BEHAVIOUR DESIGN IN MICROROBOTS: HIERARCHICAL REINFORCEMENT LEARNING UNDER RESOURCE CONSTRAINTS

THÈSE N^{O} 3682 (2006)

PRÉSENTÉE LE 26 JANVIER 2007

À LA FACULTÉ DES SCIENCES ET TECHNIQUES DE L'INGÉNIEUR

Laboratoire de systèmes autonomes 1

SECTION DE MICROTECHNIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Masoud ASADPOUR

ingénieur en informatique diplômé de l'Université de Teheran, Iran
de nationalité iranienne

acceptée sur proposition du jury:

Dr P. Hoffmann, président du jury
Prof. R. Siegwart, directeur de thèse
Prof. A. Martinoli, rapporteur
Prof. M. Nili Ahmadabadi, rapporteur
Prof. H. Ritter, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2006

Behavior Design in Microrobots: Hierarchical Reinforcement Learning under Resource Constraints

Masoud Asadpour

26 October 2006

Acknowledgement [1]

During my time as a graduate student at EPFL, I have had the privilege to meet and interact with many intelligent and inspiring people. This journey would have been much harder and a lot less fun without them.

First of all, I would like to express my gratitude to my advisor, Professor Roland Siegwart. I would like to thank him for accepting me to Autonomous Systems Laboratory, for giving me the opportunity to do robotics research, and for making the lab such a great place to work. He has been my example for management of a research lab.

Furthermore, I would like to thank Professor Majid Nili Ahmadabadi. Somehow, he has always found time to discuss whatever was on my mind, from mathematical details to higher level research and career choices, specially during his sabbatical research in our lab. I would like also to thank committee members, Profs. Alcherio Martinoli, Helge Ritter, and Patrik Hoffmann for their valuable comments.

I would like to thank our team members, Gilles Caprari, Fabien Tâche (also my officemate), Francesco Mondada, and Walter Karlen. We worked and traveled many times together for the LEURRE project. Gilles and Fabien helped me very much in programming Alice and InsBots, developing tools for them, and realizing the important parts of our contribution to the LEURRE project. I thank them for the wonderful time we spent together.

The faculty members, students, and staff at the lab all contributed in making the working atmosphere entertaining and comfortable. I would like specially to thank Marie-Jose Pellaud (our secretary) for arranging every thing from my arrival until the end; Daniel Burnier (technical assistant) for being very much help, especially providing me some extra computers to run my extensive simulation experiments, and helping me in learning French; Samir Bouabdallah for his kind helps during my stay at Switzerland; and Ahad Harati for helping me in everything, from the theory and scientific discussions to running some parts of the simulations.

I would, also, like to thank Professor Jean-Louis Deneubourg and his team, for scientific discussions and workshops, and for receiving our team in Brussels. I would, also, like, to

---

[1] This project [1] was funded by the Future and Emerging Technologies programme (IST-FET) of the European Community, under grant IST-2001-35506. The information provided is the sole responsibility of the author and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication. The Swiss participants to the project were supported under grant 01.0573 by the Swiss Government.

thank Christian Jost and Simon Garnier for their scientific discussions.

I would like to thank my friend Bijan Najafi for providing me a cluster of computers to run an essential part of my simulation. I would like to thank my friend and actually my big brother, Sohrab Amiri, for his encouragement and supports.

Let me also thank the around 60 computers! that executed my extensive simulation experiments, sometimes day and night, for short or long periods; without their help accomplishing two last chapters of this thesis would not have been possible.

I would like most of all to thank my wife, my mother, my father in heaven, and my brothers and sisters for their constant encouragement and support in my life. My wife Elaheh! you have been supporting me and indirectly contributing to every moments of my PhD. I dedicate this thesis to you. I love you!

Masoud Asadpour
Lausanne, Oct. 2006

Abstract

In order to verify models of collective behaviors of animals, robots could be manipulated to implement the model and interact with real animals in a mixed-society. This thesis describes design of the behavioral hierarchy of a miniature robot, that is able to interact with cockroaches, and participates in their collective decision makings. The robots are controlled via a hierarchical behavior-based controller in which, more complex behaviors are built by combining simpler behaviors through fusion and arbitration mechanisms. The experiments in the mixed-society confirms the similarity between the collective patterns of the mixed-society and those of the real society. Moreover, the robots are able to induce new collective patterns by modulation of some behavioral parameters.

Difficulties in the manual extraction of the behavioral hierarchy and inability to revise it, direct us to benefit from machine learning techniques, in order to devise the composition hierarchy and coordination in an automated way.

We derive a Compact Q-Learning method for micro-robots with processing and memory constraints, and try to learn behavior coordination through it. The behavior composition part is still done manually. However, the problem of the curse of dimensionality makes incorporation of this kind of flat-learning techniques unsuitable. Even though optimizing them could temporarily speed up the learning process and widen their range of applications, their scalability to real world applications remains under question.

In the next steps, we apply hierarchical learning techniques to automate both behavior coordination and composition parts. In some situations, many features of the state space might be irrelevant to what the robot currently learns. Abstracting these features and discovering the hierarchy among them can help the robot learn the behavioral hierarchy faster. We formalize the automatic state abstraction problem with different heuristics, and derive three new splitting criteria that adapt decision tree learning techniques to state abstraction. Proof of performance is supported by strong evidences from simulation results in deterministic and non-deterministic environments. Simulation results show encouraging enhancements in the required number of learning trials, robot's performance, size of the learned abstraction trees, and computation time of the algorithms.

In the other hand, learning in a group provides free sources of knowledge that, if communicated, can broaden the scales of learning, both temporally and spatially. We present two approaches to combine output or structure of abstraction trees. The trees are stored in different RL robots in a multi-robot system, or in the trees learned by the same robot but using different methods. Simulation results in a non-deterministic football learning task provide strong evidences for enhancement in convergence rate and policy performance, specially in heterogeneous cooperations.

Key words: Behavior Combination, Micro-Robots, Mixed-Society, Robot-Animal Interaction, Q-Learning, Hierarchical Reinforcement Learning, State Abstraction, Splitting Criteria, Cooperative Reinforcement Learning, Decision Tree, Merge, Balance, Prune

Résumé

Afin de vérifier les modèles des comportements collectifs de certains animaux, des robots ont été programmés pour implémenter les modèles et les confronter avec de vrais animaux dans une société-mixte. Cette thèse décrit la conception de la hiérarchie comportementale d'un robot miniature, capable d'interagir avec des blattes et participer à leurs décisions collectives. Les robots sont contrôlés par un contrôleur hiérarchique comportemental dans lequel, des comportements complexes sont établis en combinant des comportements plus simples par des mécanismes de fusion et d'arbitrage. Les expériences dans la société-mixte confirment la similitude entre les modèles collectifs de la société-mixte et ceux de la vraie société animale. D'ailleurs, les robots peuvent induire de nouveaux comportements collectifs par la modulation de quelques paramètres comportementaux.

Les difficultés dans l'établissement manuel de la hiérarchie comportementale et l'impossibilité de l'améliorer, nous ont poussés à tirer avantage des techniques d'apprentissage, afin de concevoir la hiérarchie des compartements et leurs coordination d'une manière automatisée. Nous avons conçu une méthode de Q-Learning compacte pour des micro-robots en tenant compte de contraintes de mémoire et de puissance de calcul, et nous avons essayé d'apprendre le coordination des comportements par cette méthode. La composition du comportement est encore faite manuellement. Cependant, le gros problème de la dimensionnalité rend l'introduction de ce genre d'apprentissage peu convenable. Malgré le fait que l'optimisation de ces méthodes pourrait temporairement accélérer l'apprentissage et élargir leurs possibilités d'applications, leur scalability et l'application réelle reste une question.

Dans les prochaines étapes, nous appliquons des techniques d'apprentissage hiérarchiques pour automatiser la composition et le coordination parmi des comportements. Dans certaines situations, beaucoup de caractéristiques de l'espace des états pourraient être non pertinentes pour l'apprentissage du robot. Faire abstraction de ces caractéristiques et en découvrir la hiérarchie peuvent aider le robot à apprendre la hiérarchie comportementale plus rapidement. Nous formalisons le problème d'abstraction automatique d'états avec une heuristique différente, et décrivons trois nouveaux critères de séparation qui permettent d'adapter les techniques d'apprentissage par arbre de décision pour l'abstraction d'état. La performance de ces techniques est prouvée par des résultats de simulations dans les environnements déterministes et non déterministes. Les résultats des simulations montrent des améliorations encourageantes du nombre d'étapes d'apprentissage requises, de la performance du robot, la taille des arbres d'abstraction appris, et le temps de calcul des algorithmes.

Cependant, l'apprentissage dans un groupe fournit des sources de connaissances qui, si communiquées, peuvent élargir l'échelle de l'apprentissage, temporellement et dans l'espace. Nous présentons deux techniques pour combiner le résultat ou la structure des arbres d'abstraction. Les arbres sont stockés dans différents robots s'ils font partie d'un système multi-robot, ou dans les arbres appris par le même robot mais en employant différentes méthodes. Les résultats de simulations pour un apprentissage non déterministe du jeu du football fournissent des preuves évidentes de l'amélioration du taux de convergence et des performances de la tactique, particulièrement pour de la coopération hétérogène.

Mots clés: Combinaison de comportement, Micro-robots, Société-mixte, Interaction de robots et animaux, Q-Learning, Apprentissage par renfort hiérarchique, abstraction d'état, critères de séparation, Apprentissage par renfort coopératif, arbre de décision, fuser, équilibrer, tailler

Contents

Table of Contents **x**

List of Tables **xi**

List of Figures **xiv**

List of Algorithms **xv**

List of Symbols and Abbreviations **xvii**

1 Introduction **1**
- 1.1 Behavior Coordination 1
  - 1.1.1 Fusion 1
    - Schema 2
  - 1.1.2 Arbitration 2
    - Priority-based Arbitration 3
    - State-based Arbitration 3
- 1.2 Hierarchical Reinforcement Learning 4
  - 1.2.1 Temporal Abstraction 4
  - 1.2.2 State Abstraction 5
- 1.3 Project Goal 6
- 1.4 Organization of the Thesis 7
- 1.5 Publications 7
- 1.6 Contribution of the Author 9

I Manual Design of the Behavioral Hierarchy **11**

2 Mixed-Society of Robots and Animals **13**
- 2.1 LEURRE Project 13
  - 2.1.1 Collective Decision-Making 14
  - 2.1.2 Mixed-Society Control 14
  - 2.1.3 Previous Works 15
- 2.2 Biological Experiments 15

2.2.1 Why Aggregation? 15
2.2.2 Why Cockroach? 16
Robots Inspired by Cockroach 16
Cockroach Aggregation 16
2.2.3 Aggregation Model 16
2.2.4 Mixed Aggregation 17
2.2.5 Experimental Setup 18
2.2.6 Functionality of Robots 19
2.3 InsBot: The Insect-like Robot 19
2.3.1 Sensors and Communication Tools 20
2.3.2 Control and Electronic Architecture 20
2.3.3 Calibration 21
2.4 Behavioral Architecture 21
2.4.1 Perceptual Schema 21
Shelter Detection 22
Cockroach Detection 22
Wall Detection 23
Robot Detection 24
2.4.2 Motor Schema 25
Behavior Fusion 26
Behavior Arbitration 28
2.5 Results 29
2.5.1 Performance of the Perceptual Schema 30
Cockroach and Wall Detection 30
Robot Detection 31
2.5.2 Performance of the Motor Schema 32
Collective Robotics Experiments 32
Mixed-Society Experiments 33
2.6 Conclusion and Future Works 34

II Flat Learning **37**

3 Compact Q-Learning **39**
3.1 Introduction 39
3.2 Related Works 40
3.2.1 Reinforcement Learning 41
3.3 The Compact Q-Learning Algorithm 42
3.3.1 Integer vs. floating point operators 42
3.3.2 Q-Learning problems with Integer operators 43
3.3.3 The proposed algorithm 44
3.4 The Alice Micro-Robot 45
3.5 Learning Light-Attraction Behavior 47

3.6 Learning Safe-Wandering Behavior . . . 48
3.7 Integer vs. Floating-point . . . 51
    3.7.1 Learning measures . . . 51
    3.7.2 Implementation measures . . . 52
3.8 Learning Light-Search Behavior . . . 53
3.9 Conclusion and future works . . . 54

III Hierarchical Learning 55

4 Hierarchical Reinforcement Learning 57
4.1 Introduction . . . 57
4.2 Related Works . . . 59
4.3 Formalism . . . 60
4.4 U-Tree . . . 60
    4.4.1 Splitting Criteria . . . 61
    4.4.2 Computation Time of U-Tree . . . 62
    4.4.3 Biased Sampling Problem . . . 62
4.5 New Splitting Criteria . . . 62
    4.5.1 SANDS . . . 63
        Optimality of Partitions . . . 64
        Computation Time of SANDS . . . 64
        Mont Carlo Methods for SANDS . . . 64
    4.5.2 SMGR . . . 65
        Penalty Term . . . 66
    4.5.3 VAR . . . 68
4.6 Simulation results . . . 70
    4.6.1 A Sample Tree . . . 71
    4.6.2 Unbiased Sampling . . . 72
    4.6.3 Policy Performance and Tree Size . . . 73
    4.6.4 Computation Time . . . 74
4.7 Conclusion . . . 75

5 Hierarchical Cooperative Learning 77
5.1 Introduction . . . 77
5.2 Related Works . . . 78
    5.2.1 Cooperative Reinforcement Learning . . . 78
    5.2.2 Hierarchical Reinforcement Learning . . . 79
    5.2.3 Combining Decision Trees . . . 80
5.3 Combining Abstraction Trees . . . 80
    5.3.1 Ensemble Learning . . . 80
    5.3.2 CLASS . . . 81
        Merge . . . 81

Balance . . . 84
Pruning . . . 87
5.4 Simulation Results . . . 88
5.4.1 Ensemble Learning Results . . . 89
Tree Size . . . 89
Action . . . 89
Reward . . . 92
Score Ratio . . . 92
Discussion . . . 92
5.4.2 CLASS Results . . . 95
Policy Performance . . . 95
Merge . . . 96
Pruning . . . 96
Processing Time . . . 96
Discussion . . . 99
5.5 Conclusion . . . 99

6 Conclusion and Future Works 101
6.1 Manual Hierarchy Design . . . 101
6.2 Compact Q-Learning . . . 102
6.3 Hierarchical Reinforcement Learning . . . 102
6.3.1 Splitting Criteria . . . 103
6.3.2 Cooperative Learning . . . 103

A Splitting Criteria Results 105
A.1 Unbiased Sampling . . . 105
A.2 Policy Performance and Tree Size . . . 105

Bibliography 118

Curriculum Vitae 119

List of Tables

2.1 Probability to leave the shelters 33

3.1 Comparing the floating-point with the integer operations on Alice micro-robots 42
3.2 Comparing the percentage of memory and program size 52

4.1 Comparing the biased and unbiased U-Tree construction methods in terms of the number of actions per episodes during the test phase 72
4.2 Comparing the biased and unbiased U-Tree construction methods in terms of the number of leaves in the learned trees 72
4.3 Comparing the learning methods in terms of the average number of actions per episode during the test phase 73
4.4 Comparing the learning methods in terms of the average number of leaves in the learned trees 73

5.1 Results of the individual and ensemble learning 88
5.2 Performance of the CLASS algorithm 95

A.1 Comparing the biased/unbiased U-Tree construction methods in terms of the number of actions per episodes during the test phase (best confidence threshold) 105
A.2 Comparing the biased/unbiased U-Tree construction methods in terms of the number of leaves in the learned tree (best confidence threshold) 106
A.3 Comparing the learning methods in terms of the average number of actions per episode during the test phase (best confidence threshold) 106
A.4 Comparing the learning methods in terms of the average number of leaves in the learned trees 106

List of Figures

1.1 Fusion by superposition 2
1.2 Subsumption architecture 3

2.1 Experimental setup for mixed-society experiments 18
2.2 Mixed-society of InsBots and cockroaches 19
2.3 Control and electronic architecture of InsBot 20
2.4 Cockroach detection by InsBot 22
2.5 Cockroach detection near walls 23
2.6 Wall detection by InsBot 24
2.7 Detection of other InsBots 25
2.8 Overview of the behavioral architecture 26
2.9 Converting resultant force to speed vector 28
2.10 State-machine of mixed-aggregation 29
2.11 Performance of perception methods for InsBot 30
2.12 Sensory coverage of InsBot 31
2.13 Performance of local communication between InsBots 32
2.14 Collective choice between homogeneous shelters 33
2.15 Collective choice between heterogeneous shelters 34
2.16 Modification of collective choice induced by robots 35

3.1 Simplification of Boltzmann Selection to Roulette-Wheel Selection 44
3.2 The Alice micro-robot 46
3.3 Light attraction task 47
3.4 History of the received rewards during every 15 s of light-attraction learning 48
3.5 A sample of the learned safe-wandering behavior in the cross-maze 49
3.6 History of the received rewards during safe-wandering in the cross-maze . . 49
3.7 A sample of the learned safe-wandering behavior in the H-maze 50
3.8 History of the received rewards during safe-wandering in the H-maze 50
3.9 Averaged received rewards per minute during learning and test phases . . . 52
3.10 History of the received rewards during every two minutes of light-search learning in the Cross-maze 53

4.1 state abstraction concept 58
4.2 The penalty term when $\hat{\rho}_1 = 0.01$, $\hat{\rho}_2 = 0.99$ 67

4.3 The penalty term when $\hat{\rho}_1 = 0.99$, $\hat{\rho}_2 = 0.01$ 67
4.4 The penalty term when $\hat{\rho}_1 = \hat{\rho}_2 = 0.5$ 68
4.5 A screen-shot from the simulator 70
4.6 A typical decision tree, learned in static-player case 72
4.7 Computation time of the algorithms 74

5.1 Ensemble of abstraction trees jointly selects the next action 81
5.2 An example for merging abstraction trees 82
5.3 An example for efficiency of balance and pruning algorithms 84
5.4 Balanced version of T_3 in Fig. 5.2 in terms of depth 87
5.5 history of leaves/trees for individual and ensemble learning 90
5.6 History of actions/episodes for individual and ensemble learning 91
5.7 History of sum of rewards/episodes for individual and ensemble learning . . 93
5.8 The score ratio for the individual and the ensemble learning 94
5.9 Relation between the size of the input and output trees of the merge algorithm 97
5.10 Performance of Pruning. Top: Homogeneous, Bottom: Heterogeneous . . . 98
5.11 Processing time of the CLASS algorithm 99

A.1 History of actions/episodes during learning (best confidence threshold) . . 107
A.2 History of leaves/trees during learning (best confidence threshold) 108

List of Algorithms

1 Merging abstraction trees 83
2 Balancing an abstraction tree 85

List of Symbols and Abbreviations

CRL	Cooperative Reinforcement Learning
FSA	Finite State Automata
HAM	Hierarchies of Abstract Machines
HRL	Hierarchical Reinforcement Learning
IR	Infra-Red
MDP	Markov Decision Process
RISC	Reduced Instruction Set Computer
RL	Reinforcement Learning
ROLNNET	Rapid Output Learning Neural Network with Eligibility Traces
SANDS	State Abstraction for Non-Deterministic Systems
SMGR	Softmax Gain Ratio
VAR	Variance Reduction
WSS	Weighted Strategy Sharing
A	Finite set of actions: $A = \{a_1, \ldots, a_{\lvert A \rvert}\}$
C_i	Carrying capacity of shelter i
D	Distribution of initial states of an MDP
I_i	Probability of entering shelter i
K_i	Maximal kinetic constant for entering shelter i
L	Number of leaves of a tree
M	Total samples of all sample-lists of a leaf
$O(\ldots)$	Time order of an algorithm
O_i	Probability of leaving shelter i
$Q(s,a)$	Action-value of state s and action a
R	Reward function: $S \times A \times S \to \Re$
S	Finite or infinite set of states
$\bar{S}$	An abstract state, a sub-set of the state-space
T	Transition model: $S \times A \times S \to [0,1]$
T_i	A sample of transition history: (s,a,r,s')
$V(s)$	State-value of state s
$V(T_i)$	Sample-value of the sampled data-point T_i
Z	Total number of animals in the environment
a,b	Current action, or any action in general

f	Inter-attraction factor
m	Number of samples in a sample-list
n	Number of dimensions of the state-space
p	Number of shelters in the environment
r	Immediate reward
s	Current state
$\bar{s}$	The abstract state that state s belongs to
s'	Next state
u	Number of partitions of a set-partition
x	Current state
y	Next state
z_i	Number of individuals in the shelter i
z_e	Number of individuals outside the shelters
α	Learning rate
γ	Discount factor, $\gamma \in [0,1)$
ε	Exploration rate, $\varepsilon \in [0,1]$
θ_i	Quality factor of shelter i
λ	Reference surface ratio
μ	Averaged sample-value of a sample-list
π	Policy: $S \times A \rightarrow \Re$
π^*	Optimal policy
ρ	Size of a sub-set of a sample-list to its whole size
τ	Temperature parameter

Chapter 1

Introduction

This thesis consists of an introduction, four appended papers, and a conclusion. The papers are arranged in three parts. The purpose of the introduction is to give a glimpse of behavior coordination for robots in general and overview of the papers and my contributions.

1.1 Behavior Coordination

A *behavior* maps sensory inputs to motor outputs to achieve a particular task [2]. In *behavior-based systems* a set of behavior modules is combined to provide the functionality needed for a given application [3]. *Behavior composition* is the mechanism used for building higher-level behaviors by combining lower-level ones [4]. *Behavior coordination* is concerned with how to decide which behavior to activate at each moment [5]. In fact, behavior composition specifies the hierarchy of behaviors however, behavior coordination specifies their execution timing. The activities of lower-level behaviors are coordinated within the context of the task and objective of a higher-level behavior.

Behavior coordination mechanisms are classified to *fusion* and *arbitration* [5]. Behavior fusion allows multiple behaviors to contribute to the final output, while behavior arbitration selects one behavior at a time. These mechanisms are often suitable for multiple-objective problems. For example, driving a car, and trying to keep it inside the lane, while avoiding collisions with other cars is a multiple-objective problem.

1.1.1 Fusion

Behavior fusion allows multiple behaviors to contribute to the final control of the robot. This mechanism combines recommendations from multiple behaviors to form a behavior that represents their consensus. Thus, all the behaviors simultaneously contribute to the final output in a cooperative manner.

They can be divided into [5]: *voting*, *fuzzy*, *multiple-objective*, and *superposition. Voting* [6; 7; 8] techniques interpret the output of each behavior as votes. The outputs are combined by tallying the votes. Then the action which has received the maximum number

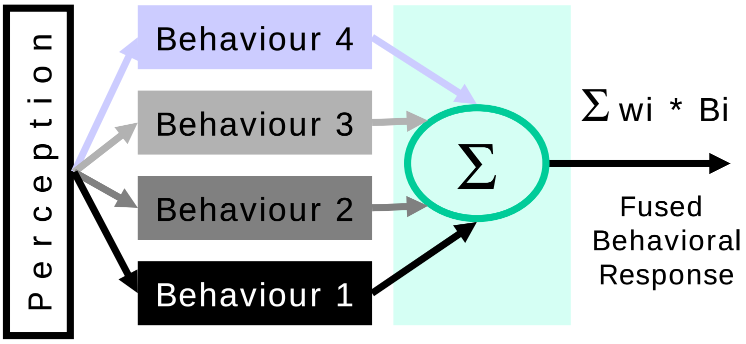


Figure 1.1: Fusion by superposition

of votes is selected. In *fuzzy* fusion mechanisms [9], fuzzy inferencing techniques are used. Multiple objective behavior coordination classes [10] , provide a formal decision-theoretic approach to making decisions based on multiple objective decision theory. Finally, *superposition* techniques combine behavior recommendations using linear combinations(Fig. 1.1).

The potential field approach [11] is a superposition-based fusion mechanism. The robot is represented as a point in the configuration space and moves under the influence of an artificial potential field. The field is produced by attractive or repulsive forces e.g. at goal configuration or obstacles. The potential field approach was introduced as an approach to motion planning. Its links to behavior coordination for behavior-based control was established in the *motor schema* [12] approach.

Schema

Schema theory [2] describes models which explain how the mental process organizes perception and coordinates responses to environmental stimuli. A *schema* is “the basic unit of behavior from which complex actions can be constructed; it consists of the knowledge of how to act or perceive as well as the computational process by which it is enacted” [11].

Schemas are parameterized potential functions that can be of two types: *motor schemas*, which are concerned with control of actuators, and *perceptual schemas*, which are concerned with sensing of features in the environment. A perceptual schema is embedded within each motor schema.

Motor schemas can have different importance. Output of each motor schema has a weight that is multiplied by its generated vector (Fig. 1.1). The weights are usually determined empirically for a given task and environment.

1.1.2 Arbitration

Behavior arbitration selects one behavior from a collection of behaviors and give it ultimate control of the output for a period of time, until the next decision cycle in which another behavior is activated. It is a competitive mechanism in which the behaviors are competing for taking control of the output.

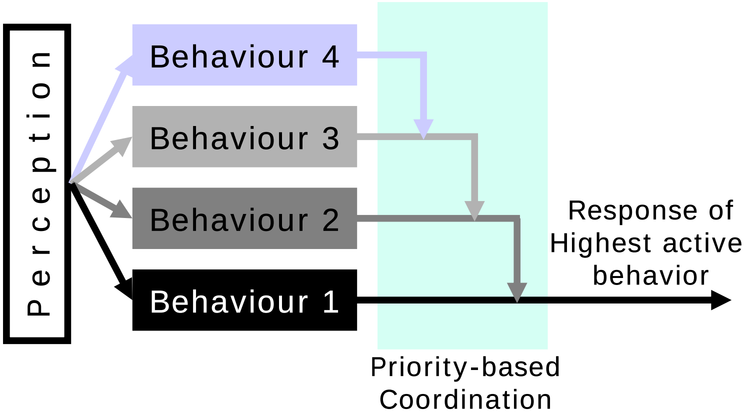

Perception
Behaviour 4
Behaviour 3
Behaviour 2
Behaviour 1
Response of Highest active behavior
Priority-based Coordination

Figure 1.2: Subsumption architecture

The criterion to select the winner is in accordance with the system's objectives and requirements under varying conditions. Arbitration mechanisms for behavior selection can be divided into [5]: *priority-based* and *state-based*.

Priority-based Arbitration

In *priority-based* mechanism a behavior is selected by another behavior in higher layers of the behavioral hierarchy based on some pre-assigned priorities. Behavior with the highest priority, then, is allowed to take control of the output of the higher-level behavior.

The subsumption architecture [13; 14; 15] is classified in this category. It consists of a series of behaviors, which constitute a network of hard-wired finite state machines. Action selection consists of higher-level behaviors overriding (or subsuming) the output of lower-level behaviors (Fig. 1.2). Each level is able to examine data from the levels below and is also permitted to inject data into the internal interfaces of the levels below, thus suppressing normal data flow.

State-based Arbitration

In *State-based* mechanism, the behavior that is the most competent of handling the given situation is selected [16; 17]. Behavior selection is done using state-transition, where upon detection of a certain event, a shift is made to a new state and thus a new behavior. Using this formalism, systems are often modeled in terms of Finite State Automata (FSA), where *states* correspond to execution of actions/behaviors. *Events*, which correspond to *observations* and *actions* cause transitions between the states.

The class of learning techniques for action/behavior selection can be classified under this category. These techniques can be used to learn how to select an action or a behavior. The most popular technique in this category is Reinforcement Learning (RL) [18].

In RL, the robot learns, by trial and error, a mapping from states to actions. Execution of actions subsequently lead to a reward. The mapping, known as a *policy*, maximizes the expected reward return. The reward function is designed so to encourage desired behaviors and suppress unwanted actions. The size of the state-space increases dramatically, as the complexity of the problems scales up. This makes the learning mechanism practically intractable.

To avoid the scaling problem, the task is broken down to a set of simpler tasks. Rather than attempting to solve the whole problem at once, a decomposition is performed to create a hierarchical structure of sub-problems [19].

1.2 Hierarchical Reinforcement Learning

Hierarchical RL [19] is a *divide-and-conquer* approach to solve the problem of large-scale reinforcement learning. Rather than attempting to find a solution for the whole problem at once, a decomposition is performed to create a hierarchical structure of sub-problems. The structure of the decomposition is often supplied as a prior knowledge to an algorithm however, we are interested in the methods that automatically derive it.

Many HRL methods are not designed to find the optimal policy. Instead, a policy with “good” performance is found in orders of magnitude less time [20]. One of these methods is *abstraction* technique. *Temporal abstraction* refers to techniques that group sequences of actions together and treat them as one abstract action. *State abstraction* refers to the technique of grouping many states together and treating them as one abstract state.

1.2.1 Temporal Abstraction

Temporal abstraction refers to techniques that group sequences of actions together and treat them as one *abstract action*. Abstract actions are closed-loop *partial* policies that are generally defined for a subset of the state set [19]. They must have well-defined termination conditions. These partial policies are sometimes called *macro-actions*, *temporally-extended actions*, *options* [21], *skills* [22], *behaviors* [13], *partial programs* [23], *sub-tasks* [24], etc. They usually are policies for efficiently achieving subgoals, where a subgoal is often a state, or a region of the state space, such that reaching that state or region is assumed to facilitate achieving the overall goal of the task [19].

Hierarchies of Abstract Machines [23] or HAMs consist of a hierarchy of learning machines, with a low-level machine representing an abstract action available to a high-level machine. Each machine is defined by a *partial program*, which constrains the range of policies the machine can learn. The program is a finite state machine with non-deterministic choice points. The optimal choice at each point is learned with RL.

The MAXQ [24] decomposition method combines both state and temporal abstraction. The task is decomposed to a hierarchy of sub-tasks. Actions consist of either executing primitive actions or policies that solve the subtasks, which can in turn invoke primitive actions or policies of other subtasks, etc. Hierarchy of sub-tasks, their termination conditions, and their pseudo-reward function are specified by the programmer. The structure of the hierarchy is summarized in a *task graph*. State abstraction could be exploited within the sub-tasks. RL finds a policy for choosing actions within each sub-task.

MAXQ [24] and HAMs [23]) have a strict, predefined hierarchy of abstract actions. Some researches try to determine useful task decompositions without prior knowledge. This obviously makes the learning problem much more difficult. McGovern [25] developed a method for automatically identifying potentially useful subgoals by detecting regions that the agent visits frequently on successful trajectories but not on unsuccessful trajectories.

HEXQ [26] sorts state features into an ordered list, beginning with the feature that changes most rapidly. It builds a task hierarchy, consisting of one level for each state feature. The very first level is the only level that interacts with the environment using primitive actions. Each level contains some sub-tasks that are connected through a set of “bottleneck” states. HEXQ is limited to considering each state feature in isolation. This approach fails for more complex problems.

Nested Q-learning algorithm [27] creates new sub-tasks, called *skills*, online while an agent is learning using Q-learning. It creates new skills by examining the frequency of state visits and the reward gradient at each state. States that are visited frequently or states where the reward gradient has high change are chosen as subgoals for new skills. These new skills take the agent from any state in the environment to the subgoal.

1.2.2 State Abstraction

State abstraction/aggregation [28] is based on grouping the sets of the original low-level states into a number of high-level *abstract states*. In some situations, knowing value of a particular feature is not necessary for selecting the optimal action. For example, what I have had last week might not be related to the action I should currently do to drive my car. By grouping the states that only differ in terms of this particular feature, the size of the state space can be reduced.

Feudal RL [29] is a *fixed-resolution* state abstraction algorithm, applied to navigational learning problems. The states are grouped according to spatial proximity. It creates a managerial hierarchy in which, high level managers learn how to set tasks to their sub-managers at the level below them, who in turn, learn how to satisfy them. The very lowest level managers can actually act in the world. The intermediate-level managers have essentially two instruments of control over their sub-managers: they can choose amongst them and they can set them sub-tasks. Managers reward sub-managers for doing their bidding, whether or not this satisfies the commands of the super-managers.

Although state abstraction is applied in Feudal RL, size of the state-space is not reduced; the managerial hierarchy helps only in decreasing the learning time. Moreover, It requires the user to pre-specify how the state space is discretized. Some styles of state abstraction attempt to automatically discover how the state should be abstracted, e.g. *variable-resolution* techniques and *regression-tree* techniques.

Parti-game [30] is a variable-resolution state abstraction algorithm. It starts with a single large abstract state. Then abstract states for which the resolution is too low, are split into smaller ones. An abstract state is divided either halfway along its longest axis or halfway along each axis.

Regression-tree techniques, e.g. U-Tree [31; 32], similar to variable resolution techniques, generate a flexible discretization of the state space using a tree structure. However unlike variable resolution techniques, they attempt to find the right *attribute* and *location* to split an abstract state, rather than just dividing down the middle.

1.3 Project Goal

This work is performed under the frameworks of the LEURRE, European project. This multi-disciplinary project gathers the competence of biologists, ethologists, chemists, and engineers coming from different European universities: Université Libre de Bruxelles, Université Paul Sabatier, Université de Rennes and École Polytechnique Fédérale de Lausanne.

The aim of the project is to understand and describe the computational behavioral nature of a group of animals, interacting and communicating together. The behavioral model must permit a description of the behaviors, from their most high-level definitions to their physical definitions and implementations. Some coarse-grain classifications must give a decomposition of the behaviors in several levels.

In order to verify the biological model of the collective behaviors, they must be implemented on a group of robots. The robots are placed in a mixed-society and interact with real animals. If the model is perfect enough, the arisen collective pattern would quantitatively be indistinguishable from the pure-animal pattern.

Biological studies give us a preliminary list of *elementary* behaviors of the animals. These behaviors are typically of the stimulus/response type. All the behaviors needed for modeling the system can be build on the elementary behaviors. It should be possible to describe more complex behaviors by combining elementary behaviors, using combinators such as logic connectors (and, or, ...) and/or modal operators.

A special robot, equipped with the necessary sensors and actuators to interact directly with the animals, must be built. Each elementary behavior must be implemented on the robot. However these behaviors, although considered as elementary for animals, might not necessarily be elementary for the robots. They could be broken to more primitive behaviors, which are meaningful to robotics engineers, but might have no counterpart in biology.

In this project, our team was mainly involved in designing and building the robots and all tools that were needed to work efficiently with them. I was, personally, responsible for developing the behaviors of the robots according to the models developed by the biologists. I should interpret the model and derive the behavioral hierarchy and the primitive behaviors for the robots.

In order to derive the behavioral hierarchy, I applied different techniques from manual design of the hierarchy to automatically learning the hierarchy. In the manual design, behavior coordination was accomplished using fusion (motor schemas and potential fields) and arbitration (finite state machines) techniques. In the automatic hierarchy extraction, new methods for state abstraction using regression trees were developed. I started from pure robot experiments (the first two parts), but as the task became unfeasible on real robots mainly due to the necessary learning time, I switched to simulation (the third part).

1.4 Organization of the Thesis

In the first part of my work, I try to specify the primitive behaviors, combine them together, and build more complex behaviors, heuristically. The hierarchy extends to a level that can implement the elementary behaviors, intended by the biological model. However, the hierarchy can be extended beyond the elementary behaviors if needed.

In the next parts, I examine machine learning techniques to automate some parts or the whole procedure of behavior composition and coordination. Specifically, in the second part, I apply flat RL to behavior coordination. The behaviors are combined and a new behavior is composed in a bottom up manner. However, the combination sequence is specified by the designer and it is not learned.

In the third part, I incorporate Hierarchical RL (HRL) techniques. There, the behavioral hierarchy is represented as a decision tree and is derived in a top-down manner. This part includes two separate papers. The first paper introduces some new splitting criteria for extraction of the decision trees. The second paper mixes HRL with Cooperative Learning (CL), in order to enable the learning robots cooperatively derive the hierarchy. There, new algorithms to combine and purify the decision trees are introduced.

The last chapter of the thesis includes summary of the results and concluding remarks. They are accompanied with some suggestions for future researches.

1.5 Publications

The thesis is constituted from four papers. The papers are extended versions of the ones that have already been published in conference proceedings and journals. The sketch of the papers is almost same as the published versions. However, mathematical proofs and detailed results, that had been ignored due to space limits, are added here. The papers constituting the thesis are:

Paper A: Mixed-Society of Robots and Animals (Ch.2)

The paper is coauthored with F. Tâche, G. Caprari, W. Karlen and R. Siegwart. It describes hardware and software implementation of the miniature robot, InsBot, built for mixed-society experiments. The robot is able to establish social interaction with cockroaches and modify their collective behavior. It is equipped with infra-red proximity sensors, luminosity sensors, and a linear camera.

The robots are controlled via a hierarchical behavior-based approach. Behaviors consist of a *perceptual schema* and a *motor schema*. Motor schemas are developed using the potential field concept. Complex behaviors are built by combining simpler behaviors through *fusion* and *arbitration* mechanisms.

Paper A corresponds to the following publications:

- **M. Asadpour**, F. Tâche, G. Caprari, W. Karlen, and R. Siegwart, "Robot-animal interaction: Perception and behavior of insbot," *International Journal of Advanced Robotics System*s, vol. 3, pp. 93–98, 2006.
- **M. Asadpour**, F. Tâche, G. Caprari, W. Karlen, and R. Siegwart, "Robot-animal interaction: Perception and behavior of insbot," in *Conceptual Modeling and Simulation Conference [CMS]*, (Marseille,France), 2005.

Paper B: Compact Q-Learning (Ch.3)

The paper is coauthored with R. Siegwart. It describes the concept and implementation of learning and combining multiple behaviors with the autonomous micro-robot, Alice. A Compact Q-Learning algorithm that is optimized in speed and memory consumption is introduced. Based on this algorithm, a simple method to compose and learn a new behavior from combination of two simpler behaviors is presented.

Paper B corresponds to the following publications:

- **M. Asadpour**, and R. Siegwart, "Compact Q-learning optimized for micro-robots with processing and memory constraints," *Journal of Robotics and Autonomous Systems*, vol. 48, pp. 49–61, 2004.
- **M. Asadpour** and R. Siegwart, "Compact Q-learning for micro-robots," in *Proceedings of the first European Conference on Mobile Robots [ECMR]*, (Poland), 2003.

Paper C: Hierarchical Reinforcement Learning (Ch.4)

The paper is coauthored with M.N. Ahmadabadi and R. Siegwart. It addresses the automatic state abstraction problem and discovering hierarchies in the state space. It formalizes the problem and derives three new criteria that adapt decision tree learning techniques to state abstraction. The first criterion maximizes the expected reward return. The second one maximizes the amount of reduction in the uncertainty on selection probability of actions. And, the last one minimizes the mean squared error on action-values.

Paper C corresponds to the following publication:

- **M. Asadpour**, M. N. Ahmadabadi, and R. Siegwart, "Reduction of learning time for robots using automatic state abstraction," in *Proceedings of the First European Symposium on Robotics* (H. Christensen, ed.), vol. 22 of *Springer Tracts in Advanced Robotics*, (Palermo, Italy), pp. 79–92, Springer-Verlag, Mar. 2006.

Paper D: Hierarchical Cooperative Learning (chapter 5)

The paper is coauthored with M.N. Ahmadabadi and R. Siegwart. It addresses two approaches to combine and purify the available knowledge in the abstraction trees, stored on different RL agents in a multi-agent system, or in the decision trees learned by the same agent using different methods.

Paper D corresponds to the following publication:

- **M. Asadpour**, M. N. Ahmadabadi, and R. Siegwart, "Heterogeneous and hierarchical cooperative learning via combining decision trees," in *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems [IROS]*, (Beijing, China), Oct. 2006.

1.6 Contribution of the Author

In paper A, all hardware parts of the robot and the necessary tools for debugging or working with them have been designed or assembled by G. Caprari and F. Tâche. The low-level layer of the local communication protocol has been developed by F. Tâche. Evaluation of the perceptual schema and some debugging have been done by W. Karlen. The other parts including sensory processing, motor control, high-level layer of local communication, perception algorithms, calibrations, and behavioral architecture have been developed by the first author.

In papers B, C, and D, the basic ideas and all theories and simulations were developed and implemented by the first author. The coauthors took the role of advisors, evaluating the developed theories, and pointing out unclear parts of arguments.

Part I

Manual Design of the Behavioral Hierarchy

Chapter 2

Mixed-Society of Robots and Animals

Masoud Asadpour, Fabien Tâche, Gilles Caprari,
Walter Karlen, and **Roland Siegwart**

Published in:
Proceedings of the Conceptual Modeling and Simulation conference, 2005 [33]
International Journal of Advanced Robotics Systems, 2006 [34]

Abstract. This chapter describes behavior implementation of a miniature robot that simulates collective behavior of cockroaches, in order to establish social interaction with them, and modify their collective behavior. Equipped with infra-red proximity sensors, luminosity sensors, and linear camera, they are able to discriminate cockroaches, walls, shelters, and other robots.

The robots are controlled via a hierarchical behavior-based approach. Behaviors consist of a *perceptual schema* that directs and organizes perception, and a *motor schema* that takes the output of the perceptual schema and produces a pattern of motor activity. Complex behaviors are built by combining simpler behaviors. Behaviors are coordinated through *fusion* and *arbitration* mechanisms.

Key words: Collective Behavior, Aggregation, Mixed-Society Control, Robot-Animal Interaction, Behavior-Based Architecture, Fusion, Arbitration

2.1 LEURRE Project

The present work is a part of the LEURRE, European project, which aims to study mixed-societies of animals and robots. This multi-disciplinary project gathers the competence of biologists, ethologists, chemists, and engineers coming from different European universities: Université Libre de Bruxelles, Université Paul Sabatier, Université de Rennes and Ecole Polytechnique Fédérale de Lausanne.

In this project, our team is mainly involved in designing and building the robots and of course all tools that are needed to work efficiently with them. Another important task is to program the behaviors according to the models developed by the biologists. The resulting system is a useful toolbox for biological researches.

The project studies two major topics:

- Collective decision-making in animals
- Controlling mixed-societies of robots and animals

2.1.1 Collective Decision-Making

Our short-term goal is to have robots which integrate into animal's society, live inside the society, and interact with the animals. The focus of our work is on the collective level. So, there is no need to have the same appearance as animals, but functionality of the robot must permit it to integrate into their society and produce *statistically* same collective behaviors.

By "*integration*" we mean not only the animal's behavior is affected by the robots, but also the robot's behavior is affected through interaction with the animals in the mixed-society. In fact every decision is made collectively by the whole society in such a way that, a top-level observer would not see any difference between the animal society and the mixed one. In our model, the animal is thus considered as a black box, and the important characteristics for our robot is to fit in the mathematical model of the collective interactions among the individuals involved in the group.

2.1.2 Mixed-Society Control

The long-term goal of the project is to manipulate the collective response of the society by modulating the behavioral parameters of the robots in order to lead the society toward a specific goal. We hope, then, to propose guidelines towards a general methodology for performing such a control on mixed-societies in general.

The mechanisms of effective leadership in biological systems have already been studied [35; 36; 37]. It is known that in many collective behaviors, only a few individuals among a group of animals that, e.g. forage [36] or travel [37], have pertinent information such as, knowledge about the location of a food source or of a migration route. Couzin et al. [35] showed by simulation that, the larger the group the smaller the proportion of informed individuals needed to guide the group, and that only a very small proportion of informed individuals is required to achieve great accuracy. Our project aims at realizing such achievements.

Potential applications of the control in mixed-societies are very vast; Stabilizing or changing the economical system of a country by controlling some informed-agents that are injected to the system, and habituating people to a specific culture can be of its potential applications. The same mechanism might be involved in the system of marketing for fashions by means of model persons. In agriculture and ecology, controlling wild populations (endangered, invasive and pest species) and managing exploitation of environment resources could be two possible applications.

2.1.3 Previous Works

Among the projects that have tried to interact with animals and influence their behavior using robots, we can mention the Robot Sheepdog [38] that controls a flock of duck by moving them safely to a pre-determined position. The W-M6 rat-like robot [39] by Ishii et al. tries to create a symbiosis between creature and robot by teaching a rat to push a lever to access a food source.

Butler et al. [40] considers the problem of monitoring and controlling the position of herd animal. They use Smart Collars, small position-aware computer device worn by the animal, which applies a stimulus to the animal as a function of its pose with respect to a virtual fence line. Multiple fence lines define a region, and the fences can be static or dynamic. The stimulus can typically be an odorous spray, a sound, or an electric shock.

These projects are different from what we are investigating in that, their robots or devices are not trying to integrate into the society. Instead, they are trying to affect or supervise the society in a centralized manner.

Boehlen developed a robot [41] that interacts with three chickens in a cage. He manipulates some techniques to mechanically reduce chickens' anxiety towards moving machinery. The goal of the robot is to integrate with the chickens but does not try to affect their behavior.

The chapter is organized as follows: first the reason behind choosing cockroach and aggregation for our studies is explained. In the next section, the biological experiments, the experimental setup, and the required functionality of the robots are described. Next, the hardware architecture of the robot is summarized. Then, the perceptual and motor schemas are presented. The chapter is finalized with the test results, conclusions, and future works.

2.2 Biological Experiments

This section explains why cockroaches and their aggregation has been selected as the base of the biological studies.

2.2.1 Why Aggregation?

One of the fundamental questions in social biology is to understand the link between individual and collective behavior, in order to address the larger picture of what happens at the group or population level, based on an understanding of the behavior of the individuals. *Self-Organization* theory [42] tries to explain how group patterns arise from the interaction of individuals in relation to their local environment, without reference to the global structure or long range information transfer. The rules specifying the interactions in a self-organized system are assumed to be on the basis of purely local information.

We believe that an elegant way to identify individual behavioral algorithms consists of replacing some animals in a group by robots, and comparing the collective responses in *mixed* and *natural* groups. If the robots are programmed according to the developed behavioral algorithms and are tested on a mixed group, the outcome can give better understanding on the computational capabilities of the animal society.

The most common collective behavior among living organisms is probably *grouping*, which occurs in a wide range of taxa, from bacteria to mammals via arthropods, fishes, or birds [43]. Depending on the species, these assemblages are labeled as herds, shoals, flocks, schools, swarms, and more broadly denoted as *aggregation* [44]. *Aggregation* therefore, is defined as any assemblage of individuals that results in a higher density than in the surrounding area [42].

2.2.2 Why Cockroach?

Ideally, biologists that study collective behaviors like to work with social insects, in particular ants. However, this is currently rather difficult because the size of the robots is constrained to centimeters, due to technological limits, that is still big compared to ants. Moreover, from the point of view of collective behavior, cockroaches are very interesting animals since their collective behavior is less complex than that of social insects [45]. They are therefore good case studies to investigate the origin of the mechanisms involved in collective behaviors.

Robots Inspired by Cockroach

Over the last decades, researchers in bio-inspired robotics have mimicked cockroaches to design hardware and software structure of the robots. The RobotV [46], RHex [47], Biobot [48], HEL-roach [49] and the hexapod micro-robot [50] are examples of legged-robots which have been mechanically inspired by cockroaches. Cowan et al. [51] present a mobile-robot doing wall-following with a simplified antenna, inspired from cockroaches.

Some researches have developed hybrid robots by mixing the artificial and biological systems. The PheGMot-III [52] by Nagasawa et al. uses real cockroach antennas as a chemical sensor to follow pheromone tracks. Holzer [53] designed a system which controls the cockroaches' actuators by electric stimulation. Hertz [54] developed the reverse case, a cockroach is used atop a modified track-ball to control a three-wheeled mobile robot. Infra-Red (IR) emitters provide navigation feedback and help the cockroach navigate the robot away from obstacles. As the robot approaches an obstacle the IR emitters send more light to the cockroach. The robot is steered into a clear space, as the cockroach moves on the track-ball to scuttle into a dark region.

Instead of building exactly the same mechanism as cockroaches, our goal is to have robots which integrate into their society, and participates in their collective decision-makings as an individual.

Cockroach Aggregation

Actually, in nature some places are more attractive for cockroaches. This attraction, thus, promotes aggregation in particular sites. For instance, cockroaches prefer to aggregate in dark places [55]. Experimentally, if one puts a dark shelter in a lighted arena, one could observe that cockroaches strongly aggregate under this shelter. And, if two or more dark shelters are placed in the arena, one can observe that the majority of cockroaches aggregates under only one of these shelters, rather than spreading evenly among all the aggregation sites [56]. Hence, cockroaches are able to collectively choose an aggregation site, even if the sites are identical.

2.2.3 Aggregation Model

In this project, cockroach aggregation is modeled based on the Self-Organization theory [42]. Self-organization relies on four basic ingredients [57]:

1. *Positive feedbacks*, that promote creation of a collective patterns e.g. reinforcement signals

2. *Negative feedbacks*, that counterbalance the positive feedback and helps to stabilize the collective pattern.
3. *Random fluctuations*, which enables exploration of the solution space.
4. *Multiple individuals*, that interact in the system.

Many studies have tried to model cockroach behaviors, both in individual level [58] and collective level [43; 59; 60]. In summary, dynamics of the aggregation process at microscopic level can be described by the following dynamic equations [60]:

$$\frac{dz_i}{dt} = I_i \cdot z_e - O_i \cdot z_i \quad i = 1,\dots,p \tag{2.1}$$

$$Z = z_e + \sum_{i=1}^{p} z_i \tag{2.2}$$

where p is the number of shelters in the system, z_i is the number of individuals in the shelter i, z_e is the number of individuals outside the shelters, I_i is the probability that an exploring individual outside the shelters joins the shelter i, O_i is the probability that an individual, that is currently inside the shelter i, leaves it and starts to explore, and Z is the total number of individuals.

I_i decreases when the ratio between the number of individuals, currently inside the shelter i, and its carrying capacity increases. It changes according to the following equation:

$$I_i = K_i \left(1 - \frac{z_i}{C_i}\right) \tag{2.3}$$

where K_i is a factor denoting the maximal kinetic constant for entering the shelter i, and C_i is the carrying capacity of the shelter i.

O_i decreases with the number of individuals z_i present in shelter i, according to the following equation:

$$O_i = \frac{\theta_i}{1 + \lambda \left(\frac{z_i}{C_i}\right)^f} \tag{2.4}$$

where λ is the reference surface ratio for estimating carrying capacities, f is the inter-attraction factor, and θ_i is a factor that depends on the quality of the shelter i.

The four ingredients of this self-organizing system are clearly distinguishable. In this model, I_i denotes the positive feedback which encourages aggregation in a specific site. In the other hand, O_i corresponds to the negative feedback that discourages aggregation in that site. Random fluctuations happen both in I_i and O_i, since they are probability numbers. Finally, it is evident that the system needs $Z > 1$ individuals to work.

2.2.4 Mixed Aggregation

To verify the biological model of cockroach aggregation [58; 43; 59; 60], it must be implemented on a group of micro-robots. The robots are then placed in a mixed-society consisting of robots and cockroaches. It must be shown that this implementation indeed results in a collective aggregation, that is quantitatively indistinguishable from pure-cockroach aggregation.

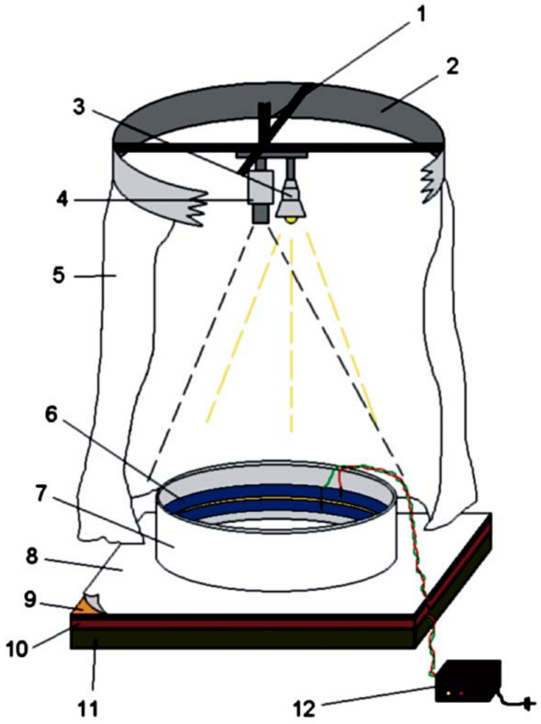


Figure 2.1: Experimental setup composed of suspension hook (1), aluminium ring (2), neon light (3), top camera (4), curtain (5), electric fence (6), white plastic arena (7), paper layer (8), phonic layer (9) and wooden layer (10), acoustic insulator (11), DC power supply (12). Shelters are not represented on the illustration

Next, when this aggregation behavior is restricted to certain zones in the environment (for instance by natural preferences for dark places as in cockroaches), the robots along with the cockroaches must preferentially aggregate in only one of these zones, i.e. they must collectively choose a single "rest" site. When these zones are of different sizes the robots must, like cockroaches, preferentially choose the biggest of the two, but without being individually able to measure their size. Finally, when the robots' preference for site selection is reversed to oppose the cockroach's preference, it must be shown that the robots can influence the cockroach and change their site selection preference, provided that the number of the robots is bigger than a limit.

2.2.5 Experimental Setup

The experimental setup (Fig. 2.1) is a circular white plastic arena (1 m diameter, 20 cm high) with an electrical fence to avoid the cockroaches standing up or escaping. The floor is composed of different layers, which reduce the amount of vibrations that could potentially frighten the cockroaches. The ground white paper is changed after each experiment to avoid cockroaches being influenced by the remaining pheromone of the previous experiments. To avoid disturbance to the cockroaches' behavior and the robots' IR sensors, illumination is given by 4 neon-light

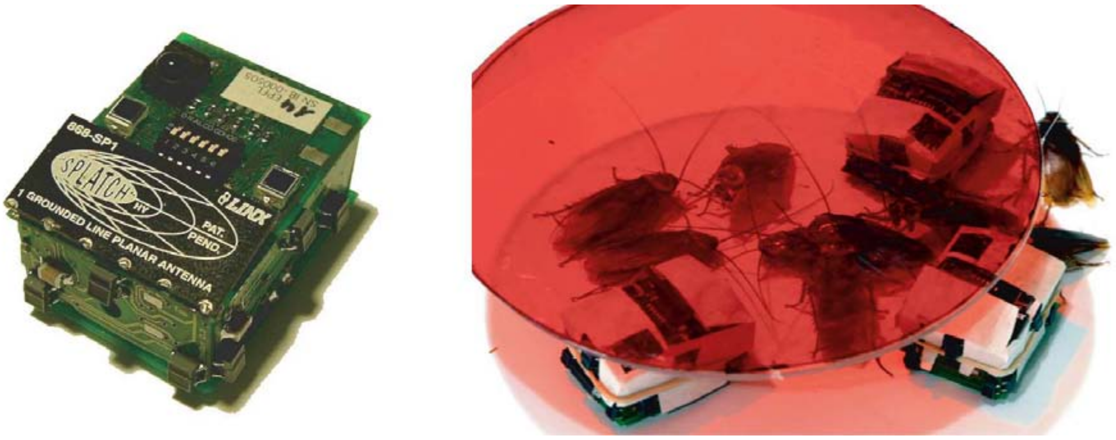


Figure 2.2: Left: InsBot without cover. Right: InsBots with their paper-covers aggregated with cockroaches under a shelter

bulbs with low IR emission. Finally, two circular shelters are suspended from the roof.

To enhance the acceptance of the robots into the cockroach's colony, they are covered by a paper that is impregnated with cockroaches' pheromone. It has been collected by extracting cuticular hydrocarbons of adult male cockroaches in dichloromethane [61].

2.2.6 Functionality of Robots

In summary, requirements of the robot[62] do not specify that it should look like a real cockroach, but it should:

- Be accepted by the cockroaches as a congener
  - Have a small size, preferably same size as cockroaches
  - "Smell" like a cockroach.
  - Behave like a real cockroach in the mixed society
  - Be able to detect them and interact with them in a safe manner
- Be able to influence the global behavior of their society
- Have an easy-to-modify behavioral architecture
- Be equipped with monitoring and debug facilities

The required sensory capability of the robot mainly depends on the environment conditions. It is also limited by its low-power consumption and small-size.

2.3 InsBot: The Insect-like Robot

The designed robots are finally $41 \times 30 \times 19\,\mathrm{mm}^3$. Figure 2.2 shows the robots with (right) and without (left) the pheromone cover. They are called **InsBot**, which stands for "*Insect-like Robot*".

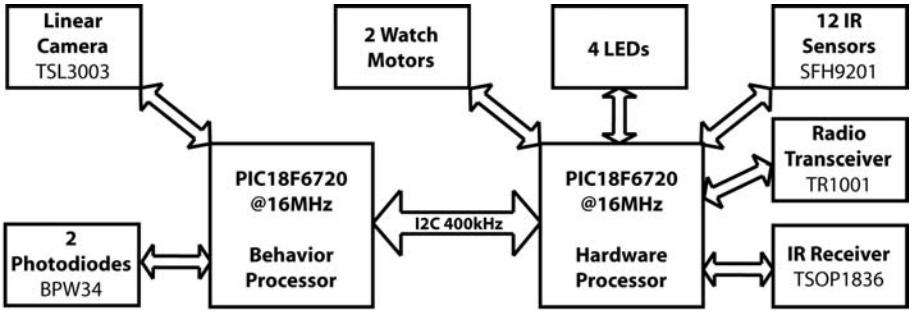


Figure 2.3: Control and electronic architecture of InsBot

Their rigid body is composed of PCBs which allow mechanical and electronic connections at once. They hold a 190 mAh Li-Poly battery that allows at least three hours of autonomy required for the biological experiments. They include two miniature step motors for locomotion in differential drive configuration. A nail head is used as the third contact point. The robots weigh 17 gr and can move up to 4 cm/s. They are equipped with two micro-processors dedicated to hardware processing and behavior generation.

2.3.1 Sensors and Communication Tools

In spite of their compact size, several sensors and communication tools are embedded in InsBots:

- Two photodiodes on top of the robots allow for shelter detection.
- 12 IR proximity sensors, three on each side, placed at different heights allow discrimination of different objects in the environment. They are also used for short range robot-to-robot communication in order to detect the other robots in vicinity.
- A linear camera in front of the robot, combined with IR sensors, enhances cockroach detection process.
- An IR receiver allows to command the robots with a standard TV remote control in order to supervise the biological experiments.
- A radio transceiver (at 868 MHz) provides a communication media with an external computer. This radio link is mainly used for debugging or monitoring purposes.

2.3.2 Control and Electronic Architecture

Each robot includes two PIC18F6720 micro-processors as depicted in Fig. 2.3. Both processors have a 16 MHz external clock, 128 KB program memory, 3840 byte of SRAM data memory and 1024 byte of ROM.

The“Hardware Processor” is connected to most of the hardware resources. It prepares sensory information and applies noise reduction, scaling and calibration techniques. The sensory information is then feeded to the perception process for sensor fusion and recognition of around objects. Sensory information and output of the perception process are periodically transmitted through a 400 KHz I^2C bus to the “Behavior Processor”.

The first plan was to run low-level hardware processing on the hardware processor and to put all behaviors on the behavior processor. The name of the processors comes from this plan. However finally, we had to distribute behavior generation on both processors. Collective behavior of cockroaches is implemented on the Behavior Processor, while individual behaviors are implemented on the Hardware Processor. Since individual behaviors need fast reaction, placing them on the Hardware Processor facilitates their access to hardware resources.

2.3.3 Calibration

Due to several reasons a calibration phase should be repeated once upon every setup change. The inclination angle of proximity sensors is hard to adjust precisely. Also, they are not precisely placed at the same height, so they have different initial values. The floor paper and its flatness highly affect the bottom sensors. The illumination condition varies in experimental setups and the amount of light under the shelters changes as well. Orientation of shelters might vary for each experimental setup, thus changing gradient of light under the shelters.

The calibration procedure developed for proximity sensors and shelters are activated via TV remote control upon the user requests. The computed calibration vectors are saved in the EEPROM and loaded after each restart. During regular process, these vectors are used to adjust the value of sensors and filter the noise part.

2.4 Behavioral Architecture

The robots are controlled via a hierarchical behavior-based approach [11; 2]. A *behavior*, maps sensory inputs to motor outputs to achieve a particular task. A behavior consists of a *perceptual schema* that directs and organizes the perception and a *motor schema* that takes the output of the perceptual schema and produces a pattern of motor activity.

The layers of the hierarchy have been designed manually. The lowest behavioral layer consists of *primitive behaviors.* Behaviors in the higher layers are constructed by combining the behaviors in the lower layers. When execution, the behaviors are coordinated through two schemes: *fusion*, and *arbitration.* In fusion, behavioral responses are represented in vectors, generated using a potential field approach. The vectors indicate the direction and the magnitude of the relevant forces. Coordination of behaviors is achieved through cooperative means by vector addition. The resultant vectors are finally mapped to proper actions. In *arbitration*, a finite-state machine stochastically selects a behavior among its subordinates.

2.4.1 Perceptual Schema

In this section we focus on the detection algorithm that has been tuned for optimal environment perception. In fact, to behave like a real cockroach, the robot must first be able to detect the relevant features of the experimental setup which are the inputs of the motor schemas. These features are the living *cockroaches*, the setup *walls*, the *shelters* and the *other robots.* The perceptual schemas have been tuned to work in this particular environment. It is executed on the Hardware Processor due to having faster access to sensors.

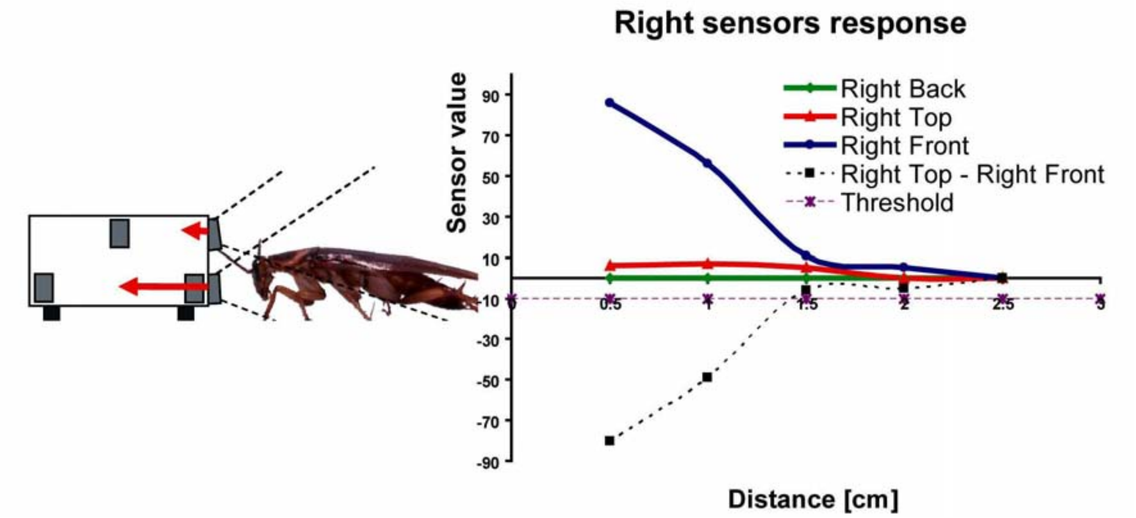


Figure 2.4: Left: IR sensors activated by a cockroach. Right: Response of IR sensors when a cockroach is placed on the right-front side of the robot

Shelter Detection

The two different circular shelters (called “dark” and “bright” shelters hereafter) are composed of dark plastic layers hanged at 5 cm from the ground. To create different levels of shadow, different number of layers are grouped.

For detection and differentiation of both shelters, the light intensity is measured by two photodiodes, mounted on top of the robot. Then, their value is compared with the thresholds, computed during the calibration procedure at the beginning of the experiment. As light intensity in the “center” of both shelters is distinct enough, detection and differentiation quality is perfect.

A gradient of light stretches from the center of the created shadow towards the borders, with the maximum light intensity at the borders. As a result, some regions have the same luminosity under both shelters. In these parts, the robot treats both shelters as the bright one. This induces no bias in the results, since real cockroaches also confront the same problem.

Cockroach Detection

The cockroaches used in the mixed-society experiments are *Periplaneta Americana* which are domiciliary species [45]. They are 24-44 mm long and shine red-brown. They have 6 legs and two long (around 3 cm) antennas. Having dark skin, they are hardly detectable by IR sensors, but thanks to the particular sensor placement on the robots, the calibration procedure and some heuristic rules, they can be distinctively detected from 1.5 cm distance.

Of the three IR sensors on each side of the robot (Fig. 2.2), the two lateral sensors are close to the ground (called “bottom sensors” hereafter) and the other one is placed at the center of each side, at the maximum possible height (called “top sensor” hereafter). Due to the short height of the cockroaches, top sensors are less affected than bottom sensors, as illustrated in the left schematic of Fig. 2.4. The solid curves on the right side show the response of the IR sensors to a cockroach placed at different distances on the right side of the robot. It shows, at far distances (more than 2.5 cm) no response from the sensors is received; the cockroach is thus

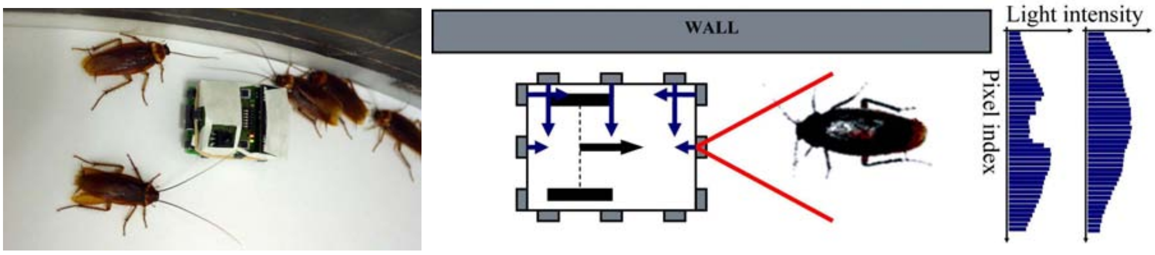


Figure 2.5: IR sensors are activated when the robot follows the wall and meets a cockroach. Left: Real situation, Middle:The front-left and rear-left sensor are activated, even without presence of any cockroach. Right: Response of the linear camera in presence/absence of cockroach.

invisible. Between 2.5 to 1.5 cm, response of both sensors are almost equal; the cockroach is visible but not differentiable from the wall(See 2.4.1 for comparison). From 1.5 cm, detection of cockroach becomes possible through thresholding the difference between the top and the bottom sensors (dashed curves of Fig. 2.4).

Enhanced Cockroach Detection: In some situations, IR sensors could not reliably help in discrimination of different objects, specially when a cockroach is located in vicinity of the wall. As illustrated in Fig. 2.5, the front-left and rear-left sensors are activated without presence of any cockroach. This is due to the large aperture angle of the IR sensors. We would like to decrease the amount of collision with cockroaches, as much as possible, by enhancement of cockroach detection algorithms. In this situation using the linear camera could help declining the number of false detections.

Therefore, the cockroach detection schema for the front side of the robot -where collision risk is the highest- uses combination of IR sensing and linear camera processing. The algorithm for processing the linear camera encounters the magnitude and the length of the discontinuous patches on the camera image (Fig. 2.5 right side), which appears due to the dark skin of the cockroaches. The length and the magnitudes are, then, compared with some thresholds that have been computed during the calibration phase.

Wall Detection

Periphery of the circular arena is composed of a white plastic, partially covered by a black colored electric fence. The electric fence starts from 3 cm height up to the top edges of the wall. So, the visible part of the wall to the IR sensors is only the lowest 3 cm section. A simple schematic of the situation in Fig. 2.6 (left side) clarifies that, value of the top sensor should be roughly close to the mean of the two lateral sensors.

The solid curves on the right graphics (Fig. 2.6) show response of the right sensors, when the robot is placed at different distances, making 45° angle with the wall. It shows, at distances further than 7 cm, no response from the sensors is received, and the wall is invisible. Between 6.5 to 0 cm, the wall becomes more visible. The sensor values confirm that, the value of the top sensor is close to the average of the bottom sensors (dashed curve). This graphics also shows that, the difference between the top and the bottom sensor (solid curve) is at some parts negative; this could make confusion with the cockroach case. That is why, the threshold for cockroach detection

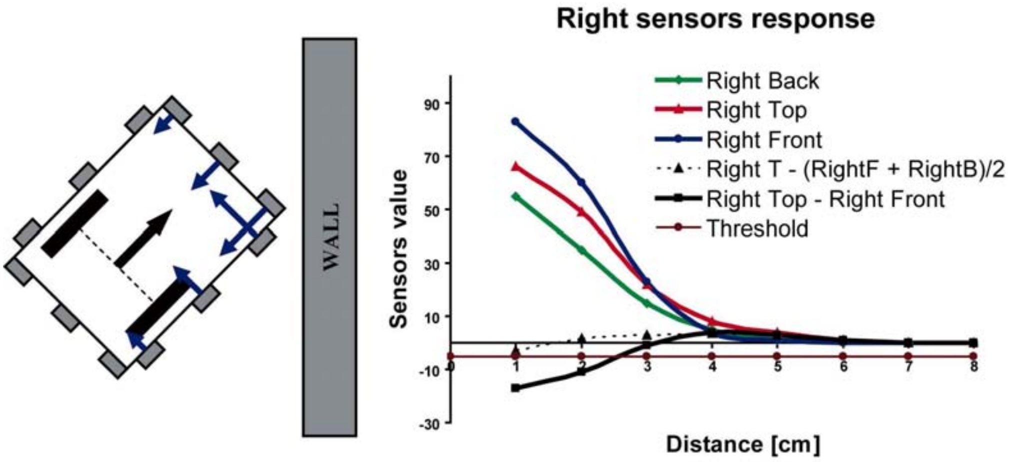


Figure 2.6: Left: IR sensors activated by the wall. Right: Response of IR sensors when the robot is placed near a wall (at 45°). For comparison with cockroach detection case, right-top minus right-front curve is also plotted.

–and consequently detection range– is decreased (Fig. 2.4 for comparison)

Robot Detection

Finally, InsBot must be able to distinguish the other InsBots, from the cockroaches and the wall. In the current experiments an InsBot is treated like a cockroach, but for future experiments, we would like the robot to interact with cockroaches and robots in different ways. However, detection of the robots using only proximity value of the IR sensors is not feasible; they might look like the wall or a cockroach, depending on their relative orientations (as depicted in Fig. 2.7). Therefore, a local communication protocol that allows for marking the robots with IR tags is developed.

Local Communication: Since the IR sensors include an emitter and receiver, they can be used as a transceiver as well as a proximity sensor. Hence, if the robot receives any signal while its emitters are silent, it means another robot is emitting, either for communication or proximity sensing. This idea is used to establish a local range, one-to-one, communication protocol between nearby robots. Due to electronic limitations and computation economy, it is implemented only on the four top sensors.

Communication Protocol: The protocol (Fig. 2.7) includes a "hello/wakeup" signal (*11* ②) in order to differentiate it with the signal, emitted during proximity measurements ① (which is similar to *1*). When this message is received by another robot (B), it becomes master and begins sending a message that starts with 3 start-bits (*111* ⑤). If the robot (A), which has sent the "hello/wakeup" signal, receives these 3 start-bits after a given time ③+④, it becomes slave and listens for the message. The message includes 8 bits, 6 bits for data ⑥ followed by 2 stop bits (*11* ⑦). The data bits specify the unique ID of the master. The perceptual schema counts the number of unique IDs to find the number of surrounding robots.

Neighborhood Range: Communication throughput with this protocol is very low (max. 7 msg/s = 56 bit/s). Therefore, the robots might not have the chance to communicate quite often. Hence, the protocol must be combined with a software solution to provide a short-term memory

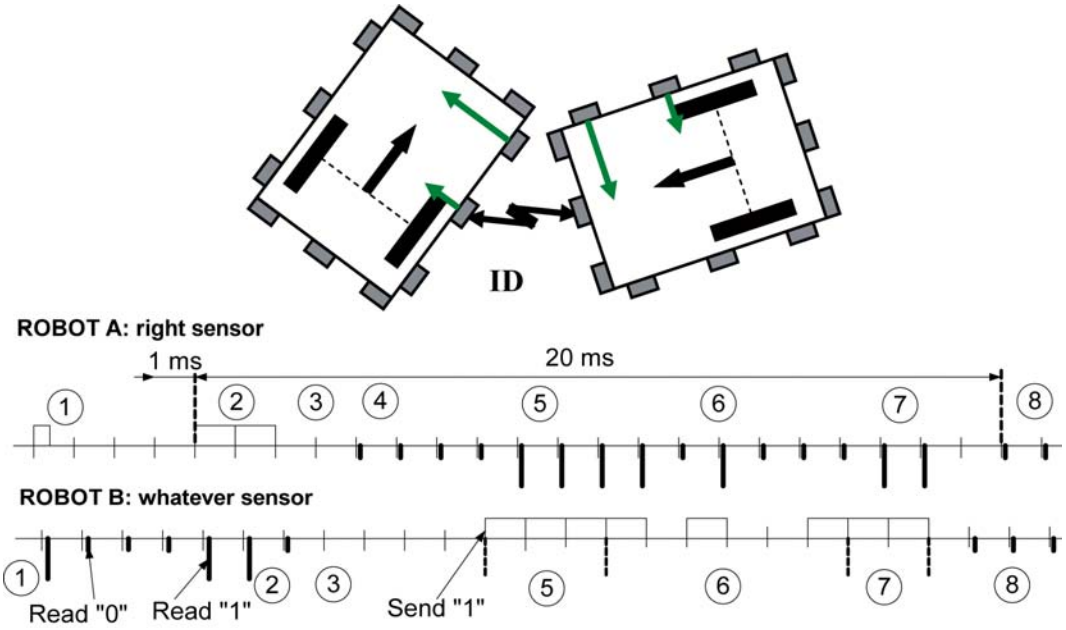


Figure 2.7: Top: Response of IR sensors when two robots meet. They might look like a cockroach (as in this case) or the wall. A local communication protocol, based on IR signals, allows for detecting and counting the robots. Bottom: Chronogram of the communication protocol

of the around robots.

The information, that the robot extracts out of the local communication, is registered in a log table. It is tagged with a time-stamp and the ID of the sensor that receives the message (It roughly indicates the relative position of the sender). The robot then has at its disposal, information about when, where, and who has been around him.

A fixed time-window of T seconds is defined to limit the neighborhood range. The message expires after this time, and only the robots that have been around within the last T seconds are counted as neighbors. The neighborhood range can increase/decrease by setting the time-window to longer/shorter duration.

The log table has a limited size. In case of experimenting with a large group of robots, it can hold only a part of the signals, the most fresh ones. If an already-registered robot is detected next time, its corresponding record is updated with the recent information. Otherwise the oldest record of the table is deleted.

2.4.2 Motor Schema

The implemented behavioral model is a hierarchical behavior-based approach [11] distributed on the two processors. Each behavior can take input from the sensors directly, the perceptual schema, and/or the other behaviors. They can send their output to the actuators or the other behaviors.

Behaviors are arranged in a hierarchy in which, the ones on the higher levels fuse or arbitrate the ones in the lower levels. At the highest level, an arbiter that implements the *mixed-aggregation*

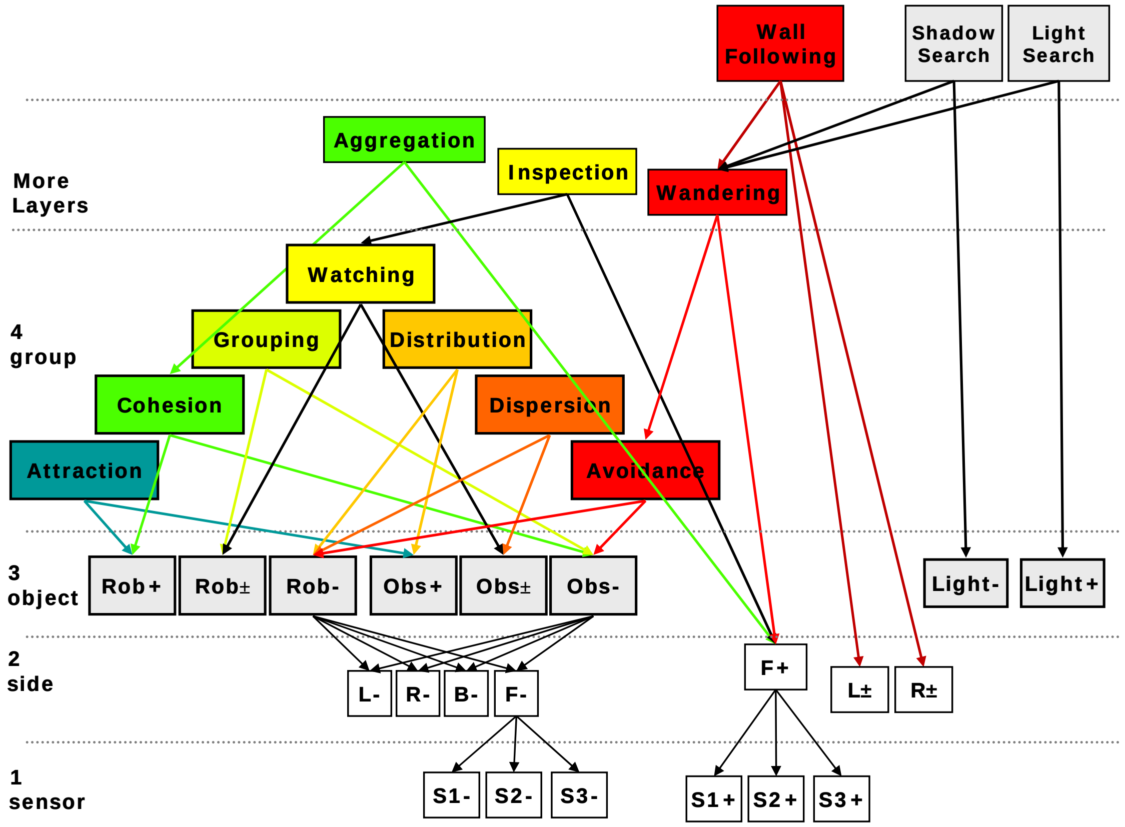


Figure 2.8: Overview of the behavioral architecture. Rob, Obs, L, R, B, F , +, -, and ± stand for robot, obstacle, left, right, back, and front, attraction, repulsion, and following (combination of attraction and repulsion), respectively. S_1, S_2, and S_3 are the three front-side sensors. Weights and some other details are not shown.

behavior decides which behavior to execute. At the next level, the selected behavior activates one or more behaviors from the lower levels. Decomposition continues downward until the primitive behaviors in the lowest layer. Behavior coordination is, therefore, *competitive* at the highest level, and *cooperative* at lower levels. The final output is the result of cooperation among the activated behaviors.

The arbiter runs on the behavior processor. Lower level behaviors (e.g. obstacle avoidance and wall following) are managed in the hardware processor, since they need faster access to the perceptual schema. The running cycle of the reactive behaviors is 10 times faster than the centralized arbiter (50 vs. 500 ms).

Behavior Fusion

Behaviors are fused by means of the potential field method [11; 63]. Each behavior generates a potential field. Potential fields map the sensory space into the motor space, through attraction or repulsion force, (r_x, r_y). The final velocity of the robot corresponds to the resultant force, (R_x, R_y), which is the weighted sum of the force vectors. The weights are specified empirically.

The resultant force is, then, transformed to the speed of the wheels.

Behavior Layers: Fig. 2.8 shows a typical architecture of the behaviors. For the sake of clarity, we had to assume the robot is equipped only with IR proximity sensors and the distinguishable entities are limited to robots, obstacles, and ambient light. Also, some behaviors in the three first layers are hidden.

The **first** layer (lowest) includes the primitive behaviors that deal with a specific sensor. Each primitive behavior assign different levels of attraction or repulsion toward the direction that the sensor actually points. The magnitude of the force is a function of the value of the sensor s, i.e. $(r_x, r_y) = f(s)$. In the simplest forms it can be constant ($f(s) = a$), proportional ($f(s) = a \cdot s$), or piece-wise e.g.

$$f(s) = \begin{cases} a & \text{if } s > c \\ b & \text{otherwise} \end{cases} \tag{2.5}$$

By defining $f(s) = a \cdot s - b$, where $a \neq 0$, a mixture of attraction and repulsion is created; when $s > \frac{b}{a}$ the force is attractive and when $s < \frac{b}{a}$ the force becomes repulsive. This form of functions is suitable to create *following* behaviors, i.e where it is expected from the robot e.g. to keep a fix distance from an object.

The **second** layer deals with a group of sensors, the ones that lie on the same side of the robot. Here a group of primitive behaviors of the first layer are combined to generate a force toward/from the left, right, front, and back directions. For instance, *move-forward* behavior is achieved by assigning a constant attraction force to the sensors in front side, or *turn-in-place* behavior is achieved by setting an attraction force toward right (clockwise rotation) or left (counter-clockwise) direction.

The **third** layer includes the behaviors that deal with the objects, perceived by the perceptual schema i.e. dealing with multiple type of sensors. For instance, *obstacle-avoidance* is sum of *left-avoidance* if an obstacle is detected at left side, plus *right-avoidance* if an obstacle is detected at right side, and so on.

The **fourth** layer deals with a *group* of objects. This layer composes collective behaviors, like dispersion, cohesion, watching, etc. For instance, *watching* is the result of *robot-following* and *obstacle-following*, or *grouping* is the sum of *robot-following* and *obstacle-avoidance.*

It is possible to add more layers and create more complex behaviors. For instance, *light-search* behavior is built from *wandering* [1] and *light-attraction*, where *wandering* is composed of *avoidance* and *move-forward*, where *avoidance* itself is composed of *robot-avoidance* and *obstacle-avoidance.*

Behavior layers are designed carefully to maintain scalability and reusability of the components, both when adding more layers and when implementing on another robot. We have implemented the same architecture (with slight weight modification) on Alice micro-robots [64].

Transformation to Speed: Finally, the resultant force (R_x, R_y) is transformed to the speed of wheels. Inspired by law of physics for force and torque, we can write for Fig. 2.9:

$$\begin{cases} \sum F = 0 \Rightarrow R_y = F_l + F_r \\ \sum \tau = 0 \Rightarrow L' R_x = L(F_l - F_r) \end{cases} \xrightarrow{L'/L=k} \begin{cases} F_l = \frac{R_y + kR_x}{2} \\ F_r = \frac{R_y - kR_x}{2} \end{cases} \tag{2.6}$$

---

[1] *Wandering* is frequently referred to, in the other papers, as obstacle avoidance. The goal is to move in the environment safely, without hitting the objects.

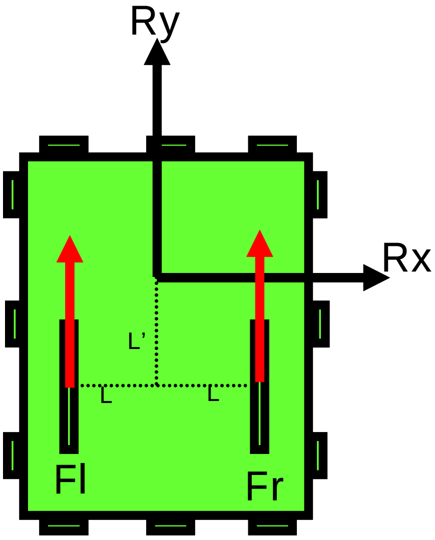


Figure 2.9: Converting resultant force to speed vector

The vector (F_l, F_r) is a force vector that must be simulated by the wheels, so that an observer perceives a force vector (R_x, R_y) affecting the movement of the robot. (F_l, F_r) is mapped to a feasible speed vector (V_l, V_r). We used $k = 2$ in our application. Bigger k values create sharper rotations, since R_x in (2.6) is magnified.

Behavior Arbitration

Mathematical model of cockroach behavior is implemented as a random-walk in form of a probabilistic Finite State Machine (Fig. 2.10). The state-machine has four states: {*random-decision*, *moving*, *turning*, *stopped*}. The trajectory of the robot is characterized by a series of straight-moves (free paths), turning angles, and stops [58].

The robot starts from the *random-decision* state. State transitions occur upon selection of a specific behavior. The selected behavior is activated and executed for a fixed period, ΔT (500 ms here). After this period, the current state returns back to the initial state, and the decision making process is repeated again and again.

When the robot is near the periphery, it can select either:

1. *Wall-following* behavior: to move along the periphery, following its circular shape
2. *Wall-avoidance* behavior: to turn away from the periphery toward the center of arena, or
3. *Stop* behavior: to turn off the motors and be immobile for ΔT seconds.

Upon selecting one of these behaviors, the current state is changed to *moving*, *turning*, or *stopped*, respectively. The robot stays in that state during the whole ΔT period and returns to the initial state, afterwards.

When the robot is in the center (i.e. not near the periphery), it can select either:

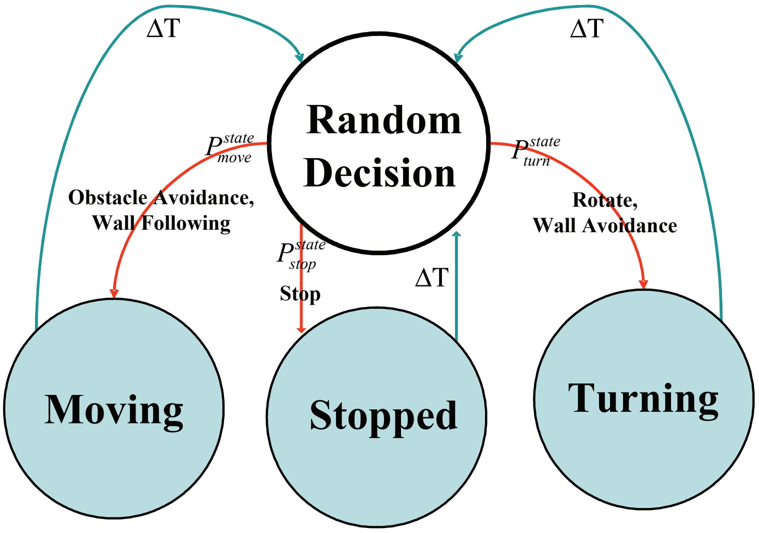


Figure 2.10: The state-machine for generation of aggregation in the mixed-society

1. *Obstacle-avoidance* behavior: to move forward while avoiding the obstacles on its way
2. *turn-in-place* behavior: to change movement direction, or
3. *Stop* behavior: to get immobile

Selecting these behaviors changes the robot's state to *moving*, *turning*, or *stopped*, respectively. The turning angle for *turn-in-place* behavior is drawn from a pre-specified angle range, with uniform random distribution.

A probability table specifies the chance of each behavior to get selected among the possible mentioned behaviors in a specific state. Entries of the table assign probability to behaviors (move, stop, or turn), based on the position of the robot (periphery or center), shelter type (dark, bright, no shelter), and number of cockroaches, and/or robots around the deciding robot (0, 1, 2, 3, ≥4 in our case). The table thus has totally 3×5+3×3×5=15+45=60 entries, 15 entries for the cases where the robot is moving along the periphery and 45 entries for the cases where the robot is moving in the center. The entries of the probability table and the other parameters have been extracted by extensive statistical data gathering on real cockroaches using a tracking software [58; 59].

In some cases, the robot has to terminate the currently running behavior, earlier than usual, and switch to the initial state. For instance, when a robot that is moving in the center encounters the periphery, or when a robot that is making a small turn accomplish the task sooner than ΔT.

2.5 Results

This section presents performance analyze of the sensor fusion algorithms and a summary of the outcomes of the biological experiments in mixed-society of InsBots and cockroaches.

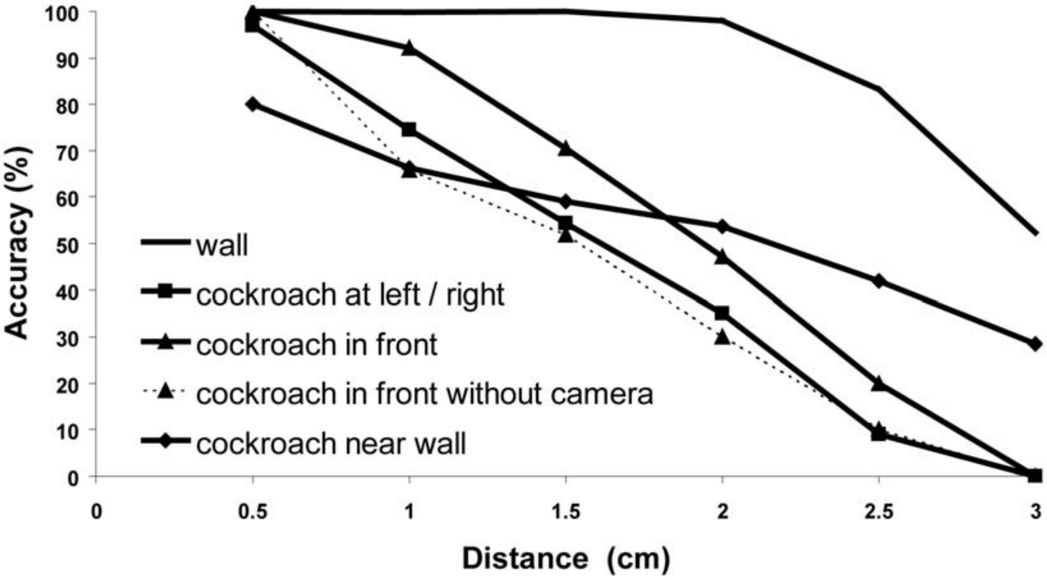


Figure 2.11: Cockroach/wall detection accuracy vs. robot-cockroach/wall distance

2.5.1 Performance of the Perceptual Schema

Cockroach and Wall Detection

Fig. 2.11 displays the accuracy of the cockroach and wall perception methods. These results were obtained by manually analyzing 900 different situations, extracted from a movie taken by the overhead camera, and linking them to the information provided by the wireless communication interface. For cockroach detection, the distance is measured from the body-borders of the robot to the closest point of the cockroach's body (excluding legs and antennas).

The graphs confirm that, cockroaches are visible from 2.5 cm but, optimal detection is reached only when they are very close to the robot. The dashed curve corresponds to the detection accuracy, based only on the IR sensors (without using the linear camera). It is clear that, accuracy in this side is close to the left and right side. However, after introducing the linear camera, performance in the front side is enhanced around 20%.

The graphs also shows that the wall, thanks to its reflective material, is detectable from further distances and with more accuracy than the cockroaches. The rather poor performance of the cockroach-detection, even at short distances, could emanate from several facts:

- Certain parts of the cockroach's body are less visible than the others. Its round head reflects IR signals appropriately, whereas the rear of its body, that is composed of thin horizontal wings, disperses IR signals. Hence, the robot detects the approaching cockroaches better, since its head appears first.

- Some positions around the robot are not entirely covered by the IR sensors, as depicted on Fig. 2.12. The dashed shapes indicate the regions where the cockroaches initially activate the IR sensors (from 2.5 cm), and the solid shapes indicate the regions where the cockroaches are effectively detected by the perception algorithms (from 1.5 cm).

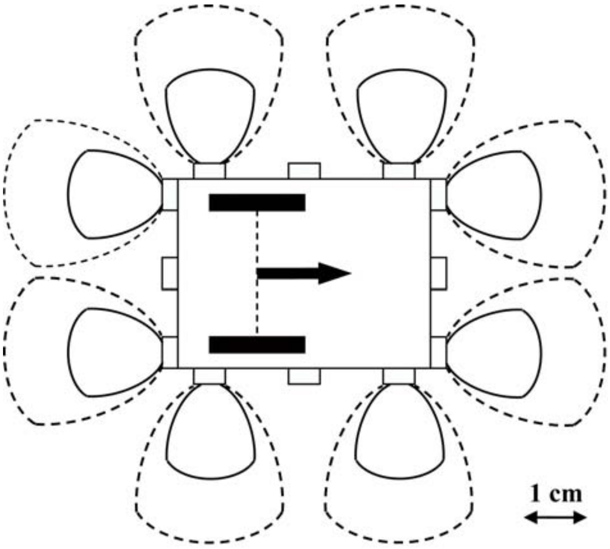


Figure 2.12: Sensory coverage around robot: Dashed shapes: indicate the regions where cockroaches start to activate the IR sensors (from 2.5 cm). Solid shapes: indicate the regions where cockroaches are effectively detected by the perception algorithm (from 1.5 cm).

Robot Detection

Robot detection mainly depends on the reliability of the local communication protocol. This is rather difficult to characterize because, it depends on several factors. The communication rate between two robots depends on the distance between them; higher rates are obtained when robots are closer to each other. Further, it also depends on the relative orientation of the robots; the more the overlap between the IR emitting and receiving cones is, the higher the communication rate is.

Fig. 2.13 shows performance of the local communication protocol. The left sub-figure shows communication rate between two robots. A robot is placed at different distances (granularity: 2cm) and with different orientations (granularity: 45°) to a fixed robot (dark square in the center of the picture). For each position, the number of received messages (8 bits including 2 stop bits) during a 30 s period is recorded. The right sub-figure shows the percentage of correctly received messages out of the set of all received ones.

The maximum achieved communication rate is 7 Hz. It is achieved when the sending and receiving sensors are nearby and face to face. These results are rather good; even in situations where communication is difficult (darker spots in Fig. 2.13-left), the percentage of correct messages is rather high (between 70-100%).

Because of the difference between length and width of the robot, communication range in left/right sides is higher than front/back sides. That is why the pixels of Fig. 2.13 form an ellipse, rather than a circle. The maximum possible distance between the robots is 22.5 cm for left/right sides and 15 cm for front/back sides. To constrain detection range of the robot to short distances, the data received via the local communication is combined with the IR proximity values, and only the robots that are in vicinity of the current robot are counted.

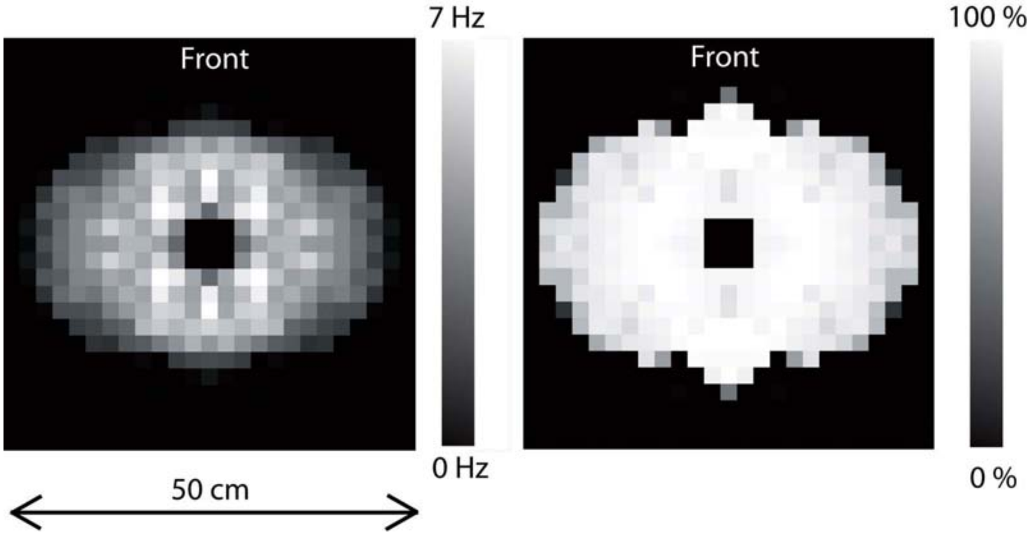


Figure 2.13: Performance of Local communication between two robots. Left: communication rate. Right: success rate

2.5.2 Performance of the Motor Schema

To show the reusability and the extensibility of the architecture, similar architectures were implemented on both Alice [64] and InsBot [62] micro-robots. The architecture was first implemented for Alice and was extended then to work for InsBot, which has more sensors. However, we had only to modify the weights of the behaviors and the perceptual schema. The hierarchy of the architecture remained untouched. Experiments with the Alice micro-robots were done in pure-robot groups, but the InsBots were tested in both pure-robot and robot-cockroach groups.

Some parts of the behavioral architecture have been re-used successfully in other projects by other people, e.g. inspection of jet turbines by a swarm of Alice robots [65], and a realistic computer simulation of mixed-societies [66].

Collective Robotics Experiments

Statistical analysis of the Alice behavior shows that most of the behavioral parameters of the Alice robot are very close to the corresponding cockroach parameters [67]. Moreover, preference for a given type of environment heterogeneities (e.g. preference for dark places in our case) can lead the group of the robots to a collective choice for an aggregation site [68].

Interestingly, when the robots are exposed to homogeneous shelters^2 with different sizes, the group of the robots are able to sense, somehow, the size of the shelters and choose the biggest one, while the robots are not equipped with any sensor for measuring the size of the shelters [68; 69]. This is an emerging consequence of the social amplification.

Table 2.1: Probability to leave the bright and the dark shelters for alone cockroaches and InsBots.

	Probability to leave (s^{-1})	
	Bright Shelter	Dark Shelter
Cockroach	0.030	0.006
InsBot	0.021	0.008

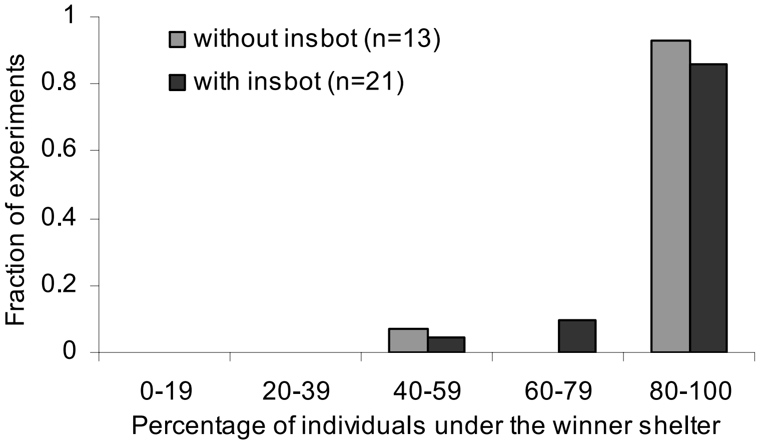


Figure 2.14: Collective choice between homogeneous shelters for pure cockroach and mixed cockroach-robot groups. ©ULB

Mixed-Society Experiments

Biological experiments[3] show that the movement pattern of the robots is similar to the ones of cockroaches. The mean speed of the robot (17.56±2.57 mm/s) is close to the one of cockroaches (19.04±2.5 mm/s) during their *calm* phase [4]. Moreover, cockroaches and InsBots have similar probabilities to leave a bright or a dark shelter (Table 2.1). The probability of leaving a shelter decreases when the shelter is darker. It also decreases when the number of neighbors increases.

Biological experiments also show that, the mixed-society of InsBots and cockroaches generate similar aggregation patterns to pure-cockroach groups. In more than 85% of the experiments, both pure cockroach and mixed groups choose only one shelter, out of two identical shelters (Fig. 2.14).

Moreover, heterogeneous shelter experiments, in around 60% of the trials, end up with selection of the dark shelter, and around 25% with the selection of the bright one, in both pure cockroach and mixed groups (Fig. 2.15). Around 15% of the experiments are finished with no clear choice between the shelters.

---

[2] shelters are built with same materials, therefore create shadows with same luminosity

[3] We can not present the detailed results of the biological experiments, instead a summary of the results is explained. The detailed results will be submitted to biological journals by other team members.

[4] During an experiment (3 hours), the mean speed of a cockroach is characterized by two distinct phases. During the first 90 minutes, it has a very mobile phase during which, it explore the experimental setup. A less mobile phase, called *calm* phase, starts 90 minutes after.

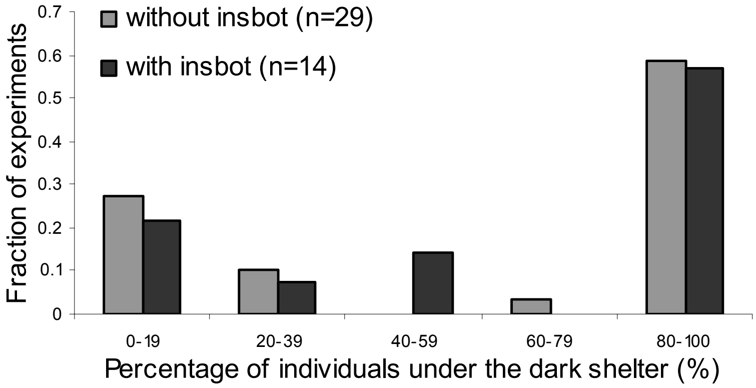


Figure 2.15: Collective choice between homogeneous shelters for pure cockroach and mixed cockroach-robot groups. ©ULB

Biological experiments, also, show that the aggregation site is selected collectively by both InsBots and cockroaches. In fact, both of them modulate each other's behavior. InsBots are found more often under the shelter that contains most of the cockroaches. Cockroaches are found more often under the shelter that contains most of the InsBots.

It is proven that the chemical marking is necessary for the InsBots to get accepted by the cockroaches as a congener [70]. The mean contact time between cockroaches and marked-robots is very similar to the encounters between cockroaches themselves.

The most important results proves that, robots are able to induce a new collective pattern and modify the collective behavior of cockroaches. InsBots in this case are programmed to prefer the bright shelter, contrary to the cockroaches, simply by swapping the entries of their probability tables. Under this conditions, the bright shelter is selected by the mixed groups in 59% of the experiments, while it was selected, before modification of the robots, only in 38% of the experiments (Fig. 2.16). Despite the individual preference of the robots, the cockroaches are able to socially drive the robots to the dark shelter in 32% of the trials.

The detailed results will be published by the group of Prof. Jean-Louis Deneubourg in Université Libre de Bruxelles, Belgium.

2.6 Conclusion and Future Works

Details of the hardware and behavior implementation of a miniature robot, InsBot, are explained in this chapter. Due to the limitations in the size of the robot, and the necessary long-time autonomy required in biological experiments with cockroaches, the hardware parts and the behavioral algorithms had to be optimized extensively.

The robots are controlled via a hierarchical behavior-based approach. Behaviors consist of a perceptual schema and a motor schema. The perceptual schema (sensor-fusion methods), associated with the heuristic rules that come from our knowledge about the experimental setup, allows the robots to well discriminate the different objects in the environment. Cockroaches and walls are detected using the IR proximity sensors, emplaced in different heights around the robot. To enhance the cockroach-detection procedure, a linear camera is added to the front side of the robot. Also, for discrimination of robots from cockroaches and wall, a simple local-range

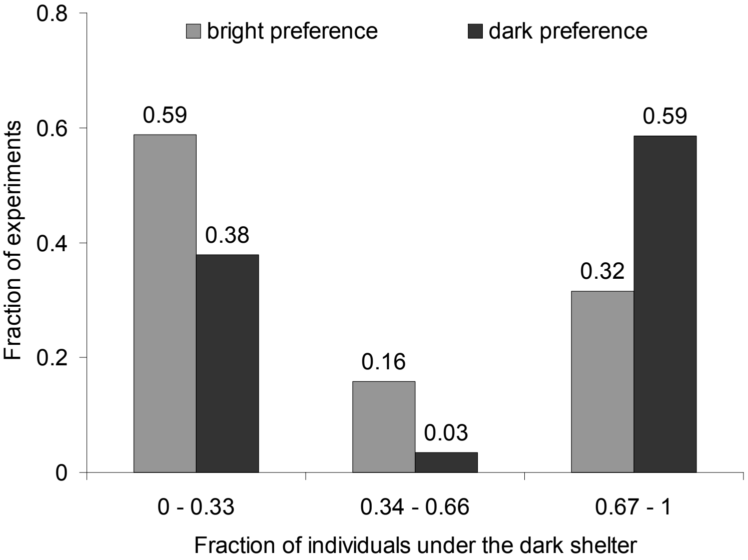


Figure 2.16: Collective choice between heterogeneous shelters in a mixed group, when preference of the robots are intentionally modified. ©ULB

communication protocol through IR sensors is established.

The motor schema takes the output of the perceptual schema and produces a pattern of motor activity. The layers of the motor schema start from some primitive behaviors at the lowest layer. The behaviors in the higher layers combine the behaviors in the lower layers and build new (more complex) behaviors. At the highest layer, the aggregation model of the cockroaches is implemented in the form of a probabilistic state-machine.

The designed hierarchy is extensible through adding more behavior layers. However it needs, like other behavior-based systems [71; 11], a high degree of hand design. It consequently becomes intractable for large scale problems [72]. Also, the major criticisms against the behavior-based approach, i.e. lack of facilities for explicit system level control and planning [73], is still an open issue here.

Biological experiments shows that, the robots are accepted by the groups of cockroaches. Moreover, the mixed-society of robots and cockroaches have statistically close characteristics to a pure-cockroach society. Also, the robots are able to control the mixed-society and induce new collective patterns.

More investigation should be carried out to completely solve the emerged problems. Using local communication for robot detection introduced noise on the sensors of the adjacent robots. This mainly rises from the periodical “hello/wakeup” signals that the robots emit to declare their presence. This noise initiates an abrupt change in the value of the IR sensors, and as a consequence diminish the performance of the perceptual schema. We must work on appropriate filters to reduce it.

The initial tests were done with cockroaches however, more experiments will be carried out with other type of animals, like sheep and chicken, in order to develop and verify a methodology for mixed-society control.

Part II

Flat Learning

Chapter 3

Compact Q-Learning

Masoud Asadpour and **Roland Siegwart**

Published in:

Proceedings of the First European Conference on Mobile Robots [ECMR], 2003 [74]

Journal of Robotics and Autonomous Systems, 2004 [75]

Abstract. Scaling down robots to miniature size introduces many new challenges including memory and program-size limitations, low processor-performance and low power-autonomy. In this chapter, we describe the concept and implementation of learning and combining multiple behaviors with the autonomous micro-robot, Alice. We propose a simplified and compact Reinforcement Learning (RL) algorithm based on one-step Q-learning, that is optimized in speed and memory consumption. This algorithm uses only integer-based sum operators, and avoids floating-point and multiplication. Performance of the method is tested on safe-wandering and light-attraction behaviors. Finally, the safe-wandering and the light-attraction behaviors are combined and the robot learns light-search behavior.

Key words: Reinforcement Learning, Q-Learning, micro-robots.

3.1 Introduction

Small-sized robots are suitable tools to study biology [67; 62]. Small mobile machines could one day perform noninvasive microsurgery [76], miniaturized rovers [77] could greatly reduce the cost of planetary missions, and tiny surveillance vehicles [78] could carry equipment undetected.

Miniaturizing robots introduces many problems in hardware modules and behavior implementations [79]. The robot parts must have low power-consumption. This fact compels the designers to add modules like sensors, conservatively. Due to the simplicity of hardware modules, then, the control program must handle all un-handled tasks, such as noise filtering. Moreover, the instruction-set of the processors is reduced. Behaviors of the robot must be coded compactly and efficiently, due to the limitations on program-size, and memory and processing-speed. Additionally, because of the limited power-autonomy, long tasks such as learning, confront with serious restrictions.

In this chapter, we describe how to practically tackle on-line learning problems on micro-robots with processing constraints. We reveal the problems that arise when modifying the one-step Q-learning algorithm, in order to fit it to the micro-robot, Alice [64]. As a result, an optimized algorithm in terms of size, processing time, and memory consumption, called **Compact Q-Learning** is introduced. It is designed on the basis of integer-calculation and low-level micro-processor instructions. The Compact Q-Learning is verified on learning of *safe-wandering*(obstacle avoidance) and *light-attraction* behaviors.

We also test the feasibility of behavior-combination using the Compact Q-Learning algorithm. We know that, learning complex behaviors requires often to combine outputs of different -and sometimes competing- behaviors. In this chapter, a simple method to learn and combine *light-attraction* and *safe-wandering* behaviors, in parallel, and create the new behavior, *light-search*, is tested.

This chapter is organized as follows. The next section deals with the previous works in micro-robots and RL. The third section explains the rising problems for floating-point based Q-learning and introduces the Compact Q-Learning algorithm. Then, learning of safe-wandering and light-attraction is described, and experimental results are demonstrated. Next, performance of the Compact Q-Learning is compared to the floating-point based Q-Learning. Then, a simple method to learn different behaviors in parallel, and combine them and create new behaviors is explained. Finally, conclusions and future works are discussed.

3.2 Related Works

Processing power of a micro-robot, which is related to its available energy, scales down by L^2 factor (where L is length) [79]. This fact drastically limits the capacity of control algorithms. As a consequence, the robot designers are compelled to reduce the calculation-power in order to increase the energy autonomy, e.g. by using 8-bit instead of 16 or 32-bit micro-controllers. Therefore, intelligence of micro-robots are somehow limited. Nevertheless, small robots, assisted by external supervisors (computer, human), appropriate collective approaches, or suitable simplification in control program, might still be able to fulfill complex learning tasks.

Researchers in micro-robots have implemented different learning algorithms. Floreano et al. [80] used evolutionary algorithms, in combination with spike neurons, to train the old version of the Alice micro-robots for safe-wandering. The spiking neural networks are encoded into genetic strings, and the population is evolved. Crossover, mutation and fitness evaluation procedures are optimized and use bitwise operators. But, since no learning is done during fitness evaluation of a newly generated individual, the training task requires a long time (3 hours), even for such a simple task.

Dean et al. [81] applied ROLNNET (Rapid Output Learning Neural Network with Eligibility Traces) neural networks to mini-robots for backing a car with trailers. ROLNNET [82] is a mixture of Neural Networks and RL. It has been designed for real robots with very limited computing power and memory. Input and output spaces are divided into discrete regions, and a single neuron is assigned to each region. Neurons are provided with regional sensitivity through using eligibility traces. Response learning takes place rapidly using cooperation among neighbor neurons. Although the mathematical formulation is simple (It consists only of summation, multiplication and division operations), the required floating-point operations might still cause processing problem on autonomous micro-robots with limited processing capabilities.

Various implementations of micro-robots have been developed for different applications. Among them we can point to Sandia MARV [83], MIT Ants [84], Nagoya MARS [85], KAIST Kity [86] and ULB Meloe [87]. However, to our best knowledge, no implementation of learning has been reported on them.

3.2.1 Reinforcement Learning

RL [18] is one of the widely used online-learning methods in robotics. In this method, the robot learns during action and acts during learning. This facility is absent in supervised learning methods. In RL, the learner perceives the state of the environment (or feature at higher-levels) and selects an action (or behavior). This action changes the environment's state, and the agent receives a feedback signal from the environment or a critic, called *reinforcement signal.* The signal indicates how much the selected action is appropriate in that situation. It can be either positive (that is referred to as *reward*) or negative (that is called *punishment*). The agent then updates the learned policy, based on the sign and magnitude of the signal.

The RL formulation that we use in this work is the *one-step Q-learning* method [88; 89]. The algorithm however, should be adapted to the limitations of our robots. In the *one-step Q-learning*, the external world is modeled as a *Markov Decision Process (MDP)* with *discrete* and *finite* states. Immediately after each action, the agent receives a scalar reinforcement signal.

The learned policy of the agent is stored in a table, called *Q-table.* Cells of the Q-table are *action-value functions* (called hereafter *Q-values*), which estimate the long-term discounted reward of the state-action pairs. Given the current state x, and the available actions, a_i , $i = 1 \dots |A|$, where A is the set of actions, a Q-learning agent selects an action, a, with the probability, P, given by the Boltzmann probability distribution:

$$P(a_i|x) = \frac{e^{Q(x,a_i)/\tau}}{\sum_{k=1}^{|A|} e^{Q(x,a_k)/\tau}} \tag{3.1}$$

where τ is a temperature parameter that adjusts randomness of action-selection. Bigger τ values give more randomness to selection.

The Boltzmann selection is able to adjust *exploration* and *exploitation* rates. At the beginning, action-values are almost equal. Therefore, selection of actions is close to uniform random selection (more exploration). As learning continues, some action-values might increase. Those actions are, then, selected more frequent than the others (more exploitation).

After selecting an action, the agent executes it, receives an immediate reward r, moves to the next state y, and updates $Q(x,a)$ as following:

$$Q(x,a) \leftarrow (1-\alpha)Q(x,a) + \alpha(r + \gamma V(y)) \tag{3.2}$$

where α $(0 \leq \alpha \leq 1)$ is the learning rate, r is the reinforcement signal that is received from the environment, and γ $(0 \leq \gamma \leq 1)$ is the discount factor, which determines the present value of future rewards. If the discount factor is zero, the agent is concerned with maximizing immediate rewards. As the discount factor approaches one, the agent takes future rewards more into account.

Table 3.1: Comparing the floating-point with the integer operations on Alice micro-robots

Floating-point operator	**number of instructions**	**Call overhead**	**% of memory for the first call**	**Execution time (μs)**
+,-	322*	26	4.25	154
×	119	25	1.76	613
/	204	25	2.8	1121
total (only for one call to each operator)			**8.81**	**1888**
8-bit integer operator	**number of instructions**	**Call overhead**	**% of memory for the first call**	**Execution time (μs)**
+,-	3	0	0.04	3
×	37	7	0.54	43
/	21	7	0.34	87
total (only for one call to each operator)			**0.92**	**133**

*Float numbers are represented in mantissa-exponent form. The generated code for float + is larger than × and / since in + the second operand is changed to have the same exponent as the first one.

$V(y)$ estimates *state-value* of y, which is the long-term discounted reward, if the agent starts from state, y, and select the next actions according to the current policy. It is estimated by the currently best action-value:

$$V(y) = \max_{b \in A} Q(y,b) \tag{3.3}$$

If the state and action spaces are finite, the agent visits every possible pair (x,a) infinitely often, the immediate rewards are bounded, and the learning rate is decayed over time so that $\sum_{i=t}^{\infty} \alpha_i = \infty$ and $\sum_{i=t}^{\infty} \alpha_i^2 < \infty$, Q-learning is proved to converge, eventually [90; 91].

3.3 The Compact Q-Learning Algorithm

The existing RL algorithms are not limited to specific type of operations. Numbers are floating-point, or at least, floating-point operations are employed. However, in order to implement Q-learning on the micro-robot Alice, we need a simplified algorithm that is able to cope with its limited memory, processing resources, and power autonomy. Thus, we propose a new algorithm based only on integer operations.

3.3.1 Integer vs. floating point operators

Floating-point operations take too much processing time and program memory. For the sake of comparison, the number of instructions generated by our C compiler [1] and the average execution time for 4 floating-point operations: $a=b+c$, $a=b-c$, $a=b*c$, and $a=b/c$ is listed in Table 3.1. They are compared to the integer-based operations.

---

[1] PCW® Compiler, from Custom Computer Services Inc.

Every instance of the operators in the program (except for integer sum) is accompanied actually with a call overhead for preparing the registers, and copying the results back to the memory. Call overheads include both processing time and program memory. Functions fill program memory once, but take processing time in every call. Therefore, we prefer to use only integer-sum operators, since they have no call overhead, require fewer instructions, and run very fast. Moreover, we prefer to use unsigned numbers to save memory bits, facilitate computations, and reduce overflow/underflow checking.

3.3.2 Q-Learning problems with Integer operators

Proofs of convergence for RL algorithms are valid when numbers are real. In this section, we discuss some problems that happen when switching to integer numbers and operations.

The first problem appears in the Boltzmann probability distribution (3.1). Because of computing powers of e. This formula is time-intensive. It needs also a float-type memory cell (at least 4 bytes) to be assigned to the probability of each action, since they will be used then to select an action accordingly. So, the action selection mechanism needs revision.

Assume at the beginning, Q cells are initialized to c. Since the Q-values must be of type integer, they must be incremented or decremented by one (not a fraction). Applying these conditions to (3.2) and assuming $\alpha = \frac{m}{n}$, $\gamma = \frac{p}{q}$, where m, n, p, and q are positive integer numbers, it is straightforward to show that value of the reward at the beginning must be at least:

$$\left\lceil \frac{n}{m} + \frac{c(q-p)}{q} \right\rceil = \left\lceil \frac{1}{\alpha} + c(1-\gamma) \right\rceil \tag{3.4}$$

to affect the new Q-value. Otherwise, the table remains unchanged and learning task will not converge. If the learning rate is fixed to e.g. 0.1, the reward must be at least 10. Moreover, value of the reward should be increased according to Q-value increments (see the c factor in (3.4)). This implies that the reward might need too many bits.

Furthermore, some state-actions are not accompanied with any reward, but lead the learner to the states in the vicinity of the goal states. Responsibility of the future reward term $\gamma V(y)$ in (3.2) is to augment the value of such state-actions. It can be shown that the value of $V(y)$ must be at least:

$$\left\lceil \frac{q}{p}(c + \frac{n}{m}) \right\rceil = \left\lceil \frac{1}{\gamma}(c + \frac{1}{\alpha}) \right\rceil \tag{3.5}$$

for example if $\alpha = 0.1$, $\gamma = 0.9$, and $c = Q(x,a) = 0$, then $V(y)$ must be at least 12 to increment the Q-value by one i.e. it takes a long time to build action chains from initial states to goal states.

Furthermore, if we take $V(y)$ out and consider only the rewards around zero, or punishments, Q-values will decrease by time, because of the integer division operator[2] i.e.:

$$Q(x,a) \leftarrow (1 - \frac{m}{n})Q(x,a) + \frac{m}{n}r \approx \left\lfloor (1 - \frac{m}{n})Q(x,a) \right\rfloor \leq Q(x,a) - 1 \tag{3.6}$$

In these cases, however, it is more appropriate to leave the Q-value unchanged.

Also, assume after some learning episodes, the value of a cell in the Q-table increases up to a large integer value. Since, Q-values are incremented at least by one, this circumstance easily happens. Therefore, the Q-value of the previous state-action would increase by a large number.

---

[2] Remember that e.g. 9/5 =1 in integer division

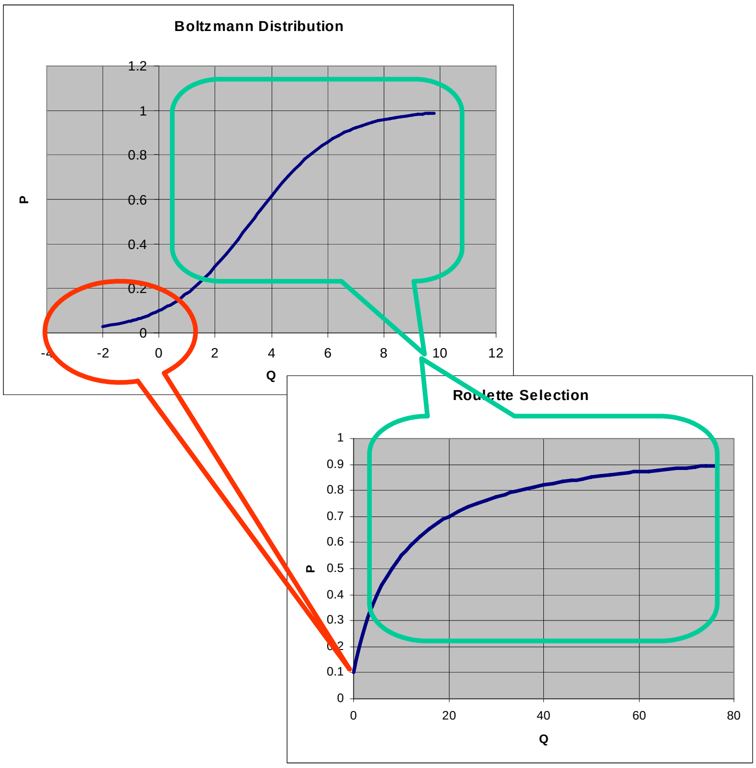


Figure 3.1: Simplification of Boltzmann Selection to Roulette-Wheel Selection

This results in overflow of all action-values in the transition history, quickly. Then, even a big punishment would not decline the Q-values.

3.3.3 The proposed algorithm

Considering the problems, described in the previous section, we propose a very simple algorithm that requires only unsigned integer summation. We assume that Q-values are unsigned integers.

The action probability assignment is changed to *Roulette-Wheel Selection* [92] as the following:

$$P(a_i|x) = \frac{Q(x,a_i)+1}{|A|+\sum_{k=1}^{|A|} Q(x,a_k)} \tag{3.7}$$

Q-values are summed by one so that, zero-valued actions have still a small chance to be selected. Roulette Selection method has been widely used in the Genetic Algorithm.

The idea behind the simplification could be illustrated by an example. Assume that, the action set includes 10 actions, nine of them have Q-values fixed to zero –at a specific state– and Q-value of the last one varies. The assigned probabilities by the two selection methods to the last action is shown in Fig. 3.1. Note that, for negative Q-values the Boltzmann function assigns small probabilities around 0.1. The Roulette selection, although dealing only with non-negative numbers (and therefore saving one bit of sign-bit), assigns them 0.1 probability as well. For positive Q-values, comparing the shape of the two curves shows both of them are log-incremental.

The Roulette-Wheel Selection is capable of adjusting exploration and exploitation rates during the learning process. At the beginning, randomness of selection is high, since all Q-values are nearly equal. But as learning continues, some Q-values might become large. Hence, the summation term in the divisor of (3.7) would also become large. Thus, selection probability of non-efficient actions declines to zero, and exploitation rate augments.

In implementation, the action selection mechanism could be optimized even more, in terms of processing time and memory. The optimized procedure is as follows: First, a uniform random number between 0 and $|A| + \sum_{k=1}^{|A|} Q(x, a_k)$ is generated. The drawn random number is then compared to the partial sum $\sum_{k=1}^{i} (Q(x, a_k) + 1)$ in a loop, starting from $i = 1$. The first action for which, the random number is less than or equal to the partial sum, is selected. This way, only summation operator is used. Also, it is not required to temporarily save a chain of probabilities.

The policy update formula is changed to:

$$Q(x, a) \leftarrow Q(x, a) + r + \nu(y) \tag{3.8}$$

where r is the immediate (positive or negative) reinforcement signal, and $\nu(y)$ is the *discount function* –like $\gamma V(y)$– that depends on the value of the state y, and could be implemented via conditional operators (or a lookup table), without using multiplication (e.g. $\textsf{if}\ (V(y) > 32)\ \textsf{then}\ \nu(y) = 1\ \textsf{else}\ \ldots$). Using the ν function limits the effect of large Q-values, discussed earlier in Sec. 3.3.2.

The reinforcement signal r could be defined as:

$$r = \begin{cases} 1 & \text{reward} \\ 0 & \text{don't care} \\ -1 & \text{punishment} \end{cases} \tag{3.9}$$

If so, it can be handled by increment and decrement operators, which exists usually in the instruction set of all micro-processors. In order to work with unsigned numbers, reinforcement signals could be shifted one unit up, i.e. to 2, 1, and 0, respectively, and then decremented one unit when adding to Q-value.

A drawback of the proposed algorithm is the lack of the learning rate, but we do not have any better choice, since each reward must result in increment of the Q-value, as described previously. In order to decrease negative effects of missing the learning rate, we have to scale down r and ν as much as possible.

3.4 The Alice Micro-Robot

The Compact Q-Learning Algorithm has been tested for light-attraction and safe-wandering tasks with the micro-robot Alice. Alice (Fig. 3.2) is one of the smallest (22 × 21 × 20 mm) and lightest (5 gr) autonomous mobile robots in the world [64]. In its basic configuration it is equipped with a PIC16F877® micro-controller[3]. It has two bi-directional watch motors for locomotion (up to 40 mm/s), four active Infra-Red (IR) proximity sensors, a NiMH rechargeable battery, and an IR TV remote receiver for communication. All equipments are mounted on a PCB and folded in a plastic frame. Its power consumption has been highly optimized to 12-17 mW, providing high

---

[3] from Microchip® Corporation

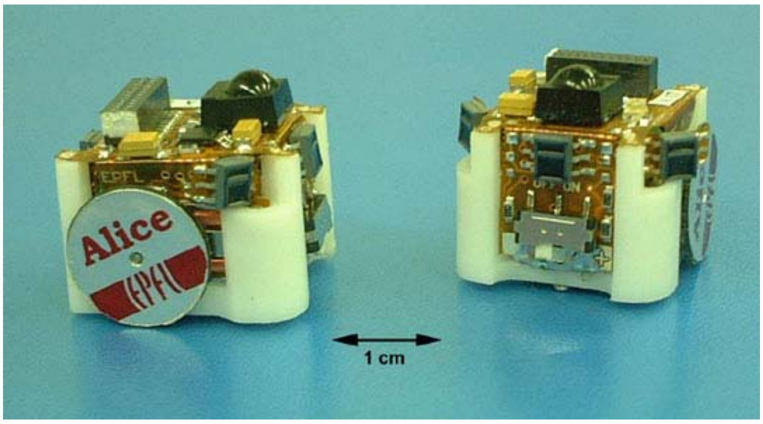


Figure 3.2: The Alice micro-robot

energetic autonomy, up to 10 hours. The autonomy can be increased by adding an extra battery, if required. This makes the robot suitable for collective robotics and learning experiments.

Alice is a programmable and modular robot and a number of modules can be added to it such as, linear camera, wireless radio or IR communication, touch sensor, interface to personal computers, and automatic recharging pack. Thanks to its programmability and interface with personal computers, it has been used in various research [64; 93; 65] and educational projects [94; 95].

The micro-processor is an 8 bit RISC (Reduced Instruction Set Computer) micro-controller. It has only 35 basic instructions, including bit-wise sum, and conditional and unconditional jumps. It has 8K×14-bit Words of flash program memory and 368×8-bit RAM data memory. It directly drives two low-power watch motors through 6 pins, and reads the value of the proximity sensors through A/D converters. To save energy, its clock speed is set to 4 MHz. Each instruction takes four clock-cycles, so each instruction takes $1\mu s$ (except for jumps which require $2\mu s$).

A reasonable portion of the memory is assigned to management of the sensors and the motors. The software core is composed of a simple operating system, which handles different real-time tasks:

1. Communication with the TV remote-control, to receive commands from an operator or an external computer. Currently, four commands for supervision of the learning process are available:
    - *Start:* Initialize the Q-table (with fixed or random values).
    - *Stop:* Pause the learning process, and demand the robot to behave according to the learned policy. The robot then selects greedy actions (actions with maximum Q-values) in each state.
    - *Save:* If the learned behavior is satisfactory, the corresponding policy could be saved into the EEPROM of the robot so that, it could be retrieved for future use, and
    - *Load:* The saved policy could be resumed, afterwards. This facility is useful when the learning task is too long to be completed within autonomy cycle of one battery.
2. Reading the proximity sensors, and updating the current state,

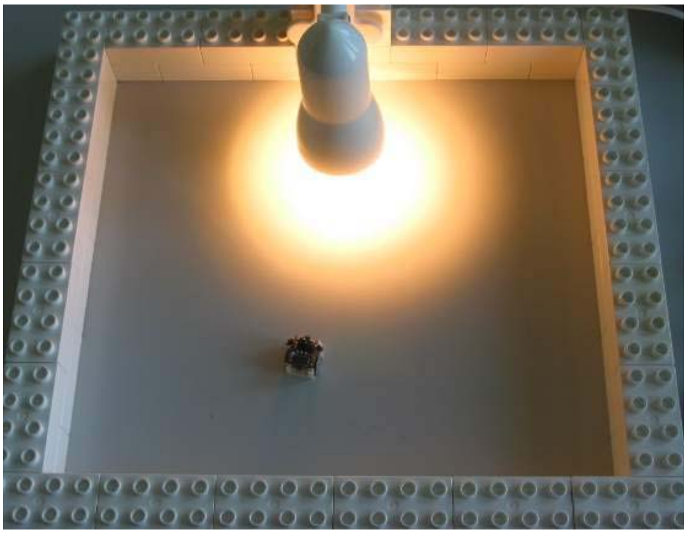


Figure 3.3: Light attraction task

3. Selecting the next action,
4. Controlling the motors to execute the selected action, appropriately. The motors are controlled by placing delays between their six operational phases.
5. Computing the reward function, and updating the Q-table accordingly, and
6. Computing performance measures, and writing them to the EEPROM for later analyze.

3.5 Learning Light-Attraction Behavior

Performance of the simplified algorithm has been tested on different tasks. In the simplest task, Alice learns the *light-attraction* behavior. The goal is to follow a moving light source, which is displaced manually around a simple rectangular maze (Fig. 3.3). The IR sensors are used to measure the ambient light on different sides of the robot. No obstacle is placed in the robot's path at this step. Learning cycle is repeated every 200 ms (5 Hz). Every 15 s, the sum of the received rewards, from the previous period till now, is recorded in the EEPROM.

The current state is specified by the most illuminated side (totally 4 states). The action set consists of three actions: {*move-forward, turn-right, turn-left*} with the fastest speed. The robot is rewarded when it is faced toward the light source, and receives punishment in the other cases. The reinforcement signal is given by a manually programmed function. Cells of the Q-table are one-byte unsigned integers and their value ranges from 0 to 240. The reason behind setting the maximum to 240 is to avoid overflows in addition or subtract operations.

The discount function, ν, is defined as the following:

$$\nu(y) = \begin{cases} 2 & 64 \leq V(y) \\ 1 & 32 \leq V(y) < 64 \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

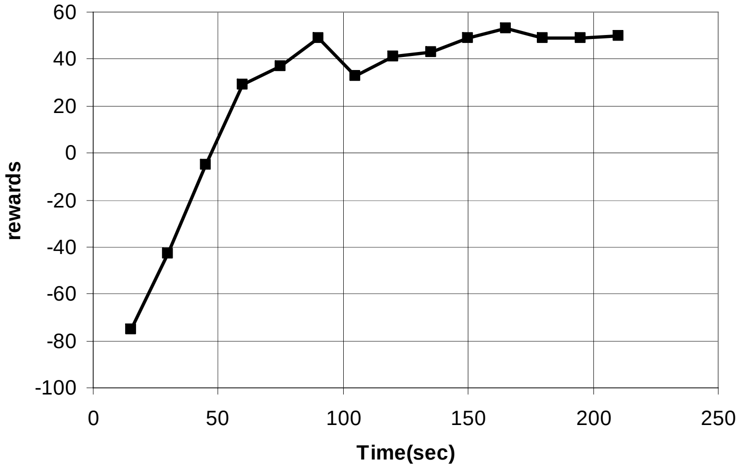


Figure 3.4: History of the received rewards during every 15 s of light-attraction learning

$V(y)$ is calculated according to 3.3.

Fig. 3.4 shows an example of the effectiveness of the algorithm. Each point in the figure shows the sum of the received rewards by the robot within every 15 s of the experiment. It needs less than three minutes to converge. The learned behavior is comparable to our hand-coded program.

3.6 Learning Safe-Wandering Behavior

The next learning task is *safe-wandering* in the two cross and H-shape mazes, shown in Figs. 3.5 and 3.7. The H-maze is a narrow labyrinth and has a complex shape. The cross-maze is simpler and its walls are at farther distance to each other. The robot's task is to wander in the maze, as straight as possible, without bumping into the walls.

The current state is defined based on the value of the four IR proximity sensor at *front*, *front-left*, *front-right* and *rear* side of the robot. Each sensor corresponds to one bit of the state, and shows presence or absence of an obstacle at its pointing direction. The number of states is 16, but it can be reduced to 15, since no situation where obstacles surround the robot from all sides occurs. The action set again includes three actions: {*move-forward, turn-right, turn-left*}. The Q-table is thus 15 × 3 = 45 bytes totally. The reward function is defined as the following:

$$r = \begin{cases} 1 & \text{forward move,} \\ -1 & \text{hitting the walls,} \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

Hitting the walls is defined as "getting closer than 0.5 cm to the walls". A critic, programmed as a procedure in the program of the robot, compares the values of the IR proximity sensors with a threshold, and sends the appropriate reinforcement signal to the learning procedure, immediately after execution of actions.

A sample of the learned safe-wandering behavior in the cross-maze is shown in Fig. 3.5. The robot has learned to avoid obstacles and move straight forward efficiently. Fig. 3.6 shows the

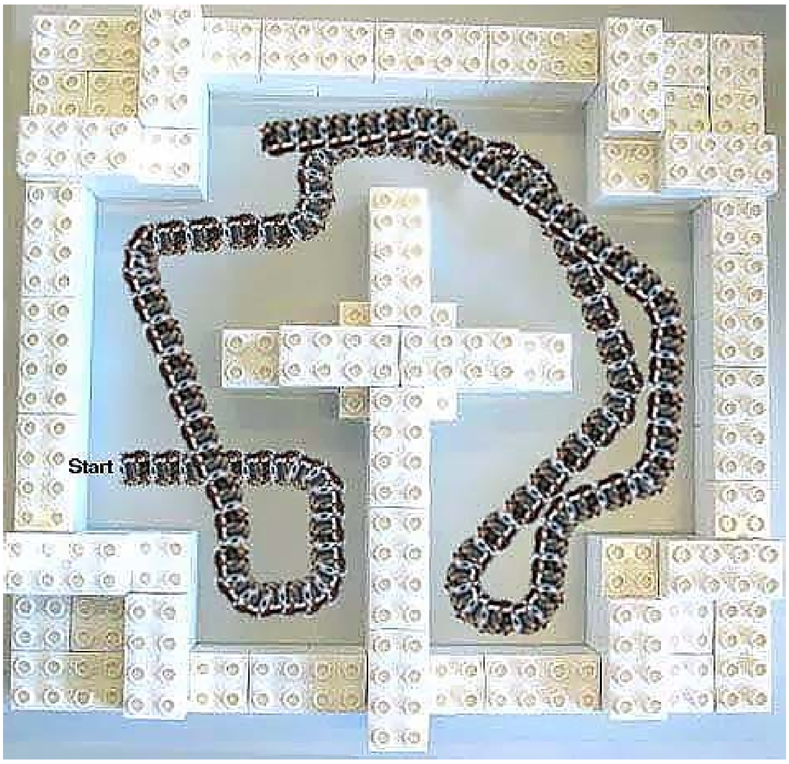


Figure 3.5: A sample of the learned safe-wandering behavior in the cross-maze

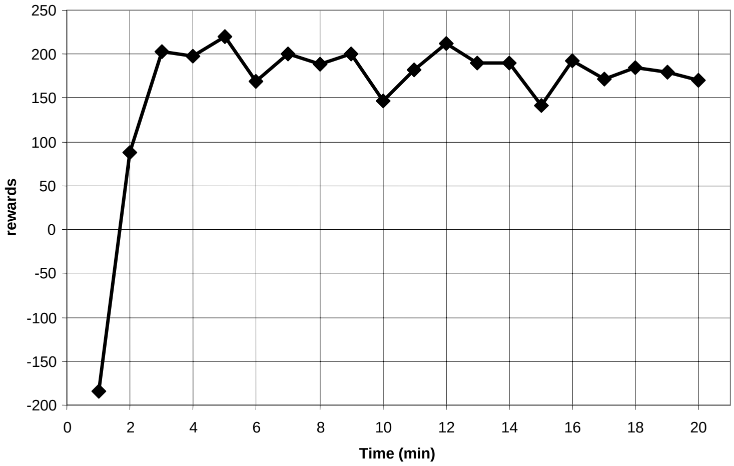


Figure 3.6: History of the received rewards during safe-wandering in the cross-maze

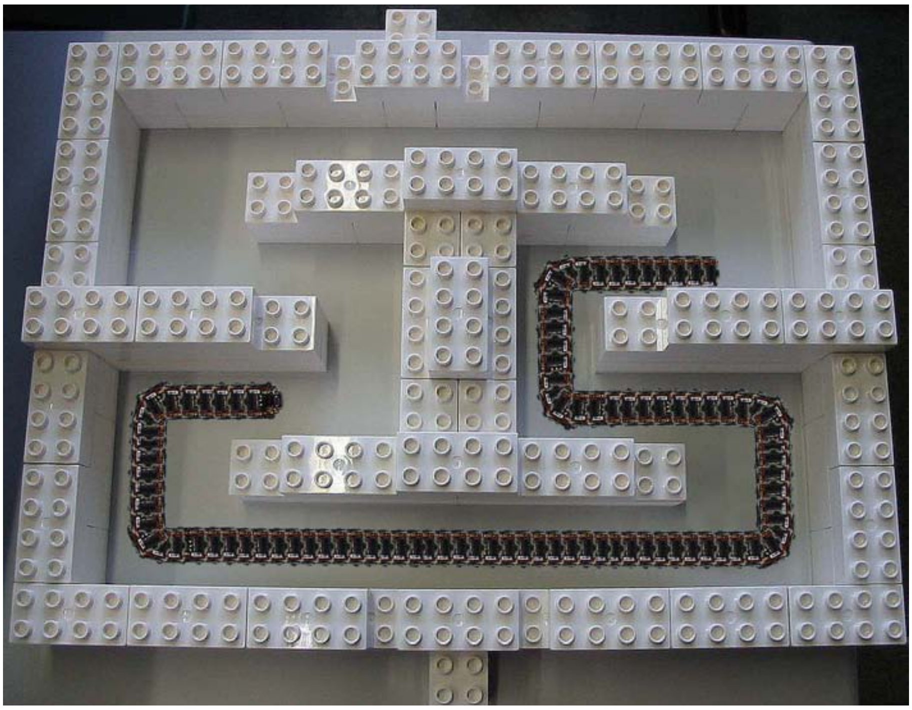


Figure 3.7: A sample of the learned safe-wandering behavior in the H-maze

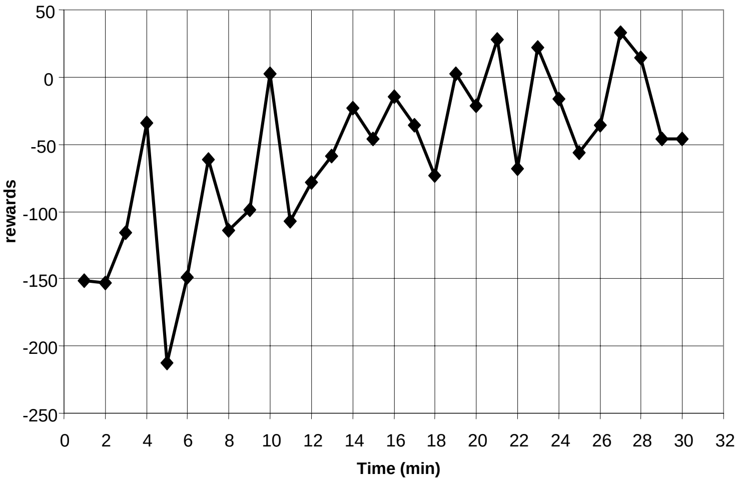


Figure 3.8: History of the received rewards during safe-wandering in the H-maze

history of the received rewards during an experiment in the cross-maze. Each point corresponds to the sum of received rewards during one minute. The experiment could be terminated in minute 7, but we intend to compare it with the floating-point based algorithm.

If the robot moves forward forever, the maximum achievable reward during one minute could be 300. But, the maximum is not reachable since the robot is forced some times to turn, due to the geometry of the maze. Also, the current state indicates only presence or absence of the obstacles (0 or 1), and does not provide more details. As a result, the robot passes some times beyond the distance threshold –especially at sidewalls– and receives punishment, although it had not actually bumped into the obstacles. However, after 7 minutes the received rewards converges to around 160.

A sample of the learned trajectory and the received rewards for H-maze is illustrated in Figs. 3.7 and 3.8 respectively. The experiment takes 30 minutes to finish. The sum of rewards changes from -213 at the initial steps to 33 finally. The poor performance in the 5th minute is because of an unexplored situation in the maze, where the robot has not learned how to deal with.

The two learning tasks converge in 7 and 30 minutes, respectively. Comparing to 10 hours autonomy of the Alice robots, it seems that implementation of more complex learning tasks is feasible.

3.7 Integer vs. Floating-point

To measure the introduced error due to the simplification of the algorithm, performance of the proposed algorithm is compared to the floating-point based algorithm (one-step Q-learning). Different combination of learning rates and temperature parameters were tested on the safe-wandering task in the cross-maze: learning rate changed from 0.01 to 0.1 (0.01 steps), and from 0.1 to 0.5 (0.1 steps). Temperature parameters changed from 0.1 to 1 (0.1 steps).

Finally, the three best results were obtained by setting the learning rate to 0.1, 0.2, 0.3, and temperature parameter to 0.5. The discount parameter was fixed to 0.9. Learning experiments lasted 20 minutes. Then, the learned behavior was tested for 10 minute. The robots are placed in the same positions, at the beginning of learning and test phases. The experiment is repeated 5 times.

The methods are compared in terms of four measures: *Convergence rate*, *performance*, *memory*, and *program size*. The first two measures are learning measures, and the two last ones are implementation measures. The averaged received rewards during learning phase gives an indirect indication of the convergence rate. The faster the learning procedure converges, the higher this average is. In the other hand, the average received rewards during the test phase gives a measure to compare performance of the learned policies.

3.7.1 Learning measures

The charts in Fig. 3.9 show the averaged received rewards during one minute of learning and test phases, respectively. Among the three float-based experiments, $\alpha = 0.1$ has the most conservative updates (the lowest α). Therefore, its average during learning is less than the others (Fig. 3.9-left). However, results of the test phase indicates that, it could come up with the best policy (Fig. 3.9-right). On the other hand, $\alpha = 0.3$ (the biggest α) has the best convergence rate but,

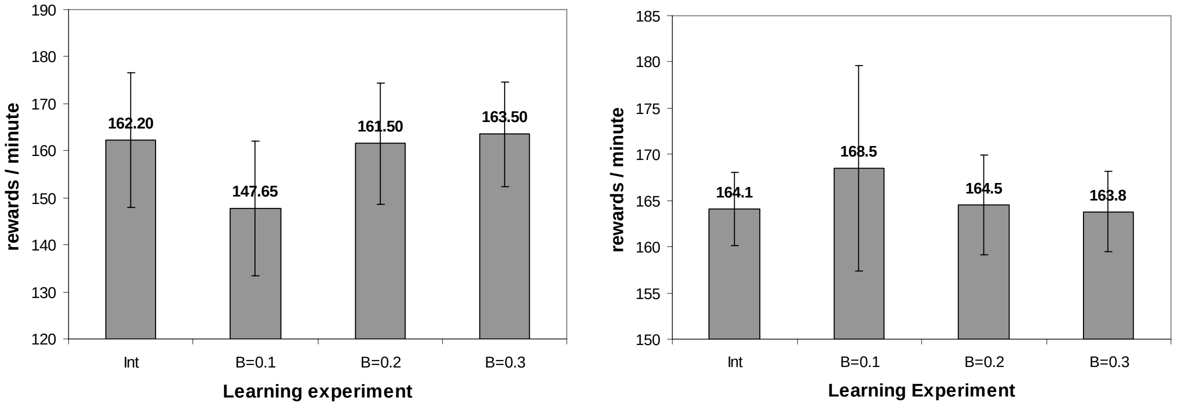


Figure 3.9: The averaged received rewards per minute during learning (left) and test (right) phases

Table 3.2: Comparing the percentage of memory and program size between the Compact Q-Learning and the floating-point based Q-Learning algorithm.

Memory	Operating System	Compact Q-Learning	Regular Q-Learning
Data	29	13	60
Program	19	25	64

it has the worst performance in the test phase. In this case, learning policy gets stuck very early in a local optimum.

Comparing these results to the integer-based algorithm, one can see that for both measures, the Compact Q-Learning algorithm stands in the middle, between the $\alpha = 0.1$ and $\alpha = 0.3$ cases. Therefore, we can argue that the proposed algorithm comprises a satisfying trade-off between the convergence rate and the optimality of the learned policy.

3.7.2 Implementation measures

Table 3.2 compares the occupied percentage of data and program memories by the Compact Q-Learning algorithm and the floating-point based one. The Compact Q-Learning takes totally (the algorithm itself and the operating system) around 40% of both data and program memories. The learning procedure plus save, load, and evaluation procedures occupy only 13% of the data and 25% of the program memories. Recall from Table 3.1 that, only three floating-point operations fill around 9% of the program memory.

On the other hand, the whole floating-point based program takes more than 80% of the both memories. Roughly speaking, since the required data memory depends mainly on the Q-table, using one-byte Q-values can reduce memory consumption to, at least, three times comparing to the (4-byte) floating-point representation.

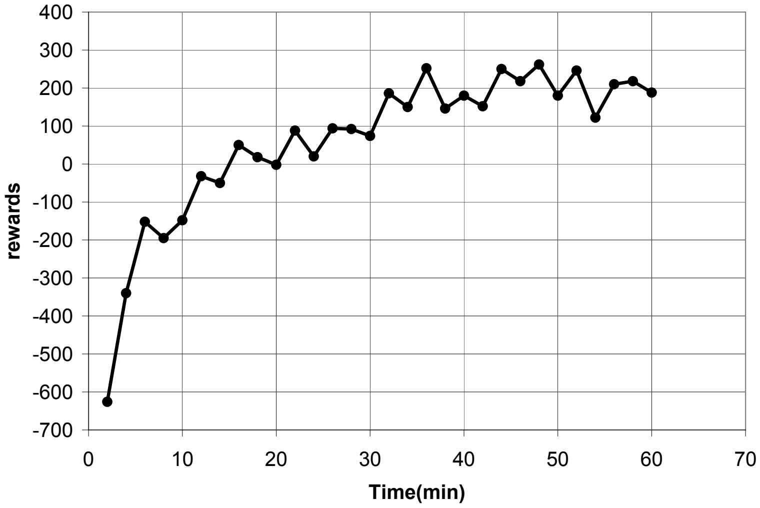


Figure 3.10: History of the received rewards during every two minutes of light-search learning in the Cross-maze

3.8 Learning Light-Search Behavior

In this section, *light-search* behavior is composed, in a bottom-up manner, from combination of the *light-attraction* and the *safe-wandering* behaviors. State vector of the light-search behavior is the Cartesian product of the state vectors of the *light-attraction* and the *safe-wandering* behaviors. In general, common features of the state vectors could be factored. In our case, the state-vectors have different features. The new vector is, therefore, composed of information about the objects and the light intensity around the robot. The number of states is 15×4=60.

Reward function of the new behavior is defined as weighted sum of the reward function of the two behaviors [96]. The weights depends on the importance of each behavior in achieving the ultimate goal. This might not be trivial to define. We define the new reward function for this case as: two times the reward function of the *safe-wandering* behavior, plus one times the one of the *light-attraction* behavior, i.e.:

$$\mathbf{r} = 2 \cdot r_{sw} + r_{la} = \tag{3.12}$$

$$\begin{cases} +3 = 2\times\ \ 1 + 1\times\ \ 1 & \text{moving forward} \bigwedge \neg \text{ hitting obstacles} \bigwedge \text{ faced toward the light source} \\ +1 = 2\times\ \ 1 + 1\times -1 & \text{moving forward} \bigwedge \neg \text{ hitting obstacles} \bigwedge \neg \text{ faced toward the light source} \\ +1 = 2\times\ \ 0 + 1\times\ \ 1 & \neg \text{ moving forward} \bigwedge \neg \text{ hitting obstacles} \bigwedge \text{ faced toward the light source} \\ -1 = 2\times\ \ 0 + 1\times -1 & \neg \text{ moving forward} \bigwedge \neg \text{ hitting obstacles} \bigwedge \neg \text{ faced toward the light source} \\ -1 = 2\times -1 + 1\times\ \ 1 & \text{hitting obstacles} \bigwedge \text{ faced toward the light source} \\ -3 = 2\times -1 + 1\times -1 & \text{hitting obstacles} \bigwedge \neg \text{ faced toward the light source} \end{cases}$$

The robot is tested in the Cross-maze (Fig. 3.5). The light source (Fig. 3.3) is manually moved along the periphery of the maze. History of the sum of received rewards during every two minutes of a learning experiment is shown in Fig. 3.10.

Maximum reward that the robot can receive in two minutes is +1800. The robot starts with around -600 punishment and ends up with around +200 rewards. The Q-table is four times bigger than the Q-table of safe-wandering. The experiment takes one hour to converge, about 8 times longer than safe-wandering in the Cross-maze. However, the quality of the learned behavior is not visually as good as that of *light-attraction*. This means the robot still needs longer training time to fine-tune its behavior in different situations.

Due to hardware limitations of our robots, we could not add more behaviors, since the new Q-table had already filled 75% of the memory (180 bytes). Although, the proposed method is simple to be implemented on the robots, it is not scalable since the size of the Q-table, as well as the required learning time, increase exponentially.

3.9 Conclusion and future works

In this chapter, the problems we faced in programming micro-robots for learning tasks are described. The major challenges with micro-robots are limitations in power autonomy, processing power, and memory (both in program and in data memory). These limitations obliged us to avoid floating-point operations and, hence, to develop a simplified learning algorithm, that rely mainly on integer-based addition and subtraction operations.

We proposed a simple and fast RL algorithm optimized for data and program memory consumption. It was implemented and verified on light-attraction and safe-wandering task with two test environments. The fast convergence of the algorithm made it possible to save at least 95% of the power autonomy, while performance of the learned policies remain competitive to regular Q-learning. The small number of required instructions leaves around 60% of both, program and data memory unused. Therefore, it offers potential for more time-critic and complex learning behaviors.

Learning combination of two behaviors was implemented by combining the state vectors of the behaviors with Cartesian product, and combining their reward functions with weighted sum. Common features of the state vectors are factored. Maximum number of states of the combined state-space is the product of the number of states of the behaviors. Weights of the reward functions might not be trivial to define.

Convergence of the proposed algorithm has been demonstrated by experiments. However mathematical analysis and prove has still to be made. Scalability of the algorithm is quite limited. In fact, representation of the state space in a table might not be optimal. Some parts of the table might be too much detailed, and some other parts might be too much general. Some features of the state space might be irrelevant to what the robot is currently learning. Abstracting these features can help the agent learn faster. We would like to work on this issue for future works.

Part III

Hierarchical Learning

Chapter 4

Hierarchical Reinforcement Learning

Masoud Asadpour, **Majid Nili Ahmadabadi**[1], and **Roland Siegwart**

Published in:

Proceedings of the First European Symposium on Robotics [EUROS], 2006 [97]

Abstract. The required learning time and the curse of dimensionality restrict applicability of Reinforcement Learning (RL) on real robots. Difficulties in inclusion of available knowledge and interpretability of learned rules must be added to the mentioned problems. In this chapter, we address the automatic state abstraction problem and discovering hierarchies in the state space, as two major approaches for reducing the number of learning trials, simplifying inclusion of prior knowledge, and making the learned rules more abstract and understandable. We formalize the automatic state abstraction and hierarchy creation and derive three new criteria that adapt decision tree learning techniques to state abstraction. Proof of performance is supported by strong evidences from simulation results in deterministic and non-deterministic environments. Simulation results show encouraging enhancements in the required number of learning trials, agent's performance, size of the learned trees, and computation time of the algorithm.

Key words: State Abstraction, Hierarchical Reinforcement Learning, Splitting Criteria

4.1 Introduction

It is well accepted that utilization of proper learning methods reduces the after-design problems. These problems are caused naturally due to use of inaccurate and incomplete models and inadequate performance measures at design time. However, confronting unseen situations and changes in the environment and the robot must be added to it.

---

[1] Control and Intelligent Processing Center of Excellence, ECE Dept., University of Tehran, Iran, mnili@ut.ac.ir

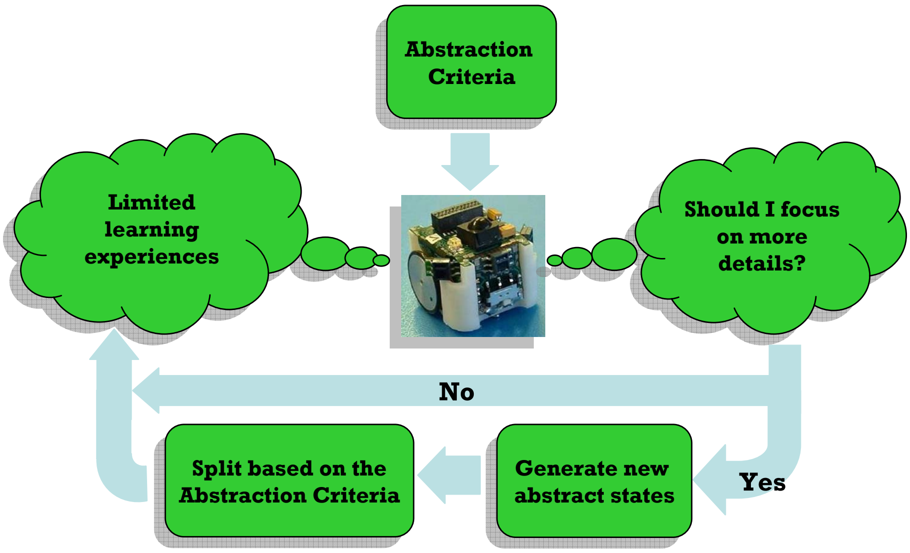

Abstraction Criteria
Limited learning experiences
Should I focus on more details?
No
Yes
Split based on the Abstraction Criteria
Generate new abstract states

Figure 4.1: state abstraction concept

The existing learning methods that are capable of handling dynamics and complicated situations are slow. Moreover, they produce rules –mostly numerical– that are not abstract, modular, and comprehensible by human. Therefore, designer's intuition cannot be simply incorporated in the learning method and in the learned rules. In addition, the existing learning methods suffer from the curse of dimensionality, requiring a large number of learning trials and lack of abstraction capability. RL [18], despite its strength in handling dynamic and non-deterministic environments, is an example of such learning methods. In this research, we try to solve some of the mentioned problems in RL domain. We have chosen RL, because of its simplicity and variety of its applications in robotics field in addition to its mentioned advantages.

It is believed that, the curse of dimensionality can be lessen, to a great extend, by implementation of state abstraction and creation of hierarchy in the behavioral architecture. Having abstract states and hierarchies in the agent's mind, accompanied by a careful design of the learning method, the learned rules would be partitioned, and the number of learning trials could be reduced. In addition, incremental improvement of agent's performance becomes straight-forward.

Different approaches have been proposed so far, for state abstraction, creation of hierarchies, and implementation of RL in those architectures. One group of these methods incorporate a fixed Artificial Neural Network, and use RL methods to learn both the hierarchy and the rules in each layer [98]. Another group of researches proposes hand-designing the sub-tasks and learning them in a defined hierarchy [99]. These two approaches require much design effort for explicit formulation of hierarchy and state abstraction. Moreover, the designer should, to some extent, know how to solve the problem before s/he designs the hierarchy and the sub-tasks. The third approach uses decision-tree learning methods to automatically abstract the states, detect sub-tasks and speed up the learning process (Fig. 4.1).

The devised approaches, however, do not take the exploratory nature of RL into account. This results in non-optimal solution. In this chapter, we take a mathematical approach to develop new criteria for utilization of decision trees in state abstraction. Our method outperforms the existing solutions in performance, number of learning trials, and size of trees.

Related works are reviewed in the next section. U-Tree algorithm and its drawbacks are briefly described in the third section. In the fourth section we formalize state abstraction according to different heuristics and derive new splitting criteria. The fifth section describes the simulation results. Conclusions and future works are discussed finally.

4.2 Related Works

The simplest idea to gain abstract knowledge is detection of sub-tasks, specially repeating ones [100], and discovering the hierarchy among them. Once the agent learned a sub-task (which is faster and more efficient, due to confronting with smaller state space), it is potentially able to use it afterwards. Sub-tasks in Hierarchical RL(HRL) are closed-loop policies that are generally defined for a subset of the state space [19]. These partial policies are sometimes called *temporally-extended-actions*, *options* [21], *skills* [22], *behaviors* [13], etc. They must have well-defined termination conditions.

Hierarchies of Abstract Machines(HAM) [23], partially reduces the state space by constraining the actions that the agent can take in each state. Although, this allows for transferring prior knowledge to an agent, it needs the agent designer to spend much more design effort than regular RL. Moreover, the number of states of the induced Markov Decision Process (MDP) is quite bigger than MDP itself. In MaxQ Value Function Decomposition [24], the core MDP is decomposed into a set of sub-tasks, each with a termination condition and a pseudo-reward function. Hierarchy of sub-tasks is summarized in a manually-designed task graph.

Although these methods give more understanding about underlying aspects of HRL, since particular features are designed manually based on the task domain, it is hard to apply them to complex tasks. The general problem that should be solved first, either when designing manually or extracting structures automatically is the *state abstraction*. In many situations, significant portions of a large state space might be irrelevant to a specific goal. They can be aggregated into few, relevant states. As a consequence, the learning agent can learn sub-task reasonably fast. Our work addresses this problem.

Techniques for non-uniform state-discretization are already known e.g. Parti-game [30], G algorithm [101], and U-Tree [31]. They start by modeling the whole world with a single state and recursively split it when necessary. Continuous U-Tree [32] extends U-Tree to work with continuous state variables. TTree [102] applies Continuous U-Tree to SMDPs. Jansson and Barto [103] applied U-Tree to options but they use one tree per option and build the hierarchy of options manually. We adopt U-Tree as our basic method, because the reported results illustrate it as a promising approach to automatic state abstraction.

In this chapter, we show that the employed U-Tree methods ignore the explorative nature of RL. This imposes a bias in the distribution of the samples that are saved for introducing new splits in U-Tree. As a result, finding a good split becomes more and more difficult and the introduced splits could be far from optimality. Moreover, U-Tree-based techniques have been excerpted in essence from decision tree learning methods. Therefore, their utilized splitting criteria are very general and can work with any sort of data. Here, we devise some splitting criteria that are specialized for state abstraction in RL. We show that, they are more efficient than the general ones both in learning performance and computation time.

Dean and Robert [104] have developed a minimization model, known as ε-reduction, to construct a partition space that has fewer number of states than the original MDP, while ensuring that utility of the policy learned in the reduced state space is within a fixed bound of the optimal policy. Our work is different because, ε-reduction does not extract any hierarchy among state partitions; while, building a hierarchy can reduce search time in partitions from $O(u)$ to $O(\log u)$, u being the number of partitions. Also, their theory is developed based on immediate return of actions instead of long-term return. We can argue that, ε-reduction is a special case of our method.

4.3 Formalism

We model the world as a MDP, which is a 6-tuple (S,A,T,γ,D,R) , where S is a set of *states* (here can be infinite), $A=\{a_1,\ldots,a_{|A|}\}$ is a set of *actions*, $T=\{P^a_{ss'}\}$ is a *transition model* that maps $S\times A\times S$ into probabilities in $[0,1]$, $\gamma\in[0,1)$ is a *discount factor*, D is the initial-state distribution from which the start state is drawn (shown by $s_0\sim D$), and R is a *reward function* that maps $S\times A\times S$ into real-valued rewards.

A policy π maps from S to A, a real-valued *value function*, V, on states or a real-valued *action-value function* (also called *Q-function*), Q, on state-action pairs. The aim is to find an optimal policy π^* (or equivalently, V^* or Q^*) that maximizes the expected discounted rewards of the agent. We assume that, each state s is a *sensory-input* vector $(x_1,\ldots,x_n)$ where x_i is a *feature* (called also *state-variable*, or *attribute*).

An *abstract state* $\bar{S}$ is a subset of state space S such that all states within it have "close" values. *Value* of abstract state $\bar{S}$ is defined as the expected discounted reward return if the agent starts from a state in $\bar{S}$ and follows the policy π afterwards:

$$V^\pi(\bar{S})=\sum_{s\in\bar{S}}\left\{P(s|\bar{S})\sum_{a\in A}\left[\pi(s,a)\sum_{s'}P^a_{ss'}(R^a_{ss'}+\gamma V^\pi(s'))\right]\right\} \tag{4.1}$$

where $\pi(s,a)$ is the selection probability of action a in state s under policy π, $P^a_{ss'}$ is the probability that environment goes to state s' after doing action a in state s, and $R^a_{ss'}$ is the expected immediate reward after doing action a in state s and going to state s'. Action-value $Q^\pi(\bar{S},a)$ is similarly defined as:

$$Q^\pi(\bar{S},a)=\sum_{s\in\bar{S}}P(s|\bar{S})\sum_{s'}P^a_{ss'}(R^a_{ss'}+\gamma V^\pi(s')) \tag{4.2}$$

4.4 U-Tree

The U-Tree [31] [32] abstracts the state space incrementally. Each leaf L_i of the U-Tree corresponds to an abstract state $\bar{S}_i$. Each leaf stores the action-values $Q(\bar{S}_i,a_j)$ for all available actions a_j. The tree is initialized with a single leaf, assuming the whole world as one abstract state. New abstract states are added if necessary. Sub-trees of the tree represent sub-tasks of the whole task. Each sub-tree can have other sub-sub-trees that correspond to its sub-sub-tasks. The hierarchy breaks down to leaves which specify primitive sub-tasks.

The procedure for construction of the abstraction tree loops through a two-phase process: *sampling* and *processing* phases. During the sampling phase, the algorithm behaves as a standard

RL algorithm, with the added step of using the tree to translate sensory input to an abstract state. In the sampling phase, a history of transition steps, i.e. $T_i = (s, a, r, s')$, composed of the current state, the selected action, the received immediate reward, and the next state is recorded. The sample is assigned to a unique leaf of the tree, based on the value of the current state.

After some number of learning episodes the processing phase starts. In the processing phase a value is assigned to each sample:

$$\begin{array}{c} V(T_i) = T_i.r + \gamma V(T_i.\bar{s}') \\ V(T_i.\bar{s}') = \max_{a \in A} Q(T_i.\bar{s}', a) \end{array} \tag{4.3}$$

where $\bar{s}'$ is the abstract state that s' belongs to.

If a significant difference in distribution of sample-values within a leaf is found, the leaf is broken up to two leaves. To find the best split point for each leaf, the algorithm loops over the features. Samples within a leaf are sorted according to a feature, and a trial split is virtually added between each consecutive pair of feature values. This split divides the abstract state into two sub-sets. A *splitting criterion* compares the two sub-sets, and returns a number indicating the different between the two distributions. If the largest difference among all features is bigger than a confidence threshold, then the split is introduced. This procedure is repeated for all leaves.

Although the original algorithm for U-Tree construction [31] can function in *partially observable* domains, for the sake of simplicity, we make the standard assumption that the agent has access to a complete description of environment. Also, since structure of the tree is not revised once a split is introduced, we assume the environment is *stationary*.

4.4.1 Splitting Criteria

Two types of splitting criteria have been used, so far, for state abstraction: the non-parametric Kolmogorov-Smirnov(KS) test [31; 32] from Statistics, and Information Gain Ratio(IGR) test [105] from Information Theory.

The non-parametric KS-test looks for violation of Markov property [2]. It computes the statistical difference between the distribution of samples on either side of the split using the difference in cumulative distributions of the two data-sets.

IGR-test measures the amount of reduction in uncertainty of sample values. Given a set of samples $T = \{T_i\}, i = 1, \ldots, M$ from abstract state $\bar{S}$, labeled with $V(T_i)$ –computed according to (4.3)– the *entropy* of the sample-values is computed as:

$$Entropy_{\bar{S}}(V) = -\sum_{v} P(V(T_i) = v|\bar{S}) \ \log_2 P(V(T_i) = v|\bar{S}) \tag{4.4}$$

where $P(V(T_i) = v|\bar{S})$ is the probability that samples in T have value v.

The *Information Gain* of splitting the abstract state $\bar{S}$ to $\bar{S}_1$ and $\bar{S}_2$ is defined as the amount of reduction in the entropy of sample-values i.e.:

$$InfoGain(\bar{S}, \bar{S}_1, \bar{S}_2) = Entropy_{\bar{S}}(V) - \sum_{i=1}^{2} \frac{M_i}{M} Entropy_{\bar{S}_i}(V) \tag{4.5}$$

---

[2] A stochastic process has the Markov property if, the conditional probability distribution of future states of the process, depends only upon the current state and not on any past states.

where M_i is the number of samples from T that fall in $\bar{S}_i$. IGR is defined as:

$$IGR(\bar{S},\bar{S}_1,\bar{S}_2) = \frac{InfoGain(\bar{S},\bar{S}_1,\bar{S}_2)}{-\sum_{i=1}^{2} \frac{M_i}{M} \log_2 \frac{M_i}{M}} \tag{4.6}$$

4.4.2 Computation Time of U-Tree

If we assume processing phase takes place after each learning episode, number of leaves [3] at episode t is $L(t)$, and sample size in each leaf is M, the best sort algorithms, i.e. Quicksort, would sort the samples based on a feature in $O(M \log M)$. Maximum number of split points could be $(M-1)$. KS and IGR tests need the samples in the either sides of the virtual splits to be sorted based on their values. This needs at least $O(2\,\frac{M}{2}\,\log\frac{M}{2}))$. Then a pass through all sorted samples needs $O(M)$. Totally the order of the algorithm, assuming n being dimension of the state vectors, is:

$$O\left(L(t)\; n\; \left[M\; \log M + (M-1)(2\,\frac{M}{2}\;\log\frac{M}{2} + M)\right]\right) = O\left(L(t)\; n\, M^2\; \log M\right) \tag{4.7}$$

4.4.3 Biased Sampling Problem

In implementation, each leaf of U-Tree could hold a limited number of samples. If the number of samples exceeds an upper limit, an old sample is selected randomly and is replaced by the new sample. However, selection probability of an action in a state depends on its action-value function. Thus, the actions that have higher action-value functions are selected more than the others. As a result, the distribution of samples in the leaves are biased. We call this the *biased sampling* problem.

The biased sampling problem has a defective effect. RL escapes from local optimums by making a trade off between exploration and exploitation. If the size of the sample lists is not big enough, then number of samples with explorative character would not be statistically significant to introduce any split, until the confidence threshold for the splitting criterion is dropped. Reducing the threshold increases the occurrence of non-optimal splits. If a split is introduced on the basis of a wrong feature, then in the next steps the correct split have to be repeated multiple times, in all sub-trees.

This problem can be solved by spreading the entries of the big sample list over the actions. Instead of one big list with capacity M, we could form $|A|$ smaller lists with capacity $\frac{M}{|A|}$. Then, the algorithm is modified so that, when new samples arrive while the sample list is full, an old sample from the list that corresponds to the current action field of the new sample is deleted randomly. As a positive side-effect sorting the new sample lists would be of order $|A|\; O(\frac{M}{|A|}\;\log\frac{M}{|A|}) = O(M\;\log\frac{M}{|A|})$ which is a bit faster than $O(M\;\log M)$. [4]

4.5 New Splitting Criteria

In this section we formalize the state abstraction problem and describe three new splitting criteria that are specially derived for HRL. They are called:

---

[3] Total number of nodes is $2\,L(t)-1$.

[4] Note however that, according to complexity theory the two time orders are still equal.

1. SANDS: **S**tate **A**bstraction for **N**on-**D**eterministic **S**ystems
2. SMGR: **S**oft**M**ax **G**ain **R**atio
3. VAR: **VA**riance **R**eduction

4.5.1 SANDS

To derive the SANDS criterion, we formulate the state abstraction as an optimization problem and solve it incrementally. From the set theory, a *set partition* of a set S, is defined as a collection of disjoint subsets of S whose union is S. So $U = \{\bar{S}_1, \ldots, \bar{S}_u\}$ is a set partition of S if and only if:

$$\begin{array}{c} \bar{S}_i \subseteq S, \forall i \in \{1, \ldots, u\} \\ S = \bigcup_{i=1}^{u} \bar{S}_i \\ \bar{S}_i \bigcap \bar{S}_j = \emptyset, \forall i, j \in \{1, \ldots, u\}, i \neq j \end{array} \tag{4.8}$$

State abstraction could be formalized as an optimization problem, which aims at finding a set partition U and a policy π on the state space S, where u the number of partitions is minimum, and the value of the learned policy π is within a fixed bound ε of the optimal policy π^* i.e.:

$$|V^{\pi^*}(s) - V^{\pi}(\bar{S}_i)| < \varepsilon, \forall s \in \bar{S}_i, \forall \bar{S}_i \in U \tag{4.9}$$

It is easy to prove that U-Tree generates a set-partition of state space. It is clear from definition of state abstraction in (4.9) that we are minimizing the difference between the optimal policy and the policy that a U-Tree represents. However, instead of walking toward the optimal policy which is unknown, we try to walk away from the current policy toward the optimal policy.

Assume that we have a U-Tree that defines a policy π. We would like to enhance it to a policy $\tilde{\pi}$ by finding a partition $\bar{S}$ (one of the leaves in the tree) and splitting it to unknown partitions $\bar{S}_1$ and $\bar{S}_2$ so that, the expected return of the resulting tree is maximized. We should maximize, then, the following goal function:

$$G = \sum_{s \sim D} P(s)[V^{\tilde{\pi}}(s) - V^{\pi}(s)] \tag{4.10}$$

where D is the distribution of the initial states.

Using the modified Bellman equations in (4.1) we can write:

$$G = \sum_{s \sim D} P(s) \left(\begin{array}{r} \sum_{a \in A}[\tilde{\pi}(s,a) \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V^{\tilde{\pi}}(s'))] \\ -\sum_{a \in A}[\pi(s,a) \sum_{s'} P^a_{ss'}(R^a_{ss'} + \gamma V^{\pi}(s'))] \end{array} \right) \tag{4.11}$$

$$G = \sum_{s \sim D} \sum_{a \in A} \sum_{s'} P(s) P^a_{ss'} \left(\begin{array}{l} R^a_{ss'}(\tilde{\pi}(s,a) - \pi(s,a)) \\ +\gamma(\tilde{\pi}(s,a)V^{\tilde{\pi}}(s') - \pi(s,a)V^{\pi}(s')) \end{array} \right) \tag{4.12}$$

$V^{\tilde{\pi}}(s')$ is unknown; we can use $V^{\pi}(s')$ as an estimation. For the states that fall outside of $\bar{S}$, the policy remains unchanged after splitting i.e $\forall s \notin \bar{S}, \tilde{\pi}(s,a) = \pi(s,a)$. For the states within $\bar{S}$ or within the new abstract states $\bar{S}_1$ and $\bar{S}_2$, we define $\pi^a = \pi(s,a), s \in \bar{S}$ and $\tilde{\pi}^a_i = \tilde{\pi}(s,a), s \in \bar{S}_i, i = 1, 2$. Hence:

$$G = \sum_{a \in A} \sum_{i=1}^{2} \left((\tilde{\pi}^a_i - \pi^a) \sum_{s \in \bar{S}_i} \sum_{s'} P(s) P^a_{ss'}(R^a_{ss'} + \gamma V^{\pi}(s')) \right) \tag{4.13}$$

According to properties of the conditional probabilities, we can write $P(s) = P(s|\bar{S}_i) \times P(\bar{S}_i|\bar{S}) \times P(\bar{S}), \forall s \in \bar{S}_i, i = 1,2$. Considering the definition in (4.2), we can write:

$$G = P(\bar{S}) \sum_{a \in A} \sum_{i=1}^{2} \left[(\tilde{\pi}_i^a - \pi^a) Q^\pi(\bar{S}_i, a) P(\bar{S}_i|\bar{S}) \right] \tag{4.14}$$

This way, the learner finds the split that maximizes performance of the tree over the "whole action set", provided that actions are always selected according to Softmax distribution. However, after learning the learned policy is freezed and the greedy action will always be selected. An important point here is that, the learner can not revise the tree after initiating the split. This means in this level, it is fixing somehow a part of the policy. Therefore, it is more valuable to judge the splits based on performance of the greedy action instead of Softmax. The optimization term, then, should change from summation to maximum. Moreover, $P(\bar{S})$ is same for all states in abstract state $\bar{S}$. So, the final formulation is simplified to:

$$\boxed{SANDS(\bar{S}) = \max_{a \in A} \sum_{i=1}^{2} \left[(\tilde{\pi}_i^a - \pi^a) Q^\pi(\bar{S}_i, a) P(\bar{S}_i|\bar{S}) \right]} \tag{4.15}$$

In fact, SANDS tries to find a point that well differentiates both value and selection probability of actions before and after introducing the split.

Optimality of Partitions

U-Tree generates a set-partition of the state space. Although the algorithm starts with one partition and adds more partitions if required, optimality of the number of partitions is not guaranteed. In fact, the splits here are Univariate [106; 107] that break the state space parallel to an axis. The necessary (but not sufficient) condition to have minimum size tree, in most tasks, is to have Multivariate splits [108; 109; 110], which can split the state space in any direction. Multivariate splits are not covered in the current version of the algorithm. We will discuss more about the size of the trees in the result section.

Computation Time of SANDS

If we assume the processing phase starts after one learning episode, number of leaves in episode t is $L(t)$, sample size of the $|A|$ sample-lists in the leaves is $m = \frac{M}{|A|}$, and space dimension is n, time order of the new algorithm would be:

$$O\left(L(t)\, n\, |A|\ [m \log m + (m-1)\, m]\right) = O\left(L(t)\, n\, |A|\, m^2\right) = O\left(L(t)\, n\, \frac{M^2}{|A|}\right) \tag{4.16}$$

which is faster than (4.7) by $|A| \log M$ factor.

Mont Carlo Methods for SANDS

Using the samples recorded during the learning episodes, we can compute the SANDS criterion by Mont-Carlo methods. Each logged sample is a 4-tuple $T_i = (s, a, r, s')$, corresponding to the current state, the selected action, the received reward, and the next state, respectively. If among

m^a samples for action a in $\bar{S}$, m_1^a and $m_2^a = m^a - m_1^a$ samples correspond to $\bar{S}_1$ and $\bar{S}_2$ respectively, we can compute:

$$\boxed{SANDS(\bar{S}) = \max_{a \in A} \sum_{i=1}^{2} (\hat{\pi}_i^a - \hat{\pi}^a)\hat{\mu}_i^a \hat{\rho}_i^a} \tag{4.17}$$

Where $\hat{\mu}_i^a$ is the action-value function $Q(\bar{S}_i, a)$ estimated from the samples, $\hat{\pi}^a$ and $\hat{\pi}_i^a$ are the selection probabilities of action a, i.e. $\pi(\bar{S}, a)$ and $\pi(\bar{S}_i, a)$, which are computed according to the Boltzmann distribution from the estimated action-value functions, and $\hat{\rho}_i^a$ is the fraction of samples that their current state fall within $\bar{S}_i$. They can be computed as follows:

$$\begin{aligned}
&\hat{\rho}_i^a = \tfrac{m_i^a}{m^a}, i = 1,2 && \hat{\rho}_i = \tfrac{\sum_{a \in A} m_i^a}{\sum_{a \in A} m^a}, i = 1,2 \\
&\hat{\mu}_i^a = \tfrac{1}{m_i^a} \textstyle\sum_{j=1}^{m_i^a} [T_j.r + \gamma \max_b Q(T_j.s', b)\,], i = 1,2 && \hat{\mu}^a = \hat{\rho}_1^a \hat{\mu}_1^a + \hat{\rho}_2^a \hat{\mu}_2^a \\
&\hat{\pi}_i^a = \tfrac{\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b/\tau)}, i = 1,2 && \hat{\pi}^a = \tfrac{\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)}
\end{aligned} \tag{4.18}$$

4.5.2 SMGR

SMGR tries to find the splits that results in more certainty on selection probability of actions. Instead of defining the entropy function on the sample-values, like what IGR does in (4.6), it is defined here on the selection probability of actions:

$$\boxed{SMGR(\bar{S}) = \max_{a \in A} \frac{-\hat{\pi}^a \log \hat{\pi}^a + \sum_{i=1}^{2} \hat{\rho}_i^a \hat{\pi}_i^a \log \hat{\pi}_i^a}{-\sum_{i=1}^{2} \hat{\rho}_i^a \log \hat{\rho}_i^a}} \tag{4.19}$$

where $\hat{\pi}^a$, $\hat{\pi}_i^a$, and $\hat{\rho}_i^a$ are defined according to (4.18).

We prove that SMGR and SANDS have some similarities in their preference over the splits. We show that SMGR combines SANDS with other heuristics. Like what we did for SANDS in Sec. 4.5.1, we initially work on summation function ($\sum_{a \in A}$) instead of maximum ($\max_{a \in A}$). By expanding $\hat{\pi}^a$ and $\hat{\pi}_i^a$ we can write:

$$\begin{aligned}
SMGR(\bar{S}) &= \sum_{a \in A} \frac{-\frac{\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)} \log \frac{\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)} + \sum_{i=1}^{2} \hat{\rho}_i^a \frac{\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b/\tau)} \log \frac{\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b/\tau)}}{-\sum_{i=1}^{2} \hat{\rho}_i^a \log \hat{\rho}_i^a} \\
&= \sum_{a \in A} \frac{-\frac{\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)} (\frac{\hat{\mu}^a}{\tau} - \log \sum_b \exp(\frac{\hat{\mu}^b}{\tau})) + \sum_{i=1}^{2} \hat{\rho}_i^a \frac{\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b\ \tau)} (\frac{\hat{\mu}_i^a}{\tau} - \log \sum_b \exp(\frac{\hat{\mu}_i^b}{\tau}))}{-\sum_{i=1}^{2} \hat{\rho}_i^a \log \hat{\rho}_i^a}
\end{aligned} \tag{4.20}$$

Substituting $\hat{\mu}^a$ with $\hat{\rho}_1^a \hat{\mu}_1^a + \hat{\rho}_2^a \hat{\mu}_2^a$ results in:

$$\begin{aligned}
SMGR(\bar{S}) &= \sum_{a \in A} \frac{\frac{1}{\tau} \sum_{i=1}^{2} \hat{\rho}_i^a \hat{\mu}_i^a (\frac{\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b/\tau)} - \frac{\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)})}{-\sum_{i=1}^{2} \hat{\rho}_i^a \log \hat{\rho}_i^a} + \\
&\quad \sum_{a \in A} \frac{\frac{\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)} \log \sum_b \exp(\frac{\hat{\mu}^b}{\tau}) - \sum_{i=1}^{2} \hat{\rho}_i^a \frac{\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b/\tau)} \log \sum_b \exp(\frac{\hat{\mu}_i^b}{\tau})}{-\sum_{i=1}^{2} \hat{\rho}_i^a \log \hat{\rho}_i^a}
\end{aligned} \tag{4.21}$$

We can approximate $\hat{\rho}_i^a$ with $\hat{\rho}_i$, and $\sum_{i=1}^2 \hat{\rho}_i^a \log \hat{\rho}_i^a$ with $\sum_{i=1}^2 \hat{\rho}_i \log \hat{\rho}_i$, in order to bring them out of summation:

$$\begin{aligned}
SMGR(\bar{S}) &= \frac{\frac{1}{\tau}\sum_{a\in A}\sum_{i=1}^2 \hat{\rho}_i^a \hat{\mu}_i^a \left(\frac{\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b/\tau)} - \frac{\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)}\right)}{-\sum_{i=1}^2 \hat{\rho}_i \log \hat{\rho}_i} + \qquad (4.22)\\
&\quad \frac{\frac{\sum_{a\in A}\exp(\hat{\mu}^a/\tau)}{\sum_b \exp(\hat{\mu}^b/\tau)} \log \sum_b \exp(\frac{\hat{\mu}^b}{\tau}) - \sum_{i=1}^2 \hat{\rho}_i^a \frac{\sum_{a\in A}\exp(\hat{\mu}_i^a/\tau)}{\sum_b \exp(\hat{\mu}_i^b/\tau)} \log \sum_b \exp(\frac{\hat{\mu}_i^b}{\tau})}{-\sum_{i=1}^2 \hat{\rho}_i \log \hat{\rho}_i}\\
&= \frac{\frac{1}{\tau}\sum_{a\in A}\sum_{i=1}^2 \hat{\rho}_i^a \hat{\mu}_i^a (\hat{\pi}_i^a - \hat{\pi}^a)}{-\sum_{i=1}^2 \hat{\rho}_i \log \hat{\rho}_i} + \qquad (4.23)\\
&\quad \frac{\log \sum_b \exp(\frac{\hat{\mu}^b}{\tau}) - \sum_{i=1}^2 \hat{\rho}_i \log \sum_b \exp(\frac{\hat{\mu}_i^b}{\tau})}{-\sum_{i=1}^2 \hat{\rho}_i \log \hat{\rho}_i}
\end{aligned}$$

We know that:

$$\begin{aligned}
\hat{\mu}^b = \hat{\rho}_1^b \hat{\mu}_1^b + \hat{\rho}_2^b \hat{\mu}_2^b &\approx \hat{\rho}_1 \hat{\mu}_1^b + \hat{\rho}_2 \hat{\mu}_2^b \qquad (4.24)\\
\Rightarrow \log \sum_b \exp(\frac{\hat{\mu}^b}{\tau}) &\approx \log \sum_b \exp(\frac{\hat{\rho}_1 \hat{\mu}_1^b + \hat{\rho}_2 \hat{\mu}_2^b}{\tau}) \qquad (4.25)\\
&= \log \sum_b \left[\exp(\frac{\hat{\mu}_1^b}{\tau})\right]^{\hat{\rho}_1} \left[\exp(\frac{\hat{\mu}_2^b}{\tau})\right]^{\hat{\rho}_2} \qquad (4.26)
\end{aligned}$$

Thus, the last terms of (4.23) can be rewritten as:

$$\log \sum_b \left[\exp(\frac{\hat{\mu}_1^b}{\tau})\right]^{\hat{\rho}_1} \left[\exp(\frac{\hat{\mu}_2^b}{\tau})\right]^{\hat{\rho}_2} - \log \left[\sum_b \exp(\frac{\hat{\mu}_1^b}{\tau})\right]^{\hat{\rho}_1} - \log \left[\sum_b \exp(\frac{\hat{\mu}_2^b}{\tau})\right]^{\hat{\rho}_2} \tag{4.27}$$

$$= \log \sum_{a\in A} \left[\frac{\exp(\hat{\mu}_1^a/\tau)}{\sum_b \exp(\hat{\mu}_1^b/\tau)}\right]^{\hat{\rho}_1} \left[\frac{\exp(\hat{\mu}_2^a/\tau)}{\sum_b \exp(\hat{\mu}_2^b/\tau)}\right]^{\hat{\rho}_2} = \log \sum_{a\in A} \prod_{i=1}^2 [\hat{\pi}_i^a]^{\hat{\rho}_i} \tag{4.28}$$

Putting (4.23) and (4.28) together we can write then:

$$SMGR(\bar{S}) = \frac{\frac{1}{\tau}\sum_{a\in A}\sum_{i=1}^2 \hat{\rho}_i^a \hat{\mu}_i^a (\hat{\pi}_i^a - \hat{\pi}^a) + \log \sum_{a\in A} \prod_{i=1}^2 [\hat{\pi}_i^a]^{\hat{\rho}_i}}{-\sum_{i=1}^2 \hat{\rho}_i \log \hat{\rho}_i} \tag{4.29}$$

The first term is the SANDS criterion (see (4.14)) and the second one is a “penalty term”. In fact, SMGR scales SANDS and combines it with another heuristic:

$$\boxed{SMGR(\bar{S}) = \frac{\frac{1}{\tau} SANDS(\bar{S}) + \log \sum_{a\in A} \prod_{i=1}^2 [\hat{\pi}_i^a]^{\hat{\rho}_i}}{-\sum_{i=1}^2 \hat{\rho}_i \log \hat{\rho}_i}} \tag{4.30}$$

Penalty Term

We give an example, in order to understand the underlying mechanism of penalizing the splits by the penalty term. Assume the action set includes only two actions $\{a, b\}$. The figures 4.2 to 4.4 show the penalty term for one of the actions in three different cases.

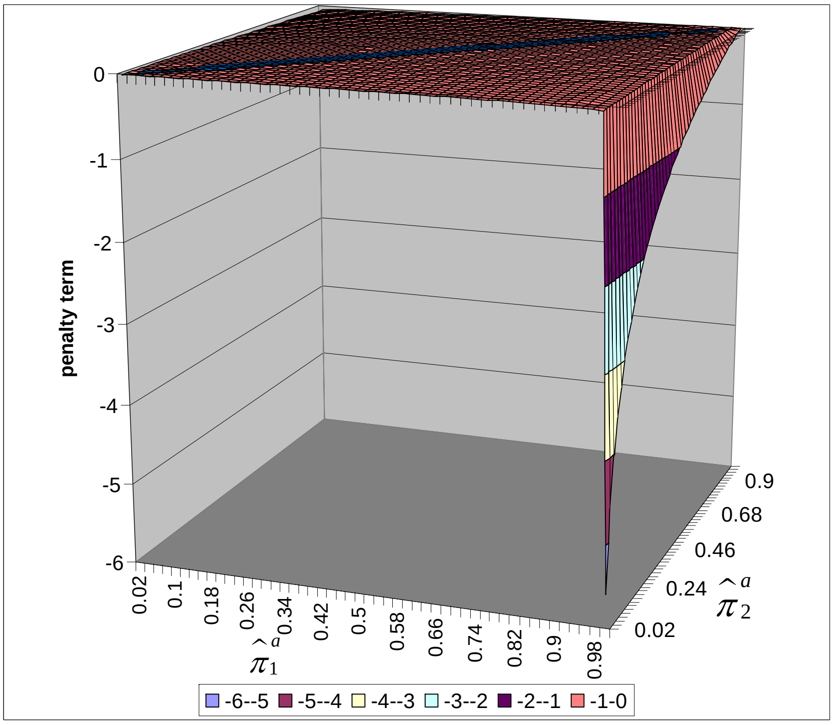


Figure 4.2: The penalty term when $\hat{\rho}_1 = 0.01$, $\hat{\rho}_2 = 0.99$

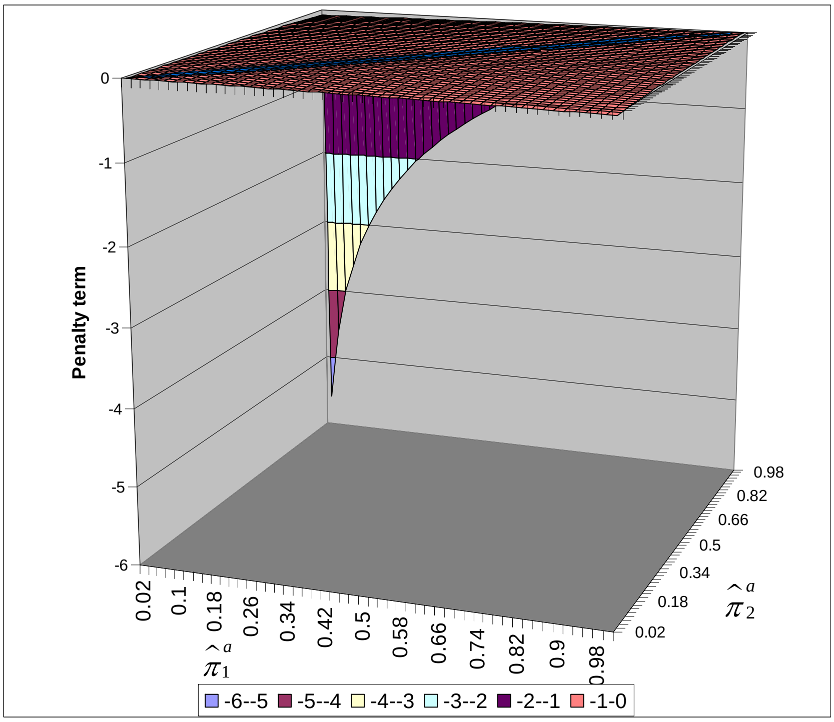


Figure 4.3: The penalty term when $\hat{\rho}_1 = 0.99$, $\hat{\rho}_2 = 0.01$

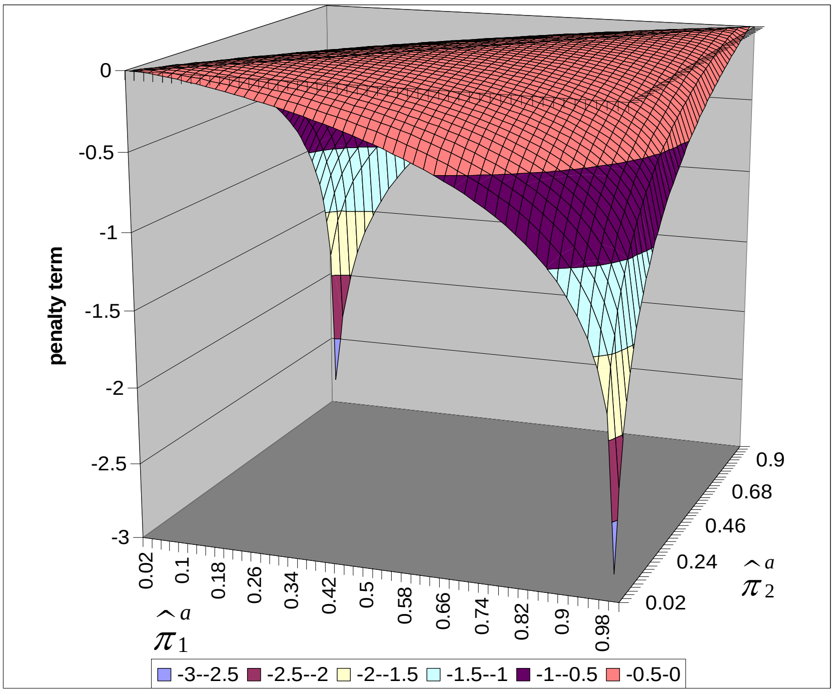


Figure 4.4: The penalty term when $\hat{\rho}_1 = \hat{\rho}_2 = 0.5$

Fig. 4.2 shows a case where, only 1% of the samples fall in the left side of the split, and 99% of them fall in the right side. The biggest penalty is given to $(\hat{\pi}_1^a = 1,\ \hat{\pi}_2^a = 0)$ point. Fig. 4.3 shows the reverse case, when 99% of the samples fall in the left side of the split, and 1% of them fall in the right side. The biggest penalty is given this time to $(\hat{\pi}_1^a = 0,\ \hat{\pi}_2^a = 1)$ point. In fact, the penalty term punishes the splits that result in giving "high" probability to a specific action without support of "enough" evidence (samples).

Fig. 4.4 shows a case where, the samples are evenly distributed between the two sides of the split. The biggest penalties are given to two points: $(\hat{\pi}_1^a = 0,\ \hat{\pi}_2^a = 1)$ and $(\hat{\pi}_1^a = 1,\ \hat{\pi}_2^a = 0)$. In both cases, the action a is estimated as the best action in one side, but at the same time, it is estimated as the worst action in the other side. In fact the penalty term punishes contradicting estimations on selection probabilities of a specific action in either side of the split.

4.5.3 VAR

The VAR criterion is developed on the basis of Mean Square Error (MSE) reduction technique on action-value functions of abstract states. Given that, a trial split breaks the abstract state $\bar{S}$ to $\bar{S}_1$ and $\bar{S}_2$, the preferred trial split would be the one that maximizes the amount of reduction in MSE over the whole action set:

$$VAR(\bar{S}) = \max_{a \in A} \left[\sigma^2 \left(Q(\bar{S}, a) \right) - \sum_{i=1}^{2} P(\bar{S}_i | \bar{S})\, \sigma^2 \left(Q(\bar{S}_i, a) \right) \right] \tag{4.31}$$

where A is the set of actions, and $\sigma^2(X)$ is the statistical variance of the random variable X.

Sampled data-points of action a in $\bar{S}$, i.e. $T = \{T_i = (s,a,r,s') | s \in \bar{S}\}$, can be grouped to two parts based on whether their current state falls in $\bar{S}_1$ or $\bar{S}_2$:

$$T^j = \{T_i \in T | T_i.s \in \bar{S}_j\},\ j = 1,2 \tag{4.32}$$

Action-values of action a in abstract states $\bar{S}$, $\bar{S}_1$, and $\bar{S}_2$ can be estimated by averaging the sample-values as mentioned in (4.18). According to definition of $\hat{\mu}^a$ and $\hat{\rho}_i^a$ we can write:

$$\hat{\mu}^a = \hat{\rho}_1^a \hat{\mu}_1^a + (1 - \hat{\rho}_1^a)\hat{\mu}_2^a \tag{4.33}$$

$$= \hat{\rho}_1^a(\hat{\mu}_1^a - \hat{\mu}_2^a) + \hat{\mu}_2^a \tag{4.34}$$

$$= \hat{\mu}_1^a + \hat{\rho}_2^a(\hat{\mu}_2^a - \hat{\mu}_1^a) \tag{4.35}$$

Statistical variance of $Q(\bar{S},a)$ is estimated from the samples as the following [5]:

$$\sigma^2\left(Q(\bar{S},a)\right) = \frac{1}{m^a}\sum_{i=1}^{m^a}(V(T_i) - \hat{\mu}^a)^2 \tag{4.36}$$

$$= \frac{1}{m^a}\sum_{i=1}^{m_1^a}\left(V(T_i^1) - \hat{\mu}^a\right)^2 + \frac{1}{m^a}\sum_{i=1}^{m_2^a}\left(V(T_i^2) - \hat{\mu}^a\right)^2 \tag{4.37}$$

$$= \frac{1}{m^a}\sum_{i=1}^{m_1^a}\left(V(T_i^1) - \hat{\mu}_1^a - \hat{\rho}_2^a(\hat{\mu}_2^a - \hat{\mu}_1^a)\right)^2 + \tag{4.38}$$

$$\frac{1}{m^a}\sum_{i=1}^{m_2^a}\left(V(T_i^2) - \hat{\rho}_1^a(\hat{\mu}_1^a - \hat{\mu}_2^a) - \hat{\mu}_2^a\right)^2 \tag{4.39}$$

$$\begin{aligned} = {} & \frac{1}{m^a}\sum_{i=1}^{m_1^a}\left[\left(V(T_i^1) - \hat{\mu}_1^a\right)^2 + (\hat{\rho}_2^a)^2(\hat{\mu}_2^a - \hat{\mu}_1^a)^2 - 2\hat{\rho}_2^a\,(\hat{\mu}_2^a - \hat{\mu}_1^a)\left(V(T_i^1) - \hat{\mu}_1^a\right)\right] + \\ & \frac{1}{m^a}\sum_{i=1}^{m_2^a}\left[\left(V(T_i^2) - \hat{\mu}_2^a\right)^2 + (\hat{\rho}_1^a)^2(\hat{\mu}_1^a - \hat{\mu}_2^a)^2 - 2\hat{\rho}_1^a\,(\hat{\mu}_1^a - \hat{\mu}_2^a)\left(V(T_i^2) - \hat{\mu}_2^a\right)\right] \end{aligned} \tag{4.40}$$

Knowing that $\sum_{i=1}^{m_1^a}(V(T_i^1) - \hat{\mu}_1^a) = 0$, and $\sum_{i=1}^{m_2^a}(V(T_i^2) - \hat{\mu}_2^a) = 0$, we can write:

$$\begin{aligned} \sigma^2\left(Q(\bar{S},a)\right) = {} & \frac{1}{m^a}\sum_{i=1}^{m_1^a}\left(V(T_i^1) - \hat{\mu}_1^a\right)^2 + \frac{1}{m^a}\sum_{i=1}^{m_2^a}\left(V(T_i^2) - \hat{\mu}_2^a\right)^2 \\ & + \frac{1}{m^a}(\hat{\mu}_1^a - \hat{\mu}_2^a)^2(m_1^a(\hat{\rho}_2^a)^2 + m_2^a(\hat{\rho}_1^a)^2) \end{aligned} \tag{4.41}$$

We know from (4.18) that $m_1^a = m^a\hat{\rho}_1^a$ and $m_2^a = m^a\hat{\rho}_2^a$. So, we can write:

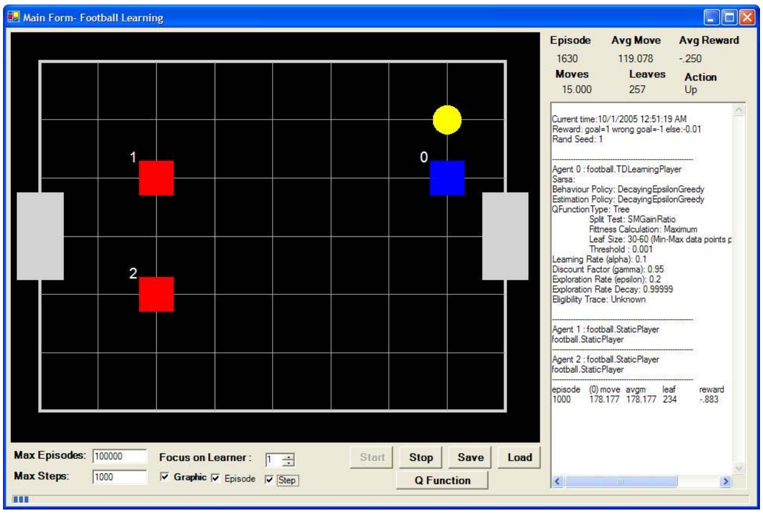
$$m_1^a(\hat{\rho}_2^a)^2 + m_2^a(\hat{\rho}_1^a)^2 = m^a\hat{\rho}_1^a(\hat{\rho}_2^a)^2 + m^a\hat{\rho}_2^a(\hat{\rho}_1^a)^2 \tag{4.42}$$

$$= m^a\hat{\rho}_1^a\hat{\rho}_2^a(\hat{\rho}_1^a + \hat{\rho}_2^a) \tag{4.43}$$

$$= m^a\hat{\rho}_1^a\hat{\rho}_2^a \tag{4.44}$$

---

[5] For unbiased estimation, $\frac{1}{m^a}$ should be replaced by $\frac{1}{m^a-1}$.



Episode 1630
Avg Move 119.078
Avg Reward -.250
Moves 15.000
Leaves 257
Action Up

```
Current time:10/1/2005 12:51:19 AM
Reward: goal=1 wrong goal=-1 else:-0.01
Rand Seed: 1

--------------------------------------------------------
Agent 0 : football.TDLearningPlayer
Sarsa:
Behaviour Policy: DecayingEpsilonGreedy
Estimation Policy: DecayingEpsilonGreedy
QFunctionType: Tree
              Split Test: SMGainRatio
              Fittness Calculation: Maximum
              Leaf Size: 30-60 (Min-Max data points p
              Threshold : 0.001
Learning Rate (alpha): 0.1
Discount Factor (gamma): 0.95
Exploration Rate (epsilon): 0.2
Exploration Rate Decay: 0.99999
Eligibility Trace: Unknown

--------------------------------------------------------
Agent 1 : football.StaticPlayer
football.StaticPlayer
--------------------------------------------------------
Agent 2 : football.StaticPlayer
football.StaticPlayer
--------------------------------------------------------
episode   (0) move  avgm      leaf        reward
1000      178.177   178.177   234         -.883
```

Max Episodes: 100000
Max Steps: 1000
Focus on Learner : 1
☑ Graphic ☑ Episode ☑ Step

Figure 4.5: A screen-shot from the simulator

Therefore, equation (4.41) can be written as:

$$\sigma^2\left(Q(\bar{S},a)\right) \quad = \quad \hat{\rho}_1^a\sigma^2(Q(\bar{S}_1,a))+\hat{\rho}_2^a\sigma^2(Q(\bar{S}_2,a))+\hat{\rho}_1^a\hat{\rho}_2^a(\hat{\mu}_1^a-\hat{\mu}_2^a)^2 \tag{4.45}$$

Putting (4.31) and (4.45) together, we can write:

$$VAR(\bar{S}) = \max_{a\in A}\left[\sigma^2\left(Q(\bar{S},a)\right) - \hat{\rho}_1^a\sigma^2(Q(\bar{S}_1,a)) - \hat{\rho}_2^a\sigma^2(Q(\bar{S}_2,a))\right] \tag{4.46}$$

$$\boxed{VAR(\bar{S}) = \max_{a\in A}\left[\hat{\rho}_1^a\,\hat{\rho}_2^a\,(\hat{\mu}_1^a-\hat{\mu}_2^a)^2\right]} \tag{4.47}$$

4.6 Simulation results

Performance of the new splitting criteria has been tested on a simplified football task. It is very similar to the classic *Taxi* problem [19], with the additional possibility of having players with different movement patterns, to create environments with different non-deterministic levels. The playing field is an 8×6 grid (Fig. 4.5). The x axis is horizontal and y axis is vertical. The goals located at $(0,3)$ and $(8,3)$.

The learning agent plays against two "passive" opponents and learns to deliver the ball to the left goal. By passive we mean the opponents move in the field, only to restrict the movements of the learner, and do not perform any action on the ball. The learner can choose among 6 actions: {*Left, Right, Up, Down, Pick, Put*}. If a player selects an action that results in touching the boundaries or going to a pre-occupied position by another player, the action is ignored. Also, if the owner of the ball hits the other player or the boundaries, the ball would fall down.

The opponents can be selected from three types: *Static, Offender,* and *Midfielder.* Static players do not move at all. Offenders move randomly in x direction (i.e. they select Left and Right actions, randomly). Midfielders move randomly in all directions.

States of the learner consist of (x,y) coordinates of the ball and the players, and status of the ball from {*Free, Picked*} states, and two terminal states for left and right goal. It sums up to $3{,}001{,}252 (= 2 \times (7 \times 5)^4 + 2)$ states. The players act in parallel. So, next state of the learner, when playing against offenders or midfielders, would be a stochastically selected state among 4 or 16 different states, respectively.

Learning experiment consists of 100,000 *episodes* of which, the last 10,000 episodes are referred as *test phase*. Episodes start by placing the players and the ball in random (and unoccupied) positions. Episodes last up to 1000 *steps*. Steps include one movement by each player in turn. An episode is finished if, either a player scores, or the maximum number of steps is passed. The learner receives +1 reward for correct scores, -1 punishment for wrong scores, and -0.01 otherwise. For each episode, the number of actions and the sum of the received reinforcement signals by the learning agent is recorded.

All methods use SARSA [18] with decaying ε-greedy for sampling phase. The learning rate, α, is 0.1 and the discount factor, γ, is 0.95. The exploration rate, ε, is initialized with 0.2. After each episode, it is decayed by multiplying with the decay factor 0.99999 . The total size of the sample lists of the leaves is 360 samples (60 samples per action).

For each state abstraction method different confidence thresholds are tried to find the best range in terms of convergence. The best range is, then, divided to 10 equal-size regions and center of each region is picked up as a confidence threshold. For each of these 10 confidence thresholds, 30 simulation runs (with different random seeds) are executed. The results that are presented in this chapter are averaged over all confidence thresholds. In Appendix A you can find the best results, acquired by the splitting criteria, among all thresholds.

The methods are compared in terms of the performance of their learned policies, the size of their learned trees, and their computation time. The performance of the policies are quantified by computing the average number of actions executed by the learner during the learning episodes (or equally the number of steps per episodes). It shows how effective has the agent learned to score in fewer moves.

As mentioned earlier, the agent receives rewards only at the end of the episodes (of course, if it scores correctly), and in the other cases it is punished. Therefore, the number of actions indirectly indicates the sum of the received reinforcement signals. The fewer the number of actions in each episode is, the more reinforcements the agent has received (except the episodes that end up with scoring in the wrong goal, which does not happen in the final episodes). That is why we do not add dedicated tables to compare the sum of the received reinforcement signals.

4.6.1 A Sample Tree

An example from the learned trees, saved from one of the simulations in the static-player case, is shown in Fig. 4.6. The static-players are located in front of the left goal at (2,2) and (2,4). The hierarchy of sub-tasks is clear in the figure. The whole task is divided to two main sub-tasks: If the learner does not own the ball, it Switches to the sub-tree that corresponds to the sub-task of navigating the learner from its current position to the ball. Otherwise, it follows the sub-tree that navigates it (while carrying the ball) to the left goal.

If the opponents occlude the learner's path to the left goal, it focuses on the obstacle-avoidance sub-task (not expanded in the figure). Otherwise, it focuses on the path to the left goal, without being worried about contacts (i.e. position of opponents are abstracted). In this case, the splits are based only on the position of the ball (which is the same as the position of the learner).

It is observed that tree structure is able to maintain not only the hierarchy of sub-tasks, but

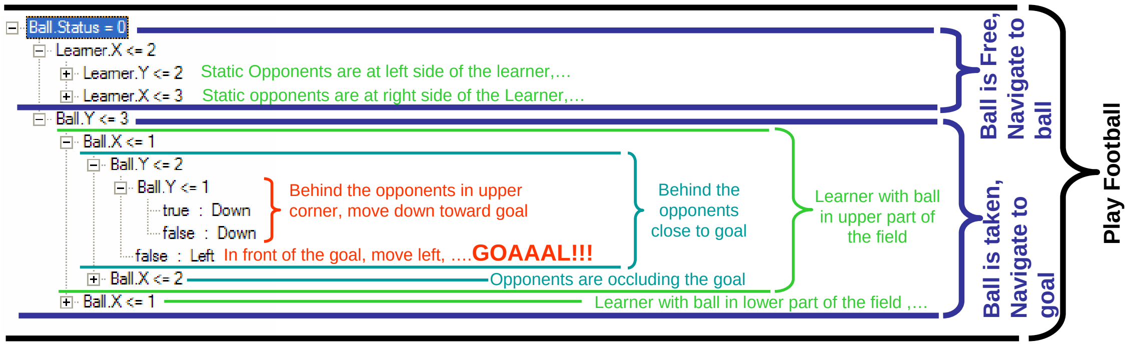

Ball.Status = 0
Learner.X <= 2
Learner.Y <= 2
Static Opponents are at left side of the learner,...
Learner.X <= 3
Static opponents are at right side of the Learner,...
Ball.Y <= 3
Ball.X <= 1
Ball.Y <= 2
Ball.Y <= 1
true : Down
false : Down
Behind the opponents in upper corner, move down toward goal
false : Left
In front of the goal, move left, ...GOAAAL!!!
Behind the opponents close to goal
Learner with ball in upper part of the field
Ball.X <= 2
Opponents are occluding the goal
Ball.X <= 1
Learner with ball in lower part of the field ,...
Ball is Free, Navigate to ball
Ball is taken, Navigate to goal
Play Football

Figure 4.6: A typical decision tree, learned in static-player case

Table 4.1: Comparing the biased and unbiased U-Tree construction methods in terms of the number of actions per episodes during the test phase. It also indirectly indicates the sum of the received reinforcement signals.

Player	KS	IGR	UKS	UIGR
Static	11.25±0.05	11.52±0.11	11.52±0.11	11.54±0.1
Offender	15.1±0.14	14.7±0.1	14.93±0.18	14.66±0.17
Midfielder	18.9±1.15	18±0.44	17.76±0.28	17.47±0.64

also the priority among them in different situations.

4.6.2 Unbiased Sampling

Table 4.1 compares biased versions of U-Tree construction methods with unbiased versions in terms of policy performance i.e. the number of actions per episodes during the test phase. UKS and UIGR are new versions of KS and IGR with unbiased sampling, respectively. Confidence intervals are computed with significance level α=0.05. Table 4.1 shows as environment becomes more non-deterministic, positive effect of unbiased sampling appears more. Although, the number of actions in UKS and UIGR is a little bit bigger than KS and IGR for static players, it is much smaller in midfielder case.

Table 4.2 compares the number of leaves in the learned trees. It is clear that, the number of leaves in the unbiased versions are always fewer than the biased versions. In the midfielder case, the unbiased versions select the splits in a more optimal way because, not only their trees are

Table 4.2: Comparing the biased and unbiased U-Tree construction methods in terms of the number of leaves in the learned trees

Player	KS	IGR	UKS	UIGR
Static	719±20.7	650±11.9	579±15.3	576±16.8
Offender	1724±113.2	1563±150.3	1365±52.2	1412±115.8
Midfielder	1960±141.1	1947±332.8	1639±61.2	1623±102.9

Table 4.3: Comparing the learning methods in terms of the average number of actions per episode during the test phase. It also indirectly indicates the sum of the received reinforcement signals.

Player	SARSA	KS	IGR	SANDS	SMGR	VAR
Static	10.85±.01	11.25±.05	11.52±.11	11.18±.02	11.18±.01	11.17±.02
Offender	19.63±.06*	15.1±.14	14.7±.1	14.46±.04	14.44±.03	14.59±.06
Midfielder	668±.1**	18.9±1.15	18±.44	17.08±.07	17.07±.06	17.26±.09

* 14.72 after 10^6 episodes ** 18.92 after 15×10^6 episodes

Table 4.4: Comparing the learning methods in terms of the average number of leaves in the learned trees. For flat Q-learning with SARSA, the number of states is mentioned.

Player	SARSA	KS	IGR	SANDS	SMGR	VAR
Static	2452	719±20.7	650±11.9	466±4.1	505±4.2	472±4.9
Offender	120052	1724±113.2	1563±150.3	1231±15.4	1194±29.5	1288±22
Midfielder	3001252	1960±141.1	1947±332.8	1406±31.6	1498±47.5	1523±39.3

smaller, but also their policy performance is higher. In the static and the offender case, the rather poor performance of the unbiased versions could possibly be due to having smaller (in fact, not enough big) trees.

The confidence intervals of the unbiased versions are always samller than the biased ones. This means that, the number of leaves in the generated trees by the biased versions change a lot. This confirms our arguments that the splits, selected by the biased versions, are not optimal sometime; therefore, some sub-trees have to be repeated multiple times.

4.6.3 Policy Performance and Tree Size

Table 4.3 shows the average number of actions per episodes during the test phase for flat SARSA and different HRL methods. Table 4.4 shows the average number of leaves in the learned trees. Results of KS and IGR are duplicates of Sec. 4.6.2 for simplicity of comparison.

Although the flat SARSA has better performance against the static players, it has quite poor performance in the other cases. Even, we let the learning agent prolong the learning episodes to 10 and 150 times more for offender and midfielder cases, respectively. But, its performance is still poor compared to performance of the hierarchical methods.

Table 4.3 shows in all cases, our proposed criteria –SANDS, SMGR, and VAR– create trees with better performance than the other methods. The difference becomes more clear when the environment becomes non-deterministic. In midfielder player case, differences between the proposed criteria and the existing ones are between one and two actions per episode. Table 4.4 shows our proposed criteria generate smaller trees in all cases. The difference becomes bigger as the environment becomes more non-deterministic.

The results confirm that, SANDS, SMGR, and VAR select the splits in a more optimal manner because, not only their tree is smaller, but also their policy performance is higher. Among our proposed criteria, SMGR shows a small enhancement in offender and midfielder case compared to SANDS and VAR. It is due to the penalty term that SMGR assigns to the splits and rejects

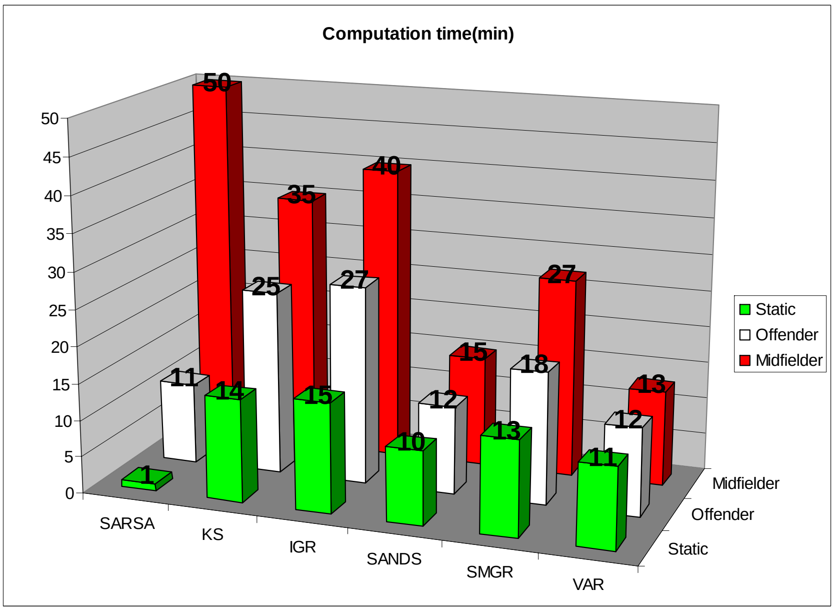


Figure 4.7: Computation time of the algorithms in minutes for 100,000 learning episodes. The flat SARSA needs 16 min to run 10^6 episodes in offender case, and 370 min for 15×10^6 episodes in midfielder case.

some inefficient ones. However, SMGR generates slightly bigger trees than SANDS. In static and offender case, it is not evident that which one generates fewer partitions, but in midfielder case, SANDS generates trees with around 100 leaves less. In terms of both measures, VAR stands after SANDS and SMGR in the third place.

4.6.4 Computation Time

Fig. 4.7 compares simulation time of the different methods [6]. It is observed in deterministic cases that, the flat SARSA needs only one minute to learn playing against the static players. However, when the environment becomes more non-deterministic, the required computation time increases rapidly.

An interesting result for HRL algorithms is that, the required learning time does not increase with the same rate as the flat SARSA. Although the order is still exponential, the increment rates are visibly smaller than that of the flat SARSA. These results show HRL have a great potential in decreasing the required learning time and the number of trials for online robot learning tasks.

As explained in Sec. 4.4.3, time order of SANDS, SMGR, and VAR is less than KS and IGR. The difference is not clear in the static player case, but in other cases, SANDS and VAR decrease the required computation time to more than half compared to KS and IGR. Among all methods, VAR and then SANDS are the fastest.

A part of the reduction in the computation time is due to the reduction in the time order

---

[6] Tests were done on a Dell® Inspiron™5150 laptop with CPU@3.06 MHz. However, since a part of the computation time depends on e.g. CPU, operating system, and compiler we compare the methods only by minutes.

of the sort algorithm in the sample lists. Another part is due to more efficient splits and as a result reduction in the number of actions per episodes. Another part is due to creating smaller trees. This is important, because it also affects the search speed and the required memory for the trees.

4.7 Conclusion

In this chapter we, formalized the state abstraction problem and derived three new criteria for split selection called SANDS, SMGR, and VAR. SANDS tries to maximize the expected reward return by finding a point that well differentiates both value and selection probability of actions, before and after introducing a split. SMGR looks for the splits that results in more certainty on selection probability of actions. VAR minimizes the MSE on the action-value functions of the abstract states over the whole action set. The new criteria adapt decision tree learning techniques to the state abstraction task. We showed in simulation that, the new criteria outperform the existing ones, not only in efficiency and size of the learned trees, but also in computation time.

Other splitting criteria, like Information-Gain [111], Gini-Index [112] and Student's T-test [31] were also tested, but the results were with limited success, as reported in [31]. Because of a variance term in its divisor, the Student's T-test has poor performance when the variance gets close to zero. Moreover, Information-Gain and Gini-Index had very poor performance in non-deterministic environments, even worse than KS and IGR.

The main reason behind the performance of our proposed methods is that, criteria like KS and IGR judge the splits based only on the "rank" of the sample-values, without taking their magnitude into account. However, value of samples is very informative for state abstraction. SANDS, SMGR, and VAR exploit this information.

Among the proposed criteria, SANDS generate the smallest trees. After SANDS, SMGR often generates smaller trees than VAR. In the other hand, SMGR generates the most efficient trees, however it needs longer time to execute. In terms of efficiency of the generated trees, SANDS criterion stands higher than VAR in the second place. Among the proposed criteria, VAR runs faster than SANDS and SMGR. However, SANDS stands in the second place in terms of execution time. In terms of all quality measures together, SANDS is the best one, since it generates the smallest trees, its efficiency is very close to SMGR, and its execution time is very close to VAR. However, attention must be paid when applying the split criteria to other tasks, as the mentioned ranking among the methods might slightly change.

At first look, size of the trees does not seem to be an important issue. However, our experiences shows it becomes a critical issue in large state spaces. Methods that create big trees completely fill memory of robot very early and will not allow the robot to enhance its knowledge anymore. The learning robot have to incorporate the splitting criteria that generate smaller trees in advance, or it should find a ways to filter the generated trees and make them more lightweight. This issue will be discussed in the next chapter.

Although the proposed criteria generate smaller trees than the existing criteria, we think their size are still big and inefficient to be applied to more complex tasks like realistic soccer learning. Like all Univariate-based methods, inability to revise the tree in non-stationary environments is another drawback of our methods. Using Multivariate splits [108; 109; 110] are perhaps more suitable candidates, because they use a linear or non-linear combination of the features at the split point. This can greatly reduce the number of required leaves. Moreover, the linear weighted sum mechanism in the Linear Multivariate Decision Trees [109] allows for revising the learned tree, through changing the weights assigned to the features in a split point.

Chapter 5

Hierarchical Cooperative Learning

Masoud Asadpour, **Majid Nili Ahmadabadi**[1], and **Roland Siegwart**

Published in:

IEEE/RSJ Conference on Intelligent Robots and Systems [IROS], 2006 [113]

Abstract. Decision trees, being human readable and hierarchically structured, provide a suitable mean to derive state-space abstraction and simplify the inclusion of the available knowledge for a Reinforcement Learning (RL) agent. In this chapter, we address two approaches to combine and purify the available knowledge in the abstraction trees, stored among different RL agents in a multi-agent system, or among the decision trees learned by the same agent using different methods. Simulation results in non-deterministic football learning task provide strong evidences for enhancement in convergence rate and policy performance, specially in heterogeneous cooperations.

Key words: Decision Tree, Merge, Balance, Pruning, Cooperative Reinforcement Learning, Hierarchical State Abstraction, Heterogeneous Cooperation

5.1 Introduction

The existing learning methods suffer from the curse of dimensionality, i.e. requiring a large number of learning trials as state-space grows. RL [18], despite its strength in handling dynamic and non-deterministic environments, is an example of such learning methods.

On the other hand, agents in a multi-agent system –although dealing with more dynamic and as a result tougher learning task– confront with two rich sources of knowledge which, if appro-

---

[1] Control and Intelligent Processing Center of Excellence, ECE Dept., University of Tehran, Iran, mnili@ut.ac.ir

priately exploited, could greatly expand horizon of the learning process: *the learned knowledge in the subtasks*, and *the existing knowledge in the agents*.

If agents encounter a multitude of learning subtasks over their entire learning time, there is not only an opportunity to transfer knowledge between them [114], but also a chance to generalize subtasks and construct a hierarchy among them. It is believed that the curse of dimensionality can be lessen, to a great extent, by implementation of state abstraction methods and hierarchical architectures. Moreover, incremental improvement of agent's performance becomes much simpler.

Most of the different approaches, proposed so far for state abstraction and hierarchy creation, require a pre-designed sub-task hierarchy (HAM [23], MaxQ [24], FeudalQ [29]). In addition to requiring intensive design effort for explicit formulation of the hierarchy and abstraction of the states, the designer should, to some extent, know how to solve the problem before s/he designs the hierarchy and subtasks. To simplify this process, automated state abstraction approaches use decision-tree learning methods and incrementally construct the abstraction hierarchy from scratch [31] [32] [97].

Every learning agent in a multi-agent system constructs it's own individual abstraction hierarchy. However, due to confronting with different situations or heterogeneity in the abstraction method that each agent utilizes, those abstraction hierarchies might be quite different in essence. Moreover, wider spatial and temporal coverage of a group [115] brings a potential benefit out of sharing, adopting, adapting, and applying the stored knowledge in the group, in a way that helps individuals to build more deliberate and accurate abstraction levels.

Many researches have been inspired by cooperative learning schemes in human and animal society. Advice taking [116], imitation [117], ensemble learning [118], and strategy sharing [119] are among many recent researches. However, none of them address utilization of both hierarchical and cooperative learning methods.

In this chapter, we present two methods to combine the output or the structure of the abstraction trees. The abstraction trees are learned online by different RL agents in the society or by the same agent but with different abstraction methods. We also introduce the methods and the criteria to balance and prune the merged decision trees, in order to purify and condense the shared knowledge, and prepare it for further developments.

This chapter is structured as follows: The next section overviews the most recent works on Cooperative RL (CRL), Hierarchical RL (HRL), and combining decision trees. The third section introduces the new algorithms to combine the output of the abstraction trees or to combine their structure through merge, balance, and pruning algorithms. The fourth section describes the simulation task and results. Conclusions and future works are discussed finally.

5.2 Related Works

This section give an overview of the most recent works on CRL, HRL, and combining decision trees and explains the ways our work is distinct

5.2.1 Cooperative Reinforcement Learning

Cooperative Learning attempts to benefit from the existing knowledge, stored in the knowledge base of different agents, through explicit and implicit exchange of the learned rules, gathered information, etc. This term is sometimes confused with, what we call *Cooperation Learning* [120], which is learning how to coordinate a joint action in cooperative multi-agent systems. We focus our attention to the methods that enable a group of agents to share their learned knowledge with their colleagues, instead of methods that require a mentor or a higher levels of cognition like imitation [117] and advice taking [116].

Cooperative learning makes continuous learning possible, regardless of the power autonomy of the individuals [115]. A group of robots is able to accumulate knowledge about the task to be learned and pass it across generations. Then, eventual "death" of a robot would not lead to lose of the learned knowledge. It is shown that, a group of robots with short life-spans benefit more, from cooperative learning, than those with long life-spans [115].

Cooperative RL aggregates different approaches from sensor sharing to strategy sharing. In its very basic form, agents serve as a scout by sharing sensory data in order to augment their eyesight [121]. In *Episode Sharing*, agents share *(state, action, reward, next state)* triples letting their colleague explore in a virtual world [121]. Although requiring long communication time, sharing Q-tables is a reasonable supplement, provided that validity and level of their encoded knowledge is regarded.

Simple Strategy Sharing [121] blends Q-table of agents into one unique table through averaging the corresponding cells. However it homogenizes the Q-tables and is unsuitable combination when levels of expertise of the agents are different. *Weighted Strategy Sharing (WSS)* [122; 123; 119] tries to resolve the problem by introducing some *expertness* measures and assigning different weights to the Q-tables accordingly. The proposed expertness is evaluated as a function of the history of the received rewards and punishments (e.g. sum of the absolute values of the received signals), as a function of action-values (e.g. certainty measure or entropy functions [124]), or as a function of state transitions [125].

As agents might have quite different experiences, they might be expert in some parts of the state-space and novice in other areas. Extensions of WSS try to discover the *areas of expertise* by measuring their expertness in state-level [124; 125; 126] or using pattern recognition techniques [127]. Being too much elaborated, these methods becomes impractical when state space enlarges.

In this chapter decision trees help us categorize similar states to one group by abstraction techniques. This provides the chance to enlighten the expertness assessment in state-level and decrease the amount of communication. Moreover, since decision trees are more human readable than Q-tables, combination of them would also result in interpretable rule sets.

5.2.2 Hierarchical Reinforcement Learning

Techniques for non-uniform discretization of state space are already known e.g. Parti-game [30], G algorithm [101], and U-Tree [31]. U-Trees use decision tree to incrementally derive the abstraction hierarchy. Continuous U-Tree [32] extends U-Tree to work with continuous instead of only propositional features.

In the previous chapter, we showed that the existing U-Tree based methods ignore explorative nature of RL. This imposes a bias on the distribution of the samples saved for introducing new splits in U-Trees. As a consequence, finding a proper split point becomes more and more difficult and the introduced splits are far from optimality.

Moreover, since U-Tree-based techniques have been excerpted in essence from decision tree learning methods, the splitting criteria that they utilize are very general. By reformulation of state abstraction with different heuristics, three specialized criteria (SANDS, SMGR, and VAR) were derived, and their efficiency was compared to some widely used ones, like non-parametric Kolmogorov-Smirnov and Information Gain Ratio tests.

5.2.3 Combining Decision Trees

Having abstraction trees provided by different sources, we are interested in combining them and extracting an enhanced interpretation. A similar problem has already been studied in data mining on very large data sets, distributed over a network of computers. Hall et al [128] describe an approach for classification problem in which, decision trees are learned from disjoint subsets of a large data set. In combination process, the learned trees are converted to rule sets, similar rules are combined into a more general one, and contradicting ones are resolved. Although suitable for off-line classification, it is not applicable to continuous online learning, since the final rule set is not (in general) convertible back to a decision tree. This is necessary in our case for further improvement and learning. Moreover, generalization and conflict resolution techniques are context-dependent and must be adapted to our case.

Another direction of combining multiple trees, known as ensemble methods, is to construct a linear combination of the outputs of some model fitting methods, instead of using a single fit. Bagging [129] involves producing different trees from different sample sets of the problem domain (or from one training set) in parallel and then aggregating the results on a set of test instances. Boosting [130] is another technique that involves generating a sequence of decision trees from the same data-set, whereby attention is paid to the instances that have caused error in previous iterations. Ultimate output of the ensemble is specified by majority voting or weighted averaging.

We will apply both merge and ensemble techniques to state abstraction task. However, in our merge algorithm, generalization and conflict resolution will be solved by applying cooperative learning techniques to leaf combination procedure.

5.3 Combining Abstraction Trees

Cooperative learning among abstraction trees [2] could be realized by combining either their output or their structure. The former is called *Ensemble Learning.* For the latter, we introduce a new algorithm to merge, balance, and prune the abstraction trees. It is called *CLASS*, which stands for *Combining Layered Abstractions of State Space.*

Cooperative learning enables us have heterogeneous cooperation between different abstraction algorithms. Here, heterogeneity refers to diversity in the abstraction methods, which can arise from variations, either in parameter settings, or splitting criteria. We argue that, introducing the heterogeneous cooperation would simplify the tough task of method selection or parameter adjustment for abstraction methods.

5.3.1 Ensemble Learning

We use the idea of Bagging [129] and Boosting [130] and let the learning agent construct multiple abstraction trees in parallel from the same learning episodes (Fig. 5.1). It is hoped that,

---

[2] To review abstraction methods using U-Tree please read Sec. 4.4 and 4.5.

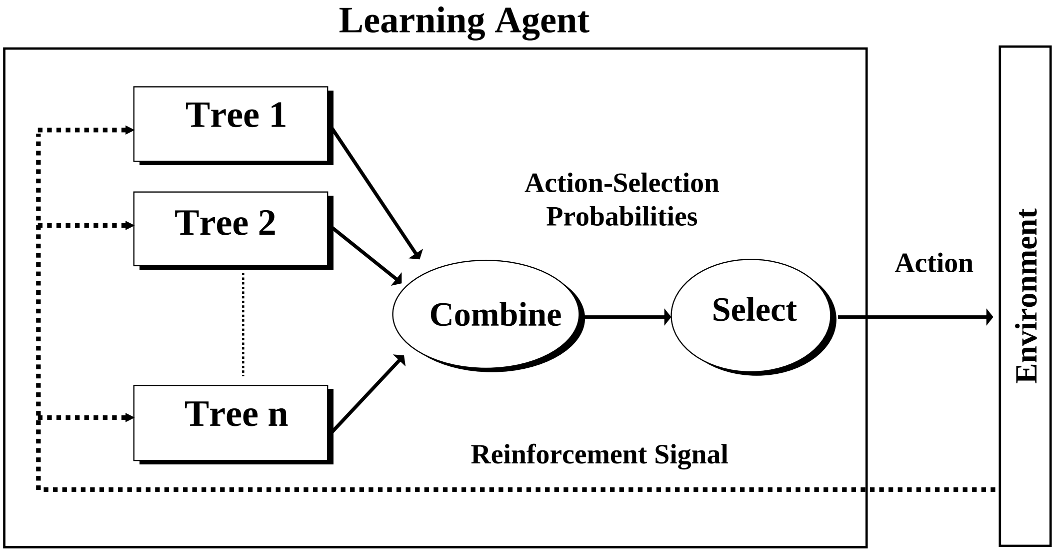


Figure 5.1: Ensemble of abstraction trees jointly selects the next action

the ensemble guide the abstraction trees to get specialized in different areas of state-space by introducing splits in different axis.

In action-selection phase of the ensemble, the agent combines the probability distribution of actions proposed by each tree by simple averaging. The action is, then, selected according to this combined distribution. Finally, the reinforcement signal is feeded back to all trees. Combining the outputs via averaging is reasonable, since all trees receive the same reinforcement signals and therefore their expertness measures are equal.

5.3.2 CLASS

The CLASS algorithm includes applying sequences of merge, balance, and pruning procedures to abstraction trees. We explain the procedures in this section. The goal is to put the knowledge, acquired from different sources, together and purify it. However, the balance and the pruning algorithms are not constrained to cooperative learning application. They could be applied to any abstraction tree in general.

The CLASS algorithm could start by merging the two abstraction trees. Then, the resulted tree is balanced and pruned. Another sequence could be to apply the balance and the pruning procedures twice, once to the abstraction trees before merge, and once to the resulted tree after merge. Our experiences show that, this way the resulted tree after merge is more lightweight. As a consequence, the balance procedure, which is the bottleneck of the algorithm, is accomplished faster.

Merge

Given the abstraction trees, T_1 and T_2, learned on the same state-space, a new decision tree $T_3 = T_1 + T_2$ is formed from pairwise intersection of the leaves of T_1 and T_2. The merge algorithm in our case is non-commutative i.e. in general $T_1 + T_2 \neq T_2 + T_1$. However, the final partitions are

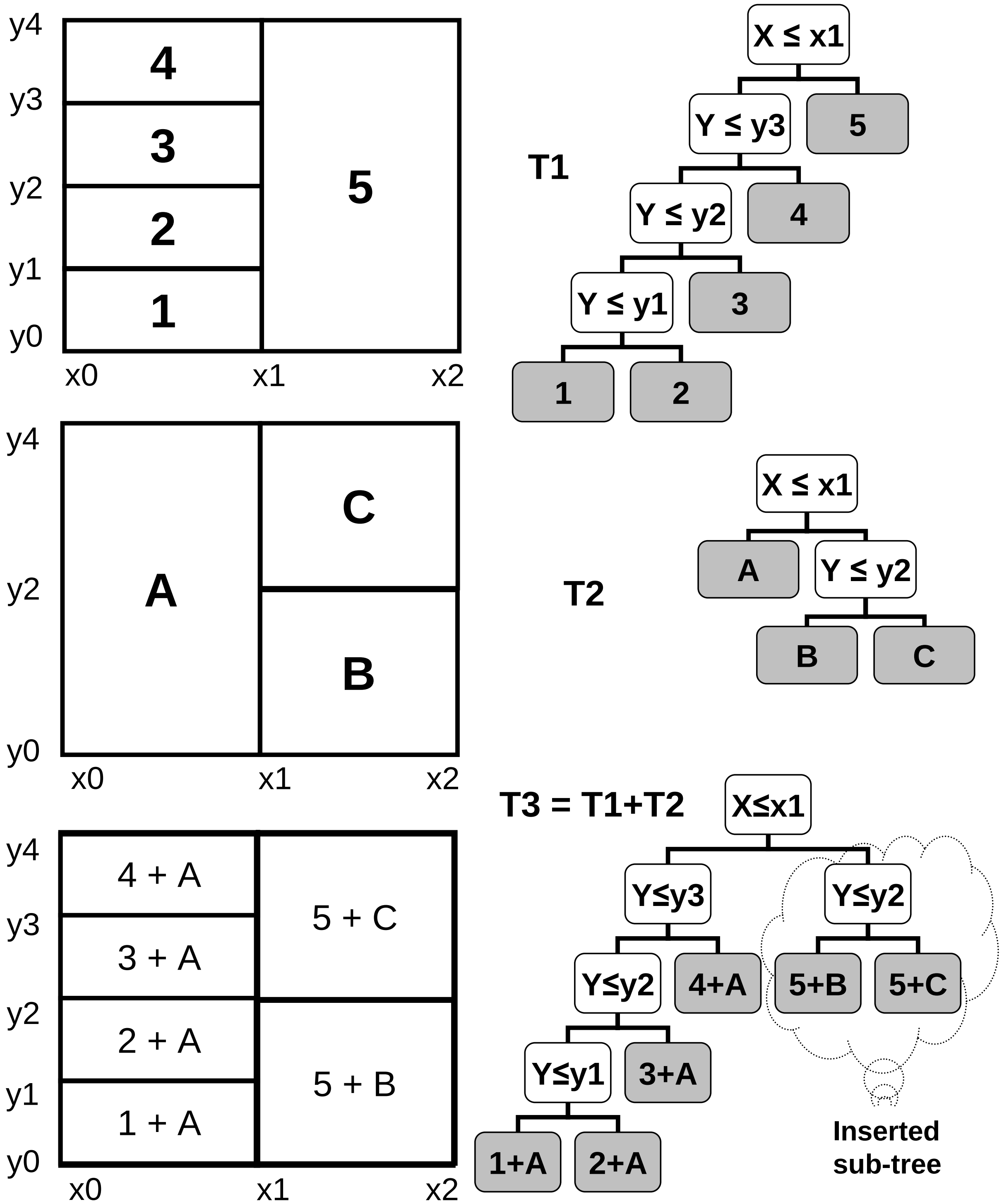


Figure 5.2: An example for merging abstraction trees. top and middle: input abstraction trees, bottom: merge result

Algorithm 1 Merging abstraction trees

Input: Two abstraction trees with root nodes r_1 and r_2

Output: Root node of the merged tree

Node MergeTree(Node r_1, Node r_2)

```
if r1.Boundary overlaps with r2.Boundary then
    if r2 is Leaf then
        MergeLeaf(r1,(Leaf)r2)
    else
        $r_1$ = MergeInternalNode($r_1$, (InternalNode)$r_2$)
        $r_1$ = MergeTree($r_1$, ((InternalNode)$r_2$).Left)
        $r_1$ = MergeTree($r_1$, ((InternalNode)$r_2$).Right)
    end if
end if
return $r_1$
```

Node MergeInternalNode(Node n_1, InternalNode n_2)

```
if $n_1$.Boundary overlaps with $n_2$.Boundary then
    if $n_1$ is Leaf then
        if $n_1$.Boundary is splittable by $n_2$.Split then
            Create a new internal node with the same split as $n_2$
            Duplicate $n_1$
            Place $n_1$ and its duplicate as children of the new internal node
            Update their boundaries and pointers
            return the new internal node
        end if
    else
        $n_1$.Left = MergeInternalNode($n_1$.Left, $n_2$)
        $n_1$.Right = MergeInternalNode($n_1$.Right, $n_2$)
    end if
end if
return $n_1$
```

void MergeLeaf(Node n_1, Leaf l_2)

```
if $n_1$.Boundary overlaps with $l_2$.Boundary then
    if $n_1$ is Leaf then
        Combine $n_1$.QFunctions with $l_2$.QFunctions (e.g.by averaging)
    else
        MergeLeaf($n_1$.Left, $l_2$)
        MergeLeaf($n_1$.Right, $l_2$)
    end if
end if
```

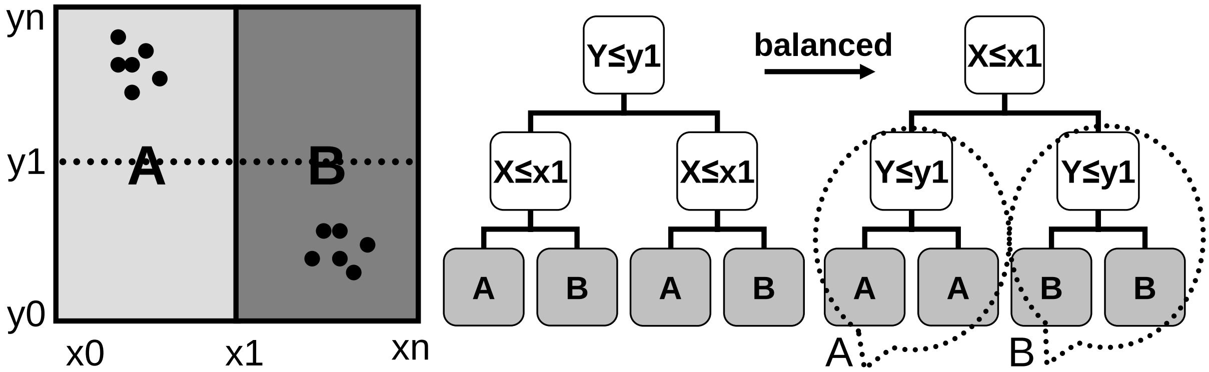


Figure 5.3: An example to show efficiency of the balance and the pruning algorithms in state abstraction. Left:actual partitions, Middle:inefficient partitioning, Right:balanced tree in terms of distance between Q-functions of leaves.

same. Merging can be illustrated by simply overlaying the two partitions on top of each other, as shown in left column of Fig. 5.2 for a simple 2D state-space. T_1 and T_2 are different abstractions of the same state-space and T_3 shows the overlayed partitions. The pseudo-code in Algorithm 1 describes the merge algorithm.

It is assumed in the algorithm that a *node* is either a *leaf* or an *internal node*. Every node has a *boundary* which specifies its corresponding *hypercube* in the state-space e.g. boundaries of the hypercube marked as 3 in Fig. 5.2 (top) is $[x_0 \rightarrow x_1, y_2 \rightarrow y_3]$. Hypercubes that correspond to leaves are *basic* hypercubes whereas the ones that correspond to internal nodes are *complex*. Internal nodes denote splits points. A *univariate* split, $X = a$, is a *hyperplane* that breaks a hypercube from point a along X axis to two smaller hypercubes (From now on, we refer to univariate splits as "split" simply). Internal nodes have two children specified by *left* ($X \leq a$) and *right* ($X > a$). A hypercube(or equally a boundary) is *splittable* by a split, $X = a$, if the split point, a, is between (and not on) the start and the end point of the hypercube boundary on the split axis, X.

The merge algorithm starts by traversing T_2 in pre-order [3] and inserting the visited nodes one by one into T_1, if required (MergeTree function in Algorithm 1, see also the trees in the right column of Fig. 5.2). If the boundary of a leaf of T_1 is splittable by a split in T_2, that split is initiated in T_1 by replacing the leaf with a sub-tree, consisting the split as root and the leaf and its copy as children (MergeInternalNode function in Algorithm 1, see also the inserted sub-tree at the bottom-right of Fig. 5.2). Merging the leaves of T_2 is done by combining their Q-functions with the one of the overlapping leaves of T_1 (MergeLeaf function in Algorithm 1, see also the leaves of T_3 in Fig. 5.2). Leaf combination process is carried out by applying WSS (Sec. 5.2.1) and expertness measures.

Balance

Mapping from partitions to abstraction trees is not always one-to-one. While only one representation is possible for T_2 (Fig. 5.2-middle), multiple trees can represent T_1 (Fig. 5.2-top). The structure of the trees, produced by abstraction methods, depends on the algorithm and sequence

---

[3] First the root, then the left sub-tree, finally the right sub-tree

Algorithm 2 Balancing an abstraction tree

Input: An abstraction tree

Output: The balanced tree

Tree Balance(Tree tree)

```
NodeList[] list = new NodeList[tree.LeafCount]
for i = 0 to tree.LeafCount-1 do
    list[0].Add(tree.Leaf[i])
end for
for i = 2 to tree.LeafCount do
    for j = $\frac{i}{2}$ down to 1 do
        NodeList $n_1$ = list[j-1]
        NodeList $n_2$ = list[i-j-1]
        if $n_1 \neq n_2$ then
            for p = $n_1$.Count-1 down to 0 do
                for q = $n_2$.Count-1 down to 0 do
                    Combine(list, $n_1$[p], $n_2$[q])
                end for
            end for
        else
            for p = $n_1$.Count-1 down to 1 do
                for int q = p-1 down to 0 do
                    Combine(list, $n_1$[p], $n_2$[q])
                end for
            end for
        end if
    end for
end for
Tree newtree = Build tree recursively from list[tree.LeafCount-1][0]
return newtree
```

void Combine(NodeList[] list, Node n_1, Node n_2)

```
if $n_1$ and $n_2$ are combinable then
    $n_{12}$ = $n_1 + n_2$
    best = list.FindNode($n_{12}$.Boundary)
    if best $\neq$ null then
        if $n_{12}$.BalanceFactor > best.BalanceFactor then
            list.Replace(best, $n_{12}$)
        end if
    else
        list.Add($n_{12}$)
    end if
end if
```

of the incoming samples (experiences). Although the structure is not important for the final policy -since all represent the same policies- it becomes crucial when traversing the tree to find the value of a specific state, and when pruning the tree to form a lightweight tree with "nearly" same efficiency.

For the decision trees that are constructed from off-line data, restructuring the tree is not relevant. however, when constructing abstraction trees from online data, we must be able to restructure the tree, since the already introduced splits might be inefficient. Fig. 5.3 (left) shows a simple case where the whole state-space is representable by only one split at x_1 and two abstract states. Now, if by chance the sampled data points are distributed like the dark circles, the abstraction algorithm might introduce a split at y_1 followed by the splits at x_1(Fig. 5.3-middle). In reality, this happens very often, since number of the sampled data-points are very few comparing to the size of the state-space. Moreover, state transitions that an online learning agent encounters would actually occur in vicinity.

Utile Distinction Memory [131] partially solves this problem by virtually generating all potential splits. When the agent is insured of the efficiency of a virtual split, it would upgrade it to an actual one. This method is not applicable to continuous features as the number of potential splits grows enormously.

A possible solution would be the combination of *balance* and *pruning* mechanisms. The balance algorithm restructures the tree according to a *balance factor* (see below). By choosing a proper balance factor inefficient splits are shifted down to leaves and removed by the pruning procedure. The pseudo-code in Algorithm 2 describes the balance algorithm.

The algorithm tries every possible tree to find the best balanced tree for the whole space. However, the algorithm does not recursively break the whole state-space in top-down manner. Instead, in order to reduce the time order of the algorithm, it combines the smaller hypercubes and build bigger hypercubes in a bottom-up manner. In top-down manner, some sub-problems (i.e. balancing subsets of state-spaces) are solved multiple times, while in bottom-up manner, the sub-problems are solved only once and the solution is saved and reused afterwards.

The algorithm starts from the basic hypercubes i.e. leaves. All leaves are tried in turn and are combined with other combinable leaves to form complex hypercubes (Balance function in Algorithm 2). Two non-overlapping hypercubes (or equally nodes) are *combinable*, if their combination also forms a hypercube, i.e. they are neighbors and their boundaries are same in all except one axes, e.g. leaf 4 in Fig. 5.2 (top-left) has two neighbors: 3 and 5, however it is combinable only with 3.

A list records the root split for the sub-spaces that have been balanced up to now e.g. this list stores $X \leq x_1$ as the root split of $[x_0 \rightarrow x_2, y_0 \rightarrow y_4]$ in Fig. 5.2(middle). If two hypercubes are combinable, then this list is checked to see whether their combination proceeds to a more balanced tree for the combined sub-space or not. If so, the new combination replaces the existing one (Combine function in Algorithm 2).

Finally, the best root split for the whole space is turned out. Putting this split as the root of the balanced tree, we can recursively build the whole tree by finding the boundaries of the sub-spaces at left and right side of the root split. The list is recursively looked up to find the root split of these two sub-spaces and so on.

Different balance factors generate different types of balanced trees. By choosing the inverse of *average path-length from leaves to root* as the balance factor, the balanced tree would have the shortest depth among all possible trees. Fig. 5.4 shows the balanced version of T_3 in Fig. 5.2 in

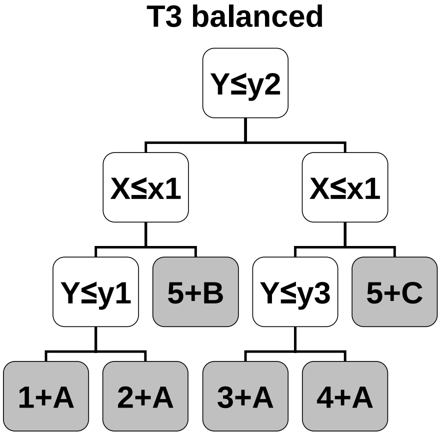


Figure 5.4: Balanced version of T_3 in Fig. 5.2 in terms of depth

terms of this factor. Combining path-length with *the fraction of samples that belong to a specific node* generates abstraction trees that are optimized for search purpose in a way that, bigger sub-spaces of the whole space are moved up, close to root, to minimize the number of conditions that are checked until a leaf is found.

Pruning

To restructure the abstraction trees for pruning purpose, the balance factor could be defined based on a *similarity measure* on the action-values of the sibling nodes. Balance according to this measure restructures the tree so that nodes with similar action-values are placed in vicinity. If the similarity between the sibling leaves is more than a threshold, they could be pruned and replaced (including their immediate parent) by a single leaf.

In this chapter, similarity of two nodes is defined as a binary function: one, if the greedy actions [4] of the nodes are equal, and zero otherwise. The balance factor for an internal node is recursively defined as: the similarity between its children (0 or 1), plus sum of the balance factors of its children.

The greedy action of a leaf is easily specified from its action-values. Since internal nodes do not hold the action-values, their equivalent values have to be computed. Action-values of an internal node, $\bar{S}$, is a weighted sum of the action-values of its children, $\bar{S}_1$ and $\bar{S}_2$:

$$Q(\bar{S},a) = \sum_{i=1}^{2} \left[P(\bar{S}_i|\bar{S}) Q(\bar{S}_i,a) \right], \forall a \in A \tag{5.1}$$

where the assigned weight to each child is the probability $P(\bar{S}_i|\bar{S})$, which is estimated by the fraction of samples that belong to the child, $\bar{S}_i$, out of the total samples that belong to the internal node, $\bar{S}$.

Fig. 5.3 (right) shows the balanced version of the abstraction tree at its left according to this balance factor. The split at y_1 is moved down and the more efficient split, x_1, is placed at root. Since the children of y_1 splits both represent the same action-values (and so they have the same greedy actions), they can be replaced by one leaf.

---

[4] The action with the highest action-value

Table 5.1: Results of the individual and ensemble learning

Method	leaf/ tree	action lrn	action tst	reward lrn	reward tst	score ratio lrn	score ratio tst
SANDS	1723	136.4	43.2	.679	.894	3.2	13.9
SMGR	1665	123.6	36.9	.718	.931	4.3	18.3
VAR	1827	179.2	54.7	.663	.919	3.3	17.9
ind. avg.	**1738**	**146.4**	**44.9**	**.687**	**.915**	**3.6**	**16.7**
SANDS-SANDS	1660	42.8	22.3	.798	.919	5.0	15.7
SMGR-SMGR	1729	57.2	23.9	.801	.942	5.5	22.1
VAR-VAR	1864	61.5	23.5	.784	.912	4.7	14.0
homog. avg.	**1751**	**53.8**	**23.2**	**.794**	**.924**	**5.1**	**17.3**
SANDS-SMGR	1666	37.5	21.4	.862	.956	7.5	27.3
SANDS-VAR	1740	45.5	21.2	.787	.920	4.7	15.4
SMGR-VAR	1780	40.1	21.4	.837	.943	6.4	22.8
heterog. avg.	**1729**	**41.0**	**21.3**	**.829**	**.940**	**6.2**	**21.9**

5.4 Simulation Results

The presented algorithms have been tested on the simplified football task, explained in sec. 4.6. However this time, the learning agent plays against an intelligent opponent. The intelligent player is programmed manually to move toward the ball, pick it and carry it to the left goal. Meanwhile, with certain probability (here 40%) it selects a wrong action to leave a bit chance for the learner to score.

Here, the total number of states is 85752($= 2 \times (7 \times 5)^3 + 2$) states. The learner receives +1 reward for correct scores, -0.1 punishment for wrong scores or scores by the opponent, and -0.0001 otherwise. All methods use SARSA with decaying ε-greedy for sampling phase. The learning rate, α, is 0.1 and the discount factor, γ, is 0.95. The exploration rate, ε, is initialized to 0.1 and is decayed by multiplying with 0.99999 after each episode. Leaf sample size is 360 (60 samples per action).

Three type of splitting criteria are examined: SANDS, SMGR, and VAR. For each state abstraction criteria, different confidence thresholds are tried in order to find the best range in terms of convergence. The best range is then divided to 10 equal-size regions and center of each region is picked up as a confidence threshold. Simulation experiments are executed with these 10 confidence thresholds.

5.4.1 Ensemble Learning Results

Table 5.1 shows final results of the *individual* and the *ensemble* learning experiments in both *homogeneous* and *heterogeneous* cooperation cases. The individual learning experiments refer to the experiments that engage only one abstraction tree. In this category, for every splitting criterion and confidence threshold, 10 simulation runs (with different random seeds) are executed. The ensembles engage two abstraction trees in parallel. In this category, all possible two-combinations of the splitting criteria and the confidence thresholds, that has been used in the individual learning experiments, are tried in turn.

Heterogeneity, here, refers to the difference between the two splitting criteria used in the ensemble. For example, SMGR-VAR is considered as a heterogeneous cooperation between SMGR and VAR. It refers to all ensembles which create two abstraction trees, one of them using SMGR and the other one using VAR. While, e.g. VAR-VAR, is considered as a homogeneous cooperation. It refers to all ensembles in which, both trees use VAR criterion (with different confidence thresholds, of course).

The first column of Table 5.1 shows the number of leaves in the generated trees averaged over all the ensembles of a specific type. The rest show the average number of actions per episodes, the average sum of the received rewards (either positive or negative) during episodes, and the score ratio for learning and test phases. The learning and test phases let us compare convergence rate and performance of learned policies, respectively. *Score ratio* is defined as: scores of the learning agent divided by scores of the programmed agent. The score ratio provides a good differentiation among the methods in terms of learning both attack and defense. If a method effectively learns both strategies, it would have higher score ratios indeed.

Tree Size

Fig. 5.5 shows history of the number of leaves in the abstraction trees, created during the individual and the ensemble learning experiments. Each individual splitting criterion is compared with the three ensembles that involve it as one part.

It is clear from the figures that, at the beginning, the individual methods are not fast as the ensembles in creating new leaves. This period starts from episode 5,000 and lasts until episode 50,000. Faster leaf creation could lead to faster convergence provided that the splits are efficient. However, the size of the generated trees are, finally, very close.

Action

Fig. 5.6 shows the (accumulated) average number of actions per episodes during learning experiments. At the beginning, the learning player learns how to prevent the intelligent opponent from taking control of the ball. That is why the number of actions per episodes increases for a period. This period matches the latency period in leaf creation of the individual methods in Fig. 5.5. As soon as the learner finds how to intercept and catch the ball, it would start learning the way to score it. At this time, the number of actions per episodes starts declining.

It is evident from the figures that, the heterogeneous ensembles converge faster than the homogeneous ones (except for SANDS-VAR that converges slower than SANDS-SANDS). Also both type of the ensembles converge reasonably faster than the individual methods.

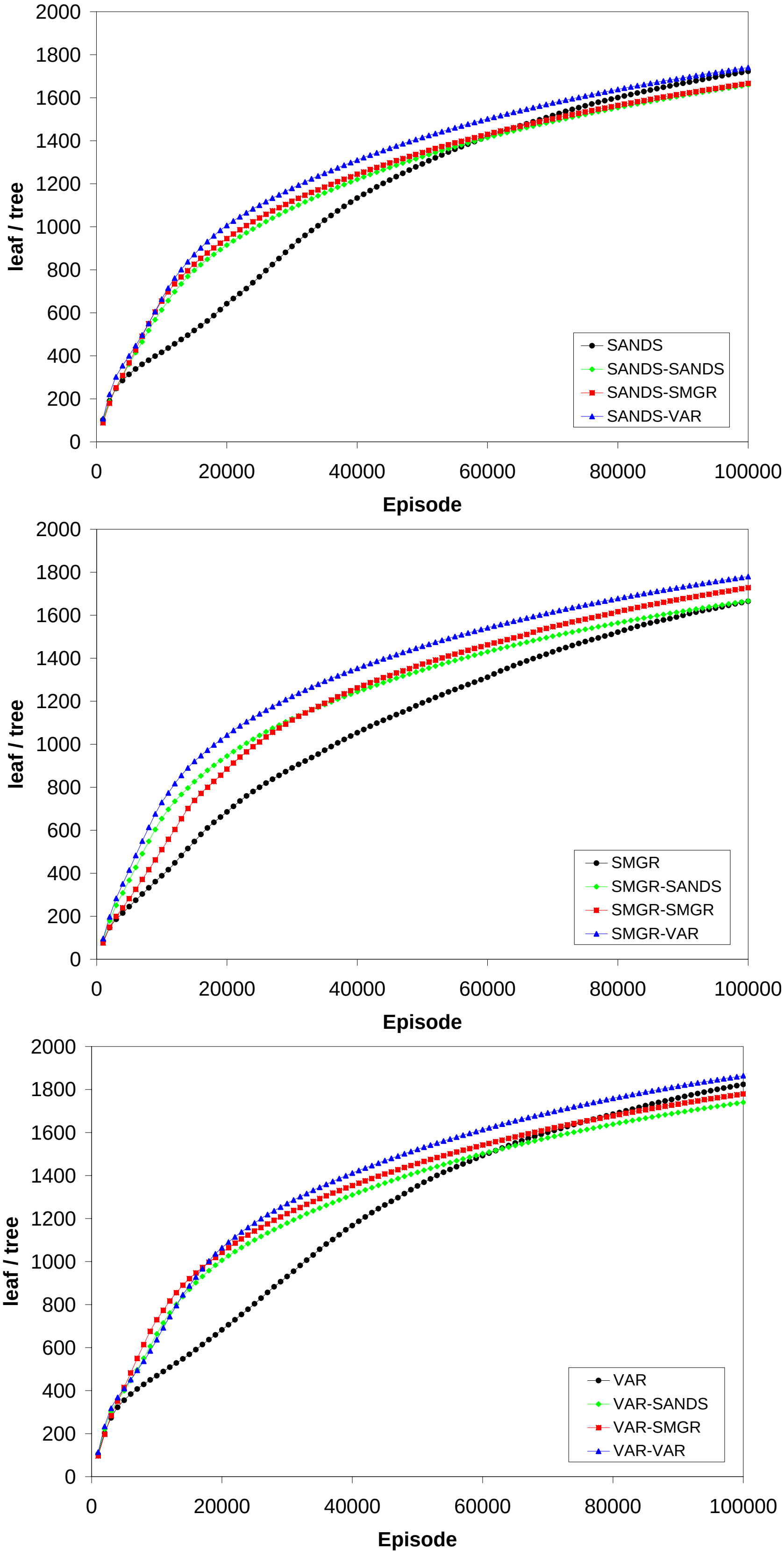


Figure 5.5: History of the number of leaves in the abstraction trees, created during the individual and the ensemble learning experiments

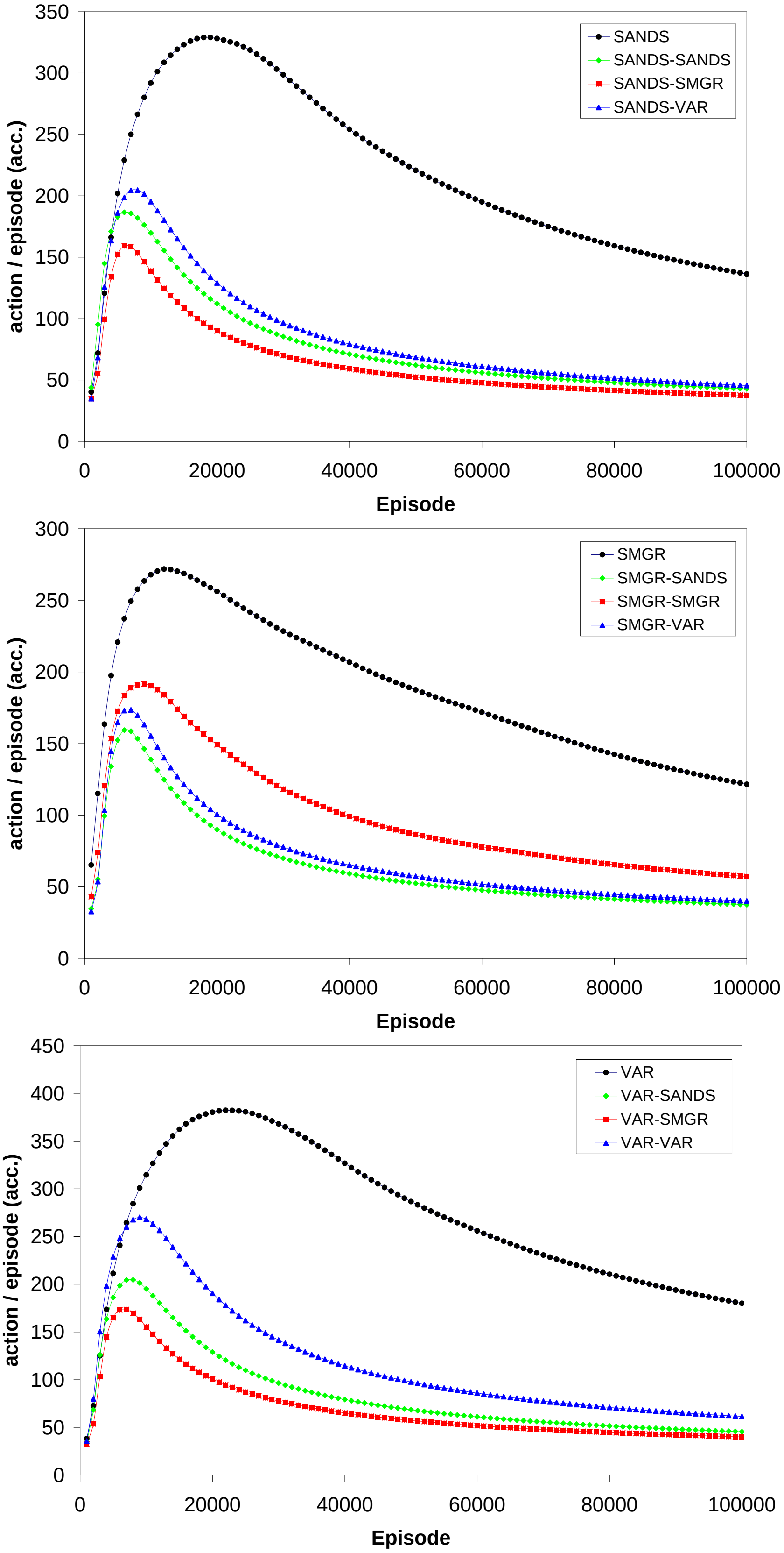


Figure 5.6: History of the (accumulated) average number of actions per episodes during the individual and the ensemble learning experiments

Reward

Fig. 5.7 shows the (accumulated) average sum of the received rewards per episodes. The heterogeneous ensembles gain more rewards than the homogeneous ones (except for SANDS-VAR), and both of them gain more rewards than the individual methods. Among the individual methods, graph of SMGR is the closest graph to the ensembles. The same thing could be perceived from the previous figures. SMGR is the fastest converging method among the individual methods.

Score Ratio

Fig. 5.8 shows history of the score ratio during the individual and the ensemble learning experiments. At the beginning, the opponent scores more than the learner. So, the ratio is less than one until episode 50,000 and 20,000 for the individual methods and ensembles, respectively. The ratio then grows with log-incremental rate.

If learning is continued the score ratio is expected to become constant at infinity. This means, even if the learning player tries its best, the programmed opponent will still have some chances to score. This small chance corresponds to the situations that, the programmed opponent is initially placed near the ball and the left goal, while the learning player in placed very far from them.

Discussion

Table 5.1 and Figs. 5.5- 5.8 show the ensemble methods, almost in all cases, have much better results than the individual methods in terms of both convergence rate and performance of the learned policy. The average number of actions for the ensembles are less than the individual cases both in learning and test phases. Also, the received rewards and the score ratios are all more than the individual cases in both phases. However, the average number of leaves per trees are close to the individual cases [5]. Though, they are slightly smaller in the heterogeneous category.

The ensembles that involve VAR criterion are exceptions. Although their convergence is faster than the individual cases, performance of their final policy degrades sometimes. In VAR-VAR and SANDS-VAR, the score ratios are smaller than their individual cases. Perhaps, in some situations multiple paths exist for catching the ball and scoring it. It is probable that one policy tries e.g. to approach the ball first in x direction and then in y, while the other one tries the opposite. Combining these conflicting policies could result in selecting an inefficient action and losing time. Meanwhile, the opponent catches the ball and scores it.

Table 5.1 and Figs. 5.5- 5.8 show in almost all cases the heterogeneous ensembles have better results than the homogeneous ones in terms of both convergence rate and policy performance. SANDS-SMGR and SMGR-VAR, which are heterogeneous ensembles, are the two top methods among all. In fact, performance of the heterogeneous combinations of the splitting criteria are better than (or very close to) the best of their homogeneous combinations. For example performance of SMGR-VAR is better than SMGR-SMGR and VAR-VAR. Also, performance of SANDS-VAR is closer to SANDS-SANDS than VAR-VAR. This can be due to variety in the split axis. If the trees in an ensemble break a subset of state space along different axis, the combined output could be like a linear multi-variate split.

---

[5] Note that the total number of leaves in the ensembles are bigger than the individual cases.

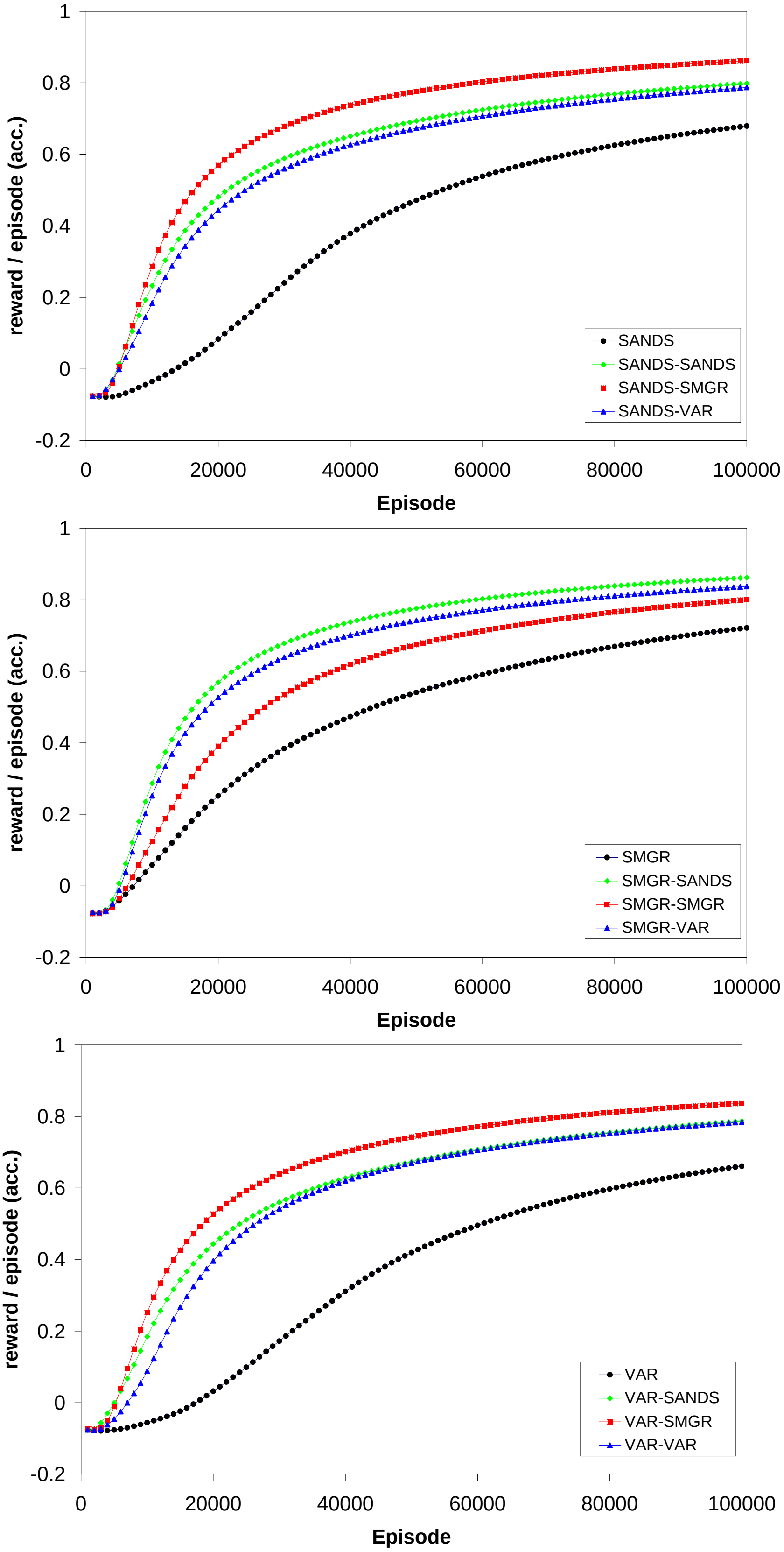


Figure 5.7: History of (accumulated) average sum of the received rewards during the individual and the ensemble learning episodes

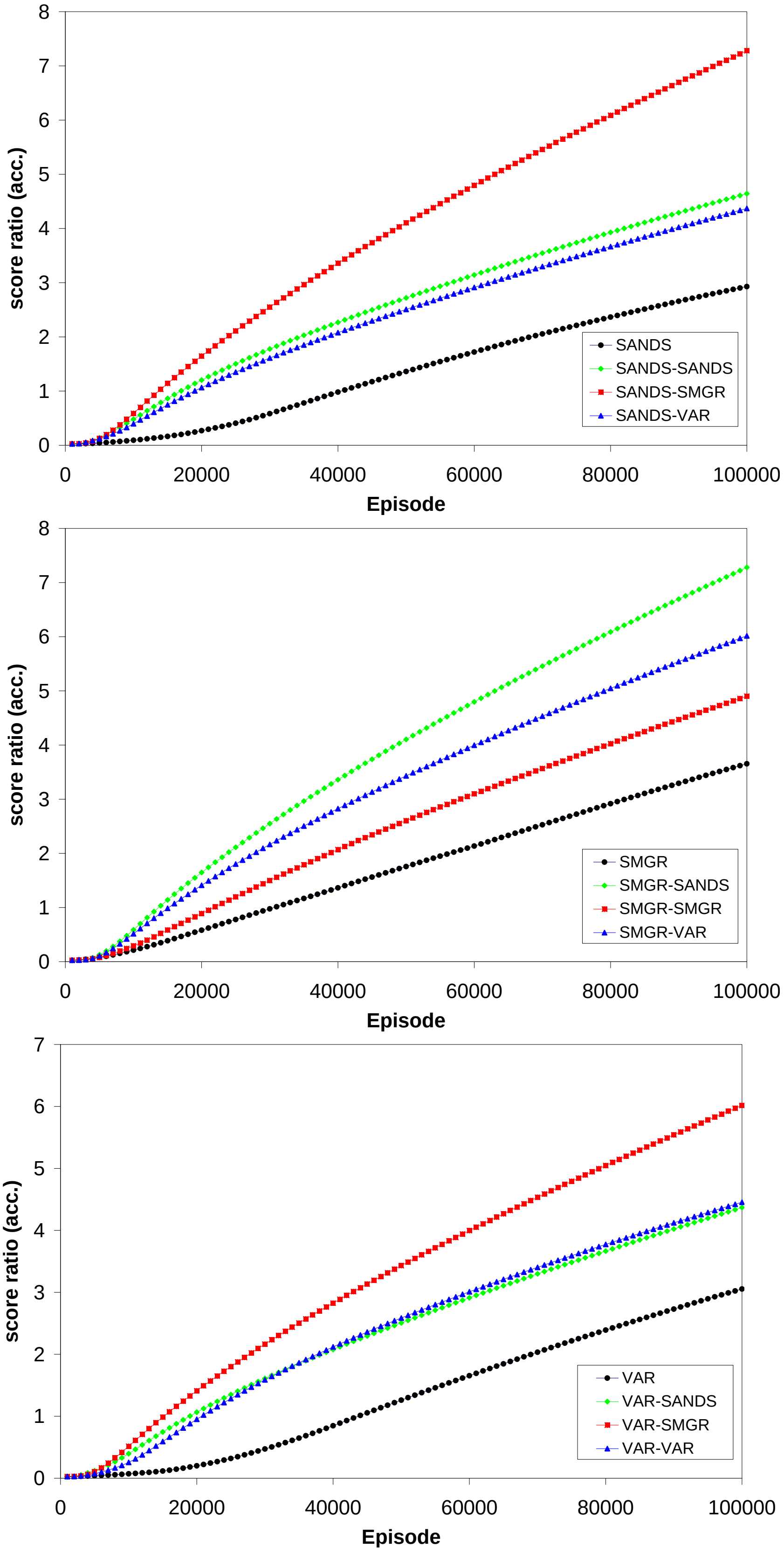


Figure 5.8: The score ratio for the individual and the ensemble learning

Table 5.2: Performance of the CLASS algorithm

	leaf			action	reward	score ratio
Method	merge	pruning	reduction %			
SANDS-SANDS	4551	2244	50.7	37.2	.904	17.6
SMGR-SMGR	4629	2254	51.3	32.9	.952	30.3
VAR-VAR	4281	2033	52.5	45.7	.929	22.8
homog. avg.	**4487**	**2177**	**51.5**	**38.6**	**.928**	**23.6**
SANDS-SMGR	6794	3175	53.3	33.6	.940	25.4
SANDS-VAR	5584	2454	56.1	39.5	.921	20.1
SMGR-VAR	7507	2964	60.5	36.9	.941	26.1
heterog. avg.	**6628**	**2864**	**56.6**	**36.7**	**.934**	**23.9**

5.4.2 CLASS Results

The second category of tests concerns the CLASS algorithm. For this purpose, the generated trees in the individual learning experiments (Table 5.1) are used. All possible two-combinations of the trees are selected and the CLASS algorithm is executed on them. Then, the resulted tree is tested in the same simulation task for 30,000 episodes. The applied CLASS algorithm, firstly, balances the input trees and prunes them in order to deal with smaller trees. Then, the input trees are merged together. The merged tree is then balanced and pruned.

Table 5.2 displays the average number of leaves in the trees after merge, followed by the average number of leaves after balance and pruning the merged trees, and percentage of reduction in the size of the trees (i.e. $100 \times \frac{initial\ size - size\ after\ pruning}{initial\ size}$). Next columns summarize the number of actions per episode, the sum of received rewards per episode, and the score ratio.

Policy Performance

Table 5.2 shows that, the combined trees by the CLASS algorithm always find faster policies that gain more scores and rewards than their individual versions (Table 5.1), e.g. the learned policies in SANDS-VAR accomplish the task in 39.5 actions that is faster than their individual versions, SANDS (43.2) and VAR (54.7). Moreover, the homogeneous combinations find policies that are, in average, 6.3 actions faster than the individual policies. the heterogeneous combinations are even better; they are 8.2 actions faster.

SMGR is the best method among the individual cases. Combining it with itself (i.e. SMGR-SMGR) results in the best method among all. Its heterogeneous combination with SANDS or VAR, also, have close performance to SMGR-SMGR. In fact, performance of SANDS-SMGR is closer to SMGR-SMGR than to SANDS-SANDS. Also performance of SMGR-VAR is closer to SMGR-SMGR than to VAR-VAR.

Merge

The heterogeneous combinations have better performance than the homogeneous cases except in the number of leaves (Table 5.2). In fact in the homogeneous cases, input trees have more similar splits. As a result, the merged trees are smaller. Fig. 5.9 emphasizes that, the number of leaves of the merged trees linearly increases with product of the number of leaves of the input trees in both homogeneous (left) and heterogeneous (right) categories. However, slope of the line in the heterogeneous merge is bigger, as was expected from Table 5.2.

Pruning

The number of leaves after pruning is around 30-40% less than the total number of leaves of the two input trees of the merge algorithm; e.g., input trees of SANDS-SANDS have totally 1723×2=3446 leaves (Table 5.1), while results of the CLASS algorithm have 2244 leaves (Table 5.2). The resulted trees are also smaller than their ensembles; e.g., results of the CLASS for SANDS-SANDS have 32% fewer leaves than SANDS-SANDS ensembles, which have 1660×2=3320 leaves (Table 5.1).

Pruning reduces the size of the input trees to less than half. Fig. 5.10 shows the number of leaves after pruning is linearly increasing with the number of leaves before pruning. However, slope of the line is smaller for the heterogeneous case. That means, percentage of reduction in the heterogeneous case is higher than the homogeneous case. We saw that, the size of the merged trees (which are inputs of the pruning procedure) are also bigger in the heterogeneous case. We might conclude that in larger scales (like 100,000 leaves), increase in the number of leaves after pruning would be log-incremental with respect to the number of leaves before pruning.

Processing Time

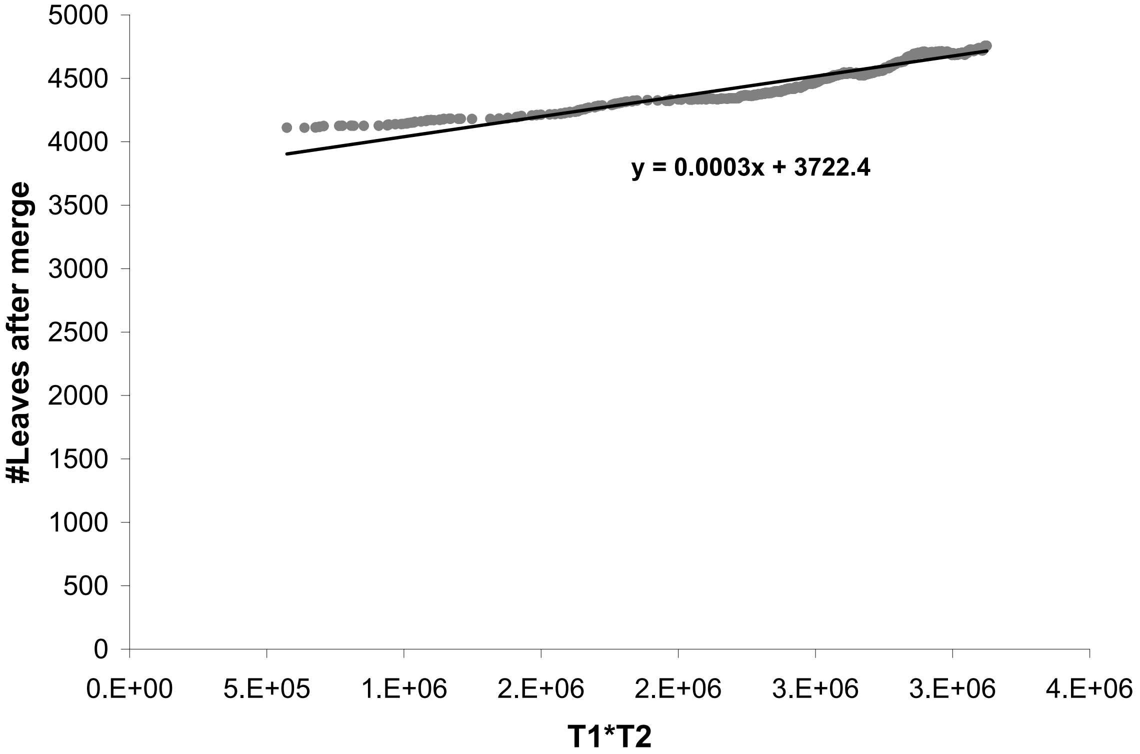
Balancing a tree with L leaves, according to Algorithm 2, needs $O(L^4)$ processing time. Pruning procedure traverses the tree once, so it is $O(L)$ [6]. Merging two trees (Algorithm 1) with L_1 and L_2 leaves needs traversing the first tree once for every nodes of the second tree, i.e. $O(L_1L_2)$.

Now assume that, the number of leaves of the input trees to the CLASS algorithm are l_1 and l_2. Also, assume after balance and pruning, their leaves are reduced to l'_1 and l'_2, respectively. Moreover, assume that the number of leaves of the merged tree is l_{12}. Balance and pruning the input trees need $O(l_1^4+l_1+l_2^4+l_2)$. Merging the pruned trees is $O(l'_1l'_2)$. Also, balance and pruning the merged tree is $O(l_{12}^4+l_{12})$. Therefore, time order of the version of the CLASS algorithm that we used for our simulations is $O(l_1^4+l_1+l_2^4+l_2+l'_1l'_2+l_{12}^4+l_{12})$.

We know from the previous sub-sections that, the size of the merged tree increases linearly with product of the sizes of the input trees, and the size of the pruned tree is linearly increasing with its size before pruning. Therefore, the time order could be rewritten as $O(l_1^4+l_1+l_2^4+l_2+l_1l_2+l_1^4l_2^4+l_1l_2)$. According to complexity theory, terms of lower degrees can be ignored. So, the order is simplified to $O(l_1^4l_2^4)$. So, time order of the CLASS algorithm is mostly affected by the balance procedure. It is specially affected by balancing the merged trees, since they have big sizes. Fig. 5.11 confirms this argument. Its shows, how processing time of the CLASS algorithm changes with respect to the number of leaves in the merged tree. It fits very well a polynomial of degree 4.

---

[6] Note that a tree with L leaves has 2L-1 nodes



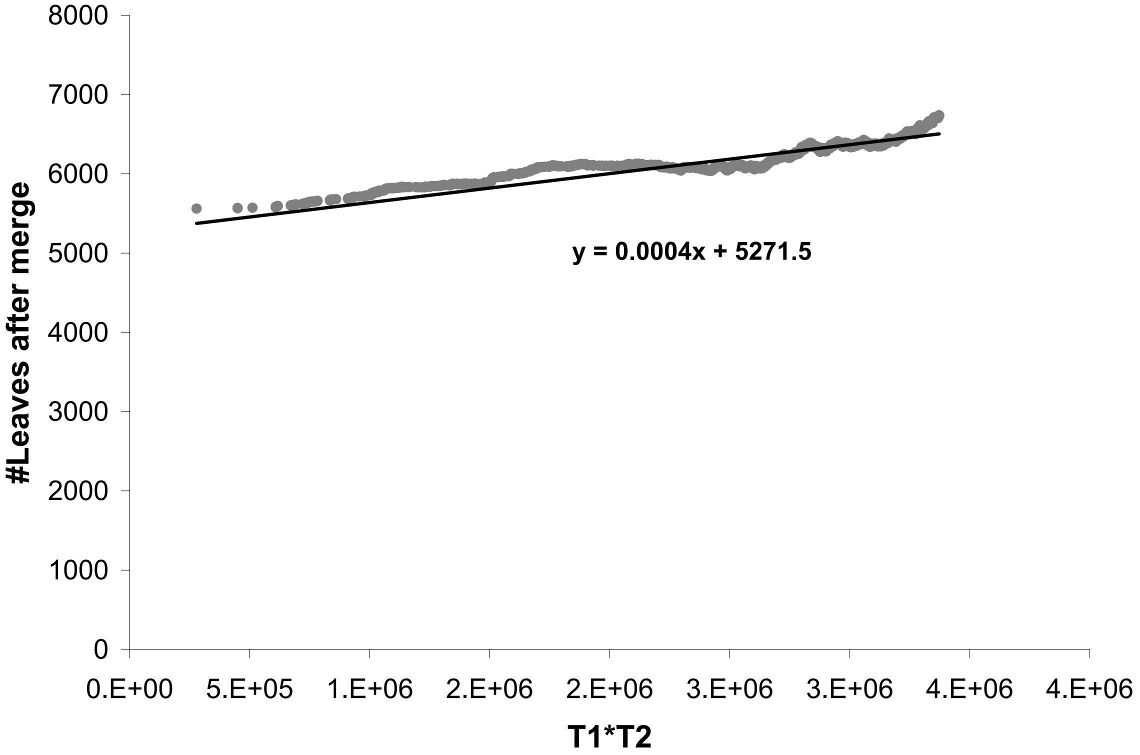
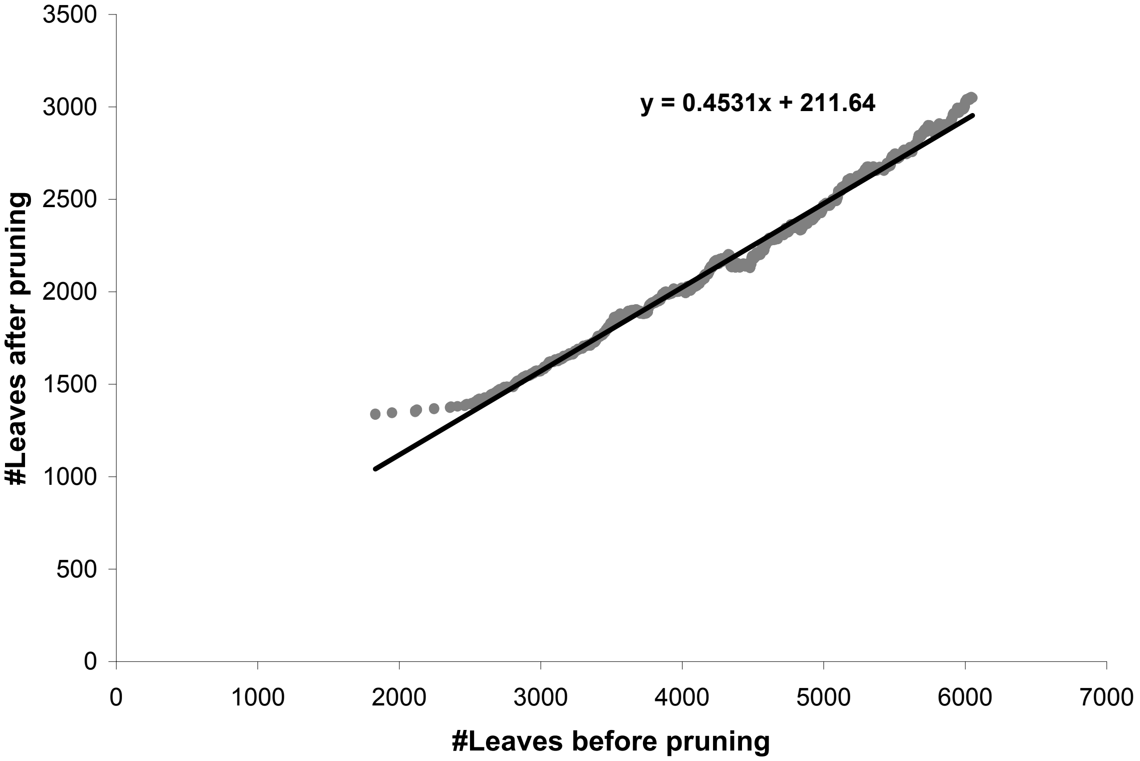


Figure 5.9: Relation between the size of the input and output trees of the merge algorithm. Top: Homogeneous, Bottom: Heterogeneous



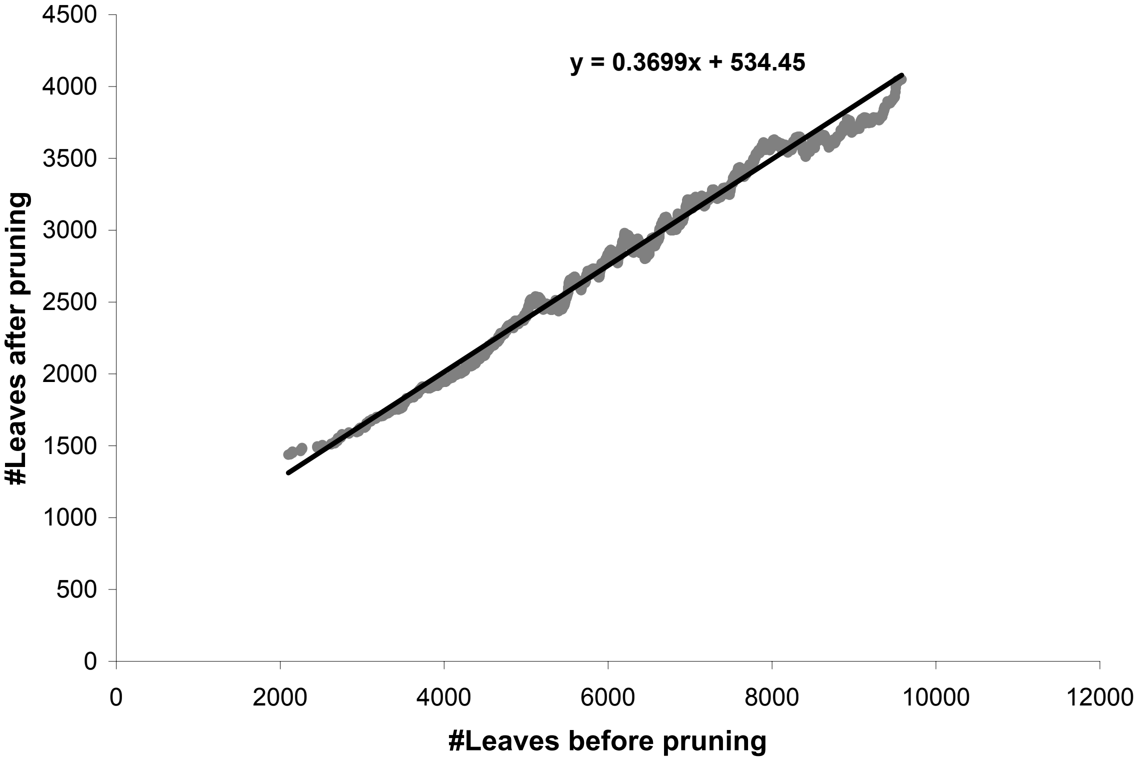


Figure 5.10: Performance of Pruning. Top: Homogeneous, Bottom: Heterogeneous

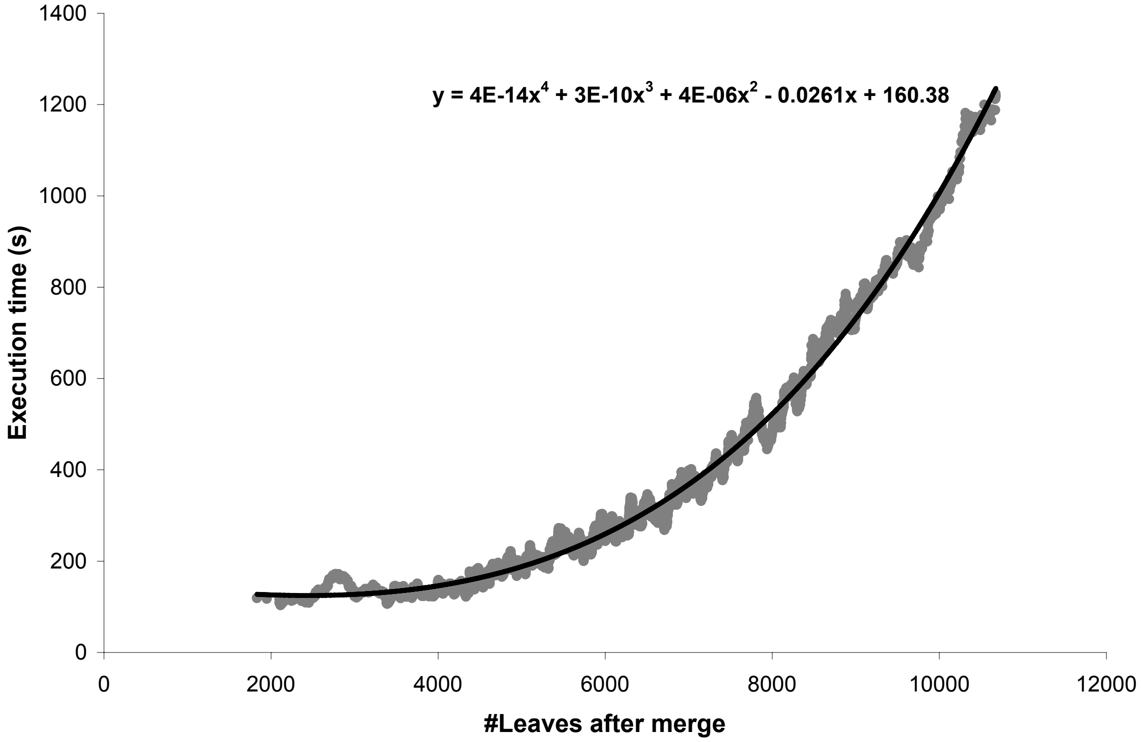


Figure 5.11: Processing time of the CLASS algorithm

Discussion

In summary, combined trees by the CLASS algorithm, always, find faster policies that gain more scores and rewards than their individual versions. Like in the ensembles, performance of the heterogeneous combinations of the splitting criteria is very close to the best of their homogeneous combinations. Therefore, in cases when, no idea about performance of different methods exists, the best selection would be to mix the methods with a cooperative method, like ensemble learning or CLASS. The mixture, then, would automatically benefit from the best of them. This outcome confirms our argument that, using heterogeneous cooperative learning could simplify the task of method selection and parameter tuning for learning algorithms.

Comparing to the ensembles, the CLASS algorithm creates smaller trees with almost better performance in terms of the received rewards and the score ratio. However, the policies found by the ensembles are faster. This means, although the developed policies by the ensembles are fast in attack, they are inefficient in defence because, they let the opponent player score more and faster. However, results of the CLASS have better defence and the episodes are, as a consequence, prolonged.

5.5 Conclusion

In this chapter two approaches to create cooperation between different sources of knowledge via ensembles of abstraction trees and via applying merge, balance, and pruning techniques (called together as the CLASS algorithm) were presented. The abstraction trees could come from different abstraction methods or different agents in a multi-agent system. Ensemble learning is suitable for single agent learning. While, the CLASS algorithm is applicable to multi-agent systems in which, knowledge sources are separated and have different learning experiences.

Simulation results in non-deterministic football learning task provided strong evidences for faster convergence rate and more efficient policies compared to individual learning. Simulation results also showed that, using heterogeneous splitting criteria results in more efficient and faster converging trees than the homogeneous ones in most cases. Using different criteria is perhaps easier when no idea about performance of different methods exists. Then, mixing them in ensembles would result in a learning system that its performance is close to the best method.

Ensemble learning and CLASS algorithm create bigger trees than individual learning. Merging the trees, that are created by the homogeneous criteria, result in smaller trees than the heterogeneous ones. Because, the homogeneous criteria generate similar splits which can be factored out. Generating bigger trees could be fine as far as memory consumption does not matter. For example, in real robot-learning tasks where faster convergence rate is more important than memory consumption, this kind of methods could induce a great reduction in the required learning time.

Cooperative learning methods have no added-value in cases where, individual learning can itself derive abstraction trees with reasonable performance in reasonable time. A similar argument has already been emphasized in [119].

For future, we would like to apply these algorithms to a realistic soccer task with multiple players in each team. Also we look for better abstraction methods to create smaller abstraction trees. We think, this is a big drawback of the current abstraction methods when they are applied to a complex task like soccer learning.

Chapter 6

Conclusion and Future Works

This chapter gives a summary of the results acquired in this thesis, accompanied by some suggestions for future researches. The goal of the thesis was to investigate methods of behavior *composition* and *coordination*; how to combine simpler behaviors and make a more complex behavior out of them, and how to coordinate their execution time.

6.1 Manual Hierarchy Design

We started by manually designing the composition hierarchy and the coordination mechanism. We applied it to the LEURRE, European project, which aimed at studying mixed-societies of robots and animals.

The biological experiments was aggregation of cockroaches under shelters. Biological studies showed that cockroaches are able to collectively choose an aggregation site, even if the sites are identical. However, the darker the site is, the more likely the cockroaches choose that site collectively.

The idea of the project was to inject some informed agents, here InsBot micro-robots, to the society of cockroaches. The robots had to integrate themselves to the society, interact with individuals (animals or robots) in the society, and participate in the collective decision makings. In the next steps, the robots in coordination with each other should be able change the collective behavior of the society through direct interactions. The robots could affect the cockroaches via chemical communication; they were covered with a paper that was impregnated with pheromone of cockroaches.

The robots were controlled via a hierarchical behavior-based controller, inspired from *schema theory*[2]. The *perceptual schema* was responsible for sensor fusion and object detection. The *motor schema* was responsible for selection of the proper behavior for the current situation.

In the highest layer of the motor schema, an arbiter stochastically selected the next behavior according to a probability table. The table was a probabilistic model of the cockroach's aggregation, acquired from extensive analysis of the experiments with real cockroaches. The selected behavior by the arbiter was then decomposed to simpler behaviors. Each behavior generated a potential field. The final outcome was the resultant vector of the forces generated by the constituting behaviors. The resultant force was mapped to an action for execution, afterwards.

Biological experiments showed that the mixed-societies of robots and cockroaches generate similar aggregation patterns to pure-cockroach groups. Biological experiments, also, showed that the aggregation site is selected collectively by both robots and cockroaches. In fact both of them modulate each other's behavior. The most important results proved that, the robots are able to induce a new collective pattern and modify the collective behavior of cockroaches. Provided that the number of robots is bigger than a threshold, they are able to bring the aggregation site from the dark shelter to the bright one.

The initial tests were done with cockroaches however, more experiments should be carried out with other types of animal, like sheep and chicken, in order to develop and verify a methodology for mixed-society control. The theory is applicable to other aspects of human life, anywhere that some informed agents in coordination with each other might be able to control the collective behavior of a large group. It can include (but not limited to) many problems in the domain of economy, education, culture and agriculture.

6.2 Compact Q-Learning

Although extensible and reusable, the manually derived behavioral hierarchy needed a high degree of hand tuning. In the next step, we benefited from machine learning techniques to automate the behavior coordination part, while still deriving the hierarchy for behavior composition manually.

We derived a Compact Q-Learning method for micro-robots with processing and memory constrains, and tried to learn behavior coordination through it. It was implemented and verified on light-attraction and safe-wandering task. Combination of multiple behaviors was implemented by combining the state vectors of the behaviors with Cartesian product, and combining their reward functions with weighted sum.

However, the problem of the curse of dimensionality made incorporation of this kind of flat-learning techniques unsuitable. Even though optimizing them could temporarily speed up the learning process and widen their range of applications, their scalability to real world applications remained under question. Size of the combined space grew very fast and learning time increased exponentially.

6.3 Hierarchical Reinforcement Learning

In the next step we automated both the behavior composition and coordination mechanisms. We know that some features of the state space might be irrelevant to what the robot is currently learning. Abstracting these features is the goal of Hierarchical Reinforcement Learning. We used the U-tree [31] algorithm for State Abstraction. It starts with assuming the whole state-space as one big abstract state. Then, it breaks the abstract state, if any violation of Markov property is detected.

6.3.1 Splitting Criteria

Violation of Markov property in the U-Tree algorithm is detected using a *splitting criteria*. We could derive three new splitting criteria by formalizing the state abstraction problem with different heuristics:

- **SANDS** (State Abstraction for Non-Deterministic Systems) maximizes the *expected reward return* by finding a splitting point that well differentiates the selection probability of actions, before and after introducing the split, considering also the Q-values.
- **SMGR** (Softmax Gain Ration) maximizes the *certainty* on the selection probability of actions. We proved that SMGR scales SANDS and combines it with a "penalty term".
- **VAR** (Variance Reduction) minimizes the *Mean Squared Error* on the action-values.

We showed that the new criteria adapt decision tree learning techniques to state abstraction, better than the generally used criteria like Kolmogorov-Smirnov or Information gain Ratio tests. Proof of performance was supported by strong evidences from simulation results in deterministic and non-deterministic football learning in grid-world. We showed that, the new criteria outperform the existing ones, not only in efficiency of the policy and size of the learned trees, but also in computation time. Among the proposed criteria, SANDS generated the smallest trees, SMGR generated the most efficient policies, and VAR executed faster than the others.

Although the proposed criteria generated smaller trees than the existing criteria, we think their sizes are still big and inefficient to be applied to complex tasks like realistic soccer learning. Inability to revise the tree in non-stationary environments is another drawback of our methods. Using Multivariate splits [108; 109; 110] might be a suitable solution.

6.3.2 Cooperative Learning

The existing knowledge in the agents of a multi-agent system, is a free source of knowledge that, if transferred, can broaden the scales of learning, both temporally and spatially. We presented two Cooperative Learning approaches for combining output or structure of abstraction trees: *Ensemble Learning* and *CLASS*(Combining Layered Abstraction of State Space) algorithm. Ensemble learning is suitable for single agent learning. While, the CLASS algorithm is applicable to multi-agent systems in which, knowledge sources are separated and have different learning experiences.

In Ensemble Learning, the learning agent constructs multiple abstraction trees in parallel from the same state-space. In action selection phase however, the agent combines the probability distribution of actions proposed by each tree via simple averaging. The next action then is selected according to this combined distribution. Finally the reinforcement signal is feeded back to all trees.

In CLASS, the structure of abstraction trees are combined with each other. The combination process is done by applying a mixture of *merge*, *balance* and *pruning* techniques. The merge algorithm creates a pairwise intersection of the leaves of the input trees. The balance and pruning techniques eliminate some inefficient splits and make the abstraction trees smaller. The balance algorithm restructures the tree so that leaves with similar Q-values are placed nearby. These similar leaves are removed then by the pruning procedure.

Ensembles enable us to have heterogeneous cooperation between different abstraction algorithms, as well as homogeneous cooperation between same abstraction methods but with different parameters. They can simplify the tough task of parameter adjustment for abstraction methods. They can guide the abstraction trees to get specialized in different areas of state-space by introducing the splits along different axis.

Simulation results in non-deterministic football learning task provided strong evidences for faster convergence rate and more efficient policies compared to individual learning. Simulation results also showed that, in most cases using heterogeneous splitting criteria results in more efficient and faster converging trees than the homogeneous ones. Using heterogeneous criteria is perhaps easier when no idea about performance of different methods exists. Then, mixing them in ensembles would result in a learning system that its performance is close to the best method.

Ensemble learning and CLASS algorithm created bigger trees than individual learning. Merging the trees, that were created by the homogeneous criteria, resulted in smaller trees than the heterogeneous ones because, the homogeneous criteria generated similar splits which could be factored out. Generating bigger trees could be fine as far as memory consumption does not matter. For example, in real robot-learning tasks where faster convergence rate is more important than memory consumption, this kind of methods could induce a great reduction in the required learning time.

We look for better abstraction methods to create smaller abstraction trees. We think, this is a big drawback of the current abstraction methods when they are applied to a complex task like soccer learning.

Appendix A

Splitting Criteria Results

In this appendix, we compare the best simulation results of our proposed criteria with the best results of the existing ones.

A.1 Unbiased Sampling

Table A.1 compares the best results acquired by the biased versions of the U-Tree construction methods with their unbiased versions in terms of the number of actions per episodes during the test phase. UKS and UIGR are new versions of KS and IGR with unbiased sampling, respectively. Table A.2 compares the number of leaves in the learned trees for the same simulation experiments.

A.2 Policy Performance and Tree Size

Table A.3 shows the average number of actions per episodes during the test phase for the flat SARSA and the best confidence thresholds of different Hierarchical RL methods. Table A.4 shows the average number of leaves in the learned trees. The results for KS and IGR are duplicates of the previous section to simplify the comparison.

Figure A.1 shows the number of actions per episodes recorded during the learning episodes against different types of players. It shows that SANDS, SMGR, and VAR converge slower than the others at the beginning. This can be problematic since performance

Table A.1: Comparing the biased/unbiased U-Tree construction methods in terms of the number of actions per episodes during the test phase (best confidence threshold)

Player	KS	IGR	UKS	UIGR
Static	11.23±.2	11.21±.05	11.35±.22	11.23±.04
Offender	14.97±.57	14.55±.15	14.91±.14	14.58±.5
Midfielder	18.41±1.2	18.65±1.73	17.39±.21	17.39±.24

Table A.2: Comparing the biased/unbiased U-Tree construction methods in terms of the number of leaves in the learned tree (best confidence threshold)

Player	KS	IGR	UKS	UIGR
Static	692±118.9	703±34.6	596±91.8	666±27.6
Offender	1476±403.8	1536±301.5	1331±78.5	1242±23.2
Midfielder	1857±253.3	2263±503.6	1651±140.6	1533±47.7

Table A.3: Comparing the learning methods in terms of the average number of actions per episode during the test phase (best confidence threshold)

Player	SARSA	KS	IGR	SANDS	SMGR	VAR
Static	10.85±.01	11.23±.2	11.21±.05	11.1±.04	11.12±.03	11.1±.04
Offender	19.63±.06*	14.97±.57	14.55±.15	14.39±.09	14.28±.07	14.4±.08
Midfielder	668±.1**	18.41±1.2	18.65±1.73	16.95±.2	16.84±.06	17.04±.16

* 14.72 after 10^6 episodes ** 18.92 after 15×10^6 episodes

Table A.4: Comparing the learning methods in terms of the average number of leaves in the learned trees. For flat Q-learning with SARSA, the number of states is mentioned (best confidence threshold)

Player	SARSA	KS	IGR	SANDS	SMGR	VAR
Static	2452	692±118.9	703±34.6	491±7.4	516±9.6	481±7
Offender	120052	1476±403.8	1536±301.5	1282±22.2	1259±26.3	1330±15.9
Midfielder	3001252	1857±253.3	2263±503.6	1313±77.5	1436±78.3	1310±98.5

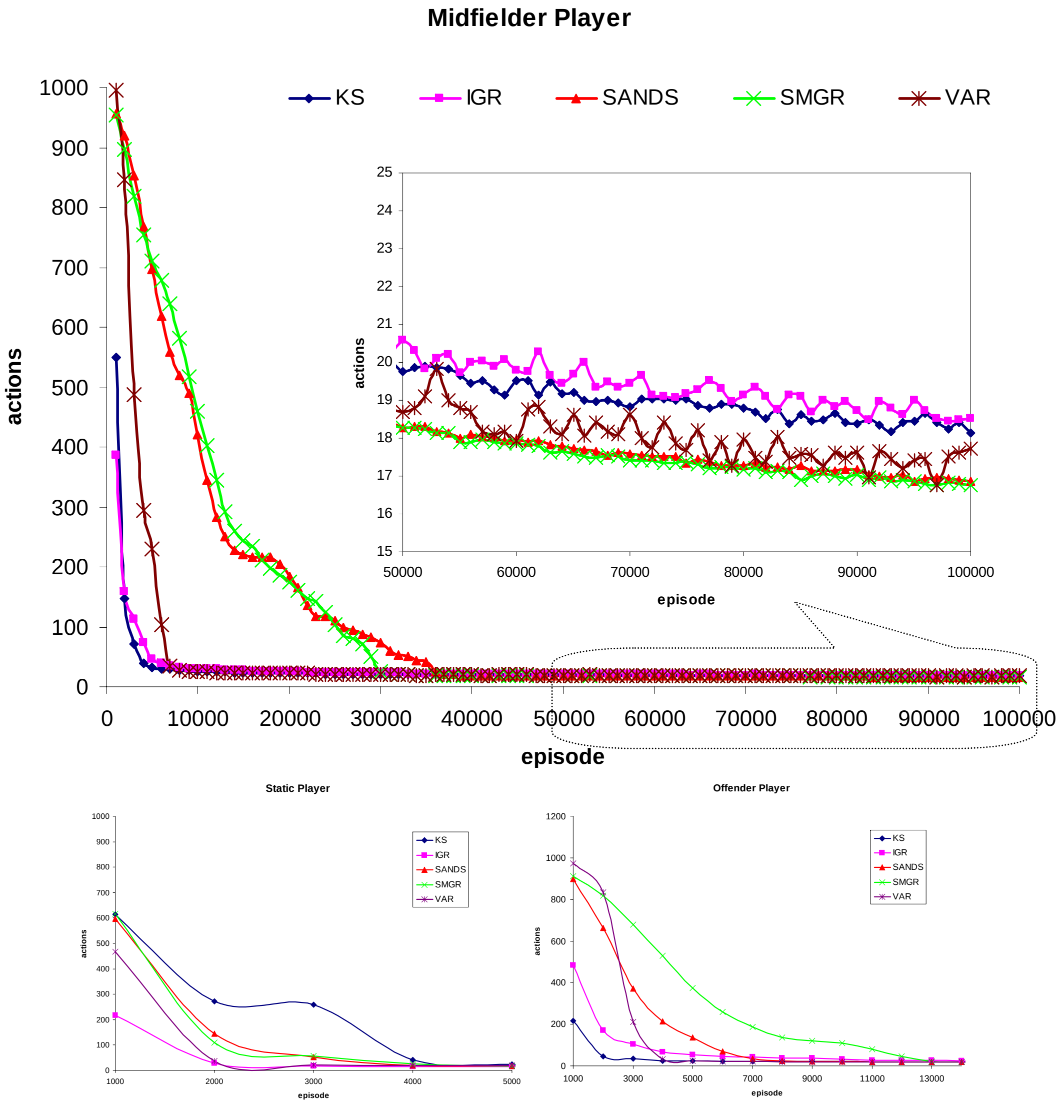


Figure A.1: History of the number of actions per episodes during the learning phase against different player types. Note that for clarity reason, *x* axis are scaled differently. (best confidence threshold)

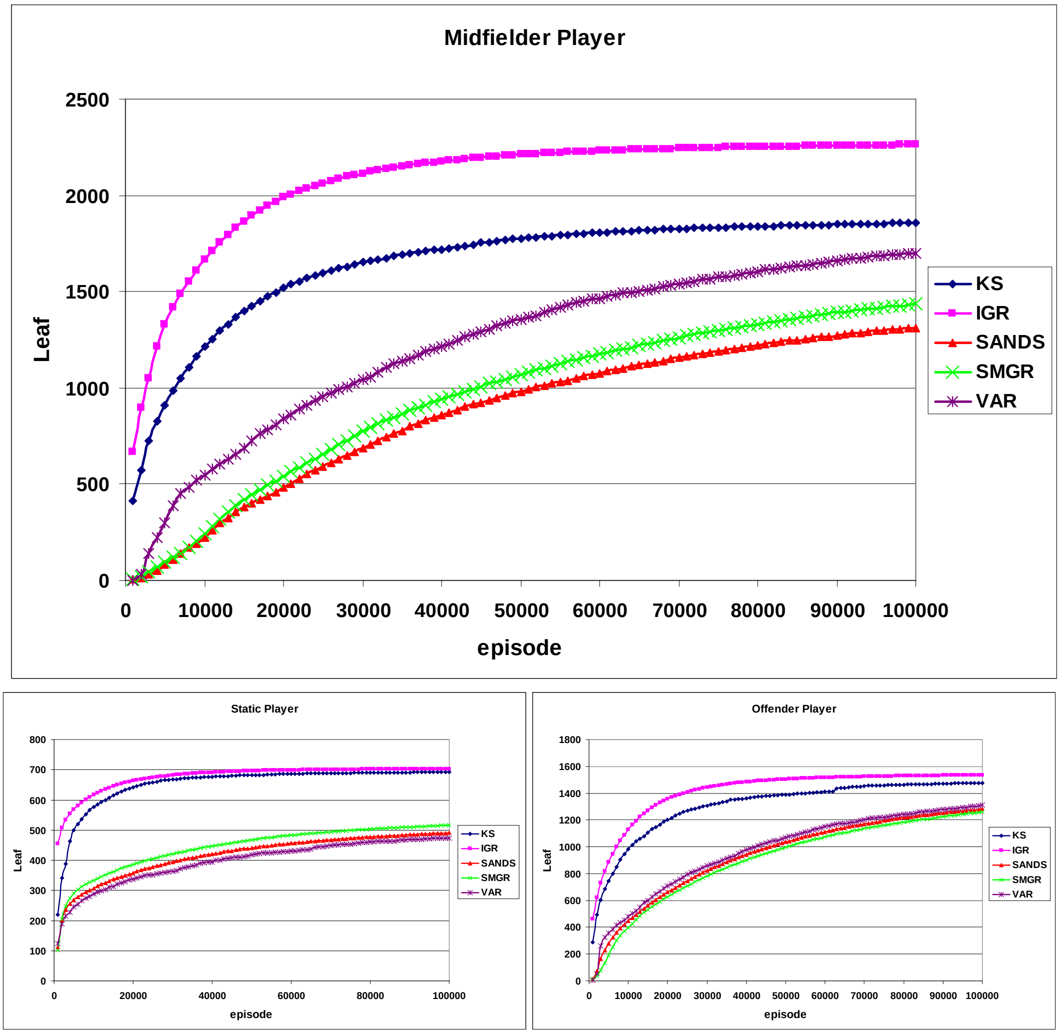


Figure A.2: History of the number of leaves in the trees during the learning phase against different player types (best confidence threshold)

of our proposed criteria does not increase that much initially. But, it is expected that after a duration, their performance anticipate the other criteria. This argument is supported by the magnified part of the top chart of Fig. A.1.

Figure A.2 shows the number of leaves recorded during the learning episodes against different types of players. The generated trees by our proposed criteria are visibly smaller.

Bibliography

[1] LEURRE Project official website, *url:* http://leurre.ulb.ac.be.

[2] M. Arbib, ed., *The Handbook of Brain Theory and Neural Networks*, ch. Schema Theory, pp. 830–834. Cambridge,MA: MIT Press, 1995.

[3] R. Arkin, *Behavior-based Robotics.* Cambridge,MA: MIT Press, 1998.

[4] H. Aghazarian, P. Pirjanian, P. Schenker, and T. Huntsberger, "An architecture for controlling multiple robots," Tech. Rep. NPO-30345, NASA's Jet Propulsion Laboratory, Pasadena, CA, Oct. 2004.

[5] A. Saffiotti, "The uses of fuzzy logic in autonomous robot navigation: a catalogue raisonné," Tech. Rep. 2.1, IRIDIA, Université Libre de Bruxelles, Nov. 1997.

[6] J. Hoff and G. Bekey, "An architecture for behavior coordination learning," in *IEEE Int. Conf. on Neural Networks*, vol. 5, (Perth, Australia), pp. 2375–2380, Nov. 1995.

[7] J. Riekki and J. Röning, "Reactive task execution by combining action maps," in *IEEE Int. Conf. on Intelligent Robots and Systems [IROS]*, (Grenoble, France), pp. 224–230, 1997.

[8] J. K. Rosenblatt, "Damn: A distributed architecture for mobile navigation," in *AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, (Stanford, CA), AAAI Press, Mar. 1995.

[9] J. Yen and N. Pfluger, "A fuzzy logic based extension to payton and rosenblatt's command fusion method for mobile robot navigation," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 25, pp. 971–978, Jun. 1995.

[10] V. Chankong and Y. Y. Haimes, *Multiobjective Decision Making - Theory and Methodology*, vol. 8. North-Holland, 1983.

[11] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The Int. Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986.

[12] R. C. Arkin, "Motor schema based navigation for a mobile robot an approach to programming by behavior," in *IEEE Int. Conf. on Robotics and Automation*, pp. 264–271, 1987.

[13] R. Brooks, “A robust layered control system for a mobile robot,” *IEEE Journal of Robotics and Automation*, vol. 2, pp. 14–23, Mar. 1986.

[14] R. Brooks, “Achieving artificial intelligence through building robots,” Tech. Rep. A.I. Memo 899, AI Lab., MIT, Cambridge, MA, 1986.

[15] R. Brooks, “The behavior language; user’s guide,” Tech. Rep. A.I. Memo 1227, AI Lab., MIT, Cambridge, MA, Apr. 1990.

[16] R. C. Arkin and D. MacKenzie, “Temporal coordination of perceptual algorithms for mobile robot navigation,” *IEEE Trans. on Robotics and Automation*, vol. 10, pp. 276–286, Jun. 1994.

[17] J. Koŝsecká and R. Bajcsy, “Discrete event systems for autonomous mobile agents,” in *Proc. of Intelligent Robotic Systems*, (Zakopane), pp. 21–31, Jul. 1993.

[18] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press, 1996.

[19] A. Barto and S. Mahadevan, “Recent advances in hierarchical reinforcement learning,” *Discrete event systems*, vol. 13, no. 4, pp. 41–77, 2003.

[20] M. Grounds, *Scaling-Up Reinforcement Learning.* PhD thesis, Department of Computer Science, University of York, 2004.

[21] R. Sutton, D. Precup, and S. Singh, “Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning,” *Artifcial Intelligence*, vol. 112, pp. 181–211, 1999.

[22] S. Thrun and A. Schwartz, “Finding structure in reinforcement learning,” in *Advances in Neural Information Processing Systems [NIPS]* (G. Tesauro, D. Touretzky, and T. Leen, eds.), (Cambridge, MA), pp. 385–392, MIT Press, 1995.

[23] R. Parr and S. Russell, “Reinforcement learning with hierarchies of machines,” in *Advances in Neural Information Processing Systems* (M. I. Jordan, M. J. Kearns, and S. A. Solla, eds.), vol. 10, The MIT Press, 1998.

[24] T. G. Dietterich, “Hierarchical reinforcement learning with the MAXQ value function decomposition,” *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 2000.

[25] A. McGovern, *Autonomous Discovery of Temporal Abstractions from Interaction with An Environment.* PhD thesis, University of Massachusetts, 2002.

[26] B. Hengst, “Discovering hierarchy in reinforcement learning with hexq,” in *ICML*, pp. 243–250, 2002.

[27] B. L. Digney, “Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments,” in *From Animals to Animats: Simulation of Adaptive Behavior [SAB96]*, 1996.

[28] S. Singh, T. Jaakola, and M. Jordan, “Reinforcement learning with soft state aggregation,” in *Advances in Neural Information Processing Systems [NIPS] 7* (G. Tesauro, D. Touretzky, and T. Leen, eds.), (Cambridge, MA), MIT Press, 1995.

[29] P. Dayan and G. E. Hinton, “Feudal reinforcement learning,” in *Advances in Neural Information Processing Systems 5, [NIPS]*, (San Francisco, CA, USA), pp. 271–278, Morgan Kaufmann Publishers Inc., 1993.

[30] A. W. Moore, “The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces,” in *Advances in Neural Information Processing Systems* (J. D. Cowan, G. Tesauro, and J. Alspector, eds.), vol. 6, pp. 711–718, Morgan Kaufmann Publishers, Inc., 1994.

[31] A. McCallum, *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, Computer Science Dept., University of Rochester, 1995.

[32] W. T. Uther and M. M. Veloso, “Tree based discretization for continuous state space reinforcement learning,” in *AAAI/IAAI*, pp. 769–774, 1998.

[33] M. Asadpour, F. Tâche, G. Caprari, W. Karlen, and R. Siegwart, “Robot-animal interaction: Perception and behavior of insbot,” in *Conceptual Modeling and Simulation Conf. [CMS]*, (Marseille,France), 2005.

[34] M. Asadpour, F. Tâche, G. Caprari, W. Karlen, and R. Siegwart, “Robot-animal interaction: Perception and behavior of insbot,” *Int. Journal of Advanced Robotics Systems*, vol. 3, pp. 93–98, 2006.

[35] I. Couzin, J. Krause, N. Franks, and S. Levin, “Effective leadership and decision-making in animal groups on the move,” *Nature*, vol. 433, pp. 513–516, 2005.

[36] S. Reebs, “Can a minority of informed leaders determine the foraging movements of a fish shoal?,” *Animal Behavior*, vol. 59, pp. 403–409, 2000.

[37] M. Lindauer, “Communication in swarm-bees searching for a new home,” *Nature*, vol. 179, pp. 63–66, 1957.

[38] R. Vaughan, N. Sumpter, J. Henderson, A. Frost, and S. Cameron, “Robot control of animal flocks,” in *Proc. of the IEEE Int. Symposium on Intelligent Controll [ISIC]*, (Gaithersburg, MD), 1998.

[39] H. Ishii, M. Nakasuji, M. Ogura, H. Miwa, and A. Takanishi, “Accelerating rat’s learning speed using a robot - the robot autonomously shows rats its functions,” in *Proc. of the Int. Workshop on Robot and Human Interactive Communicaiton*, (Kurashiki, Okayama Japan), 2004.

[40] Z. Butler, P. Corke, R. Peterson, and D. Rus, “From animals to robots: virtual fences for controlling cattle,” in *Proc. of the Int. Symposium on Experimental Robotics*, (Singapore), 2004.

[41] M. Boehlen, “A robot in a cage,” in *Int. Syposium on Computational Intelligence in Robotics and Automation [CIRA]*, IEEE, 1999.

[42] S. Camazine, J. Deneubourg, N. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau, *Self-organization in biological systems.* Princeton: Princeton University Press, 2001.

[43] J. Gautrais, C. Jost, R. Jeanson, and G. Theraulaz, “How individual interactions control aggregation patterns in gregarious arthropods,” *Interaction Studies*, vol. 5, no. 2, pp. 245–269, 2004.

[44] W. Allee, *Animal aggregations: a study in a general sociology.* Chicago: University of Chicago Press, 1931.

[45] W. Bell and K. Adiyodi, *The American Cockroach.* London: Chapman and Hall Ltd, 1982.

[46] D. Kingsley, R. Quinn, and R. Ritzmann, “A cockroach inspired robot with artificial muscles,” in *Proc. of the 2nd Int. Symposium on Adaptive Motion of Animals and Machines*, (Kyoto, Japan), 2003.

[47] U. Saranli, M. Buehler, and D. Koditscheck, “Rhex - a simple and highly mobile hexapod robot,” *Int. Journal of Robotics Research*, vol. 20, no. 7, pp. 616–631, 2001.

[48] F. Delcomyn and M. Nelson, “Architectures for a biomimetic hexapod robot,” *Robotics and Autonomous Systems*, vol. 30, pp. 5–15, 2000.

[49] N. Kagawa and H. Kazerooni, “Biomimetic small walking machine,” in *Int. Conf. on Advanced Intelligent Mechatronics*, (Como, Italy), 2001.

[50] Y. Guozheng and D. Yi, “A novel biomimetic hexapod micro-robot,” in *Int. Symposium on Micromechatronics and Human Science*, 2002.

[51] N. Cowan, J. Lee, and R. Full, “Task-level control of wall following in the american cockroach,” *Journal of Experimental Biology*, vol. 209, no. 9, pp. 1617–1629, 2006.

[52] S. Nagasawa, R. Kanzaki, and I. Shimoyama, “Study of a small mobile robot that uses living insect antennae as pheromone sensors,” in *Proc. of the 1999 Int. Conf. on Intelligent Robots and Systems*, 1999.

[53] R. Holzer and I. Shimoyama, “Locomotion control of a bio-robotic system via electric stimulation,” in *Proc. of the Int. Conf. on Intelligent Robots and Systems*, 1997.

[54] G. Hertz, “The animal-machine: Biorobotics, war and animalized technologies,” in *Proc. of the 2005 Conf. on Defense: Models, Strategies, Media*, (Irvince,CA), 2005.

[55] M. K. Rust, J. M. Owens, and D. A. Reierson, *Understanding and controlling the german cockroach.* Oxford: Oxford University Press, 1995.

[56] A. Ledoux, “Étude experimentale du grégarisme et de l’interattraction sociale chez les blattidés,” *Annales des Sciences Naturelles Zoologie et Biologie Animale*, vol. 7, pp. 76–103, 1945.

[57] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems.* Oxford: Oxford University Press, 1999.

[58] R. Jeanson, S. Blanco, R. Fournier, J.-L. Deneubourg, V. Fourcassié, and G. Theraulaz, “A model of animal movements in a bounded space,” *Journal of Theoretical Biology*, vol. 225, pp. 443–451, 2003.

[59] R. Jeanson, C. Rivault, J.-L. Deneubourg, S. Blanco, R. Fournier, C. Jost, and G. Theraulaz, “Self-organised aggregation in cockroaches,” *Animal Behaviour*, vol. 69, pp. 169–180, 2005.

[60] J.-M. Amé, J. Halloy, C. Rivault, C. Detrain, and J.-L. Denebourg, “Collegial decision making based on social amplification leads to optimal group formation,” *Proc. of the National Academy of Sciences [PNAS] of the USA*, vol. 103, pp. 5835–5840, April 2006.

[61] I. Said, G. Costagliola, I. Leoncini, and C. Rivault, “Cuticular hydrocarbone profiles and aggregation in four priplaneta species (insecta: Dictyopetra),” *Insect Physiology*, vol. 51, pp. 995–1003, 2005.

[62] A. Collot, G. Caprari, and R. Siegwart, “Insbot: Design of an autonomous mini mobile robot able to interact with cockroaches,” in *Proc. of the Int. Conf. on Robotics and Automation*, (New Orleans), pp. 2418–2423, 2004.

[63] C. Reynolds, “Steering behaviors for autonomous characters,” in *Proc. of Game Developers Conf.*, (San Jose, CA), pp. 763–782, 1999.

[64] G. Caprari, P.Balmer, R. Piguet, and R. Siegwart, “The autonomous microrobot "alice": a platform for scientific and commercial applications,” in *Proc. of the 9th Int. Symposium on Micromechatronics and Human Science*, (Nagoya Japan), 1998.

[65] N. Correll and A. Martinoli, “Collective inspection of regular structures using a swarm of miniature robots,” in *Proc. of the 9th Int. Symposium on Experimental Robotics [ISER]* (M. Ang and O. Khatib, eds.), vol. 3630 of *Springer Tracts in Advanced Robotics*, pp. 375–385, Springer-Verlag, 2006.

[66] A.-E. T. Qui, N. Correll, and A. Martinoli, “Modeling of mixed animal-robot societies,” Tech. Rep. SWIS-SP8, SWIS Lab., EPFL, 2005.

[67] C. Jost, S. Garnier, R. Jeanson, M. Asadpour, J. Gautrais, and G. Theraulaz, "The embodiment of cockroach behaviour in a micro-robot," in *Proc. of the 35th Int. Symposium on Robotics*, (Paris, France), Mar. 2004.

[68] S. Garnier, C. Jost, R. Jeanson, J. Gautrais, M. Asadpour, G. Caprari, and G. Theraulaz, "Collective decision-making by a group of cockroach-like robots," in *Proc. of the IEEE Swarm Intelligence Symposium*, (Pasadena, CA), June 2005.

[69] S. Garnier, C. Jost, R. Jeanson, J. Gautrais, M. Asadpour, G. Caprari, and G. Theraulaz, "Aggregation behaviour as a source of collective decision in a group of cockroach-like robots," in *Proc. of the 8th European Conf. on Artificial Life [ECAL]* (M. Capcarrere, ed.), vol. 3630 of *Lecture Notes in Artificial Intelligence*, (Berlin Heidelberg), pp. 169–178, Springer-Verlag, September 2005.

[70] J.-M. Amé, C. Rivault, and J.-L. Deneubourg, "Cockroach aggregation based on strain odour recognition," *Animal Behaviour*, vol. 68, pp. 793–801.

[71] R. Brooks, "Intelligence without reason," *Artifficial Intelligence*, 1991.

[72] J. Tsotsos, "Behaviorist intelligence and the scaling problem," tech. rep., Department of Computer Science, University of Toronto, 1986.

[73] J. Kosecka, "Experiments in behavior composition," *Journal of Robotics and Autonomous Systems*, vol. 21, pp. 37–51, 1997.

[74] M. Asadpour and R. Siegwart, "Compact q-learning for micro-robots," in *Proc. of the first European Conf. on Mobile Robots [ECMR]*, (Poland), 2003.

[75] M. Asadpour and R. Siegwart, "Compact q-learning optimized for micro-robots with processing and memory constraints," *Journal of Robotics and Autonomous Systems*, vol. 48, pp. 49–61, 2004.

[76] K. Ikuta, K. Yamamoto, and K. Sasaki, "Development of remote microsurgery robot and new surgical procedure for deep and narrow space," in *Proc. of the 2003 IEEE Int. Conf. on Robotics and Automation [ICRA]*, (Taipei, Taiwan), pp. 1103–1108, September 2003.

[77] B. Wilcox, "Mars microrover navigation: performance evaluation and enhancement," in *Proc. of the Int. Conf. on Intelligent Robots and Systems [IROS]*, vol. 1, (Washington, DC, USA), p. 433, IEEE Computer Society, 1995.

[78] A. Beyeler, C. Mattiussi, J. Zufferey, and D. Floreano, "Vision-based altitude and pitch estimation for ultra-light indoor aircraft," in *IEEE Int. Conf. on Robotics and Automation [ICRA]*, (Orlando, FL), pp. 2836–2841, 2006.

[79] G. Caprari, *Autonomous Micro-robots: Applications and Limitations.* PhD thesis, EPFL, Switzerland, 2003.

[80] D. Floreano, N. Schoeni, G. Caprari, and J. Blynel, "Evolutionary bits'n'spikes," in *Proc. of the 8th Int. Conf. on Artificial Life* (R. Standish, M. Bedau, and H. Abbass, eds.), MIT Press, 2002.

[81] F. Dean, M. Gini, , and J. Slagle, "Rapid unsupervised connectionist learning for backing a robot with two trailers," in *IEEE Int. Conf. on Robotics and Automation [ICRA]*, 1997.

[82] D. Hougen, "Use of an eligibility trace to self-organize output," in *Science of Artificial Neural Networks II, Proc. of SPIE* (D. W. Ruck, ed.), vol. 1966, pp. 436–447, 1993.

[83] R. Byrne, D. Adkins, S. Eskridge, J. Harrington, E.J.Heller, and J. Hurtado, "Miniature mobile robots for plume tracking and source localization research," *Journal of Micromechatronics*, vol. 1, no. 3, pp. 253–261, 2002.

[84] J. McLurkin, "Using cooperative robots for explosive ordnance disposal," tech. rep., Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge MA, 1996.

[85] T. Fukuda, H. Mizoguchi, K. Sekiyama, and F. Arai, "Group behavior control for mars (micro autonomous robotic system)," in *Proc. of the 1999 IEEE Int. Conf. on Robotics and Automation [ICRA]*, pp. 1550–1555, 1999.

[86] J. Kim, M. Jung, H. Shim, and S. Lee, "Autonomous micro-robot 'kity' for maze contest," in *Proc. of Int. Syposium on Artificial Life and Robotics*, pp. 261–264, 1996.

[87] Meloe Micro Robots, Active Structure Laboratory, Université Libre de Bruxelles, http://www.ulb.ac.be/scmero/robotics.html#micro.

[88] C. Watkins, *Learning from Delayed Rewards.* PhD thesis, King's College, 1989.

[89] C. Watkins and P. Dayan, "Q-learning (technical note)," *Machine Learning: Special issue on reinforcement learning*, vol. 8, 1992.

[90] C. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[91] T. Jaakkola, M. Jordan, and S. Singh, "On the convergence of stochastic iterative dynamic programming algorithms," *Neural Computation*, vol. 6, no. 6, pp. 1185–1201, 1994.

[92] J. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Proc. of the 2nd Int. Conf. on Genetic Algorithms and their Application*, (Hillsdale, NJ), pp. 14–21, 1987.

[93] G. Caprari, K. Arras, and R. Siegwart, “Robot navigation in centimeter range labyrinths,” in *Proc. of the 5th Int. Heinz Nixdorf Symposium: Autonomous Mini-robots for Research and Edutainment [AMiRE]* (U.Rückert, J.Sitte, and U.Witkowski, eds.), 2001.

[94] G. Caprari, K. Arras, and R. Siegwart, “The autonomous miniature robot alice: From prototypes to applications,” in *Proc. of the 2000 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems [IROS]*, pp. 793–798, IEEE Press, 2000.

[95] R. Siegwart, C. Wannaz, P. Garcia, , and R. Blank, “Guiding mobile-robots through the web,” in *Workshop Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems [IROS]*, (Victoria, Canada), 1998.

[96] N. Mehta, S. Natarajan, P. Tadepalli, and A. Fern, “Transfer in variable-reward hierarchical reinforcement learning,” in *Advances in Neural Information Processing Systems [NIPS], workshop on transfer learning*, 2005.

[97] M. Asadpour, M. N. Ahmadabadi, and R. Siegwart, “Reduction of learning time for robots using automatic state abstraction,” in *Proc. of the First European Symposium on Robotics* (H. Christensen, ed.), vol. 22 of *Springer Tracts in Advanced Robotics*, (Palermo, Italy), pp. 79–92, Springer-Verlag, Mar. 2006.

[98] R. Parekh, J. Yang, and V. Honavar, “Constructive neural network learning algorithms for multi-category pattern classification,” *IEEE Trans. on Neural Networks*, vol. 11, no. 2, pp. 436–451, 2000.

[99] B. Bakker and J. Schmidhuber, “Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization,” in *Proc. of the 8th Conf. on Intelligent Autonomous Systems [IAS]* (F. Groen, N. Amato, A. Bonarini, E. Yoshida, and B. Kröse, eds.), (Amsterdam, Netherlands), pp. 438–445, 2004.

[100] W. Uther and M. Veloso, “The lumberjack algorithm for learning linked decision forests,” in *Proc. of PRICAI*, 2000.

[101] D. Chapman and L. Kaelbling, “Input generalization in delayed reinforcement learning: An algorithm and performance comparisons,” in *Proc. of 12th Int. Joint Conf. on Artificial Intelligence [IJCAI]*, pp. 726–731, 1991.

[102] W. Uther, *Tree Based Hierarchical Reinforcement Learning.* PhD thesis, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2002.

[103] A. Jonsson and A. G. Barto, “Automated state abstraction for options using the u-tree algorithm,” in *Advances in Neural Information Processing Systems [NIPS]*, (Cambridge, MA), pp. 1054–1060, MIT Press, 2001.

[104] T. Dean and G. Robert, “Model minimization in markov decision processes,” in *Proc. of AAAI-97*, p. 76, 1997.

[105] M. Au and F. Maire, "Automatic state construction using decision tree for reinforcement learning agents," in *Int. Conf. on Intelligent Agents, Web Technologies and Internet Commerce [CIMCA]*, (Gold Coast, Australia), 2004.

[106] S. K. Murthy, "Automatic construction of decision trees from data: A multi-disciplinary survey," *Data Mining and Knowledge Discovery*, vol. 2, no. 4, pp. 345–389, 1998.

[107] W. Loh and Y. Shih, "Split selection methods for classification trees," *Statistica Sinica*, 1997.

[108] C. E. Brodley and P. E. Utgoff, "Multivariate decision trees," *Machine Learning*, vol. 19, no. 1, pp. 45–77, 1995.

[109] O. T. Yildiz and E. Alpaydin, "Linear discriminant trees," in *Proc. of the 17th Int. Conf. on Machine Learning*, pp. 1175–1182, Morgan Kaufmann, San Francisco, CA, 2000.

[110] O. Yildiz and E. Alpaydin, "Omnivariate decision trees," *IEEE Trans. on Neural Networks*, vol. 12, no. 6, pp. 1539–1546, 2001.

[111] R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[112] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees.* Belmont, CA: Wadsworth, 1984.

[113] M. Asadpour, M. N. Ahmadabadi, and R. Siegwart, "Heterogeneous and hierarchical cooperative learning via combining decision trees," in *Proc. of the IEEE/RSJ Conf. on Intelligent Robots and Systems [IROS]*, (Beijing, China), Oct. 2006.

[114] S. Thrun, "Lifelong learning: A case study." Carnegie Mellon University: CS-95-208, 1995.

[115] I. Riachi, M. Asadpour, and R. Siegwart, "Cooperative learning for very long learning tasks: A society inspired approach to persistence of knowledge," in *European Conf. on Machine Learning, Cooperative Multi-Agent Learning workshop*, (Porto, Portugal), 2005.

[116] R. Maclin and J. W. Shavlik, "Creating advice-taking reinforcement learners," *Machine Learning*, vol. 22, no. 1-3, pp. 251–281, 1996.

[117] G. Hayes and J. Demiris, "A robot controller using learning by imitation," in *Proc. of Int. Symposium on Intelligent Robotic Systems* (A. Borkowski and J. Crowley, eds.), vol. 1, pp. 198–204, Lifia Imag, Grenoble, France, 1994.

[118] T. G. Dietterich, "Ensemble methods in machine learning," *Lecture Notes in Computer Science*, vol. 1857, pp. 1–15, 2000.

[119] M. N. Ahmadabadi and M. Asadpour, "Expertness based cooperative q-learning," *IEEE Trans. on Systems, Man and Cybernetics, Part B*, vol. 32, no. 1, pp. 66–76, 2002.

[120] Y. U. Cao, A. S. Fukunaga, and A. Kahng, "Cooperative mobile robotics: Antecedents and directions," *Autonomous Robots*, vol. 4, p. 7, Mar. 1997.

[121] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative learning," in *Readings in Agents* (M. N. Huhns and M. P. Singh, eds.), pp. 487–494, San Francisco, CA, USA: Morgan Kaufmann, 1997.

[122] M. N. Ahmadabadi, M. Asadpour, S. H. Khodaabakhsh, and E. Nakano, "Expertness measuring in cooperative learning," in *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems [IROS]*, vol. 3, pp. 2261–2267, 2000.

[123] M. N. Ahmadabadi, M. Asadpour, and E. Nakano, "Cooperative q-learning: the knowledge sharing issue," *Advanced Robotics*, vol. 15, no. 8, pp. 815–832, 2001.

[124] S. M. Eshgh and M. N. Ahmadabadi, "An extension of weighted strategy sharing in cooperative q-learning for specialized agents," in *Proc. of 9th Int. Conf. on Neural Information [ICONIP]*, pp. 106–110, 2002.

[125] S. M. Eshgh, B. Araabi, and M. N. AhmadAbadi, "Cooperative q-learning through state transitions: A method for cooperation based on area of expertise," in *Proc. of 4th Asia-Pacific Conf. on Simulated Evolution And Learning [SEAL]*, (Singapore), pp. 61–65, 2002.

[126] S. M. Eshgh and M. N. Ahmadabadi, "On q-table based extraction of area of expertise for q-learning agents," in *World Automation Congress [WAC]*, 2004.

[127] M. N. Ahmadabadi, A. Imanipour, B. N. Araabi, M. Asadpour, and R. Siegwart, "Knowledge-based extraction of area of expertise for cooperation in learning," in *Proc. of the IEEE/RSJ Conf. on Intelligent Robots and Systems [IROS]*, (Beijing, China), Oct. 2006.

[128] L. O. Hall, N. Chawla, and K. W. Bowyer, "Decision tree learning on very large data sets," in *IEEE Conf. on Systems, Man, and Cybernetics*, 1998.

[129] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[130] Y. Freund and R. Schapire, "Experiments with a new boosting algorithm," in *Proc. of 13th Int. Conf. on Machine Learning*, (San Francisco), pp. 148–156, Morgan Kaufmann, 1996.

[131] A. McCallum, "Overcoming incomplete perception with utile distinction memory," in *Int. Conf. on Machine Learning*, pp. 190–196, 1993.

Curriculum Vitae

Masoud Asadpour was born in Lar, Iran, in 1975. He received his B.Sc. degree in Computer Software Engineering from Sharif University of Technology, Tehran, Iran, in 1997. He got his M.Sc. degree in Machine Intelligence and Robotics from the University of Tehran, in 1999. He received the *Best Researcher Award* from University of Tehran, in 2000. He worked for two years as a researcher at the Intelligent Systems Research Center, Institute for Studies on Theoretical Physics and Mathematics (IPM), Tehran. He started his PhD at Autonomous Systems Lab., École Polytechnique Fédéral de Lausanne in Dec. 2002. His research interests are Machine Learning, Swarm Intelligence, Cooperative Robotics, and Multi-Agent systems.

Publication List

Theses

1. **M. Asadpour**, *Study of Cooperative Learning in a Team of Object Transferring Robots*, MSc Thesis, University of Tehran, 2000 (in persian).
2. **M. Asadpour**, *A Persian Web Browser*, BSc Thesis, Sharif University of Technology, 1997 (in persian).

Journal Papers

1. **M. Asadpour**, F. Tâche, G. Caprari, W. Karlen, and R. Siegwart, "Robot-animal interaction: Perception and behavior of insbot," *Int. Journal of Advanced Robotics Systems*, vol. 3, pp. 93–98, 2006.
2. **M. Asadpour**, and R. Siegwart, "Compact Q-learning optimized for micro-robots with processing and memory constraints," *Journal of Robotics and Autonomous Systems*, vol. 48, pp. 49–61, 2004.
3. M. N. Ahmadabadi and **M. Asadpour**, "Expertness based cooperative Q-learning," *IEEE Transaction on Systems, Man and Cybernetics, Part B*, vol. 32, no. 1, pp. 66–76, 2002.
4. M. N. Ahmadabadi, **M. Asadpour**, and E. Nakano, "Cooperative Q-learning: the knowledge sharing issue," *Journal of Advanced Robotics*, vol. 15, no. 8, pp. 815–832, 2001.

Conference Papers

1. **M. Asadpour**, M. N. Ahmadabadi, and R. Siegwart, “Heterogeneous and hierarchical cooperative learning via combining decision trees,” in *Proc. of the IEEE/RSJ Conf. on Intelligent Robots and Systems [IROS]*, (Beijing, China), Oct. 2006.
2. M. N. Ahmadabadi, A. Imanipour, B. N. Araabi, **M. Asadpour**, and R. Siegwart, “Knowledge-based extraction of area of expertise for cooperation in learning,” in *Proc. of the IEEE/RSJ Conf. on Intelligent Robots and Systems [IROS]*, (Beijing, China), Oct. 2006.
3. **M. Asadpour**, M. N. Ahmadabadi, and R. Siegwart, “Reduction of learning time for robots using automatic state abstraction,” in *Proc. of the First European Symposium on Robotics* (H. Christensen, ed.), vol. 22 of *Springer Tracts in Advanced Robotics*, (Palermo, Italy), pp. 79–92, Springer-Verlag, Mar. 2006.
4. I. Riachi, **M. Asadpour**, and R. Siegwart, “Cooperative learning for very long learning tasks: A society inspired approach to persistence of knowledge,” in *European Conf. on Machine Learning, Cooperative Multi-Agent Learning workshop*, (Porto, Portugal), 2005.
5. **M. Asadpour**, F. Tâche, G. Caprari, W. Karlen, and R. Siegwart, “Robot-animal interaction: Perception and behavior of insbot,” in *Conceptual Modeling and Simulation Conf. [CMS]*, (Marseille,France), 2005.
6. F. Tâche, **M. Asadpour**, G. Caprari, W. Karlen, and R. Siegwart, “Perception and behavior of insbot : Robot-animal interaction issues,” in *IEEE Conf. on Robotics and Biomimetics [ROBIO]*, (Hong Kong), 2005.
7. S. Garnier, C. Jost, R. Jeanson, J. Gautrais, **M. Asadpour**, G. Caprari, and G. Theraulaz, “Collective decision-making by a group of cockroach-like robots,” in *Proc. of the IEEE Swarm Intelligence Symposium*, (Pasadena, CA), Jun. 2005.
8. S. Garnier, C. Jost, R. Jeanson, J. Gautrais, **M. Asadpour**, G. Caprari, and G. Theraulaz, “Aggregation behaviour as a source of collective decision in a group of cockroach-like robots,” in *Proc. of the 8th European Conf. on Artificial Life [ECAL]* (M. Capcarrere, ed.), vol. 3630 of *Lecture Notes in Artificial Intelligence*, (Berlin Heidelberg), pp. 169–178, Springer-Verlag, September 2005.
9. C. Jost, S. Garnier, R. Jeanson, **M. Asadpour**, J. Gautrais, and G. Theraulaz, “The embodiment of cockroach behaviour in a micro-robot,” in *Proc. of the 35th Int. Symposium on Robotics*, (Paris, France), Mar. 2004.
10. **M. Asadpour** and R. Siegwart, “Compact Q-learning for micro-robots,” in *Proc. of the first European Conf. on Mobile Robots [ECMR]*, (Poland), 2003.
11. Mohammad Ali Abbasi, M.N. Ahmadabadi, **M. Asadpour**, “A study on effects of reward distribution on learning of a team of distributed agents,” *The 7th Iranian Computer Society Conf.*, 2002 (in Persian)
12. M.N. Ahmadabadi, **M. Asadpour**, S.H. Khodaabakhsh, and E. Nakano, “Expertness measuring in cooperative learning,” in *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems [IROS]*, vol. 3, pp. 2261–2267, 2000.

13. **M. Asadpour**, and M.N. Ahmadabadi, “Weighted Strategy Sharing: A New Multi-Agent Cooperative Learning Method Based on Expertness Measuring,” *8th Iranian Conf. on Electrical Engineering*, Isfahan, Iran, Mar. 2000 (in Persian)

14. H. Khodaabakhsh, **M. Asadpour**, M.N. Ahmadabadi, “Cooperative Learning in Object Pushing,” *The 1st Iranian Symposium on Intelligent Systems*, Tehran, Iran, 2000 (in Persian)