesian unit. This can be viewed as a simple optimization problem (maximize sum of probabilities of decisions taken) or as a constraint satisfaction problem (CSP) like "no unit should be left behind/die/dissatisfied". Related to 5.5

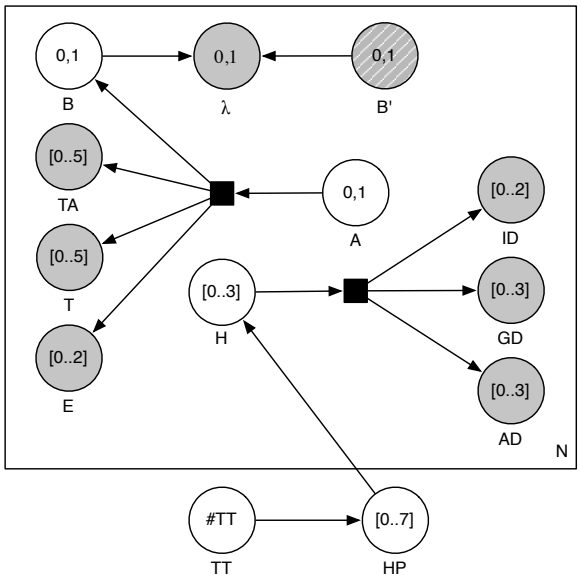


Figure B.4: Plate diagram of the Bayesian tactical model in prediction (we know ID, AD, GD)