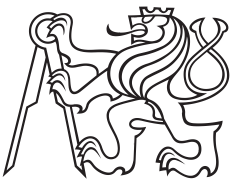
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Learning to play real-time strategy games from demonstration using decentralized MAS

Jan Malý
Study Programme: Open informatics
Field of Study: Artificial Intelligence

May 2017
Supervisor: RNDr. Michal Čertický, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science

DIPLOMA THESIS AGREEMENT

Student: Malý Jan

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: Learning to play real-time strategy games from demonstration using decentralized MAS

Guidelines:

1. Review current approaches for building expert level real time strategy game-playing AI.
2. Identify decision making processes realized by human player in expert level gameplay of real-time strategy game StarCraft: Brood War.
3. Implement game-playing agent for StarCraft Brood War. To reduce complexity and amount of required expert knowledge, design the game-playing AI as a highly decentralized multiagent system with the ability to learn from demonstration (watching replays of StarCraft: Brood War games played by humans).
4. Use Markov decision processes and inverse reinforcement learning to train decision modules for individual MAS agents.
5. Evaluate and discuss the capabilities of game-playing AI and compare it to state of the art AIs.

Bibliography/Sources:

[1] BEN G. WEBER. Integrating learning in a multi-scale agent [online]. 2012. Santa Cruz: University of California, 2012. ISBN 14-776-1473-7.
[2] D. Churchill, M. Preuss, F. Richoux, G. Synnaeve, A. Uriarte, S. Ontanón, M. Čertický. StarCraft Bots and Competitions. Chapter in Encyclopedia of Computer Graphics and Games (ECGG). Springer International Publishing. ISBN: 978-3-319-08234-9. 2016.
[3] SHOHAM, Yoav. a Kevin LEYTON-BROWN. Multiagent systems: algorithmic, game-theoretic, and logical foundations. 2008. New York: Cambridge University Press, 2009. ISBN 05-218-9943-5.
[4] ROBERTSON, Glen; WATSON, Ian D. An Improved Dataset and Extraction Process for Starcraft AI. In: FLAIRS Conference. 2014.
[5] ABBEEL, Pieter; NG, Andrew Y. Apprenticeship learning via inverse reinforcement learning. In: Proceedings of the twenty-first international conference on Machine learning. ACM, 2004. p. 1.

Diploma Thesis Supervisor: RNDr. Michal Čertický, Ph.D.

Valid until the end of the summer semester of academic year 2017/2018

L.S.

prof. Dr. Michal Pěchouček, MSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 19, 2017

Acknowledgement / Declaration

I would like to thank my supervisor Michal Čertický for the valuable comments and remarks he has given me during the creation of this work.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 26. 5. 2017

.....

Abstrakt / Abstract

Přes úsilí vynaložené na výzkum Umělé Inteligence, boti do strategických her reálného času nedokáží ohrozit profesionální lidské hráče. Stále je tu spousta výzev, které výzkumníci musí překonat, aby AI mohla prazit experty. V této práci se zabýváme třemi výzvami: adaptivním plánováním, integrací doménové znalosti a integrací AI technik do jednotné architektury. Představujeme použití techniky Inverse Reinforcement Learning jako způsobu, jak se naučit dělat rozhodnutí na základě pozorování her hraných hráči. Abychom mohli integrovat Inverse Reinforcement Learning společně s dalšími technikami vyžadovanými pro realizaci kompletního bota, postavili jsme naší AI na nové jednotné architektůře v podobě vysoce decentralizovaného Multi-agentního systému. Potom, co jsme použili malou množinu ukázek her, náš bot se by shopný naučit strategii, která dokáže v některých scénářích porazit zabudovanou AI. Bot také vykazuje schopnost uzpůsobit své chování situaci. Náš přístup demonstruje nový způsob jak s minimem doménových znalostí vyvíjet bota, který bude výzvou pro lidské hráče.

Klíčová slova: Inverse Reinforcement Learning; Multiagentní system; strategie reálného času; bot

Překlad titulu: Učení se hraní strategických her reálného času z demonstrací s využitím decentralizovaného MAS

Despite the amount of effort put in Artificial Intelligence research, bots for real-time strategy games present no threat for professional human players. There are still many challenges to overcome by researchers to develop AI able to beat experts. In this work, we deal with three challenges: adaptive planning, domain knowledge integration and integration of AI techniques to unified architecture. We introduce the usage of Inverse Reinforcement Learning as a new approach for decision-making based on the observation human gameplay. To be able to integrate Inverse Reinforcement Learning with other techniques needed for the complete bot, we build our AI on new unified architecture in the form of a highly decentralized Multi-agent system. After using a small set of replays, our bot was able to learn strategy which beats built-in AI in some scenarios. The bot also shows the ability to adapt its behavior to the situation. The approach presents a novel way of developing challenging bots with little to none domain expert knowledge.

Keywords: Inverse Reinforcement Learning; Multi-agent system; real-time strategy game; bot

Contents /

Tables / Figures

Chapter 1
Introduction

Despite the amount of effort put in AI research to develop a program capable of playing real-time strategy (RTS) video games and recent successes of programs such as AlphaGo [1] and DeepStack [2], professional players in complex RTS games remain unchallenged by an expert level AIs. Making the game more fun to play for humans by developing advanced AI is not the only one benefit of AI research in this area. Tasks carried out in hardly predictable environments of those games, resemble real-world military scenario which demands to solve many complex issues quickly and satisfactory. Most of the decision making in real-world has similar nature. Therefore one can expect that techniques used in AI able to master an RTS game could perform well in other domains.

1.1 Motivation

There are many challenges to overcome by AI researchers to be able for their bots to match human expertise in RTS games. In our work we would like to contribute to following ones:

- **Adaptive planning**: most bots are not able to adapt their strategy to counter the opponent. Instead, they use some hard-coded triggers to select between sets of strategies or mix between them based on performance or by random.
- **Domain Knowledge integration**: this is related to lack of ability to adapt as many bots creators struggle to incorporate various forms of domain knowledge to their bots. Excellent sources of domain knowledge for the game are demonstrations of human play. Observing other players is a natural way for a human to learn to play the game but it does not hold for bots.
- **Domain Knowledge integration**: playing RTS is multi-scale AI problem. Bots creators know that and decompose the problem to sub-problem. However, approach how they do that varies from bot to bot.

The test-bench for our research is RTS game Starcraft: Brood War where all three challenges and the ones described in 3.1 remain valid. Researchers have been working on techniques to solve the issues of bots in many RTS games over a decade. However, StarCraft: Brood War has become the first choice for this kind of research in recent years. It has one significant advantage over other similar games in a number of competitions for AI StarCraft bots (organized by AIIDE, CIG, and SSCAIT [3]) so one can easily see how created bot is doing against others. Another advantage is that players' community around the game is still active even though the game was released back in 1998.

1.2 Objectives

Our primary goal of this work is to develop a bot for StarCraft: Brood War which learns its decision-making processes from the demonstration. To meet this aim we set up following guideline:

1. Review current approaches for building expert level real time strategy game-playing AI.
2. Identify decision-making processes realized by a human player in the expert level gameplay of real-time strategy game StarCraft: Brood War.
3. Implement game-playing agent for StarCraft Brood War. To reduce complexity and amount of required expert knowledge, design the game-playing AI as a highly decentralized multiagent system with the ability to learn from demonstration (watching replays of StarCraft: Brood War games played by humans).
4. Use Markov decision processes and Inverse Reinforcement Learning to train decision modules for individual MAS agents.
5. Evaluate and discuss the capabilities of game-playing AI and compare it to state of the art AIs.

1.3 Contribution

By meeting our objectives, we deliver following contributions:

- By learning decision-making in a different situation of the game by observing experts, we eliminate most of the hard-coded behavior in our bot in the sake of better adaptability. It is also a good example how to encode part of the domain knowledge which is otherwise hard to get on the highest level of abstraction.
- Using Markov decision processes and Inverse Reinforcement Learning is way how to explain/decide behavior in given situation. Approaching problem trough utility based planning has some advantages over goal-oriented planning used in most current bots (more on differences in 5.1.1).
- Decomposition of the problem is inevitable as hardly only one technique can be used to solve all subproblems (which is in alignment with results of famous No Free Lunch Theorems[1]), we developed a prototype of unified architecture in the form of an opinionated framework based on the multi-agent system. Not only our StarCraft bot project can benefit from this as our framework is domain independent.

1.4 Outline

In 2 we give a broader introduction to the domain of RTS games, StarCraft and analyze decision making involved to identify decision-making processes. 3 is a continuation of 2 and presents challenges and techniques in RTS game AI development. Another purpose of that chapter is to introduce ways how bots are being developed. All of this is useful in our attempt to build our bot. In 4 we provide a description of Inverse Reinforcement Learning technique which we use to learn to make decisions in the same way as humans do it in given situation. 5 presents an explanation of what we understand under term agent as we see our bot as an agent. An important part of this chapter is to explain types

[1] `http://www.no-free-lunch.org`

of the agents to give some classification of them as our bot is partially concerning utility in contrast to common bots. We also provide a description of the multi-agent system as we use it for task decomposition in the form of the opinionated framework described in 6. 6 is also a place where we bring everything together as we present way how we integrated decision making to our bot and how other techniques can be incorporated. In 7 we present our initial version of the bot with examples of the framework usage and IRL integration. This chapter also evaluates its current performance and gives an analysis of current issues. In last 8, we give a recapitulation of our work and discuss directions for future work.

Chapter 2
Identification of decision-making processes realized by humans in Starcraft

In this chapter we want to introduce real-time strategy (RTS) games, give an overview of decision-making processes of player in those games, especially in the domain of StarCraft: Brood War, and show that mastering any RTS game is nothing trivial, even for a human.

2.1 Real-time strategy games and their complexity

RTS is one of the sub-genre of strategy games. In those kinds of games, the player usually needs to build economy (collect resources and construct buildings) and military power (by training and upgrading units in buildings for gathered resources) to defeat his opponents (by destroying his army or economy). The „real-time“ gives RTS games other dimensions compare to a classic game of Chess. Each player has a small time frame to decide the next move in an environment where players' actions are simultaneous as players can issue play command in same time. Most of the actions are not instantaneous; it takes some time to complete them to see the result. On top of that, the RTS games are partially observable as players do not have full perception of the state of the affair in the world. To this situation is referred as fog-of-war because the player can not see unexplored parts of the maps and do not know situations in parts of the map where he does not have his units. Moreover, those games are usually non-deterministic as actions may not succeed given their chance of failure, and most importantly the complexity of the state and action space is enormous.

In [4] author gives decision space complexity (set of possible actions which can be executed at a particular moment) estimation of RTS trough StarCraft as follows:

$$(1)\ O\left((W \cdot A \cdot P)(T \cdot D \cdot S) + B \cdot (R + C)\right)$$

- W - number of workers
- A -number of the type of worker assignments
- P -average number of workplaces
- T -number of troops
- D -number of movement directions
- S -number of troop stances (Attack, Move, Hold)
- B -number of buildings
- R -average number of research options at buildings
- C -average number of unit types at buildings

Given an extremely simplified scenario of the game on 256x256 tile map in SC: BW with 50 workers results in 1 000 000 000 possible actions which is orders of magnitude higher comparing it to the complexity of Chess. Same goes for state complexity. For example, Chess is estimated to be around 10^{50} and Go around 10^{170}. However, StarCraft

scenario on the typical map is believed to be many orders of magnitude larger. More detailed discussion of StarCraft complexity can be found in [5].

Figure 2.1. AlphaGo plays Lee Sedol in 2016 (from: [6]).

2.2 Decision-making processes in RTS games

Due to the complex nature of problem playing RTS games, the most common approach is by decomposition of the problem into a collection of subproblems which can be solved independently. Conventional subdivision (not the exclusive one) is according to [5] as follows:

- The **Strategy** is the most abstract level of game comprehension. It corresponds to the high-level decision-making process and concerns all units and building as well as properties of the environment. Finding successful strategy against given opponent is key to defeating him.
- **Tactics** is a way how to realize strategy. It focuses on groups of units and implies theirs positioning, movements, timings and so forth.
- **Reactive control** is the implementation of tactics concerning particular unit. It involves moving, targeting, fleeing and so on.
- **Terrain analysis** is part of environment analysis (map specifically), the primary goal is for example to identify strategic locations, resources, and distances. This knowledge is then employed in other processes.
- **Intelligence gathering** that corresponds to information collection due to the partial observability of environment to gain intelligence on the opponent.

Levels of abstraction described above and their relation to uncertainty coming from partial observability and not knowing specific intentions of the opponent, timing corresponding to a duration of behavior switching, and spatial and temporal reasoning are on figure 2.2.

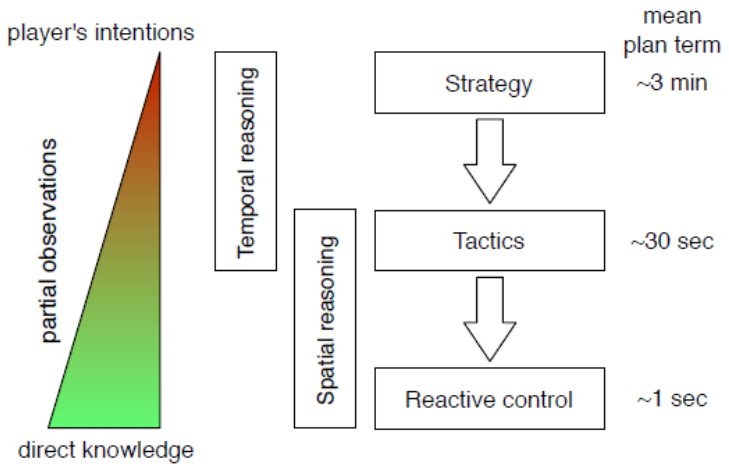


Figure 2.2. RTS AI levels of abstraction and theirs properties (from: [5]).

Churchill in [7] gives a deeper elaboration of Strategy, Tactics and Reactive Control description and their subtasks as can be seen on figure 2.3. On this figure can also be seen information flow hierarchy between those subtasks similar to the military command structure.

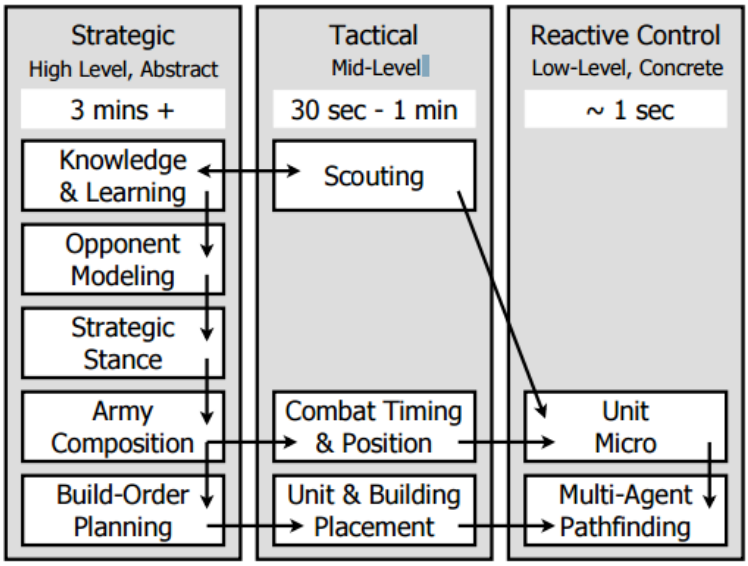


Figure 2.3. The main sub-problems in RTS AI research categorized by their approximate time scope and level of abstraction (from: [7]).

Subtasks for **Strategy** are as follows:

- **Knowledge and learning** which are vital for playing the game (rules, unit properties, openings, knowledge about the opponent and so long). This knowledge can be further extended by playing matches and gathering additional information about the game itself and opponents.
- **Opponent modeling and prediction** is useful to the player because the environment is only partially observable so one can not be sure what is his opponent doing. In this case, modeling and prediction come in handy as its enables player to exploit perceived weaknesses.

- **Strategic Stance** corresponding to the ability to balance between aggression and economic expansion. The choice of particular stance influences things such as army composition and attack timing.
- **Army composition** is decided by Strategic Stance, Opponent modeling and prediction, and by the current situation.
- **Build-Order Planning** goal is realizing decided army composition by gathering resources and building infrastructure to have resources, capacity and meet requirements to train desired units.

As for **Tactics** subtasks are following:

- **Scouting** is a way to gather intel on the opponent as areas of RTS maps are typically covered by fog-of-war which unable vision of other sectors and enemies expect those in the direct vicinity of friendly units. Sometimes is possible to use technology or ability to uncover part of the map. Scouting is a vital part of the game as it provides information to the player to adjust his play and make right decisions.
- **Combat Timing and Positioning** are crucial in RTS games as it involves decision where and when to attack the opponent. For example attacking opponent's new expansion when it still under construction can help the player get an edge in a match.
- **Building Placement** plays most important role in the game, especially in the beginning and is very influenced by map, opponent, strategy and so. For example, one can imagine situation by placement of additional defensive buildings player can discourage the enemy from attack, or at least delay it.

For **Reactive Contro**l Churchill in [7] list following subtasks:

- **Unit Micro** is mostly referred to in combat situations as it dictates individual units behavior. Decision processes here are particularly hard as one needs to manage many units and their actions simultaneously.
- **Multi-Agent Pathfinding and Terrain Analysis** are an integral part of RTS games. Pathfinding is related to Unit Micro and involves pathfinding and other more complex optimizations to avoid getting damage and so. Terrain analysis gives player not just necessary inputs for pathfinding algorithms but high-level info about properties of the map such as resources location.

Despite the decomposition, most of the task remains quite complex as is illustrated in [8] where many games (even StarCraft) are analyzed and from the discussion of researchers and authors of SC: BW agents in [9]. On top of that for many of the tasks, the optimal solution does not exist which is shown in 3.

2.3 StarCraft specifics

StarCraft and its expansion StarCraft: Brood War was released in 1998 by Blizzard Entertainment and became tremendously popular. To win the game, mechanics are similar to traditional RTS games. The player needs to collect resources (in this case mineral and gas) to be able to construct buildings, train units and unlock upgrades for them. Resources further can be spent on defensive buildings to help units to defend strategic points of a map given the specific situation. Player uses units, not just for defense; he distributed them among other tasks such as scouting, attack, maneuvering them when they meet the enemy, and various „mind games“ to confuse the opponent.

Figure 2.4. StarCraft: BroodWar in-game screen.

The game is set in a science-fiction universe where the player takes the role of commander of an army of one of the three races:

- Zergs as an insectoid alien race with cheap and weak units which can be produced fast giving the player ability to overwhelm opponent simply by numbers.
- Protoss is also alien race similar to humans. Units are the exact opposite of Zergs'. They have expensive production regarding costs and manufacturing times. Playing this race means to concentrate more on quality over quantity.
- Terrans represents humans in this world with units balanced between Zergs and Protoss.

Despite the fact that each race contains 30-35 unique types of buildings and units (most of them has own special abilities) is extremely well balanced. The typical game takes place on the map of dimension between 64x64 to 256x256 build tiles (32x32 squares of pixels). The game does not allow each player to control more than 200 units at the time, but the player can have an unlimited number of buildings. All of this makes StarCraft challenging on the one hand, but extremely fun to play (watch) on the other. Thanks to this StarCraft became famous eSport matter with contestants earning a considerable amount of money [10].

In comparison to 2.2 when humans playing StarCraft they typically abstract decision making in this manner:

- **Micromanagement** (Micro) corresponds to Reactive Control and partially to Tactics described in previous section. It involves the ability to control units individually. Good micro players are more likely to keep their units alive longer.
- **Macromanagement** (Macro) corresponds almost to everything except Reactive Control and Tactics (except the part from micro). It represents the capacity to produce right units and expand at appropriate times to ensure that production of units is flowing. A player who is macro-oriented has a usually larger

This chapter is especially vital for our bot development as we had very sparse knowledge about StarCraft: BroodWar game. In 7 we present implementation of our bot based on knowledge from this chapter.

Chapter 3
Challenges and techniques in RTS Game AIs development

In previous section 2 we showed that playing RTS games can be quite challenging, even for a human. In this chapter, we present the challenges in the development of RTS Game AIs and how game agents creators and researchers approach them.

3.1 Challenges in RTS Game AIs

According to [5] current problems in RTS Game AIs can be grouped in 6 different areas as follows:

- **Planning**: As was mentioned in 2.1 due to the enormous size of action and state space, planning in this domain presents a problem where standard adversarial planning approaches are not directly applicable. A common approach is to use multiple levels of abstractions as was shown in 2.2. However, this may not be enough as we explain on common techniques in the following section.
- **Learning**: due to limitations of usage of standard adversarial planning techniques researchers have been training to employ learning techniques to improve game AIs. In this area, most efforts were put into Prior Learning as a way how to learn appropriate strategy before game using for example replays and specific map information. Another are of focus is deploying online learning techniques allowing agents to improve their play while playing a game. It is called In-game learning. Researchers are also interested in Inter-game learning with the goal to increase the chance of victory in next game by learning from previous one.
- **Uncertainty**: what makes planning in this domain even harder is uncertainty as adversarial planning under uncertainty in domains of the size of RTS games is still an open question. Uncertainty comes from 2 sources – environment is partially observable, and a player cannot predict opponent's actions.
- **Spatial and Temporal Reasoning**: be able to position unit and building well in right time is a vital part of each game playing agent for strategy and tactic execution. Therefore, RTS AIs developers pay much attention to Spatial and Temporal Reasoning.
- **Domain Knowledge Exploitation**: in the case of RTS games (compare to for example board games) exploitation of domain knowledge remains quite an uncharted territory. There have been two main directions: in most common researchers are focusing on hard-coding strategies to agents, so agents have to only decide on an action from the predefined set of strategies. In other researchers are trying to learn plan, strategies, or trends from replays. However, how to learn any of this in games like StarCraft automatically is still uncertain.
- **Task Decomposition**: due to many challenges mentioned before and the fact that decomposition is natural even for humans (as is seen in 2.2), a decomposition is a

preferred approach for developing game AI. However, even it presents many challenges. The significant one is on design architecture which would enable individual AI techniques to work well together.

As was mentioned in 1 we are concerned on: Adaptive planning, Domain Knowledge integration and Domain Knowledge integration.

3.2 Techniques for RTS AI development

To develop game playing agent, one needs to address most of the previously mentioned problems so classification according to this problems is hard. So [5] categorize techniques rather to 3 branches: strategy, tactic and reactive control. What each branch represents is described in detail in previous section 2.2.

3.2.1 Strategy

Making decisions on strategy level presents still an open problem in the domain of RTS games because of the size of the search space. Solutions based on Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs) are hardly applicable here. Over the years there have been many other simpler approaches how to address those problems from planning based methods, hard-coded solutions to machine learning methods. We give a short description for just a few of them in the following text (more details can be found in [5]). Is worth to mention that most of the techniques mentioned assume complete information which may be problematic in an environment of imperfect information.

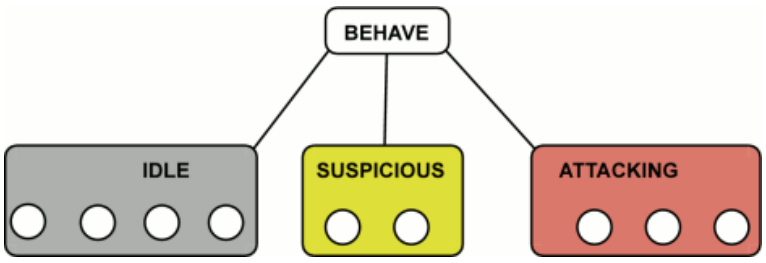


Figure 3.1. Behavior Trees: A tree made of modular behaviors (from: [11]).

Most commonly used techniques are ones that are hard-coded. That especially holds in commercial RTS game industry. The most typical case is to use finite state machines (FSM) where AI behavior is decomposed to manageable states with conditions triggering transitions between them. Despite FSMs popularity and level of adoption by developers of game AIs, they are easily exploitable by opponents who can adapt, because they struggle to encode dynamic and adaptive behavior. Other approaches based on FSMs such as Hierarchical FSMs and Behavior trees (BT) (shown in figure 3.1) remain exploitable too. Other well-explored techniques providing more flexibility are approaches employing planning techniques such as Case-based planning (CBP) and Hierarchical Task-Networks (HTN). Many researchers have been trying to approach the problem from machine learning point of view employing massive data sets of replays available. Very popular technique among researchers is case-based reasoning (CBR). In [12] it is used for Army Compositions. Examples of CBR and other less common machine learning techniques such as Hidden Markov Models, Bayesian networks, and evolutionary algorithms, are listed in [13].

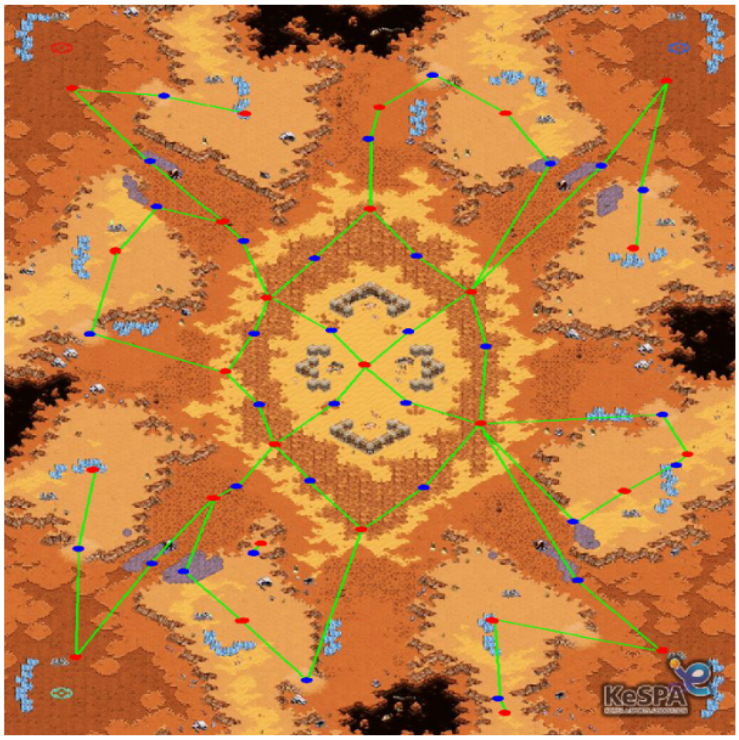


Figure 3.2. Starcraft map with regions and choke points (from: [14]).

3.2.2 Tactics

Tactics involve two part: reasoning about tactical decisions in a battle and terrain analysis. When concerning those parts of tactics composed mainly from spatial and temporal reasoning for fighting battles, all the techniques mentioned in section 3.2.1 can be applied as well. On top of that literature such as [5] gives examples of other techniques such as Answer Set Programming (ASP) for walling (intentionally blocking the entrance to base shown in figure 3.3), UCT algorithm (a Monte Carlo Tree Search algorithm) for tactical decisions, and so long. Most of the work done in Terrain analysis is usually performed off-line before a game as most of the information gathered about map holds for the whole match. Terrain analysis techniques are for example based on influence maps, application of Voronoi decomposition to detect regions and choke points, and others (example of analyzed map is in figure ter]).

3.2.3 Reactive Control

There are many useful techniques to maximize the effectiveness of units, but Potential fields and Influence maps are the most prominent ones. Those techniques are used for things like obstacles avoidance or for staying in maximum shooting distance to minimize taken damage. There are other popular approaches based on usage of simple pathfinding algorithms represented in many cases by A* (example is on figure 3.4 and 3.5). Many researchers have been exploring options to use machine learning techniques already mentioned in 3.2.1 and 3.2.1 for reactive control. On top of that, many of them have tried to employ reinforcement learning (RL) of some kind.

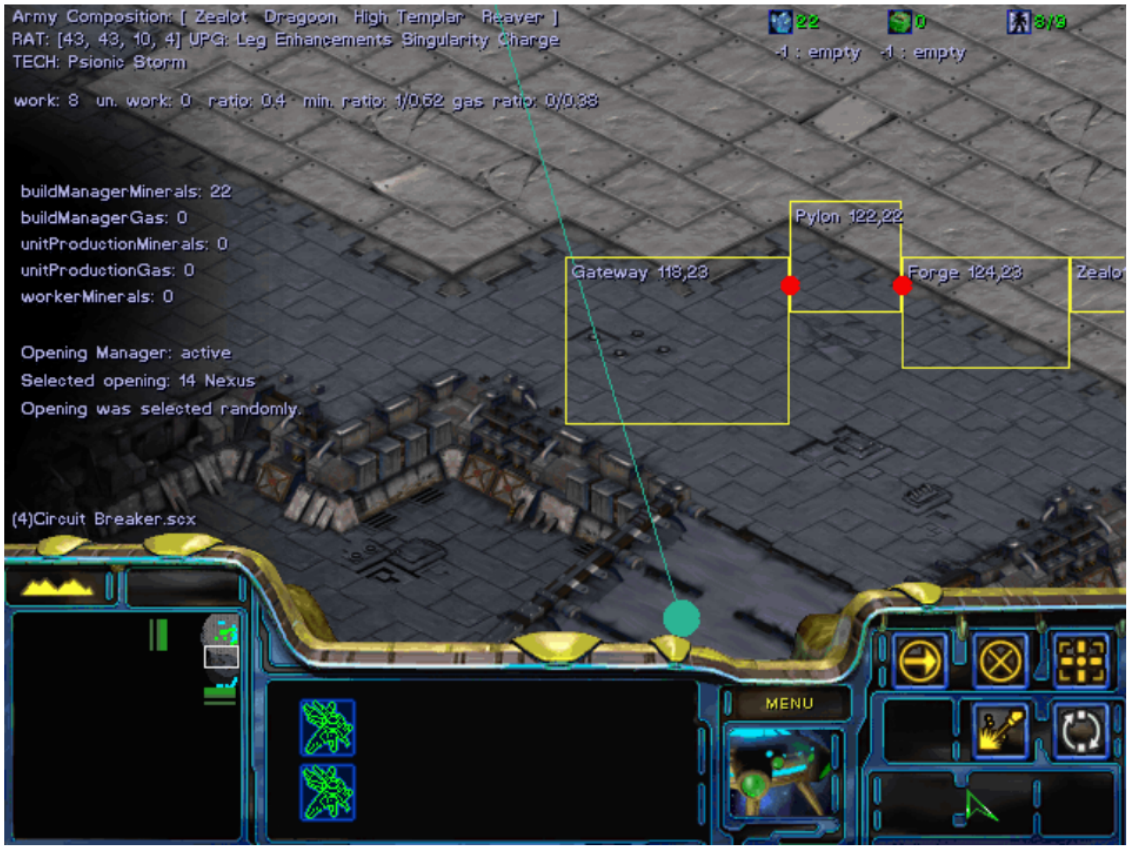


Figure 3.3. Example of wall-in placement as a Protoss. The wall consists of a Gateway, Forge and Pylon structures and a Zealot unit. In CSP terms, variables from X = $\{Gateway, Pylon, Forge, Zealot\}$ are assigned the values of (118, 23),(122, 22),(124, 23) and (126, 23) respectively (from: [15]).

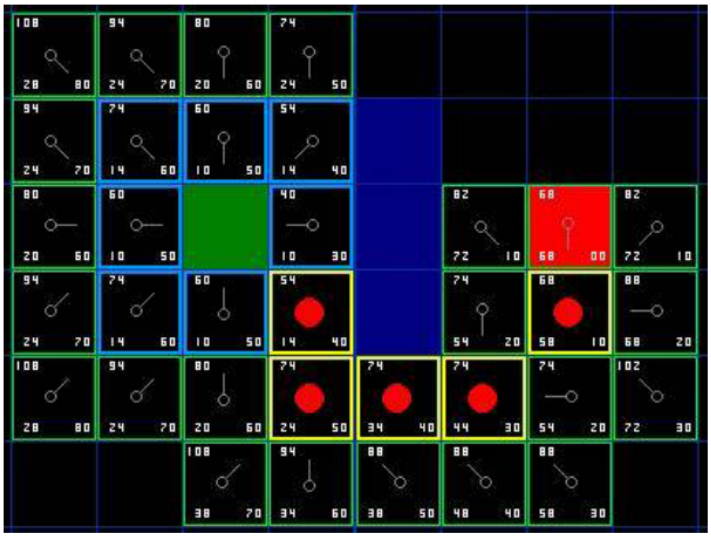


Figure 3.4. A* example (from: [14]).

Figure 3.5. A* in Starcraft (from: [14]).

3.3 Design of the state of the art game playing (StarCraft) bots

Authors of [5] give a practical overview of architectures (see figure 3.6) used in StarCraft bots participating in StarCraft AI competitions. The current situation on the field of complete game playing agents for StarCraft does not differ much from the one in 2013. Developer and researchers have been working to some extent on integrating many of the techniques introduced earlier to complete bots. The truth is that incorporating techniques alone to bot is not enough to match human ability to play RTS game, so designers use a lot of domain knowledge to improve the play of their agents. The typical approach is to divide the problem to subproblem which alone is technique how to handle such complex issues. Creators then can for each type of subproblem choose appropriate method how to deal with the problem. The real art of complete bot development is designing architecture which can integrate many techniques together to get an intelligent agent. By analyzing top bots' structures [5] identifies following tools used by creators to help them achieve the goal:

- **Abstraction**: it is very common for AI agents in StarCraft to reason about the task on different levels of abstraction to make the problem easier to solve. For example playing a game can be seen from a high level as deploying strategy and from low-level spectrum as issuing commands to individual units. The usual practice is to develop a module for each level of abstraction and use outputs of reasoning on a high level of abstraction as input for the lower level. For example, the top-level module will select a strategy to execute, and the lower level module uses this to come up with build order which is then performed by the lowest level module.
- **Divide-and-conquer**: playing the game can be divided to separate task which can be under some assumptions handle relatively independently of each other so one module can, for example, concentrate on gathering resources and other on managing units in the battlefield.

Lots of bots use a combination of those two tools. A good example of a combination of both is using multi-agent system (MAS) architecture ([4], [16] and [14] to some

extent). As we are using MAS as well, we discuss decomposition using MAS at the end of section 5.2.

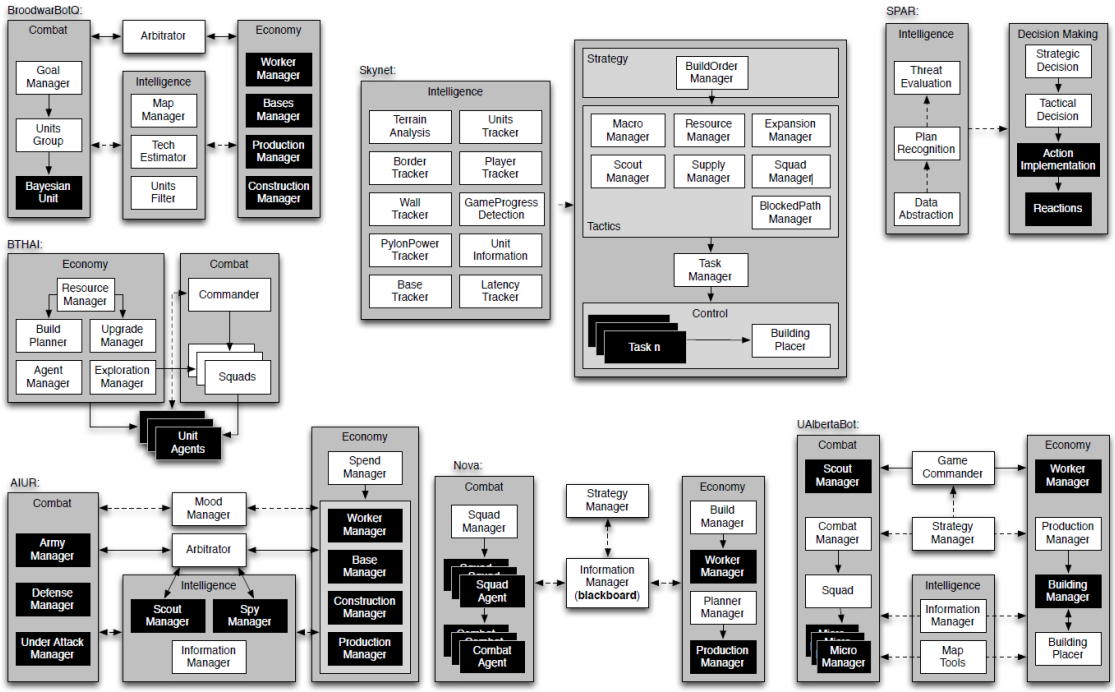


Figure 3.6. Architecture of 7 StarCraft bots obtained by analyzing their source code. Modules with black background sent commands directly to StarCraft, dashed arrows represent data flow, and solid arrows represent control (from: [5]).

Interesting fact about most current bots is that on the higher level they are usually scripted which usually involves using a set of predefined strategies to be executed by the agent. In better scenarios agents are mixing between those strategies. However, even this is not enough to match human player who can adapt to those strategies in worst case after few games as he can easily predict what will bot play based on previous games. On top of that agents lack adaptivity because once strategy is selected bot will follow it for the rest of the match.

Chapter 4
Inverse Reinforcement Learning

In this section, we would like to introduce Inverse Reinforcement Learning (IRL) as an extension of Reinforcement Learning (RL) technique. Motivation for this technique was [17] presenting autonomous helicopter capable of performing aerobatics after observing experts (see figure 4.1).

Figure 4.1. One helicopter while performing one of the airshows. It was trained trough IRL by observing experts (from: [17]). airshows.

4.1 Reinforcement Learning and Markov Decision Process

Reinforcement Learning is very popular machine learning technique to solve situations where the agent does not know the payoff it will receive for its actions, so at first it has to explore the environment by taking random steps to learn what is an actual payoff for any action in given situation. To some extent, this can be seen as a very simple

form of trial and error learning similar to learning experienced by humans during their lives.

In RL agent's goal is to choose right action in any states he is currently in to maximize his feature discounted reward. That is, to find optimal **policy**. To give a formal definition of problem we need to formalize (finite) Markov decision process (MDP) first. An MDP notation in [18] is given as a tuple $(S, A, P_{sa}, \gamma, R)$, where

- S is a finite set of **states** with cardinality N
- $A = \{a_1, \cdots, a_k\}$ represents set of k **actions**
- $P_{sa}(.)$ are the state **transitions probabilities** upon taking action a in state s
- $\gamma \in (0, 1)$ is the **discount factor**
- $R : S \rightarrow R$ is the **reinforcment (reward) function** is bounded in absolute value by R_{max}

Further on we will write notation R(s, a) as R(s) for simplicity in equations. A policy is defined as any mapping $\pi : S \rightarrow A$, and the **value function** for a policy π is given as the expectation over the distribution of the sequence of states we visit by executing policy π:

$$(1)\ V^{\pi}(s_1) = E\left[R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + ... | \pi\right], s_{1...N} \in S$$

Q-function is given as:

$$(2)\ Q^{\pi}(s, a) = R(s) + \gamma E_{s' \sim\ P_{sa(.)}}[V^{\pi}(s')]$$

where notation $s' \sim P_{sa(.)}$ means the expectation with respect to state s' distributed according to $P_{sa(.)}$. Assuming that MDP can model all agent's decision one can find optimal policy π^* such that $V^{\pi}(s)$ is maximized. Algorithms finding optimal policies are based on two basic properties of MDPs. All $s \in S, a \in A, V^{\pi}$ and Q^{π} for an MDP satisfy (Bellman Equations):

$$(3)\ V^{\pi}(s) = R(s) + \gamma \sum_{s'} P_{s\pi(s)}(s') V^{\pi}(s')$$

$$(4)\ Q^{\pi}(s, a) = R(s) + \gamma \sum_{s'} P_{sa}(s') V^{\pi}(s')$$

moreover, the policy is optimal policy for an MDP if and only if, for all $s \in S$ holds (Bellman Optimality):

$$(5)\ \pi(s) \in \arg max_{a \in A} Q^{\pi}(s, a)$$

Illustration of RL process to obtain optimal policy is in figure 4.2. Inputting environment model (MDP) and specifying reward function R(s) one can get optimal policy trough Reinforcement Learning.

4.2 Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) is a form of Apprenticeship Learning in MDP where in contrast to RL reward function is not known. Instead, few trajectories demonstrating expert performing the task to learn are available. The goal of IRL is then to use those trajectories and learn reward function that can explain expert's behavior. Recovering reward function first and using it to generate desirable behavior (trough RL) is what makes IRL standout from other supervised learning techniques as most of them is learning policy as a mapping from states to actions right away. According to [18]:

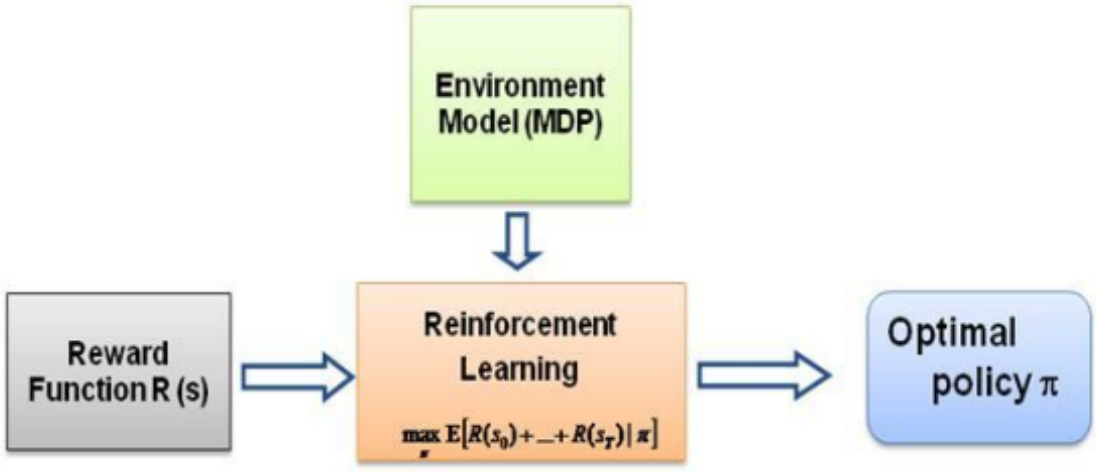


Figure 4.2. The reinforcement learning framework (from: [19]).

„reward function provides a much more parsimonious description of behavior." As is mention in [18]: „the entire field of RL is founded on the presupposition that reward function, rather than the policy, is the most succinct, robust, and transferable definition of the task." IRL is especially useful when we are attempting to learn intelligent agent that can behave successfully in our domain where is hard to define reward function; it is extremely labor or designer has only limited knowledge of it. RTS games are excellent examples of that as results of actions are not immediately observable. In figure 4.3 one can see IRL framework in contrast to RL framework illustrated in figure 4.2 we are using optimal policy to learn reward function.

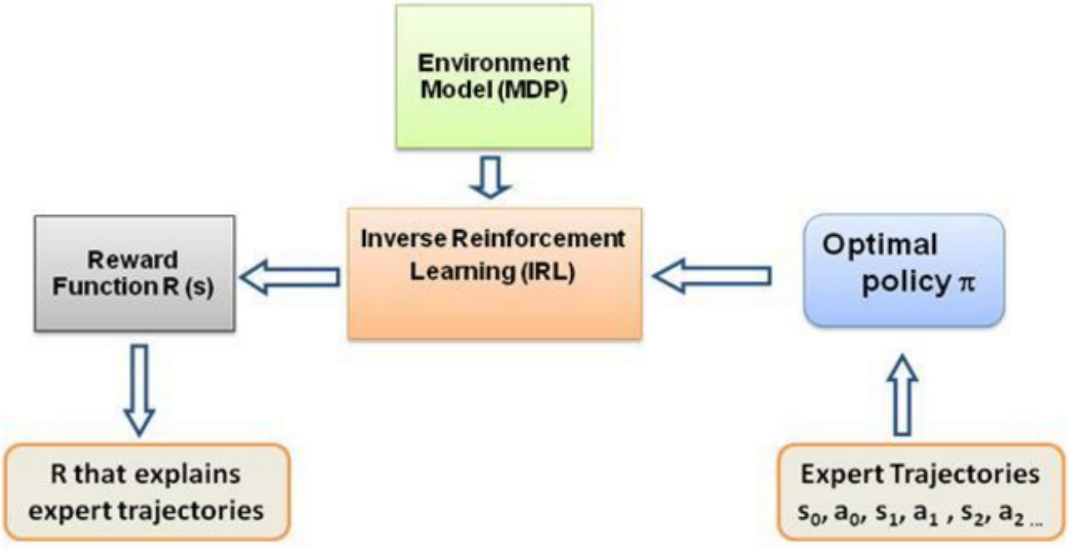


Figure 4.3. The Inverse reinforcement learning framework (from: [19]).

4.3 Solving Inverse Reinforcement Learning problem in Finite State Space

As was shown in [18] a solution to IRL problem in Finite State Space (for finite MDP) can be found trough **linear programming** (LP). LP can be formulated based on the characterization of the **Solution Set** consisting of all reward functions for which given policy is optimal. Following theorem from [18] characterizes the Solution Set:

Definition 4.1. Let a finite state space S, a set of actions $A = \{a_1, \cdots, a_k\}$, transition probability matricies $\{P_a\}$, and a discount factor $\gamma \in (0, 1)$ be given. Then the policy π given by $\pi(s) \equiv a_1$ is optimal if and only if, for all $a = a_1, \cdots, a_k$, the reward R satisfies

$$(6)\ (P_{a_1} - P_a)(I - \gamma P_{a_1})^{-1} R \succeq 0$$

Proof of 4.1 is based on substitution equation (3) to (4) from theorem written above and can be found in [18]. Before giving LP formulation, two problems with this characterization of the Solution Set need to be considered. First, any constant vector is always a solution of R, and second, many solutions satisfying equation (6) exist so choosing one R may impose challenge. One of the ways described in [18] to overcome the problem of choosing R is to demand that selected R makes given policy π optimal and favors solutions that make any deviation from π costly as possible. From all R functions satisfying (6) and $|R(s)| \leq R_{max} \forall s$, we pick one maximizing:

$$(7)\ \sum_{s \in S} Q^{\pi}(s, a_1) - \max_{a \in A \setminus a_1} Q^{\pi}(s, a)$$

Then we can formulate the optimization problem as LP which we want to maximize:

$$\sum_{i=1}^{N} \min_{a \in \{a_2, \cdots, a_k\}} \{(P_{a_1}(i) - P_a(i))(I - \gamma P_{a_1})^{-1} R\}$$

$$\text{s.t. } (\mathrm{P}_{a_1} - P_a)(I - \gamma P_{a_1})^{-1} R \geq 0,\ \forall a \in A \setminus a_1$$

$$|R_i| \leq R_{\max}, i = 1, ..., N$$

In our opinion, IRL can make a useful addition to already existing approaches (more on this in section 3.2) how to employ huge data sets with examples of professional play in games like StarCraft and bootstrap some of the expert level domain knowledge to AI agents.Due to the complexity of our domain and computational feasibility, we currently restrict ourselves to the simplest cases of IRL. We consider only IRL in Finite State Space for our experiments (to do that we are using clustering as presented in 6). Nevertheless, human StarCraft players seem to also reason about finite number of typical game situations (states). Evidence of that in StarCraft domain is the existence of many guides[1] how to play in given state of affairs.

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

Chapter 5
Agents and Multiagent Systems

In this section, we give one of the definitions for term agent as it is understood by [20] and [21] in the field of Artificial Intelligence. Understanding what (intelligent) agent is in terms of AI, has particular importance for us because our goal is to create one. More precisely an agent – bot that would be able to play StarCraft and learn some of the decision making involved from professional players. Because it is intractable to learn whole decision-making trough IRL at once, and we need to use other techniques in our bot (actually many techniques are used together to do the job as is shown in section 3.2) decomposition of some kind is inevitable. We decided to design our agent as Multiagent system (MAS) to help us decompose the problem of playing StarCraft as we think that it is the natural way how to approach decomposition. Therefore we also give a characterization of Multiagent Systems in this chapter to show that MAS is a logical way how to approach the problem. Using MAS of some sort for architecture is nothing new in StarCraft bot development as we show in 3.3. At the end of this chapter, we present the reasoning behind the idea of using MAS as a tool for decomposition.

5.1 Description of (intelligent) agent

We use the definition from [20] and [21] which defines an agent as an **entity** that perceives its **environment** through **sensors** and **acting** upon that environment through **actuators**. This idea is illustrated in figure 5.1. Despite the vagueness of this definition, it can frame the thing to which we refer as game playing agent well. In the case of a human agent playing StarCraft, we can consider player's eyes as sensors and his hands as actuators giving commands to the game. On the contrary, given this definition, one can consider even the game program as an agent as it receives commands as sensory inputs and acts on the environment by displaying the game.

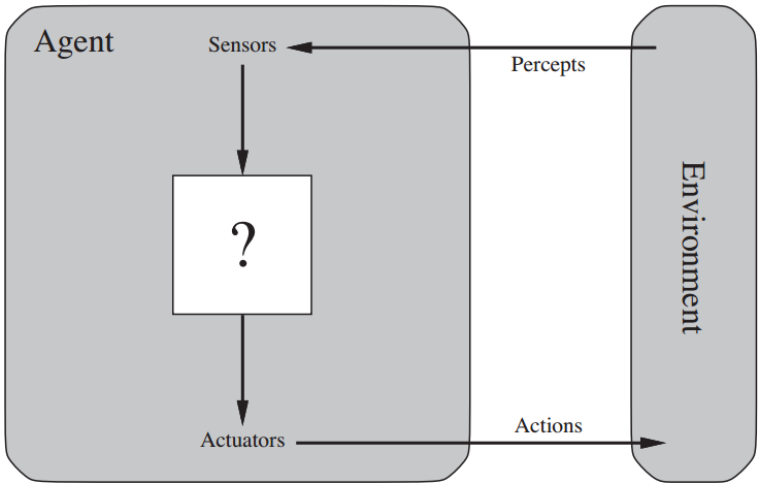


Figure 5.1. Agents interact with environments through sensors and actuators (from: [20]).

One of the differences between those two agents is **autonomy** – one agent decides on commands to send, and another one executes them. Another key difference is that the first agent mentioned can be considered as**rational**. One of the definitions for rational agents based on [20] can be stated in the following manner. For each possible sequence of sensory input, a rational agent should select an action that is expected to maximize desirability of the resulting situation of the environment for the agent (design objective), given the evidence provided by sensory input and built-in knowledge agent has. A rational agent autonomously acting according to its best interest in every situation can be to some extent considered **intelligent**. Environment properties play a major role in the design of intelligent agent. As was shown in section 3.1 designing a rational agent to play StarCraft still imposes significant challenge. Current agents are not flexible enough in every situation. They very often lack **reactivity** - the ability to respond in timely fashion to changes that occurred.

5.1.1 Types of agents

According to [20], there are four basic types of agent models embody the principles of intelligent agents:

- **Simple reflex agents** are also the simple kind of agents. Any action is taken only based on current sensory inputs. The action picking is based on if-then rules. A good example of such agents in StarCraft is a unit controllers programmed only in reactive fashion.
- **Model-based agents** keep some internal state of affairs of the environment that depends on the history of their sensory inputs using a model. The model represents knowledge how the world works. The action is then selected not just according to current sensory inputs, but state plays an important role too. Many agents in StarCraft use this approach as they are employing variants of finite state machines.
- **Goal-based agents** are based on the idea that knowing about the present situation of the environment may not be enough. Agents need goals to describe situations which are desirable. They are combining way how model-based agents choose actions with emphasis on actions which may lead to a goal. An example of AI agent in StarCraft following this principles is in [4].
- **Utility-based agents** try to solve the problem of the goal-based agents that defining goals may still not be enough in complex environments like StarCraft. So, they work with utility as a measure of desirability (preferences) for particular states of the environment. The architecture of this agent is on figure 5.2. This type is also in the vast interest of ours as Inverse Reinforcement Learning described in chapter 4 is a good example how to get a policy for such an agent. In this work, our main goal is to bring utility-based decision even to higher level of decision making (most utility-based decisions are currently restricted only to reactive control).

5.2 Description of Multiagent Systems

As was mentioned in previous section agent is in an environment. It is common that many agents share the same environment and there is a subset of agents where each agent must interact with agent different from itself. We refer to those kinds of environments with interconnected agents as Multiagent systems (MAS). Due to properties of agents, the system is highly distributed in nature, and one can perceive MAS as a

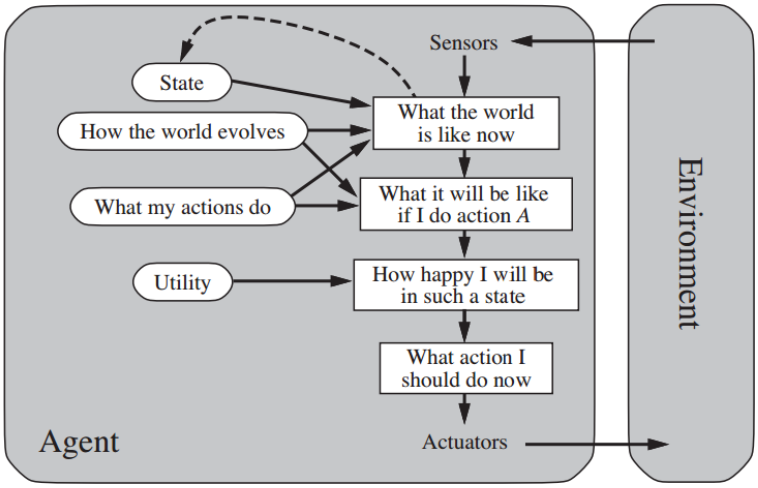


Figure 5.2. A model-based, utility-based agent (from: [20]).

form of distributed artificial intelligence (DAI). Solving problems using MAS is according to [21] best approach in situations where the multi-scale problem with following characteristics exists:

- The problem has many subproblems. Some of the subproblems can be (geographically) distributed and are heterogeneous.
- It has large content as it has a broad scope and covers a major part of a significant domain. Therefore there are many concepts to work with. Concepts in many cases work with huge amounts of data.
- The topology of the subproblems is dynamic, and content may change rapidly. Maintaining consistent information is hard.

The reasoning for using MAS in these situations is to let agents cooperate in solving the problems which are impossible to be solved by them individually using centralized approaches. Distribution of computation can allow solving situations which are otherwise impossible to solve or bring novel solutions. MAS forces system developers to implement the system in a modular fashion, to represent multiple viewpoints and knowledge of experts, therefore resulting system can be expected to be more fault tolerant and reusable. On top of that unpredictable interactions between agents make developers more likely to use declarative approach when defining agents.

From now on when the term MAS is referred to, it means a MAS where agents cooperate to reach common goals. For agents to be able to cooperate, they need to be able to communicate. Communication is necessary to achieve common goals as some form of coordination needs to take place between agents. Cooperation puts another requirement on most (rational) agents in those systems; they need to have the **social ability** – be able to communicate with others.

At the end of 3.3 we mentioned two tools for decomposition – **Abstraction** and **Divide-and-conquer**, MAS enables both of them. Using different components in the form of agents can be seen as a natural way for Divide-and-conquer approach and structure of the system – relation of the agents is a kind of abstraction. For example in StarCraft domain, one can see a unit as a vivid candidate on being an agent. So the problem of unit behavior is decomposed to smaller subproblems which can be than influence by other agents in higher hierarchy – commander of some kind who issues commands to the unit. For example telling unit by agent higher in command is the abstraction. Another advantage of MAS is a loose definition of an agent which can be decomposed

as well. All of the properties makes from MAS great tool for decomposition and way how to integrate many techniques to the system.

Chapter 6
Problem decomposition and learning integration

Decision-making is just one of many competencies required by a bot to be able to play games like StarCraft. In 2 we presented various other skills vital for successful bot and showed that those skills might have certain depth as it is in the case of decision making. To create a capable bot for games like StarCraft one must use many techniques presented in 3. Inverse Reinforcement Learning is just one of the techniques to obtain one particular skill, and it alone is not enough to realize bot. Therefore we developed an opinionated framework based on ideas of distributed multi-agent systems described in 5 to be able to integrate other techniques to our bot and due to the necessity to decompose the problem of playing StarCraft to more manageable subproblems. In the following chapter, we present our framework, the high-level architecture of our bot and integration of learning. The reasoning behind using MAS can be found at the end of the previous 5. All code related to our work can be found in public repository[1]. We give concrete examples on usage in 7.

6.1 Framework

First of all, it is important to explain why we bothered developing something such as framework in the first place when many others bots such as [4] and [14] use MAS ideas to some extent, and even some previous works aim at developing framework of some kind [16]. Simply it was not enough for our use case as those works are even more opinionated than our intended framework. In our opinion, it would put many restrictions on the future development as they do not meet our requirements:

- We do not want to enforce some static form of structure as relations between subproblems are dynamic and vary trough the time and situation. We want for parts of the system to make proposals of the goals that they think are right for the collective and augment the environment for others. Others can then decide their level of involvement and behavior for the situation. We want to have our components as simple as possible by reducing communication and focusing more on properties of the environment. This kind of contract introduce a loose way of coupling for individual components, enables greater abstraction with better decomposition and make problems simple.
- We want our component to be able to change its behavior based on circumstances. As we are dealing with complex problems, we want to make things easier by reasoning what our search space looks like first and then plan on this space.
- We want our framework to be modular and easily extensible with the possibility to integrate many techniques.

[1] `https://github.com/honzaMaly/kusanagi`

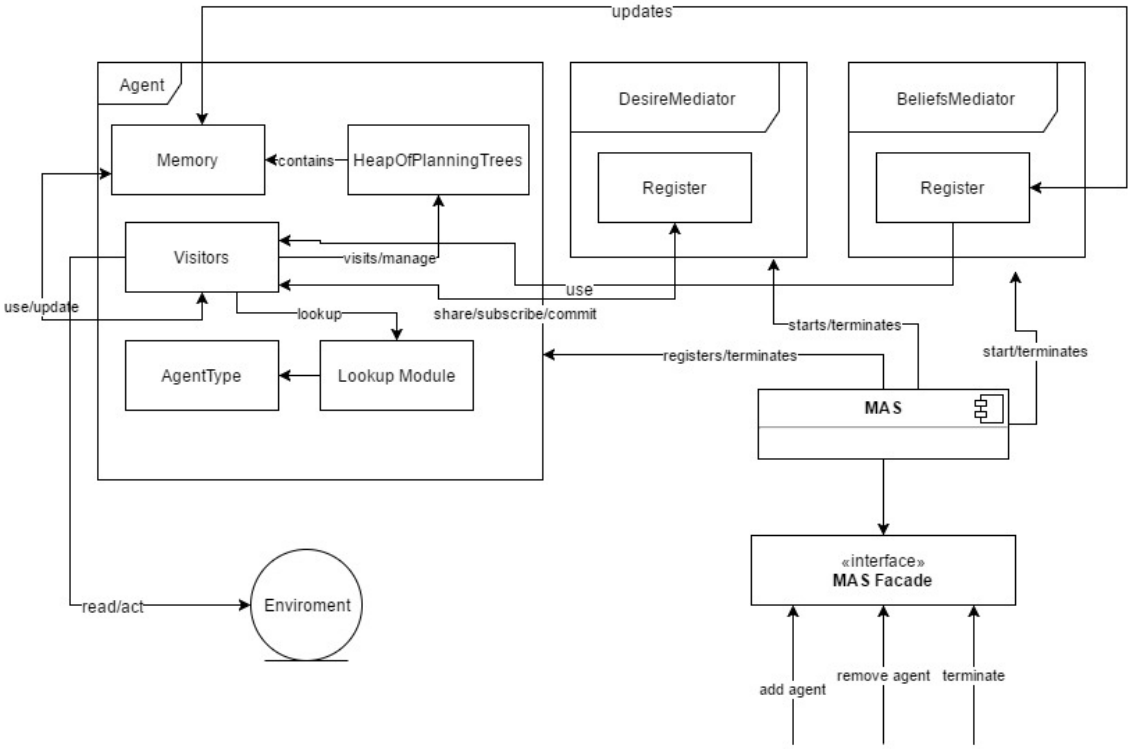


Figure 6.1. View on main components and their relations in our framework. The framework is based on the decentralized multi-agent system. We use two kinds of mediators to reduce communication required by agents to achieve simple and loosely coupled agents.

- We want for our framework to be portable; we may not want to use it just for StarCraft, and declarative to be able to add functionality to encode all system using JSON or XML configuration files.

In our framework, we use terms **Beliefs, Desire, and Intention** architecture from **BDI** [21] to model agents. It is one way how to model rational agents. From now on, to beliefs we refer as the information set available to the agent, desires are objectives which agent may want to achieve, and intentions are objectives agent has committed to accomplish. Intention also contains a **plan** to reach the goal. To illustrate the relation between those three terms we give following situation from StarCraft where the player has many options when choosing units to build (desires). Based on his beliefs about his and enemy army composition (beliefs) he starts to create a particular unit mix to adapt his army (intentions).

6.1.1 Architecture overview and basic properties

Based on our requirements we designed our framework as distributed multi-agent system. High-level architecture overview is in figure 6.1 where building blocks of our system are shown. Main components of the system are agents and mediators for desires and beliefs. The system is accessible trough facade.

The central part of our framework is an agent. Its implementation is in alignment with agent's principles outlined in 5. It has an own memory to represent beliefs where it stores data from sensors of the environment, additional facts it may deduce and accessible beliefs shared by other agents trough mediator. An important part of the memory is a belief about its current plan or plans of others. The goal for agent design

was to be as simple as possible to ease complexity and enable better decomposition. We made communication as part of the environment – trough desire and belief mediator.

Each agent periodically consumes desires from the mediator. Parameters of shared desires are initialized by beliefs of the original agent who propose them. This creates a contract between agents. An agent treats own and shared desires in the same way. The agent commits to desire based on current settings and state of beliefs; it is only reacting to the situation of the environment. Therefore agents are very loosely coupled and simple. The way how agents make proposals to the system is based on **Blackboard** architecture [21]. In this architecture, experts share expertise by stating what may be good for collective to let others do that. Bot Nova [14] also uses this kind of architecture. Using this architecture allows topology of the agents to be dynamic as agents decide which desires to follow and only than hierarchy is introduced.

6.1.2 Planning, decision making, and techniques integration

We design our planning as another way how to do problem decomposition. Our approach is similar to Behaviour Trees or Hierarchical Finite State Machine with the difference that we simplify the structure and add the ability to create structure dynamically. The architecture of our planning algorithm is in figure 6.2. Each agent contains component referred to as „**HeapOfPlanningTrees**“ which has set of trees to represent search space for desires and their realizations. Each planning tree is composed of modules where each instance is desire or intention. When a module has a form of intention, it defines an internal action for an agent to take – to reason about something, execute some action in the environment, share desire with the system or extend current tree by adding additional nodes. On top of that, each agent has its lookup library to choose appropriate intention for each desire it has. Selection of desire is currently based only on key-value lookup where current desire and its parent is the key. Replacing key-value approach by modular one in the future will grant us with the possibility to integrate other techniques to improve search for right behavior in given situation.

Every module is declarable by the user, so there are endless possibilities how to implement new techniques or to further decompose the problem. For deliberation, we use submodules named as

Various visitors periodically visit every tree – two to decide commitment and third to execute executable content of terminal modules. The way how they do that is shown in figure 6.3 which represent an abstract overview of agent routine. User-defined configuration initializes each agent by declaring desires agent may want to pursue, beliefs it works with and lookup library to build up trees. The important part of the process is to keep most current beliefs and desires as they are a vital part of agent's decisions.

6.1.3 Implementation of framework

We implemented our framework as domain independent where agents run in parallel synchronizing only trough mediators. To achieve good performance and robustness communication with mediators is asynchronous. Each mediator keeps a queue of sharing requests which are incorporated to working register. Mediator also maintains a read-only version of register available for agents. This register is periodically updated by working one in a way that it can be accessed concurrently by agents. The downside of this approach is the fact that data contained may not be most recent. However, this is the obvious downside of most distributed systems.

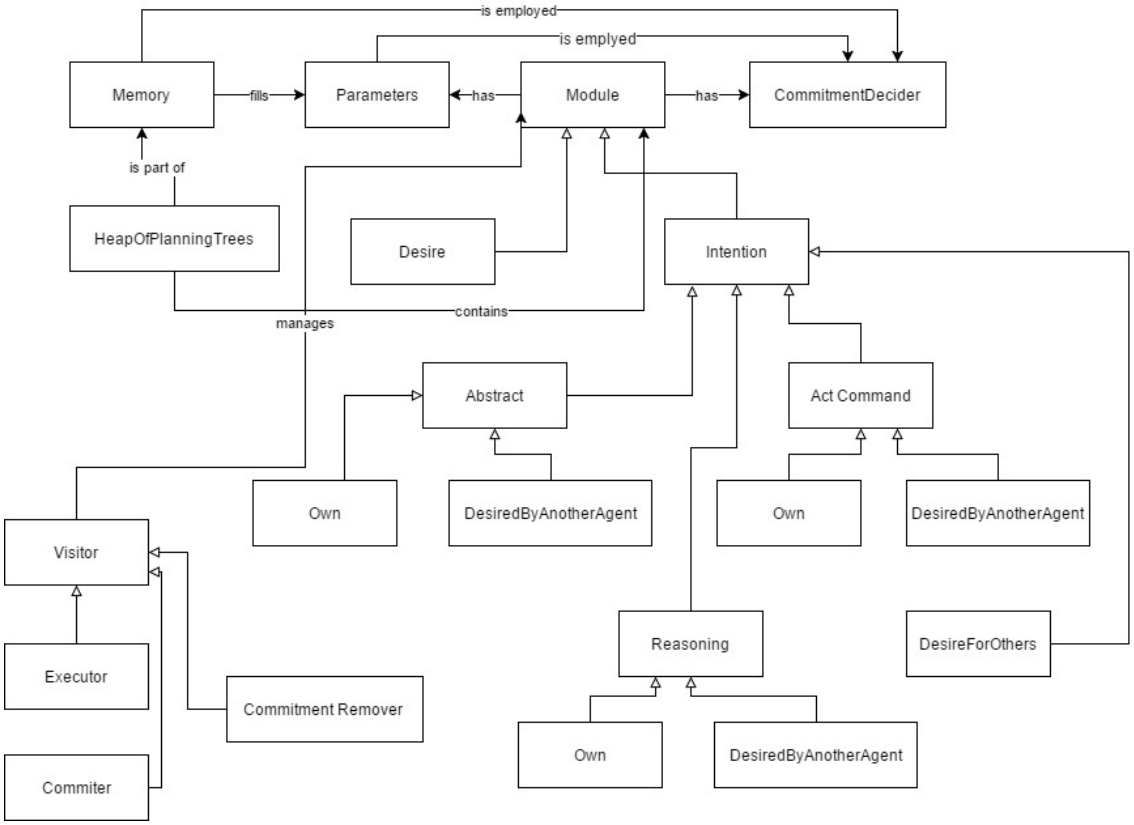


Figure 6.2. A simplified view of components and some of the relations between them for our planning algorithm. Objects with the arrow to Intention are the concrete instantiation of it and contain user-specified code.

As the language of implementation, we choose **Java 8**. Java is portable and strongly typed Object-Oriented programming language. On top of that, it supports generic programming and has some functional features. Together with good dependency management and a lot of available libraries for our use case, it makes an excellent choice. Only one downside for usage of Java is a lack of multiple inheritance support. In figures 6.4 and 6.5 we present our declarations of an agent representing Hatchery and declaration of one module using trained decision module to make a proposal to the system. We give an overview of agents in 7.

6.2 Integrating Learning

To develop bot and integrate Inverse Reinforcement Learning to it, we add three different packages to our bot. Executable packages are Replay Parser and Bot. Both share package Abstract Bot. We present those packages together with our implementation specifics and way how we Integrate Inverse Reinforcement Learning in the following text.

6.2.1 Abstract Bot

As our framework is domain independent, we need to introduced another layer to our bot to be able to play StarCraft. Abstract Bot contains wrappers for StarCraft objects which we are accessing trough **BWMirror**[1] API written in Java. This library maps on

[1] `https://github.com/vjurenka/BWMirror`

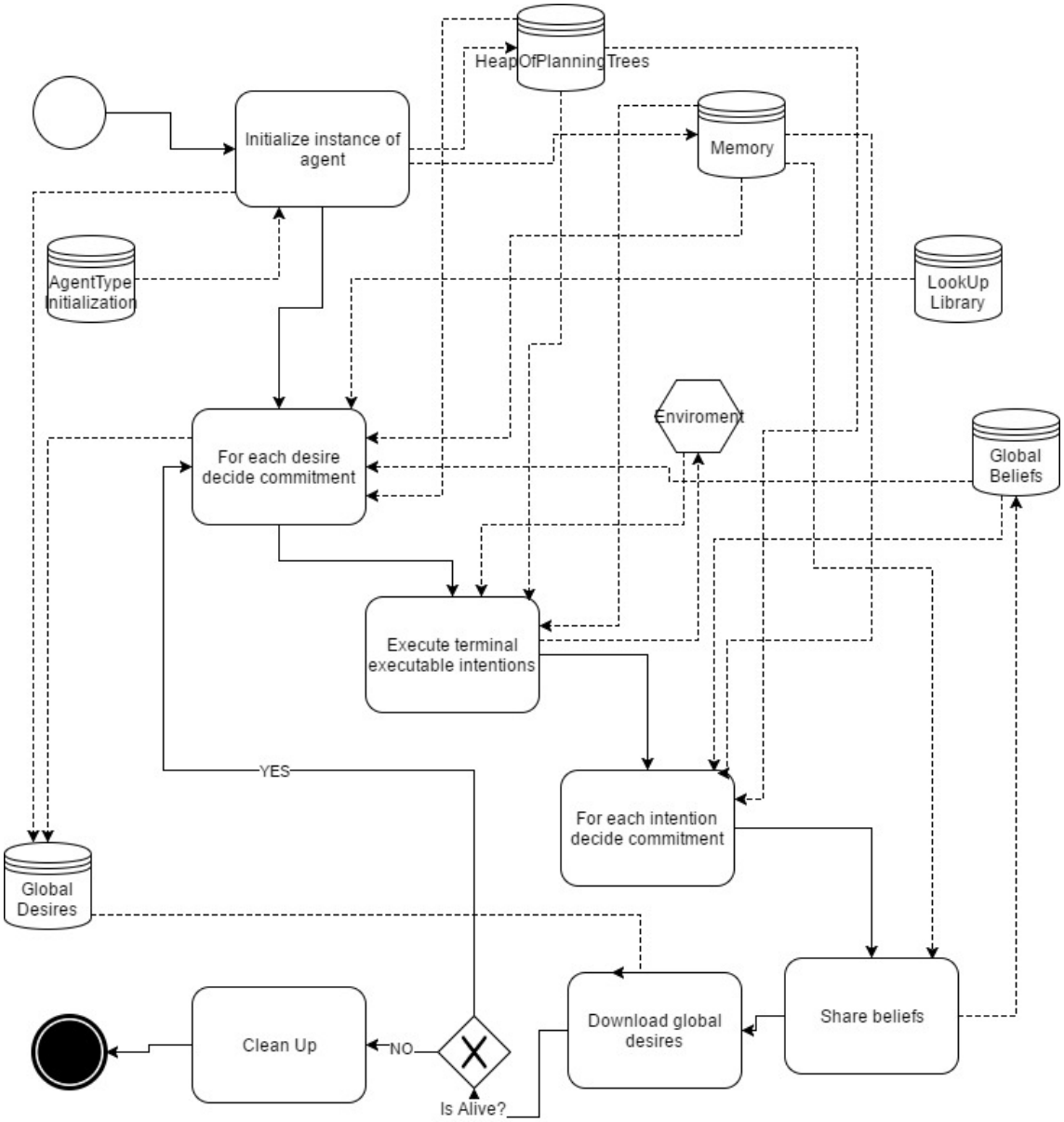


Figure 6.3. A simplified model of Agent's routine representing whole Agent's life cycle in the form of BPMN process. An Agent works with many data sources. It accesses global beliefs and desires managed by mediators. It has its private memory which contains plans in the form of HeapOfPlans. Important stores are the ones with initializations. Initializations are declarative.

BWAPI[1] connected to the game. Due to properties of our framework, we had to write wrappers for most of the objects we use from BWMirror as it does not support concurrent access and does not allow to work with objects outside of the main routine. We accompanied wrappers by the cache to store wrapped instances of game-related objects. There are other performance benefits of it. As agents access game independently – we can serve them cached objects already accessed by different actors.

Another important part of Abstract Bot package are declarations for two remaining packages such as types of agents, their desires, feature container headers to describe

[1] `https://bwapi.github.io/`

```
//BUILDINGS
public static final AgentTypeUnit HATCHERY = AgentTypeUnit.builder()
        .agentTypeID(AgentTypes.HATCHERY)
        .initializationStrategy(type -> {
            type.addConfiguration(UPDATE_BELIEFS_ABOUT_CONSTRUCTION, beliefsAboutConstruction);

            //upgrade to lair
            ConfigurationWithCommand.WithActingCommandDesiredByOtherAgent upgradeToLair = ConfigurationWithCommand.
                    WithActingCommandDesiredByOtherAgent.builder()
                    .commandCreationStrategy(intention -> new ActCommand.DesiredByAnotherAgent(intention) {
                        @Override
                        public boolean act(WorkingMemory memory) {
                            return intention.returnFactValueForGivenKey(IS_UNIT).get().morph(LAIR_TYPE);
                        }
                    })
                    .decisionInDesire(CommitmentDeciderInitializer.builder()
                            .decisionStrategy((dataForDecision, memory) ->
                                    dataForDecision.getFeatureValueGlobalBeliefs(COUNT_OF_MINERALS) >= LAIR_TYPE.getMineralPrice()
                                            && dataForDecision.getFeatureValueGlobalBeliefs(COUNT_OF_GAS) >= LAIR_TYPE.getGasPrice()
                                            //is on start position or there is no base on start position
                                            && (memory.returnFactValueForGivenKey(REPRESENTS_UNIT).get().getNearestBaseLocation().get().isStartLocation()
                                            || memory.getReadOnlyMemoriesForAgentType(AgentTypes.HATCHERY)
                                            .map(readOnlyMemory -> readOnlyMemory.returnFactValueForGivenKey(REPRESENTS_UNIT).get().getNearestBaseLocation().get())
                                            .noneMatch(ABaseLocationWrapper::isStartLocation))
                            )
                            .globalBeliefTypesByAgentType(new HashSet<>(Arrays.asList(COUNT_OF_MINERALS, COUNT_OF_GAS)))
                            .build()
                    )
                    .decisionInIntention(CommitmentDeciderInitializer.builder()
                            .decisionStrategy((dataForDecision, memory) -> false)
                            .build())
                    .build();
            type.addConfiguration(UPGRADE_TO_LAIR, upgradeToLair);
        })
        .usingTypesForFacts(new HashSet<>(Arrays.asList(IS_BEING_CONSTRUCT)))
        .desiresWithIntentionToReason(new HashSet<>(Collections.singletonList(UPDATE_BELIEFS_ABOUT_CONSTRUCTION)))
        .build();
```

Figure 6.4. An example of a declaration of an agent representing Hatchery. It declares the desire to upgrade to Lair. However, ECO Manager needs to make a proposal to the system first for the hatchery to consider commitment.

```
//hold air
ConfigurationWithSharedDesire holdAir = ConfigurationWithSharedDesire.builder()
        .sharedDesireKey(HOLD_AIR)
        .reactionOnChangeStrategy((memory, desireParameters) -> memory.updateFact(TIME_OF_HOLD_COMMAND,
                memory.getReadOnlyMemoriesForAgentType(PLAYER)
                        .map(readOnlyMemory -> readOnlyMemory.returnFactValueForGivenKey(MADE_OBSERVATION_IN_FRAME))
                        .filter(Optional::isPresent)
                        .map(Optional::get)
                        .findAny().orElse( other: null)))
        .decisionInDesire(CommitmentDeciderInitializer.builder()
                .decisionStrategy((dataForDecision, memory) -> memory.returnFactValueForGivenKey(IS_ENEMY_BASE).get()
                        && Decider.getDecision(AgentTypes.BASE_LOCATION, DesireKeys.HOLD_AIR, dataForDecision, HOLDING))
                .globalBeliefTypes(HOLDING.getConvertersForFactsForGlobalBeliefs())
                .globalBeliefSetTypes(HOLDING.getConvertersForFactSetsForGlobalBeliefs())
                .globalBeliefTypesByAgentType(HOLDING.getConvertersForFactsForGlobalBeliefsByAgentType())
                .globalBeliefSetTypesByAgentType(HOLDING.getConvertersForFactSetsForGlobalBeliefsByAgentType())
                .beliefTypes(HOLDING.getConvertersForFacts())
                .beliefSetTypes(HOLDING.getConvertersForFactSets())
                .build())
        .decisionInIntention(CommitmentDeciderInitializer.builder()
                .decisionStrategy((dataForDecision, memory) -> !memory.returnFactValueForGivenKey(IS_ENEMY_BASE).get()
                        || !Decider.getDecision(AgentTypes.BASE_LOCATION, DesireKeys.HOLD_AIR, dataForDecision, HOLDING))
                .globalBeliefTypes(HOLDING.getConvertersForFactsForGlobalBeliefs())
                .globalBeliefSetTypes(HOLDING.getConvertersForFactSetsForGlobalBeliefs())
                .globalBeliefTypesByAgentType(HOLDING.getConvertersForFactsForGlobalBeliefsByAgentType())
                .globalBeliefSetTypesByAgentType(HOLDING.getConvertersForFactSetsForGlobalBeliefsByAgentType())
                .beliefTypes(HOLDING.getConvertersForFacts())
                .beliefSetTypes(HOLDING.getConvertersForFactSets())
                .build())
        .build();
type.addConfiguration(HOLD_AIR, holdAir);
```

Figure 6.5. Declaration of the module for BaseLocation agent using trained decision module to decide if an instance of agent make the proposal to the system to hold the position. Each air unit sees this desire and can decide to follow it.

IRL states, and beliefs. We use common declarations to make sure that data used in both packages check.

6.2.2 Replay Parser

The Replay Parser package contains two executable programs. One is for game observation, and other is to learn decision-making trough Inverse Reinforcement Learning using those observations. The part to observe the replays consists of observers representing agents we would like to learn to make decisions based on observation of players. We reduce decision-making problem on two options. Was agent committed to desire or not. Each observer has predefined set of desires to track. It tracks values of features defined in „“ to describe the state, and decisions made in the form of trajectory. After each replay, trajectories are saved.

To learn decision making, we use the second program. For each agent and type of desire, it loads saved trajectories. To be able to use IRL we have to reduce the number of states from trajectories. As a baseline for state compression, we use **K-Means clustering with** Euclidian distance, **Z-Score** normalization and the user-defined number of clusters. For clustering, we employ **JSAT library**[1] which provides an implementation of **Minibatch K-Means** [22]. Our approach is currently very biased and far from optimal but computationally bearable in our setting. Using states and trajectories, we then can create MDP with additional dummy state. We use this state as a destination for transitions which did not take place in replays. During the run of IRL algorithm, we keep reward for this state as small as possible to learn a policy which will not use unknown transitions. For IRL we extended algorithms and MDP implementations of **BURLAP library**[2] to match our use-case. The learned policy is then saved as decision module which is then loaded by bot.

6.2.3 Bot

The figure 6.6 represents high-level architecture overview of our bot and shows the integration of components. It contains package Bot with the concrete implementation of agents – units, abstract agents and even agents for places. We present additional implementation details on our agents in next chapter.

An important part of this package is „**BotFacade**“ implementing method for various events called by the game. This way bot can communicate with the game by issuing commands or reading data. To do that we introduce „**GameCommandExecutor**“ which manages requests of individual agents on game trough queue. It has a time window for handling requests on game call. It also keeps execution time of the request type to plan ahead as it makes sure that time will not be exceeded.

[1] `https://github.com/EdwardRaff/JSAT`
[2] `http://burlap.cs.brown.edu/index.html`

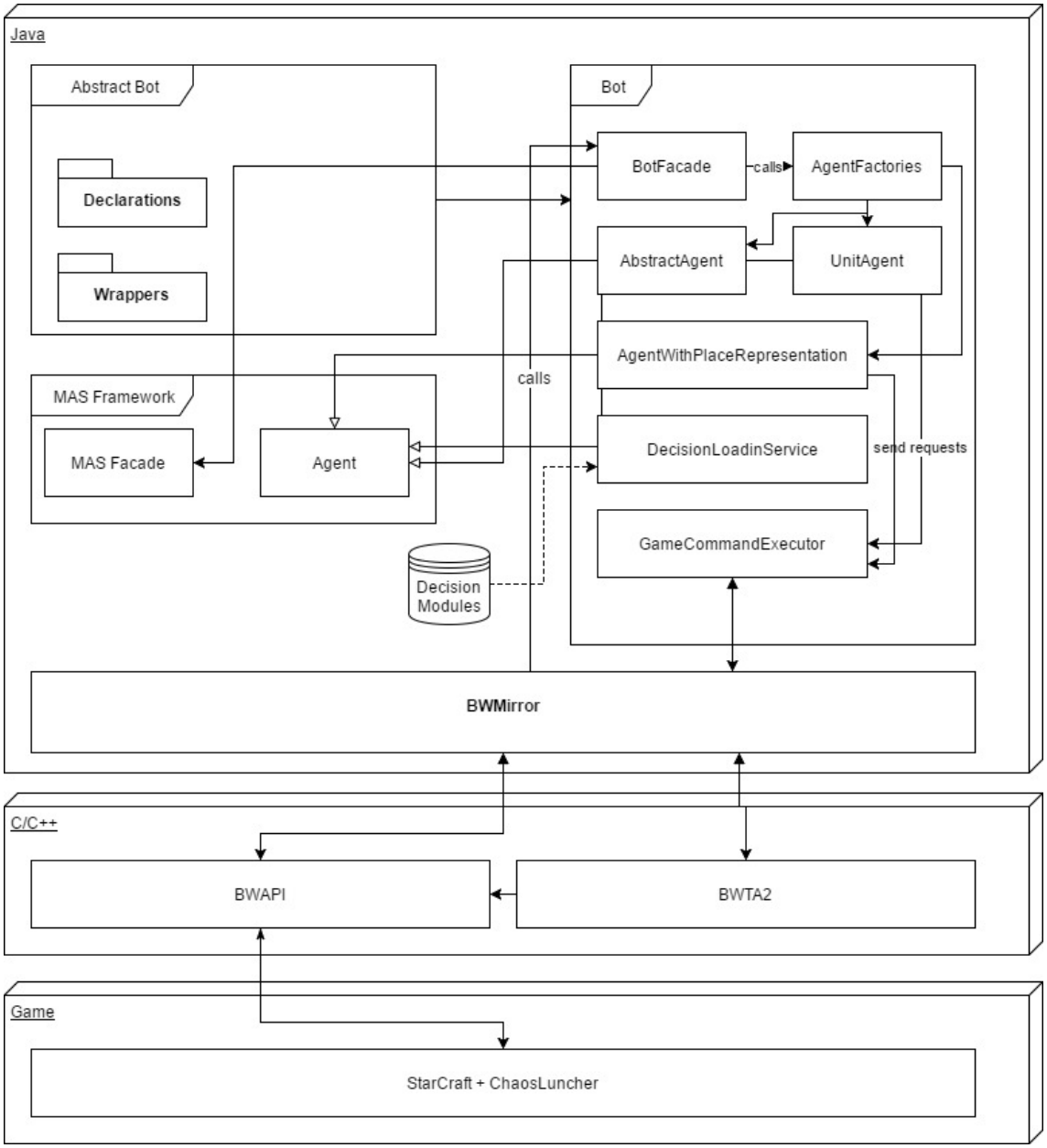


Figure 6.6. Overview of bot's architecture. There are three different layers. We implement top layers (except for BWMirror). The bot is based on our framework. The top layer contains domain-specific packages and abstract declarations of our bot. Declarations are shared with Replay Parser.

Chapter 7
Presentation of the bot and its Experimental Evaluation

In the previous chapter, we gave an overview of the architecture of our bot with description how we integrated Inverse Reinforcement Learning to it. In this chapter, we present examples of our implementation with an overview of agents, and their desires. We also discuss the behavior of our bot and results against built-in AI. As our bot is still more prototype than any serious threat to current the state of the art competition bots, we analyze and explain its present performance its current performance to help us in future development.

7.1 Realization of bot

We decided for our bot to play Zerg. In figure 7.1 we present high-level overview of the implementation of agents with their desires and relations. Agents representing units and agents to handle the economic aspect of the game or build orders are standard even in other MAS systems such as those defined in [14] or [21]. We introduce a new type of agent representing the location where a base can be built. In our opinion using this kind of agent could be better than the central solution when coordinating multiple attacks.

Each base location has the same set of desires as others. Those are desires such as build static anti-ground or anti-air defense, or to send ground or air units to hold this base. A base decides on a commitment to those desires using decision modules learned trough IRL. When a base makes a commitment, it shares desire with system together with its location as a parameter. This desire then propagates to units' „HeapOfPlans“ to let them decide if they want to realize the plan they have for this kind of situation. For example, realization in the case of worker and desire to build static ground defense has a form of abstract plan. A worker commits to it when it thinks it is nearest to this location. Then it starts to execute individual steps of the abstract plan by moving to the site, selecting a suitable place for building and finally building it. In our implementation, each worker operates with some time limit to meet this intention. After that, it may stop pursuing it and let others try it. To describe states for a base location with our base on it, we use features related to units (buildings, army – for the enemy and us) on the site, and economic value (workers mining resources). On top of that, everything is compared to global values by introducing another set of features. For the enemy base, we use a similar set of features.

Using MAS has other benefits besides decomposition. For example, workers can easily cooperate on gathering resources by making reservations of the resources which are currently gathered. Other workers can use this knowledge to decide on mining other resources.

Trough IRL we are learning decisions for the managers and the base locations. ECO Manager uses learned decision modules to decide when to expand, build another worker,

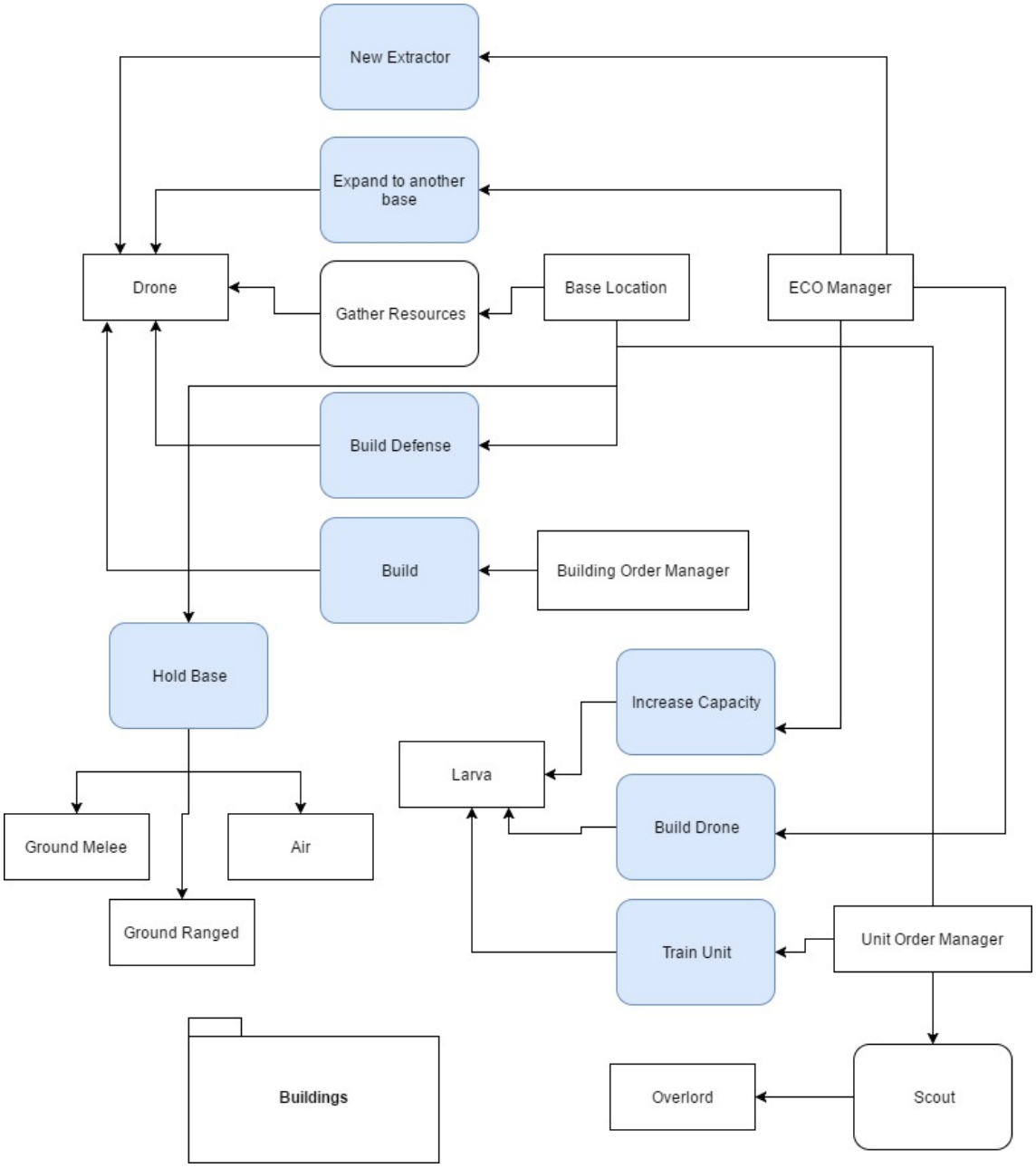


Figure 7.1. A high-level overview of the implementation of agents with their desires and relations. Blue ones have decision modules trained trough IRL by observing gameplays.

increase population capacity or build another gas extractor. So far, we have added support only for three military type of units – melee and ranged ground, and air. Evan though, we omit research and other unit types, it should provide bot with some level of flexibility to start with. Decisions, when to build any unit or infrastructures, are made by Unit Order Manager or Building Order Manager respectively. Those are trained from replay observation using IRL. Same goes for decisions when to attack and when to build any defense of each base location. Rest of the stuff is currently hard-coded.

To learn decision-making trough observation, we composed dataset of roughly 500 replays. We download those replays from forum thread[1] where the users upload hand

[1] `http://www.teamliquid.net/forum/brood-war/310883-replays`

curated packages of interesting plays. There are many large datasets available[1]. However, we currently restricted ourselves to hand curated replays as we did not have the capacity to parse larger datasets.

7.2 Analysis of bot's behavior and its performance

To show our bots current capabilities, we put videos[2] of our bot beating default Zerg AI on 1v1 maps. The base was able to build is in figure 7.2. It was able to transit to the mid-game meeting are requirements for training air units.

Figure 7.2. Demonstration of our bot capabilities.

In our setting, we experiment with a dataset of various sizes; we let our bot to see some replays ranging from 250 to 500. For each dataset, we use all replays for the definition of MDPs, and at most 35 replays for learning reward function using IRL. On one machine employing more replays is very time-consuming. To speed things up and complexity restriction, we limit our MDPs to 2000 states. Sadly, 2000 as K for clustering is in most cases not sufficient enough to get the best partitioning as we discovered when we did some data exploration. Results of one training session are in table 7.1.

After each training session, we tested our bot against built-in AIs on different competitive maps varying in size. It is fascinating to observe that the bot does different

[1] http://www.starcraftai.com/wiki/StarCraft_Brood_War_Data_Mining
[2] https://youtu.be/hjL2Srf08pI and https://youtu.be/Y-CMgxLju04

Decision in desire	Committed	States
ENABLE GROUND MELEE	112	2000
UPGRADE TO LAIR	89	2000
ENABLE AIR	141	2000
ENABLE GROUND RANGED	101	2000
ENABLE STATIC ANTI AIR	250	2000
BOOST GROUND MELEE	710	2000
BOOST GROUND RANGED	616	2000
BOOST AIR	438	1999
BUILD CREEP COLONY	183	1000
HOLD GROUND	477	750
BUILD SPORE COLONY	52	1000
HOLD AIR	304	750
BUILD SUNKEN COLONY	172	1000
BUILD WORKER	818	2000
BUILD EXTRACTOR	500	1999
INCREASE CAPACITY	524	2000
EXPAND	946	2000

Table 7.1. Results of decision-making training. For each dataset, we use all replays for the definition of MDPs, and at most 35 replays for learning reward function using IRL.

build orders against various opponents. In many cases, the build order is also influenced by map size. Transitions are also different and may depend on the situation. The acting is in many cases amusing. However, using the right set of default replays for learning and the whole set of replays for MDPs, bot learned a strategy able to beat zerg opponent in 1on1 maps in few scenarios. When comparing two provided videos illustrating this strategy transitions are different – in one setting bot build mid-game infrastructure in other it does the massive expansion.

By testing our bot, we analyzed following issues which in our opinion undermine the performance of our bot most:

- Bot builds basic infrastructure at the beginning but sometimes is idle until it has another vision on the enemy. To reduce the number of states, we eliminated time from our features. We based them on the current situation only. Together with insufficient scouting micro, it presents a big problem for bot ability to survive. Slow transition in most cases leads to certain death.
- The composed dataset may be very opinionated and too small. With a narrow set of situations as most of it are games from professional players. Some of the features may not describe the problem well. Mentioned time is one of the most prominent examples.
- Our bot is currently lacking any serious micro. We also omit many capabilities of the full-scale player. Many capabilities are interrelated, and their absence can degrade other ones.
- There are many parameters settings we have not optimized well yet due to the state of the development.
- Learning single decision-making module for all races and map sizes shows its limitations. The number of examples limits us when training decision modules. Situations are also very different.

Despite the fact that we identified many problems lowering performance of our bot. The results show great promise as it seems that bot could learn to play like a beginner. We are optimistic about the possibility of our bot attending competitions in future.

Chapter 8
Conclusion

In this work, we develop a prototype of bot for StarCraft: Brood War which learns its decision-making processes from demonstrations. We decompose the problem of playing RTS game using our framework based on Multi-agent system to integrate Inverse Reinforcement Learning with other techniques. Using the right set of replays to learn decision-making trough game-play observation our bot was able to learn strategy which can beat built-in AI for Zerg in some 1on1 map scenarios as we show in the last chapter. The bot also shows the ability to adapt to the situation.

8.1 Discussion and future work

With the current state of bot it seems that we are still not at the end of the road to introduce game AI which could be a severe threat to the present state of the art bot and maybe even for humans. However, results present great promises. The latest version of bot shows some level of adaptive behavior and is even able to learn some winning strategy. However, there are still challenges to be addressed before bot developers can take full advantage of IRL technique or presented framework. In the case of our framework, we see great promise in the implementation of following features:

- It would be good to have some local belief mediator for shared desires between the contractor and committed agents to the abstract developer from accessing global beliefs to find information which can be shared locally.
- Another functionality increasing the level of flexibility for plan definition would be the ability to use different techniques to select appropriate implementation of the module.

For Inverse Reinforcement Learning usage in the domain of RTS, we see following directions for future work:

- Exploring usage of infinite states MDPs.
- Combining IRL with other algorithms to adjust policy and reward function.
- Apply this technique to train a bot to play a different game (or at least different RTS).

We are also interested in the possibility of using symbolic regression in combination with our framework to model the system. In our case, a model of bot is provided by the user which may introduce bias and can limit the performance of the system as some of the essential domain knowledge may not be know by a designer. An instance of symbolic regression could eliminate those issues by finding a structure which can fit dataset by composing system from blocks of behaviors.

References

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George vanden Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016-1-27, vol. 529 (issue 7587), 484-489. DOI 10.1038/nature16961.

[2] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. DeepStack. *Science*. eaam6960-. DOI 10.1126/science.aam6960.

[3] David Churchill, Mike Preuss, Florian Richoux, Gabriel Synnaeve, Alberto Uriarte, Santiago Ontannón, and Michal Čertický. StarCraft Bots and Competitions. *Encyclopedia of Computer Graphics and Games*. 2016, 1. DOI 10.1007/978-3-319-08234-9$_1$8 – 1.

[4] Ben G. Weber. *Integrating learning in a multi-scale agent*. 2012 edition. Santa Cruz: University of California, 2012. ISBN 14-776-1473-7. `http://alumni.soe.ucsc.edu/~bweber/bweber-dissertation.pdf`.

[5] Santiago Ontanon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*. 2013, vol. 5 (issue 4), 293-311. DOI 10.1109/TCIAIG.2013.2286295.

[6] *DeepMind's AlphaGo is secretly beating human players online*. `https://www.newscientist.com/article/2117067-deepminds-alphago-is-secretly-beating-human-players-online/`.

[7] David Churchill. *Heuristic Search Techniques for Real-Time Strategy Games*. 2016.

[8] Giovanni Viglietta. Gaming Is a Hard Job, but Someone Has to Do It!. *Theory of Computing Systems*. 2014, vol. 54 (issue 4), 595-621. DOI 10.1007/s00224-013-9497-5.

[9] *A Starcraft AI seems incredibly difficult*. 2013. `https://day9.tv/d/jbigg2012/a-starcraft-ai-seems-incredibly-difficult/`.

[10] *StarCraft: Brood War Prize Pool & Top Players*. `http://www.esportsearnings.com/games/152-starcraft-brood-war`.

[11] *Understanding Behavior Trees*. 2007. `http://aigamedev.com/open/article/bt-overview/`.

[12] Martin Certicky, and Michal Certicky. *Case-Based Reasoning for Army Compositions in Real-Time Strategy Games*. In: *Scientific Conference of Young Researchers 2013*. 2013. 70–73.

[13] Gabriel Synnaeve, and Pierre Bessiere. A Dataset for StarCraft AI & an Example of Armies Clustering. *Artificial Intelligence in Adversarial Real-Time Games 2012*. 2012, 7.

[14] Alberto Uriarte Pérez. *Multi-Reactive Planning for Real-Time Strategy Games*. 2011.

[15] Michal Certicky. Implementing a wall-in building placement in starcraft with declarative programming. *arXiv preprint arXiv:1306.4460*. 2013,

[16] David Fiedler. *Použití metod multiagantních systémů pro implementaci umělé inteligence v real-time strategiích*. 2016.

[17] P. Abbeel, A. Coates, and A. Y. Ng. *Autonomous Helicopter Aerobatics through Apprenticeship Learning.* In: *The International Journal of Robotics Research*. 2010-11-05. 1608-1639.
`http://ijr.sagepub.com/cgi/doi/10.1177/0278364910371999`.

[18] Pieter Abbeel, and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. *Proceedings of the twenty-first international conference on Machine learning. ACM*. 2004,

[19] *Apprenticeship learning using Inverse Reinforcement Learning*. 2016.
`https://jangirrishabh.github.io/2016/07/09/virtual-car-IRL/`.

[20] Stuart J. Russell, Peter. Norvig, and Ernest. Davis. *Artificial intelligence*. 3rd ed. edition. Upper Saddle River: Prentice Hall, c2010. ISBN 978-0136042594.

[21] Gerhard Weiss. *Multiagent systems*. Reprint. edition. Cambridge, Massachusetts: The MIT Press, 2001. ISBN 978-026-2731-317.

[22] D. Sculley. Web-scale k-means clustering. *Proceedings of the 19th international conference on World wide web - WWW '10*. 2010, 1177-. DOI 10.1145/1772690.1772862.