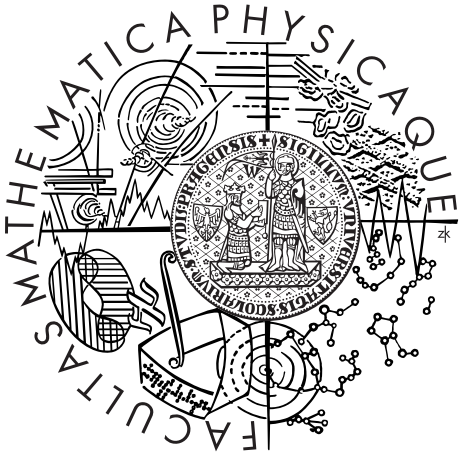
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Ondřej Filip

Distributed Monte-Carlo Tree Search for Games with Team of Cooperative Agents

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Viliam Lisý, MSc.

Study programme: Theoretical Computer Science

Specialization: Nonprocedural Programming and Artificial Inteligence

Prague 2013

I would like to thank my supervisor Mgr. Viliam Lisý, MSc., for all his effort and valuable advice. I also want to thank my girlfriend, family, closest friends and roommates for their support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Distribuovaný Monte-Carlo Tree Search pro hry s týmem kooperujících agentů

Autor: Bc. Ondřej Filip

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Viliam Lisý, MSc., Centrum agentních technologií, České vysoké učení technické v Praze

Abstrakt: Cílem této práce je návrh, implementace a experimentální evaluace distribuovaných algoritmů pro plánování akcí týmu kooperujících autonomních agentů založených na Monte-Carlo tree search algoritmu. Jednotlivé algoritmy vyžadují rozdílné množství komunikace. V práci jsou shrnuty relevantní poznatky o Monte-Carlo tree search algoritmu, jeho paralelizaci a distribuovatelnosti a algoritmech pro distribuovanou koordinaci autonomních agentů. Navržené algoritmy jsou testovány v prostředí zjednodušené hry Ms Pac-Man. Testována je síla jednotlivých algoritmů v závislosti na času výpočtu, množství komunikace a robustnosti vůči selhání komunikace. Jednotlivé algoritmy jsou dle těchto charakteristik porovnány.

Klíčová slova: Multi-agentní systémy, Monte-Carlo Tree Search, distribuované algoritmy

Title: Distributed Monte-Carlo Tree Search for Games with Team of Cooperative Agents

Author: Bc. Ondřej Filip

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Viliam Lisý, MSc., Agent Technology Center, Czech Technical University in Prague

Abstract: The aim of this work is design, implementation and experimental evaluation of distributed algorithms for planning actions of a team of cooperative autonomous agents. Particular algorithms require different amount of communication. In the work, the related research on Monte-Carlo tree search algorithm, its parallelization and distributability and algorithms for distributed coordination of autonomous agents. Designed algorithms are tested in the environment of the game of Ms Pac-Man. Quality of the algorithms is tested in dependence on computational time, the amount of communication and the robustness against communication failures. Particular algorithms are compared according to these characteristics.

Keywords: Multi-agent systems, Monte-Carlo Tree Search, distributed algorithms

Contents

Introduction

Monte-Carlo tree search turned out as an algorithm reaching unprecedent results in playing a game of Go [4] and a variety of other games. According to properties of the algorithm, Monte-Carlo tree search has been suggested as an algorithm suitable for multi-agent coordination, such as in pursuit-evasion games. A special class of multi-agent coordination problems is coordination of a team of robots, for example in a contest environment or autonomous fighting robots, such as unmanned aerial vehicles. Common algorithms for team coordination consider reliable inter-agent communication. In our thesis, we, in opposite to previous approaches, face the nature of real-world communication problems such as limited communication, transmission delays or communication failures.

In our thesis, we aim to design and experimental evaluation of algorithms based on Monte-Carlo tree search suitable for distributed planning of actions of a team of autonomous agents. For simplicity, we consider a game with a team of cooperative players as an environment for team coordination with respect to the real-world communication problems.

The initial point of our research was a review of parallelization approaches to Monte-Carlo tree search [3, 4, 15] that gave us a solid cornerstone for the further work. Second source of initial inspiration, especially for a discussion on communication between agents, was distributed constraint optimization algorithms [16]. Finally, the third influence for the thesis was articles dealing with application of Monte-Carlo tree search to a domain of a game of Ms Pac-Man [8, 10].

The thesis is divided into four chapters. Chapter 1 contains general notes on perfect-information games with teams in extensive form, including definition of such games and discussion on inter-agent coordination and communication.

In Chapter 2, relevant research on Monte-Carlo tree search is reviewed, together with MCTS parallelization.

Once Monte-Carlo tree search is reviewed and games with teams are defined, we propose our distributed algorithms in Chapter 3. Next to simple algorithms serving for comparison (independent agents, joint-action exchanging agents), we propose two algorithms based on existing parallel algorithms (root exchanging agents, simulation results exchanging agents) and one algorithm in which we attempted to create an algorithm possessing good qualities of former two algorithms but suppressing bad ones. General properties and comparison of the algorithms are discussed in Section 3.4.

Chapter 4 contains an evaluation of proposed algorithms on a domain of a game of Ms Pac-Man. In particular, we describe the framework in Section 4.1 together with rules of the game and simplifications of the game we have done for our purposes. We give relevant notes on implementation of Monte-Carlo tree search on the chosen domain and simulation of distributed environment with real-world properties (communication failures etc.) in Section 4.2. Before we have ran evaluational experiments, we had devoted some time for basic tunings of parameters of Monte-Carlo tree search (Section 4.4). Finally, we propose results of the experiments together with discussion on them and comparison of the algorithms in terms of the results in Section 4.5.

1. Multi-Agent Cooordination

In this chapter, we define perfect-information game in extensive form and perfect-information game with teams in extensive form, discuss the coordination of agents and inter-agent communication and shortly review two articles related to multi-agent coordination.

1.1 Perfect-Information Game

This thesis deals with the problem of coordination of agents in games. Before work on this topic will be reviewed and our algorithms proposed, it is necessary to define what the game in our context is. Finite, discrete, fully-observable, deterministic and static environment will be considered in which we will talk about *perfect-information game*. The following definition is based on the definition proposed in [13] and generalized for our purposes (original definition does not include a possibility of multiple players playing simultaneously and forces a play of exactly one player each turn).

Definition (Perfect-information game)**.** *A (finite)* ***perfect-information game*** *(in extensive form) is a tuple* $G = (P, A \cup \{\lambda\}, S = S_n \cup S_p, \chi, \sigma, u)$*, where:*

- P *is a set of* n *players;*
- A *is a (single) set of actions;*
- S_n *is a set of nonterminal choice states;*
- S_t *is a set of terminal states, disjoint from* S_n*;*
- $\chi : S_n \times P \mapsto 2^A \cup \{\lambda\} \setminus \varnothing$ *is the action function, which assigns to each choice node and player a set of possible actions,* $\lambda \notin A$ *is a neutral action played by the players not being on turn;*
- $\sigma : \{(s, a) | s \in S_n, a \in \prod_{i \in P} \chi(s, i)\} \mapsto S$ *is the successor function, which maps a choice node and an action tuple to a new choice node or terminal node such that for all* $s_1, s_2 \in S_n$ *and* $a_1 \in \prod_{i \in P} \chi(s_1, i), a_2 \in \prod_{i \in P} \chi(s_2, i)$*, if* $\sigma(s_1, a_1) = \sigma(s_2, a_2)$ *then* $s_1 = s_2$ *and* $a_1 = a_2$*; and*
- $u = (u_1, \ldots, u_n)$*, where* $u_i : S_t \mapsto \mathbb{R}$ *is a real-valued utility function for player* i *on the terminal nodes* S_t*.*

We say that a perfect-information game is ***turn-based*** *if each turn at most one player plays an action, in other words, if the following restriction on the action function* χ *is satisfied:*

$$\forall s \in S_n \; |\{i \in P | \chi(s, i) \neq \{\lambda\}\}| \leq 1 \tag{1.1}$$

Note that our definition allows nodes with no player being on turn. This is reason why Equation 1.1 contains non-strict inequality.

Contrariwise a perfect-information game will be called ***simultaneous*** *if simultaneous plays of multiple players are permitted and a state forcing a play of at least two players exists (and so Equation 1.1 is not satisfied).*

Because we deal only with perfect-information games in our thesis, we will usually refer to the perfect-information game and its variants as to *game*, *turn-based game* and *simultaneous game*. We will also bound a utility function to $[0, 1]$ (without loss of generality).

The aim of a player i is reaching a terminal node s with the greatest possible utility $u_i(s)$.

1.1.1 Conversion between Turn-Based and Simultaneous Games

Because a variant of MCTS used in this thesis requires turn-based games to ensure the convergence (as discussed in Section 2.2) and our algorithms, which are based on this specific MCTS variant, will be evaluated in the domain of a naturally simultaneous game, we will need a conversion from a turn-based game to a simultaneous game, which will be described here. Simultaneous nodes expansion modifies the game but our goal is not to develop the strongest possible players for the chosen domain but to study a coordination algorithm related to MCTS. Therefore, retaining of algorithm properties has been chosen.

We need to expand simultaneous actions to a sequence of single actions. To determine the order of such an expansion, we can simply define linear ordering on a set of players. Then each simultaneous action is splitted into multiple single-actions, one for each player on turn.

There is, of course, a catch in this approach since the turn-based game created by this conversion modifies game rules and adds an advantage to players which play later in originally simultaneous turn. This is because players playing later have more information than in the original simultaneous game as they additionally know actions performed by players playing in same simultaneous turn.

In MCTS, when playing a simultaneous game, the calculated action will be committed to the simultaneous game so the imbalance originating from the conversion will not affect the played game but only the algorithm itself. We face the decision which order of players to choose and we distinguish two main variants - *optimistic* and *pessimistic expansion*.

Optimistic expansion considers the player performing MCTS calculation playing after all the other players. This approach is called optimistic because the player supposes that other players will play as he expects. On the other hand, pessimistic expansion lets the player in MCTS calculations play first and gives other players information about his action which is more pessimistic variant for the player. Expansion of a simultaneous is illustrated by Figure 1.1.

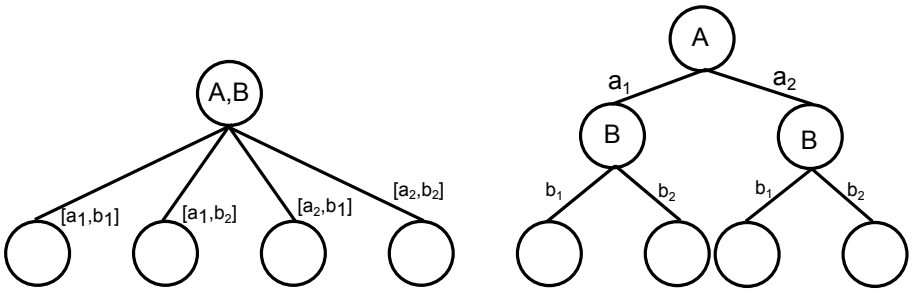


Figure 1.1: Simultaneous node expansion.
Left - the node represents a simultaneous play of players A and B with sets of moves $\{a_1, a_2\}$ and $\{b_1, b_2\}$. Right - expanded simultaneous node. Player A is chosen to play first. If the player A is an owner of the tree, the expansion is pessimistic; otherwise, the expansion is optimistic.

1.2 Team Coordination

1.2.1 Games with Teams of Players

Definition of perfect-information game serves (without further modifications) as an environment for *teams of players*. A team of players (or simply a team) is a set of agents attempting to reach the best result possible collectively. In other words, agents of a team share the utility function.

This can be simply applied to a perfect-information game. A team can be represented as a single player of a perfect-information game where actions of such a player are a joint-action composed of all actions performed by team members. The second possibility how to model a team in perfect-information game is adding teams as new entities in the game and considering players of a single team to share the same utility. Since it is a more natural concept allowing the team player to be viewed as independent agents, we will use this concept in our thesis. The following definition is a formalization of the concept.

Definition (Perfect-information game with teams)**.** *A (finite)* ***perfect-information game with Teams*** *(in extensive form) is a tuple* $G = (P, A \cup \{\lambda\}, S = S_n \cup S_p, \chi, \sigma, T, \tau, u)$*, where:*

- P*,* n*,* A*,* λ*,* S*,* S_n*,* S_t*,* χ *and* σ *are same as in perfect-information game;*
- T *is a set of teams;*
- $\tau : P \mapsto T$*, assigns players to the teams; and*
- $u = (u_1, \ldots, u_m)$*, where* $u_i : S_t \mapsto \mathbb{R}$ *is a real-valued utility function for team* i *on the terminal nodes* S_t*.*

  As in case of plain (without teams) perfect-information game, we distinguish turn-based and simultaneous game with teams of players. We say that a game with teams is ***turn-based*** *if only players of a same team play in all nodes. If we consider players of plain game being members of virtual distinct teams, such a condition is generalization of plain turn-based game. The condition is formalized by Equation 1.2.*

$$\forall s \in S_n \, |\{t \in T | \exists i \in \tau^{-1}(t) \; \chi(s,i) \neq \{\lambda\}\}| \leq 1 \tag{1.2}$$

Specially, we distinguish ***pure turn-based game*** *with teams if only one player plays in each node (as requires plain turn-based game without teams in Equation 1.1) and* ***weak turn-based game*** *not satisfying the condition.* ***Simultaneous game with teams*** *is a game with teams not satisfying Equation 1.2.*

A tuple of actions played by single team T_a in state S_b will be called *joint-action* (formally $(a_p^{S_b})_{\tau(p_a)=T_a}$).

1.2.2 Distributed Coordination

In this subsection, we briefly review two related approaches to the team coordination. The first one is using of the distributed constraint optimization (DCOP, [13]). The second approach is coordination by Monte-Carlo tree search.

The distributed constraint optimization has been successfully applied to the problem of the coordination of a large team of sensing agents [16]. The purpose of sensing agents is to dynamically cover given area. In the article, DCOP has been applied as follows. Each agent has its position, a sensing range (range covered by agent's sensors), a mobility range (area reachable by an agent in one iteration of an algorithm) and a credibility. The environmental requirement function then says how should the environment be covered by the sensors. In addition, events influencing agents' credibilities or the environmental requirement function occur at certain times (defined by the number of iterations from the beginning of the computation). This problem setting can be modelled by extended DCOP, named as DCOP_MST (DCOP for mobile sensing teams) in the article. Position of the agents is changed after each iteration. Adjusted MGM algorithm (a local incomplete algorithm for DCOP solving) were used. Experiments proposed in the article compares usage of MGM with two other search algorithms and showed that only MGM gives satistying results with fast convergence to local minima.

In [10], Monte-Carlo tree search has been applied to the problem of the coordination of a team of ghosts in the game of Ms Pac-Man. Actions of the ghosts are planned centrally by the MCTS computation. In addition, several improvements are used. Most important is that a tree built by the algorithm does not contain Pac-Man moves. Instead of that, the most probable move is guessed and suggested to be played. The ghosts controller stores all moves played by Pac-Man during the game in the space of several game characteristics approximing the full game settings. Pac-Man's move is then predicted according to moves stored with the k-nearest neighbours. The ghosts controller driven by MCTS won the CEC 2011 Ms Pac-Man vs Ghost Team Competition [11].

1.2.3 Communication and Coordination

Let us consider players of a team as defined in Section 1.2.1. Such players share the utility function, attempting its maximization. The definition of perfect-information game with teams doesn't talk about the coordination; players are viewed as autonomous agents sharing no information about their intentions or

reasoning. But sharing such information is natural requirement for a team and is the way how to effectively coordinate the players of the team.

Agents can share information by passing messages. In real-world applications, messages are transmitted over a communication channel which has limited capacity and may be possibly unreliable; it is necessary to consider length of message encoded to a particular form. These properties, in addition, force us considering the delay in transmission.

As an example of usage of a message passing between independent agents is the distributed constraint optimization problem [16].

2. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is an iterative best-first search with stochastic positional evaluation, anytime property and fast convergence. As Chaslot summed up in [6], MCTS was simultaneously developed in three variants [5, 7, 9] in 2006. Specific variant used in this thesis along with all important details and explanation of the properties is discussed in this chapter. In addition, section 2.3 sumarizes current approaches to parallelization of MCTS.

2.1 Algorithm Description

Chaslot, Winands, and Van Den Herik provide good description of the Monte-Carlo Tree Search algorithm in [4]. A variant of MCTS as well as the terminology used in this thesis are based mainly on this paper. Related pseudocodes follows this specific variant.

Algorithm 2.1: *Select*(*tree*)

```
/* algorithm selecting a node for running a playout          */
```
Data: *tree*... MCTS tree
Result: Leaf node chosen by the selection is returned. The selection is determined by a particular *SelectionStep*.

```
1 curr_node ← Root(tree)
2 while curr_node ∈ T do
3 |   last_node ← curr_node
4 |   curr_node ← SelectionStep(curr_node)
5 return last_node
```

Algorithm 2.2: *Backpropagate*(*tree*, *node*, *rewards*[ ])

```
/* simulation results propagation                            */
```
Data: *tree* ... MCTS tree
node ... node to which the rewards are being backpropagated
rewards[ ] ... array of rewards
Result: *tree* with playout rewards added to all *node*'s ascendants

```
1 AddRewards(node, rewards[ ])
2 parent ← Parent(node)
3 if parent ∈ tree then
4 |   Backpropagate(tree, parent, rewards[ ])
```

Monte-Carlo Tree Search is iteratively building a search tree as depicted by Figure 2.1 (originally published in [6]) and Algorithms 2.1, 2.2 and 2.3 which follow the method in its generality. Further details of chosen variant of MCTS are discussed in subsequent sections and their related Algorithms, namely 2.4, 2.5, 2.6 and 2.7.

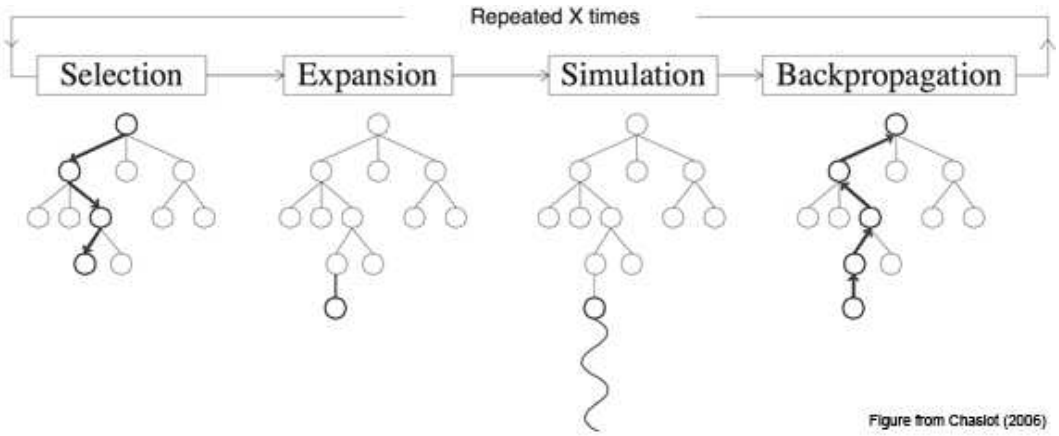


Figure 2.1: Monte-Carlo Tree Search.
Monte-Carlo tree search consists of iterating over four phases and can be interrupted at any time. Figure originates from [6].

Algorithm 2.3: $MCTSLoop(tree)$

```
/* main MCTS computation loop                                    */
```
Data: $tree \ldots$ MCTS tree
Result: $tree$ is enlarged by newly expanded nodes and results of playouts performed are added. Node representing the best evaluated position reachable by one action is returned

1 **while** $EnoughTime()$ **do** `// Main MCTS loop`
2 $\quad node \leftarrow Select(tree)$ `// Phase 1: Selection`
3 $\quad node \leftarrow Expand(node)$ `// Phase 2: Expansion`
4 $\quad rewards[\,] \leftarrow Playouts(node)$ `// Phase 3: Simulation`
5 $\quad Backpropagate(tree, node, rewards[\,])$ `// Ph 4: Backpropagation`
6 **return** $\underset{n \in Children(root)}{\arg\max} \; c_n$ `// Return most visited child`

Each node i of the tree t contains at least two values - visit count n_i saying how many random position evaluations have been performed from i and all its descendants and actual value v_i which aggregates the results of these evaluations (usually as an average). In addition, the game position represented by i, $Position(i)$, and pointers defining tree structure have to be included.

Position p represents the current game state. Following functions are appliable to p: $Actions(p)$ returns set of actions performable in p, $Step(p, a)$ returns new position reachable by performing action a in p and $Score(p) \in [0; 1]$ returns static evaluation of the position (e.g. current game score).

$Parent(i)$ is pointing to the parent of the node or to the $NullNode$ if i is the root of the tree. $Children(i)$ is the set of i's children nodes. Once i is expanded, $Children(i)$ contains nodes representing positions reachable by set of distinct actions $Actions(p)$ performable from $p = Position(i)$. Particular action required to reaching i's child j is $Action(j)$.

The tree itself provides the pointer to its root $Root(t)$. Function $EnoughTime()$ returns *true* if there is enough time to pass at least one more MCTS iteration and return the best evaluated node. We use also other notations in pseudocodes of our thesis which are omitted here. All such notations are at least covered by a comment in a pseudocode.

Each iteration of MCTS consists of four phases - *selection*, *expansion*, *simulation* and *backpropagation*. During the selection phase the algorithm passes through the tree to a particular leaf i where better-evaluated but less-visited nodes are preferred. Appropriate balance between these claims is the main objective of this phase. Once a leaf node is selected, the expansion phase follows. In this phase, according to a certain condition (which is usually condition on node's visit count), i itself is selected, or new nodes reachable by actions from $Actions(i)$ are added to set $Children(i)$ and any of these nodes is selected. The next phase, simulation (also called *playout*), plays a random game (or several games) starting in position defined by expanded node to the end (simulation may be possibly terminated earlier). Results from the simulations are then backpropagated to all expanded node's ancestors during the fourth phase - backpropagation. The phases of the MCTS and their specific variants are further discussed in subsequent sections.

2.1.1 Selection

The process of selection consists of selection steps passing from a node to one of its children. Each such a step meets exploration-exploitation problem where exploitation tends to choose the so far best evaluated node and exploration on the other side promotes undiscovered ways in the tree. This problem can be viewed as an instance of a well-known problem called the Multi-Armed Bandit Problem (MAB). Its definition is adopted from [1] and [9]:

Definition (K-armed bandit problem)**.** *Let us have independent random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$. Each i is the index of a gambling machine and $X_{i,1}$, $X_{i,2}$,... are identically distributed rewards with unknown expected value μ_i yielded by successive plays of machine i. For the simplicity the rewards are bounded to $[0, 1]$.*

*A **policy** A is an algorithm that chooses the next machine to play based on the sequence of past plays and obtained rewards. Let n_i be the number of times machine i has been played by A during the first n plays and I_i^A be the index of a machine played in nth play. Then the regret of A after n plays is defined as*

$$R_n^A = n\mu^* - \sum_{j=1}^{K} n_j \mu_j, \text{ where } \mu^* \stackrel{\text{def}}{=} \max_{1 \leq i \leq K} \mu_i \tag{2.1}$$

*thus the **regret** R_n^A is the loss caused by the policy not always playing the best machine.*

***K-armed bandit problem** consists in finding optimal policy A^* minimizing expected regret R_n^A.*

Auer, Cesa-Bianchi, and Fischer introduced computationaly effective optimal policy UCB1 [1] having regret bounded to $O(\log n)$. Abbreviation UCB stands for *Upper-Confidence Bound* and refers to value maximized by the policy. Let us define the value of UCB1:

$$UCB1(\bar{X}_{i,n_i}, n, n_i) = \begin{cases} \bar{X}_{i,n_i} + \sqrt{\frac{2 \log n}{n_i}} & if \quad n_i > 0 \\ \infty & otherwise \end{cases} \tag{2.2}$$

.

Then the policy itself chooses the machine as follows:

$$I_{UCB1}(n+1) = \arg\max_{i \in 1,\ldots,K} UCB1(\bar{X}_{i,n_i}, n, n_i) \tag{2.3}$$

.

UCB1 consists of an average of previous rewards and a bias decreasing with number of the machine's plays. In addition, the significance of the bias is growing with the total number of plays what leads to increase of exploration. Each machine is tried once before further selection based on previous evaluation and the bias is used. This is forced by setting UCB1 value to ∞ for machines not yet tried.

UCB1 applied to Trees [9] is a specific selection strategy we used in our thesis. It is derived from UCB1 using values v_i and n_i stored in particular node i. In addition, the bias coefficient C is introduced in [4]. The purpose of this coefficient is tuning the balance between exploration and exploitation. UCB1 contains fixed value of C equal to $\sqrt{2}$ which can be used as a good reference value. Tuning of C is usually done experimentally. The form of the formula for UCT and related selection applied in node p is as follows:

$$UCT(v_i, n_p, n_i) = \begin{cases} v_i + C\sqrt{\frac{\log n_p}{n_i}} & if \quad n_i > 0 \\ \infty & otherwise \end{cases} \tag{2.4}$$

$$I_{UCT}^p = \arg\max_{i \in Children(j)} UCT(v_i, n_p, n_i) \tag{2.5}$$

According to the UCT, the selection passes down through the tree starting in the root where the child with greatest UCT value is chosen in each node until a leaf is reached. As mentioned in [4], experimental results show that the UCT value does not guide the selection well until enough trials have been performed in node

p (n_p is too small). To overcome this empirical finding, the selection is guided by *simulation strategy* which uses domain-specific information (described in section 2.1.3) until n_p does not reach certain threshold T. Selection phase consisting of selection steps is depicted by Algorithm 2.1. The selection steps using UCT selection strategy is then depicted by Algorithm 2.4. In our implementation of MCTS, we use the UCT. In MCTS for a game of Go in [4] the constants used are $C = 0.7$ and $T = 30$.

Algorithm 2.4: *UCTSelectionStep*(*node*)

```
/* one step of selection phase, an unvisited child, or a child
   having best UCT of given node is selected                    */
```

Data: *node*... a node of MCTS tree

Result: a children node chosen by a selection

1 **if** $Children(node) = \varnothing$ **then** // *node* is a leaf or terminal node
2 ⌊ **return** *NullNode*
3 **if** $n_{node} < T_s$ **then** // Not enough simulations played
4 ⌊ **return** *SimulationStep*(*node*) // Follow simulation strategy
5 **if** $\exists i \in Children(node)$ *such that* $n_i = 0$ **then**
6 ⌊ **return** i // Try all children first
7 **return** $\underset{i \in Children(node)}{\arg\max} \left(v_i + C\sqrt{\frac{\log n_{node}}{n_i}} \right)$ // Follow UCT value

2.1.2 Expansion

Algorithm 2.5: *Expand*(*node*)

```
/* expansion of an unexpanded node                              */
```

Data: *node*... a leaf node of MCTS tree

Result: *node* itself is returned, or new children reachable by actions valid in position defined by *node* are added to *node*. One if these children is then returned.

1 **if** $n_{node} < T_e$ **then** // Not enough simulations performed
2 ⌊ **return** *node*
3 **if** $Actions(node) = \varnothing$ **then** // *node* is a terminal node
4 ⌊ **return** *node*
5 **foreach** $action \in Actions(node)$ **do**
6 ⌊ $next\_pos \leftarrow Step(Position(node), action)$
7 ⌊ $child \leftarrow new\ Node(next\_pos, action)$
8 ⌊ $Children(node) \leftarrow Children(node) \cup \{child\}$
9 **return** *child*

The expansion decides whether or not the selected leaf will be expanded and so if the simulation will begin in the leaf itself or in one of its children. The most common variant used also in our thesis is that the node is expanded once it

reaches a particular visit count T_e. Greater value in such a condition causes higher simulation count in each node and therefore a stronger evaluation. On the other hand, the process of building tree is slowed down. With $T_e = 1$ the expansion becomes even simplier and the leaf is simply always expanded. Algorithm 2.5 ilustrates expansion performed after T_e simulations have been performed from a leaf.

2.1.3 Simulation

Algorithm 2.6: *Playouts*(*node*)

```
/* simulation phase returning rewards from performed playouts
   */
Data: node... a node of MCTS tree
Result: An array of results of performed simulations is returned.
1 for i ← 1 to SIMULATIONS_COUNT do
2     p ← Position(node)
3     while not TerminalCondition(p) do
4         p ← Step(p, SimulationAction(p))
5     rewards[i] ← Score(p)
6 return rewards[ ]
```

The simulation, or *playout*, performs random play starting in the position defined in the node selected by previous phases - selection and expansion. Simulation steps can be plain random actions or, to improve the strength of simulation, the simulation can follow a pseudo-random *simulation strategy* designed according to domain-specific information. A proper compromise between playing randomly and following deterministic heuristic has to be chosen.

In original Monte-Carlo tree search, the simulation is terminated at the end of simulated game. To spare time spent in simulations, we can also terminate it earlier on certain condition [8] (e.g. number of simulations can be limited). After the termination, player's position score is returned (chosen, without loss of generality, from interval $[0; 1]$). Additionally, to speed up the simulation, the simulation steps can be performed in simplified game environment.

Even if it is better to perform selection before each simulation to avoid multiple evaluations of weak positions, some cases requires running multiple simulations in this phase (e.g. *leaf parallelization* described in section 2.3.2). This is the reason why the array of results is returned by *Playouts*(*node*) depicted by Algorithm 2.6 Number of simulations performed is denoted as $SIMULATIONS\_COUNT$, action selected by simulation strategy in position p as $SimulationAction(p)$ and terminal condition as $TerminalCondition(p)$.

2.1.4 Backpropagation

The last phase of the MCTS iteration is the backpropagation. The task of this phase is to propagate rewards provided by the simulation to all node's predecessors. Various backpropagation strategies may be used, however, the plain

average is crucial for good properties of UCT and performs well in practise. Hence the result set $rewards[\,]$ obtained by simulation is added to the node's value containing average of simulations performed in its subtree which leads to the update formulae for node p used in Algorithm 2.7.

Algorithm 2.7: *AddRewards*$(node, rewards[\,])$

```
/* adds backpropagated simulation results to node's value
   (average of al simulations) and adjust visit count
   appropriately                                          */
```

Data: $node$... a node of MCTS tree
$rewards[\,]$... array of simulation rewards
Result: Simulation results $rewards[\,]$ are added to $node$'s value.

1 $rlen \leftarrow Length(rewards[\,])$
2 $rsum \leftarrow \sum_{i=1}^{rlen} rewards[i]$
3 $v_{node} \leftarrow \frac{v_{node} n_{node} + rsum}{n_{node} + rlen}$
4 $n_{node} \leftarrow n_{node} + rlen$

2.1.5 Game Steps in the Tree

In this section, we describe simple technique which we use. It provides repeated usage of certain parts of MCTS tree in multiple game steps.

Algorithm 2.8: *MCTSGame*()

```
/* complete game play with MCTS player                    */
1  game ← new Game
2  mctree ← new McTree(game)// Empty tree rooted in new game
   position
3  P ← Player(this)// our player
4  while game is not over do
5      actions ← new ActionArray // empty map
6      foreach opponent on turn O do
7          actions[O] ← GetAction(O) // let opponent calculate turn
8      best ← MCTSLoop(mctree) // perform MCTS
9      actions[P] ← Action(best)
10     GameStep(game, actions)
       // Replace mctree with a subtree defined by actions
11     foreach player ∈ Keys(actions) ordered as in mctree do
12         Root(mctree) ← Child(Root(mctree), actions[player])
```

After the computation of the MCTS tree, the best action according to visit counts of root's children is returned and the underlying game object is updated by our action and actions of opponents. After that, computation of new MCTS tree begins. Since we work with fully observable environment, we know the action performed by opponents. In a tree built in previous game round we have

a node corresponding with the game object after the update so we don't need to build the very new tree from a scratch and instead of that, we can use a subtree corresponding with updated game object.

Since UCT selection leads to most promising nodes, it is highly probable that actions played by opponents will lead to a subtree with high visit count (our action leads to a subtree with highest visit count) and so we can use simulations of the subtree with a benefit as shown by Algorithm 2.8 which depicts MCTS together with game objects and opponents.

2.2 Convergence of UCT for Two-Player Games

The UCT algorithm (as a variant of MCTS) has been suggested as converging to optimal solution for the single-player games [9]; the counter-example of the convergence for two-player games with simultaneous moves has been proposed in [12]. In order to make the convergence to the optimal strategy possible in general, we consider turn-based games (weak turn-based games with teams eventually) for the building of a MCTS tree using UCT selection and for purposes of playing simultaneous games, we perform expansion of simultaneous nodes described in Section 1.1.1.

2.3 Parallel Monte-Carlo Tree Search

Our objective in this thesis is a design of algorithms for distributed MCTS computation among cooperative autonomous agents. Closest to such algorithms are works on parallelization of MCTS which are strong inspiration for algorithms proposed in Chapter 3.

Several publications dealing with parallelization of Monte-Carlo Tree Search have been published recently [3, 4, 15]. Two kinds of the parallelization are discussed in these papers - multi-core parallelization and cluster parallelization.

Multi-core parallelization refers to computation on a symmetric multiprocessor computer (SMP). In this case memory is shared among all processor cores and and can be accessed with same (generally low) latency by any core and access to this memory is the same and low. Access synchronization has to be managed using a mutex mechanism. Two techniques for parallelization with no need for locking (*leaf parallelization*, *root parallelization*, *simulation results passing*), and two variants of parallelization with locking (*tree parallelization*) have been proposed so far.

Cluster parallelization is a more generalized approach to parallelization. Memory is not shared between parallel processes, and so the inter-process communication is realized by *message passing*. Latencies in inter-process communication have to be taken in consideration but on the other hand, no memory locking is necessary. Algorithms mentioned in previous paragrapth as not using memory locking are suitable also for cluster parallelization.

For the simplicity and due to fact that all algorithms can be used in multi-core environment, we use multi-core terminology in descriptions of algorithms, so parallel computational flows will be referred as *threads* and computational nodes as *cores*.

Strength and speed measures for comparing of algorithms used in this section is described in following section.

2.3.1 Comparison Measures for Parallel MCTS Algorithms

We will briefly describe measures used in comparisons of parallel MCTS algorithms described in following sections. Strength-speedup measure is taken from [4]. Other measures described here are inspired by the same article.

Strength of a player choosing actions for playing a game G against opponents O (for case of multi-player game) is an expected value of player's score at the end of the game. For an experimental evaluation of the strength, a finite set of games is played and an average of scores obtained is used together with 95% confidence interval. Our players are guided by MCTS-based algorithms. Their strengths are dependent on time T provided to the algorithms for computation. In case of parallel MCTS algorithms, each thread has time T for computation. Since parallel algorithms run in multiple threads, each having same amount of time as plain MCTS, its strength should be greater than in case of plain MCTS and should grow with number n of cores involved. For more ilustrative comparison of parallel algorithms, *strength-speedup* measure will be used. Let us consider a parallel algorithm having time T_{par} for its running and its strength S. Plain MCTS needs time T_{pl} to achieve same strength S. Then strength-speedup is equal to ratio of $\frac{T_{pl}}{T_{par}}$. In other words, strength-speedup is the increase of time needed to achieve the same strength by plain MCTS. For better comparison of parallel algorithms running on various numbers of threads, the third measure will be used which is a ratio of strength-speedup to the number of cores available for parallel algorithm. The measure is called *strength-efficiency*. This ratio can be also seen as ratio between resources needed by plain MCTS (only T) and parallel MCTS (T multiplied by number of cores).

Another measure appliable to MCTS is *Simulations-per-Second*. It simply says how many simulations per second are being performed. Analogically to strength measures, *Simulations-per-Second speedup* and *Simulations-per-Second efficiency* are defined as ratio of Simulations-per-Second of plain MCTS and parallel MCTS and same ratio divided by number of cores.

2.3.2 Leaf Parallelization

Leaf parallelization is a simple algorithm originally introduced in [3] as *at-the-leaves parallelization*. Parallelization is used only during simulation phase where multiple simulations are executed.

Master thread traverses the root to the leaf according to selection strategy and expands selected leaf at first. Then for each available processor core, one simulation is performed from position defined by leaf. Master thread then waits until all simulations are finished and backpropagates their results up through tree. Algorithm 2.9 depicts simulation phase of leaf parallelization.

This approach is very easy to implement since no complicated synchronization is used which is its main advantage. However, several disadvantages have to be mentioned. First, cores are not fully loaded for two obvious reasons. The algorithm is single-threaded during selection, expansion and backpropagation phas-

Algorithm 2.9: *LeafParallelizationPlayouts*(*node*)

```
   /* simulation phase of leaf parallelization                    */
   Data: node... a node of MCTS tree
   Result: Array of simulation resulults is returned
 1 tree ← new McTree // Initialize empty tree
 2 root ← Root(tree)
 3 foreach available processor core C do
 4 |   thread_count ← thread_count + 1
 5 |   threads[thread_count] ← new PlayoutThread
 6 |   Run(thread)
 7 while AnyThreadRunning(threads) do wait for simulations
 8 |   Wait()
 9 for i ← 1 to Length(threads[ ] do
10 |   rewards[i] ← Reward(threads[i])
11 return rewards[ ]
```

es and so slave threads are sleeping. In addition, running times of simulation threads differ due to their high unpredictability and threads which already finished simulation have to wait for the last one. Second disadvantage is that leaf parallelization forces multiple simulations per position. It leads to better position evaluation but slows the speed of tree building. Last disadvantage is that in comparison with plain MCTS which performs selection after each simulation, the leaf parallelization has to perform full set of simulations per each node and so some of unnecessary simulations are uselessly performed because first few of them can serve as good evidence for considering the position as a bad candidate for further exploration. This can be worked around if simulation threads are terminated once such a consideration is made according to already finished simulations.

Leaf parallelization is suitable for using in both multi-core and cluster environment.

2.3.3 Root Parallelization

Another approach first proposed in [3] is *root parallelization* being originally named as *single-run parallelization*. In root parallelization, an independent MCTS instance is executed for each core with different random-seeds so the trees built by individual instances differ. Finally, roots with their children from all instances are collected in master thread and merged. The result is set of nodes containing information from all MCTS instances which are used for selecting the best action to perform. Main advantages are simple implementation and a minimal amount of communication so root parallelization can be successfully used both in cluster and multi-core environment.

The algorithm is ilustrated by Algorithms 2.10 and 2.11 (1 master together with several slaves are run). In the algorithm, we use procedure *BuildNextActionTreeCut* for extraction of tree root. Description of the procedure and related pseudocode can be found in Section 3.3.4.

There is a disadvantage resulting from missing communication during running

Algorithm 2.10: *RootParallelizationMaster*(*random_seed*)

```
/* master thread of root parallelization                    */
```
Data: *random_seed*... a random seed distincts from seeds of other threads
Result: Best action calculated by root parallelization alrogithm

```
1  tree ← new McTree // Initialize empty tree
2  MCTSLoop(tree) // Perform MCTS iterations
3  roots ← new RootSet // Empty set
4  roots ← roots ∪ BuildNextActionTreeCut(tree) // Add own root
5  foreach Team-mate P do // Receive roots
6      received_root ← Receive(P)// Wait for a root from P
7      roots ← roots ∪ received_root
   // Merge roots
8  merged ← new McTree
9  foreach root ∈ roots do
10     foreach leaf ∈ Leaves(root) do
11         node ← LookupNode(merged, Path(leaf)) // new node is
           created if not found
12         Backpropagate(merged, node, RewardsOf(leaf) // leaf is
           merged into the tree
13 return  arg max_{n∈Children(Root(merged))} c_n // Return most visited child
```

Algorithm 2.11: *RootParallelizationSlave*(*random_seed*, *master_thread*)

```
/* slave threads of root parallelization                    */
```
Data: *random_seed*... a random seed distincts from seeds of other threads
master_thread... master thread
Result: Nothing. Best action is returned by master thread.

```
1  tree ← new McTree // Initialize empty tree
2  MCTSLoop(tree) // Perform MCTS iterations
3  root ← BuildNextActionTreeCut(tree) // Extract root
4  Send(master_thread, Message(root))
```

of MCTS instances. Because each MCTS tree is built separately each instance have to perform similar exploration or in other words same set of positions have to be evaluated before interesting positions can be inspected. Due to this disadvantage, plain MCTS, having proportionally more time for running, would still probably perform better because no duplicit exploration have to be done and such a saved time can be used for deeper exploitation. However, according to comparisons resumed in Section 2.3.6, the root parallelization performs, at least for the game of Go, approximately equally to plain MCTS having adequately more time for computations (time used by root parallelization algorithm times number of processors used).

2.3.4 Tree Parallelization

The reasons why leaf parallelization wastes too much time leaving cores in sleep are removed by *tree parallelization*. It is done by effective tree locking so individual threads can access and modify the tree on their own and so particular simulations follow independent selections and need not to wait for the end of other simulations. The tree can be locked with *global mutex* or with *local mutexes* placed in all nodes. The global mutex is a simplier method saving costs associated with locking but the maximum speedup is significantly reduced because the average percentage of time spent in the tree (denoted as x for now) is usually relatively high ($25\% - 50\%$) and so it is not possible to reach higher speedup than $100/x$. This is because if each of n threads is accessing the tree exclusively and no one have to sleep, percentage of time spent in tree is equal to nx and so $n \leq \lfloor \frac{100}{x} \rfloor$. Due to this limitation, local mutexes are necessary for systems with greater number of cores. To beat the fact of high costs of locking and unlocking, fast-access mutexes should be used (i.e. spinlocks).

One more disadvantage of root parallelization should be handled. Tree parallelization may, at one time, run multiple evaluations of a single node in situation when firstly finished simulations would mark a node with bad value and so other evaluations are not necessary. To beat such behaviour, *virtual loss* is added to the value of nodes traversed by selection of each thread and then removed during backpropagation phase. Virtual loss suppresses multiple simulations on single node in case of weak actual value but for promising positions, more than one simulation can be performed at a time.

2.3.5 Simulation Results Passing

Simulation results passing can be viewed as a trial of adaptation of *tree parallelization* to cluster environment. These two algorithms share the property of high amount of communication which is naturally a bigger problem in the distributed environment.

Unlike tree parallelization, the simulation results passing is building search a tree separately in each thread and synchronizes these trees by broadcasting messages containing identification of expanded node and resulting value of performed simulation. This value is then backpropagated in trees of receiving threads. Master thread finally returns best action according to its tree. Simulation results passing is depicted by Algorithm 2.12.

Algorithm 2.12: *SimulationResultsPassingLoop*(*tree*)

```
   /* a thread of simulation results passing algorithm          */
   Data: tree ... MCTS tree
   Result: tree is enlarged by calculated and received simulations
 1 while EnoughTime() do // Main MCTS loop
 2     while receiving queue Q not empty do // Receive messages
 3         (path, rewards[]) ← Pop(Q)
 4         node ← LookupNode(tree, path) // new node is created if
           not found
 5         Backpropagate(tree, node, rewards)
 6     node ← Select(tree)
 7     node ← Expand(node)
 8     rewards[] ← Playouts(node)
 9     Backpropagate(tree, node, rewards[])
10     foreach Team-mate P do // Send messages
11         SendSimulationResult(P, Message(Path(node), rewards[]))
12 return  arg max_{n∈Children(root)}  c_n // Return most visited child
```

This algorithm relies on high throughput of the cluster environment. Requirements on the throughput grows with number of parallel threads so algorithm is not suitable for massive parallelism. The algorithm is designed for usage in cluster environment but can be of course used also in multi-core environment.

2.3.6 Comparison and Conclusion

We have summarized known algorithms for parallelization of MCTS in both multi-core and cluster environment. Three of them (leaf parallelization, root parallelization and tree parallelization) have been experimentally evaluated in [4] in context of playing Go in multi-core environment. For this purpose the *strength speedup* measure has been used. It is equal to the ratio of time needed for plain MCTS to reach the same strength. Brief summary of results is covered by Table 2.3.6 and Figure 2.2. The winner of comparison is root parallelization performing even little better than single-threaded MCTS in case of 2 and 4 parallel threads. Second best performance showed tree parallelization using virtual loss. Other algorithms performed very poorly especially in case of higher number of threads.

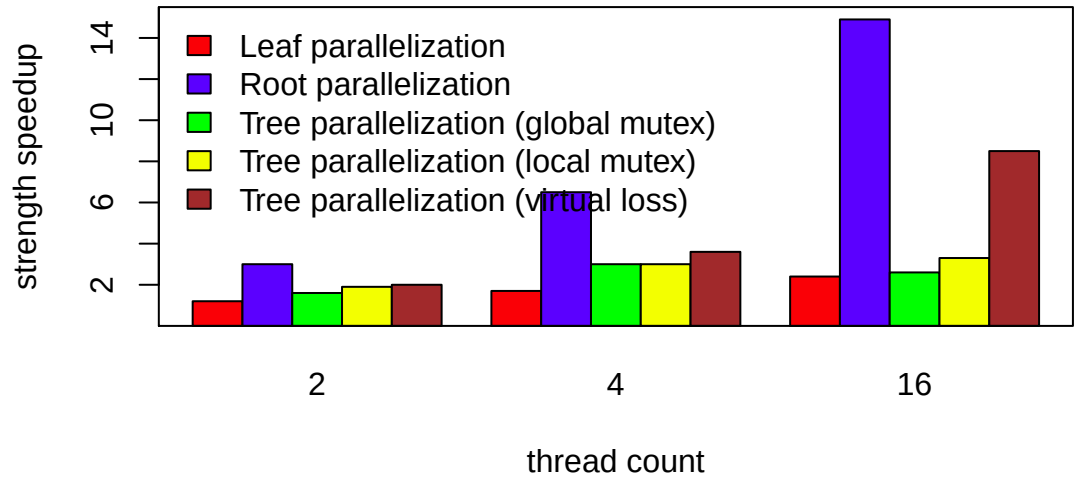


Figure 2.2: Parallel MCTS algorithms - comparison.
Strength-speedup of parallel MCTS algorithms experimentally evaluated on Ms Pac-Man game in [4]

Algorithm	2 threads	4 threads	16 threads
Leaf parallelization	1.2	1.7	2.4
Root parallelization	3.0	6.5	14.9
Tree parallelization (global mutex)	1.6	3.0	2.6
Tree parallelization (local mutex)	1.9	3.0	3.3
Tree parallelization (virtual loss)	2.0	3.6	8.5

Table 2.1: Parallel MCTS algorithms - comparison.
Strength-speedup of parallel MCTS algorithms experimentally evaluated on Ms Pac-Man game in [4]

3. Distributed MCTS Algorithms for Cooperating Teams

The purpose of this chapter is to design distributed coordination algorithms based on Monte-Carlo Tree Search. Presented algorithms will work with MCTS tree kept independently in each cooperative agent. The tree will contain full information about actions done by all cooperative agents and also agents from opponent teams. This approach is supposed to be appropriate for games with small number of players but with increasing number of players, the branching factor grows and keeping full MCTS tree turns to be ineffective. The tree used in distributed algorithms along with general description of common parts of the algorithms are described in Section 3.3.1. Before we will propose the algorithms' description, it is necessary to introduce additional notations for this chapter in Section 3.1 and briefly discuss measures used for comparison of distributed algorithms in Section 3.2.

3.1 Useful Notations

In our algorithms, we will describe, among others, an exchange of various parts of the MCTS trees of the team-mates and it will be useful to give a formal definition of one of them, the so-called *tree cut*.

Definition (Tree cut)**.** *Let us have a tree T with a set of nodes N.* ***Tree cut*** *C_T is a subset of N such that there is no path between any ascendant a of a node from C_T and any descendant d of a node from C_T and no node from C_T is an ascendant of any other node from C_T. In other words, nodes from a tree cut separates a subtree composed of nodes from C_T and their ascendants from the descendants of nodes from C_T. An example of a tree cut is depicted by Figure 3.1.*

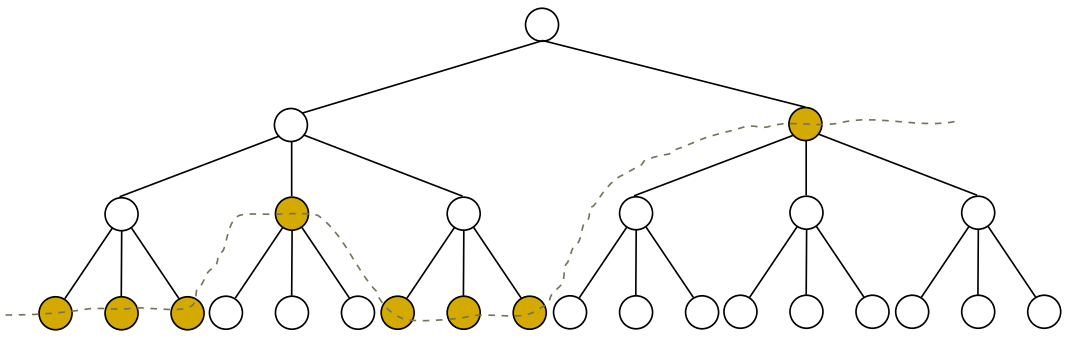


Figure 3.1: Tree Cut Example.
Yellow nodes form a tree cut which separates ascendants and descendants of the nodes of the cut. Each path from root to leaf contains exactly one node from the cut.

3.2 Comparison Measures

Measures suitable for plain Monte-Carlo Tree Search and parallel Monte-Carlo Tree Search have been already described in Section 2.3.1. We will show that these measures are also suitable for comparison of distributed MCTS algorithms for games with team of cooperating agents. In addition, we will compare the amount of communication needed by the algorithms and the robustness of the algorithms against communication failures. For such comparisons, we will evaluate strength measures depending on the amount of communication and its robustness. The amount of communication will be simply the total length of messages exchanged. Some environments (e.g. radio transmissions or Ethernet over coax) provide the possibility of broadcasting messages for the cost of passing single message whereas others don't; so, the cost of a broadcasted message equals to the cost of separate messages to all receivers. We will distinguish between these environments since some of the algorithms advantage from cheap message broadcasting.

We have described two classes of measures for parallel MCTS, *strength-* and *simulations-per second*-based measures. The former one works with score obtained at the end of a game and the latter one counts number of MCTS iterations handled per time unit. Details of these measures are discussed in Section 2.3.1. Obviously we can use simulations-per-second measures; distribution is not an obstacle, simulations are still performed and these measures show the computational costs of distribution of computation. Similar is the case of strength measueres. The only difference is that agents decide the actions independently. From outer point of view, the joint action is the same as if returned by plain or parallel MCTS, therefore these measures can be used. Strength measures say how strongly guided by an algorithm a team is.

Similar is the case of strength measures where the only difference is that agents decide the actions independently but from outer point of view the joint action is the same as if plain or parallel MCTS calculate action. Strength measures say how strong is team guided by an algorithm.

3.3 Proposed Algorithms

In the following subsections, proposed distributed MCTS algorithms for games with a team of cooperative agents are described together with a discussion on their general properties. Since we design algorithms for distributed MCTS-planning, the algorithms drive particular agents, and so all algorithms have the work *agents* in their name.

3.3.1 Backbone of the Distributed Algorithms

MCTS-based agents will iteratively build MCTS tree exactly as a plain MCTS algorithm does. In addition, before a certain set of MCTS iterations, the agent receives messages from its team-mates and performs appropriate actions. Similarly after the set of MCTS iterations, the agent broadcasts messages to the team-mates. The communicaton between agents is the point differencing particular distributed algorithms. Algorithms can also differ in other details such as

Algorithm 3.1: *DistributedMCTSLoop*(*tree*)

```
/* general distributed MCTS loop                                    */
```
Data: *tree*... MCTS tree

Result: *tree* is enlarged by newly expanded nodes and results of playouts performed are added. Additional actions are performed during message receiving. Node representing the best evaluated position reachable by one action is returned

```
1  i ← 1
2  while EnoughTime() do // Main MCTS loop
3      if i = 0 then
4          ReceiveMessages()
5      node ← Select(tree) // Phase 1:  Selection
6      node ← Expand(node) // Phase 2:  Expansion
7      rewards[] ← Playouts(node) // Phase 3:  Simulation
8      Backpropagate(tree, node, rewards[]) // Ph 4:  Backpropagation
9      if i = 0 then
10         SendMessages()
11     i ← (i + 1) mod N_it
12 return  arg max_{n∈Children(root)}  c_n // Return most visited child
```

random seed intialization. Following subsections describe particular distributed algorithms.

3.3.2 Independent Agents

We will first describe the algorithm which is supposed to be the weakest one since no communication between agents is performed. It serves as a lower bound on performance of the algorithms. The only trick used for the algorithm is setting of the random seed for the simulation. Because there is no communication between agents, it can easily happen that agents discover different local minima. To avoid such a behaviour, the random seeds for all agents are set to the same value during agent initialization. And because the agents have equal time to compute the answer, the size of trees at the end of computation will be similar (this may not be true for agents with inequal computational strength). If we suppose that all agents calculate exactly the same number of iterations, time proposed to the agents is t and the number of agents is N, then the strength of this algorithm is supposed to be equal to the plain MCTS algorithm with computation time equal to $\frac{t}{N}$. This is obvious since the size of the tree of such a plain MCTS algorithm is equal to trees computed by the independent agents. The only difference is that independent agents have to calculate their trees N-times since no communication is allowed.

Advantage of the algorithm is that no communication is needed so it is not affected by unreliability of the communication network.

3.3.3 Joint-Action Exchanging Agents

Simple way how to synchronize the movement of the agents is exchanging of the messages containing the information about the actual best joint-action (tuple of actions of all cooperating agents). The joint-action exchanging agents uses such a synchronization. When it is a time to decide which action should be played, simple voting is used. Each agent summarizes most recent joint-actions received from all team-mates and uses the most frequent one. For situations when multiple joint-actions with same frequency occurs, ordering on agents is defined action played by agent higher in the ordering is played. To improve advantages of the algorithms, different random seeds are used for initialization of the agents and so possibly different local minima are considered during voting phase.

Such a communication is very low-cost and as such does not require wide network channel. Since the actual best joint-action does not change after very often, we can send the message only when it is changed to further reduce costs on channel and computations related to networking. For cases when network is too slow to handle all joint-action messages, all unsent messages (except one which is being already transmitted) are flushed and the recent one is enqueued.

3.3.4 Root Exchanging Agents

Joint-action exchanging agents are voting for the best joint-action but the tree from which the joint-action is calculated contains only simulations done by the winning agent. Root exchanging agents increase the strength of chosen action by merging of strengths of all possible joint-actions calculated by all agents and so resulting joint-action is based on results from trees of all team-mates. This strategy is based on root parallelization algorithm described in Section 2.3.3.

Since the results from trees are merged, we want to build different trees in each action, so random seeds are initialized to different values for each agent. The term "root" in this case means set of actions $Actions(r)$ playable from the root r of the MCTS tree together with their visit counts $n_i, i \in Children(r)$. After the root merging, all agents have an almost identical set of joint-action – visit-count pairs (some differences may occur since not the most recent roots may be received at time of merging). Joint-action with highest summed visit count is selected (ordering on joint-actions is used when multiple joint-actions have same visit count).

In case of the single-player game, the previous description of algorithm works well but there is a little difficulty if any opponent is considered because set $Children(r)$ may not be a set of actions of our team. We can simply wait for opponents' turn and exchange our messages during the period when our team is on turn but this approach has two weaknesses. The first one is that this periods may be very short what may, along with unreliable networking channel, lead to no exchanges performed at all. The second weakness is that such a period for message exchanging may not be present at all in case of turn-based game converted from simultaneous game if current state was created by splitting the simultaneous turn (see Section 1.1.1). We resolve this situation by extending definition of the root for turn-based games with multiple teams.

For multi-player games, the algorithm will exchange *next-action tree cut* of a player P being on turn. The next-action tree cut is defined as follows:

Definition (Next-action tree cut)**.** *Let us have a tree cut C_1 composed of nodes in which player P is on turn and which are the first nodes occuring on the path from tree root to its leaves. For all ascendants of nodes from C_1, any player different from P is on turn. Then, we will call a set of children of nodes from C_1, which also composes a tree cut, a* ***next-action tree cut*** *for a player P.*

Construction of next-action tree cut is described by Algorithm 3.2.

Algorithm 3.2: *BuildNextActionTreeCut(tree, player)*

```
/* construction of next-action tree cut                    */
Data: tree ... MCTS tree
player ... player constructing
Result: Algorithm returns the Next-Action Tree Cut of tree for a player P
C_curr ← {Root(tree)} // cut growing from root
C_next_action ← ∅ // final next action cut
while C_curr ≠ ∅ do
    foreach node ∈ C_curr do
        C_curr ← C_curr \ {node}
        if OnTurn(node) = P then
            C_next_action ← C_next_action ∪ Children(node)
        else
            C_curr ← C_curr ∪ Children(node)
return C_next_action
```

Next-Action Tree Cut is being sent as an array containing pairs of visit counts of nodes and corresponding paths to the nodes represented as a list of actions leading to the nodes. For single-player game case, this approach leads to the exchange of set of joint-action – visit-count pairs of root's children, as described earlier.

There is another difficulty with exchanging of Next-Action Tree Cut which is that the size of the cut may grow excessively with growing number of players playing simultaneously so this algorithm is not suitable for such games. This fact should be kept in mind when appropriate algorithm for a particular game is being chosen. If a reliable communication between agents is guaranteed, it is possible to safely exchange the cuts only in turns in which the player is on turn (considering simultaneous nature of the game). In case of unreliable communication, on the other hand, information contained in cuts from turns when the player is not on turn can be used with benefit if no communication can be done during the player's turn.

3.3.5 Simulation Results Exchanging Agents

Simulation Results Exchanging Agents directly follow the idea of Simulation Results Passing algorithm described in 2.3.5. Since Simulation Results Passing algorithm is designed for cluster environment, we have to deal with two main differences in distributed coordination environment which are agents' independent reasoning and limited and potentially unreliable communication.

The former difference is simply managed by abandoning the idea of master thread and letting all agents to play a calculated action. The latter one, limited communication, is a more serious issue. We cannot transmit all simulations calculated as far as the channel is too narrow and so we have to choose which of simulation results will be transmitted. More recent simulation results will be prioritized because such results carry information about the interesting path in the MCTS tree (in comparison with less recent ones which also carry the interesting path but not so long).

Mechanism of prioritizing of more recent simulation results is realized by message buffer where, in opposite of common message sending, new messages with simulation results are inserted at the beginning of the buffer. Once the buffer is full, the least recent messages are thrown away.

3.3.6 Tree-Cut Exchanging Agents

Root exchanging agents share the information from their MCTS trees by sending messages containing as little information as possible what spares the capacity of communication channels but trees contain only calculations done by a particular agent. In opposite, simulation results exchanging agents share full information about all simulations, and so each particular MCTS tree contains simulations from all agents and if the channel is reliable and wide enough, it contains full information calculated together by all agents and all trees are the same. This approach, in comparison with root exchanging agents, brings some advantages and also some disadvantages.

The main advantage is higher robustness against communication failures longer than time available for computation of one game step. Since simulation results are applied into team-mates' trees, they don't vanish after a game step and not even after several steps. Thanks to UCT mechanism, most of simulations are performed in a subtree which is finally chosen and all these simulations can be used during calculations of the next action. This is beneficial also if no communication failure occurs.

The most serious disadvantage of simulation results passing agents is that it requires wide communication channel and if it is not available, only a proportional amount of simulations is exchanged. Second disadvantage is that each received simulation has to be backpropagated through the tree which requires additional time for computation.

Tree-cut exchanging agents algorithm tries to find a compromise between root exchanging and simulation results passing. It keeps the advantage of received simulation results stored directly in MCTS trees but aggregates them adequately to fit the width of the communication channel. As a side effect of this approach, less calculation with received simulation results is required.

Root exchanging agents algorithm constructs and sends the next-action tree cut. Similarly the tree-cut exchanging agents algorithm sends a *visit count tree cut* having following properties: the entire tree cut can be transmitted multiple times during one game step and visit counts of nodes of the tree cut are similar. Let us denote C_{cut} the number of tree cuts transmitted per a game step. Then the more cuts transmitted per a game step, the more robust our algorithm is and more recent simulations are transmitted - last cut has to be sent at most

$t = \frac{gamesteplength}{C_{cut}}$ seconds before the end of the game step; thus, the algorithm regrets transmission of simulations performed during last t seconds (at least $\frac{1}{C_{cut}}\%$ of simulations is not transmitted).

Similarly as in case of root exchanging agents, tree-cut exchanging agents send a tree cut once a sending queue is almost empty (all messages would be sent during next simulation). At that moment, a tree cut is constructed and pushed at the end of the queue.

We define the visit count tree cut by introducing pseudocode for its construction, Algorithm 3.3. Algorithm starts with a tree cut containing only the root of the MCTS tree and iteratively selects the node with highest visit count and replaces the node with its children. The algorithm ends when desired byte size of the tree cut is reached.

Algorithm 3.3: $BuildVisitCountTreeCut(tree, byte\_size)$

```
/* construction of visit count tree cut                    */
Data: tree ... MCTStree
byte_size ... desiredsizeofthetreecutinbytes
Result: Visit count tree cut of a given tree possible to transmit using less
        than byte_size bytes
1  cut ← {Root(tree)} // cut growing from root
2  curr_byte_size ← ByteSize(Root(tree))
3  while true do
4      most_visited ← arg max_{i∈cut} n_i
5      children ← Children(most_visited)
6      curr_byte_size ← curr_byte_size + ByteSize(children)
7      if curr_byte_size > byte_size then
8          break
9      cut ← cut \ {most_visited} ∪ children
10 return cut
```

Each agent has to keep information about last received tree cut. Once a new tree cut is received, an algorithm removes previous one from the tree and applies the new one.

3.4 Conclusion

In this chapter, we have proposed a total of five synchronization algorithms with various communication requirements.

Independent agents work with no communication at all, being synchronized only by keeping the same random seed for building trees. Such agents cannot exceed strength speedup value of 1.0 but can be seen as a very simple and unpretentious way of coordination without sharing of calculated results.

Joint-Action exchanging agents are based on a very simple idea of coordination based only on exchanging the best currently calculated action that requires only a little amount of communication. In addition, the ghosts vote for the action being played. Thanks to voting mechanism, the algorithm, in addition to simple

coordination, chooses among local minima and the best one is chosen. So strength speedup may reach a value of strength speedup over 1.0 with almost no regret on number of simulations calculated per second (joint-action exchanging is very trivial operation taking very little time compared to one simulation).

Root exchanging agents is an algorithm strongly inspired by *root parallelization* of MCTS. Each agent is receiving a list of next joint-actions together visit counts of corresponding nodes. Just before the end of turn, the visit counts of individual joint actions are summed (also with agent's visit counts) and the best joint-action is played. This algorithm is similar to joint-action exchanging agents in the timing of usage of the information. Joint-action exchanging agents vote the best action at the end of the turn same as root exchanging agents do. The difference is in complexity of the information transmitted. More complex information transmission requires more communication but the requirements on the communication channel are still quite low for this algorithm. Root exchanging agents are supposed to reach high value of strength speedup since previous experiments with root parallelization in [4] shows such a results.

First algorithm projecting received messages immediately after their transmission is *simulation results passing*. The algorithm is exchanging information about each simulation backpropagating it to all trees. Thanks to it, the trees grow faster and deeper exploitation can be performed. On the other hand, multiple backpropagation of each simulation brings some regret on the amount of simulations calculated per second. This approach also requires high amount of communication.

Tree-cut exchanging agents is a trial of compromise between previous two algorithms. Agents are exchanging the results of simulations in aggregated form of *visit count tree cut*, propagating them to the tree. Aggregation reduces the amount of communication and regret on simulation calculation in comparison with simulation results exchanging agents. On the other hand, received simulations are usually backpropagated multiple times so the simulation calculation regret depends on the parametrization of the algorithm.

In the next chapter, the algorithms are applied to a specific domain of Ms Pac-Man and compared and evaluated according to the obtained results.

4. Evaluation of Distributed MCTS Algorithms

4.1 Ms Pac-Man vs Ghosts Framework

For the purposes of evaluation of algorithms proposed in Chapter 3, we have chosen the Ms Pac-Man vs Ghosts Framework [11] which is an easy-to-use framework allowing implementation of players for well-known old game Pac-Man in Java. Here we will extract basics of the game rules used in the framework and afterwards we will describe modifications of the rules we have done and reasons for them.

4.1.1 Game Rules

Ms Pac-Man is a game played in a maze in which two sides compete; the Pac-Man and four ghosts. There are pills everywhere in the maze and Pac-Man's purpose is to gather all the pills, each for 10 points. Once all the pills are eaten, the game continues in a next maze. To complicate the life of Pac-Man, ghosts are moving around the maze pursuing the Pac-Man and trying to minimize its score. If the Pac-Man is caught, it loses one life and if it has any life remaining, it starts again from its starting position. Pills remain eaten after the life loss. Ghosts appear in so-called lair at the beginning of each round and after Pac-Man's life loss from where they start after several times (different for each ghost). Beside regular pills, each maze contains four power pills which are awarded with 50 points and when eaten by Pac-Man, ghosts become edible and twice slower for certain period of time. Pac-Man can eat ghosts during this period for reward of 200 points for first eaten ghost and 400, 800 and 1600 points for other ghosts. Eaten ghost starts in lair again. Exact rules of the Ms Pac-Man vs Ghosts game can be found on the project webpage.

The game was additionally modified for purposes of testing of our algorithms. Our goal isn't to create a strong player for Ms Pac-Man vs Ghosts competition but we want to test proposed coordination algorithms and for such purposes, simplier testing environment giving clearer results suits better. The biggest change of rules is removing power pills from the mazes together with entire edible ghosts mechanism. A reason for this decision is lowering the variance of results. When Pac-Man is pursuing edible ghosts, there is big difference in score if it eats different number of ghosts. For the same reason three more modifications have been done. Pac-Man has only one life, game ends after the first maze is cleared and random ghosts' reversal is suppressed. Random ghosts' reversal is a rule of original Ms Pac-Man game bringing stochasticity to the game. If the rule is on, there is a 0.15% probability in each tick of a game that all ghosts accidentally change their direction. Original game has a limit on maximum length of each round after which score for remaining pills is added to Pac-Man's score and game continues to the next level. This limit is set to 2000 what we keep as a rule, so our simplified game ends after at most 2000 ticks. Limiting the number of rounds played together with limit on game length also lets us run more tests. An example of a situation

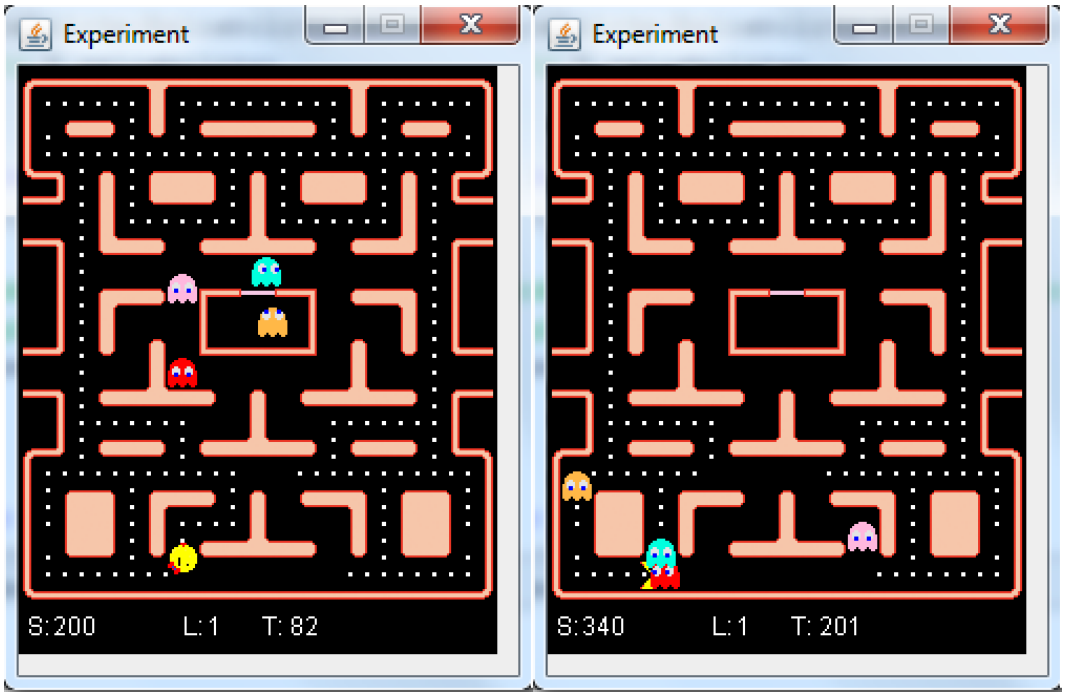


Figure 4.1: Simplified game of Ms Pac-Man.
Left picture shows the game few moments after it began. Three ghosts started to hunt the Pac-Man while the orange one is still waiting in lair. Right picture, on the other hand, shows the game just before the end. Pac-Man does not have any escape path and is being caught with in a while with no life remaining.

from the game without power pills is depicted by Figure 4.1.

Pac-Man has the same movement speed as non-edible ghosts. Paths inside the maze are divided into small segments, four between two neighbouring pills. Pac-Man has to play an action after each segment, turning back in the middle of path or at the crossroad is allowed and so Pac-Man has always at least two action to choose between. Contrarily, ghosts' movement is restricted. Additional rule on ghosts' movement is that ghosts cannot turn back in the middle of a path nor at a crossroad. That means that once a ghost chooses one way from a crossroad, it has to continue to the end of the path. Despite of that, ghosts have to play an action on each segment even though there is only one action to choose from.

So Pac-Man and ghosts play their actions at each segment of the maze but they, of course, have to play an action in a limited time. Ms Pac-Man vs Ghosts framework provides by default 40 ms to play what corresponds with the natural game timing when a human player controls Pac-Man.

4.1.2 Framework Details

In our thesis, we use Ms Ghost vs Pacman framework version 6.2. The framework is written in Java what simplifies the development of controllers of the players. In original game, a player was able to control only Pac-Man and ghosts were controlled by simple AI. In the framework, it is also possible to develop a controller for the ghosts. The only work a user is supposed to do is to implement a controller

of Pac-Man or ghosts, methods for running a game or a set of games are prepared for usage.

A player controller is a class implementing an interface having only one method. In case of ghosts controller, method's header is

```
public EnumMap<GHOST,MOVE> getMove(Game game,long timeDue)
```

where `EnumMap<GHOST,MOVE>` is a joint-action of ghosts, `game` contains information of current game and `timeDue` contains time until which the method has to return an action. If an action is not returned on time, certain default action is chosen by framework itself.

4.1.3 Pac-Man Opponents and Simple Ghosts Controllers

To evaluate the strength of ghost controllers based on distributed MCTS algorithms, we need to choose an appropriate Pac-Man opponent to play against. Unfortunately, experiments with Pac-Man controllers included in Ms Pac-Man vs Ghosts framework showed that these controllers are too weak for evaluation. Thus, it was necessary to look for better Pac-Man controllers.

Main purpose of the Ms Pacman vs Ghosts project is organizing competitions between existing controllers. Competitions are usually held on various conferences and meanwhile there is also a league in which all controllers committed into the website compete. Thanks to the league results we were able to contact authors of the league leading Pac-Man and obtained source coof ICEP_IDDFS [14] which is a controller based on iterative deepening depth-first search approach. We haven't received more details about the controller. However, experiments running against it have given promising results, and so the controller was chosen as an opponent to compare with.

4.2 Implementation Notes

In this section, we will discuss important implementation details of our controllers. As mentioned in the previous section, framework used for the experiments is written in Java and so controllers are also supposed to be written in this language. We provide additional information about software work in Attachment 2.

4.2.1 Tree Construction

Here we will talk about the way the concrete realization of MCTS tree for the game of Ms Pac-Man. The most direct approach to a construction is to keep a node of the tree for each game step and store, besides the value and the visit count, actions performed by Pac-Man and all ghosts. By exposing and resolving of problems of this solution, we have reached the construction used in our controllers.

First problem to be solved is that MCTS tree should not work with a simultaneous actions of multiple teams as discussed in Section 2.2. Ms Pac-Man satisfies the definition of simultaneous game with teams, since multiple ghosts together with Pac-Man may be on turn at the same time. Section 1.1.1 gives us a recipe to deal with simultaneous actions by convertion of a simultaneous game to a (weak) turn-based game by splitting simultaneous nodes. The conversion is not done

on the unterlying Ms Pac-Man game itself but only for purposes of expansion of nodes. When a simultaneous node is being expanded, instead of creating of a child node for each Pac-Man-ghosts joint-action, only children for Pac-Man actions are created and each of these children are immediately expanded with ghosts actions. By this approach, simultaneous nodes are splitted according to optimistic expansion since in such a tree the ghosts suppose to know the action of Pac-Man played in the simultaneous node. Our implementation, in addition, supports the pessimistic expansion where ghosts actions are expanded first in simultaneous nodes.

Second problem is quite high branching factor caused by Pac-Man which is able to change its direction at any time. To reduce the branching factor, we consider additional rules of Pac-Man's movement for purposes of node expansion. We allow Pac-Man to change its direction only at crossroads, in the neighbourhood of a segment with power pill not yet eaten and at segments in the middle of path when last segment allowing the direction change is exactly 6 segments far. When computing the MCTS tree for Pac-Man, this reduction does not bring any difficulty since the player controlled by the algorithm follows additional rules. But if we consider Pac-Man playing against the MCTS algorithm, it may play an action not corresponding with these rules what directly leads to desynchronization between game state and built tree. Next time any ghost reaches a crossroad and the desynchronization is detected, ghosts play an action according to the tree but then instead of using a subtree defined by the action for further computations, a new tree is built from scratch.

Finally, after expansion of simultaneous nodes and reduction of Pac-Man actions, the tree will contain nodes having only one child which are also removed; that adds a necessity to keep lengths of edges between nodes.

4.2.2 Pac-Man Playout

In our work, we decided to keep the simulation strategy simple, not leveraging much of domain-specific information. This decision is driven by experiments from early phases of development. During this experiments, both Pac-Man and ghosts players were, with a certain probability, led by some heuristic algorithm inspired by original legacy behaviour in case of ghost player and the StarterPacman example controller which is provided as a part of framework. With supplementary probability, players performed random actions. Because we haven't discovered any significant strength gain by tuning the probability of heuristic steps, we omitted heuristics at all.

The only additional knowledge put into the simulation strategy is that Pac-Man is not allowed to change its direction in the middle of a path during simulation. The point of this restriction is that once Pac-Man enters a path, it usually wants to continue to a next crossroad. In addition, such a restriction heightens expected size of area visited by Pac-Man because situations of Pac-Man idling in the middle of path are suppressed.

We also use maximum depth of simulations. Tuning of this parameter is described in Section 4.4.

Simulations have to be evaluated after they finish. At first, we decided to use following simple formula:

$$\frac{score\ reached}{maximum\ reachable\ score} \tag{4.1}$$

But after some experiments, we realized that the formula does not motivate ghosts enough to hunt Pac-Man so we decided to add a bonus if ghosts successfully catch Pac-Man:

$$(1-\alpha)\frac{score\ reached}{maximum\ reachable\ score} + \alpha \begin{cases} 1 & if pacman\ was\ eaten \\ 0 & otherwise \end{cases} \tag{4.2}$$

We call an α coefficient *death_weight* and tune it in Section 4.4.

4.2.3 Communication

Main method of a player controller (`getMove`) returns the joint-action of the ghosts but for purposes of distributed approach, each ghost is supposed to reason individually returning its action. To fulfil this, four subcontrollers, returning single actions are started in four separated threads and virtual bidirectional communication channels are created between each pair of subcontrollers.

Channels provide interfaces for both senders and receivers and work with real time. A channel consists of a queue of messages to be sent and a queue of received messages. Messages from the sending queue to the receiving queue are transmitted every time a method on a channel is called. The amount of messages transmitted corresponds with time elapsed since last channel event ending with nonempty sending queue.

4.3 Experiments Setup

All experiments were performed in virtualized environment of CentOS release 6.3 (Final) having assigned 12 CPUs and 16 GiB of memory. Underlying host server disposes of 2 Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz processors (6 cores/12 threads each) and total of 128 GiB of memory. Beside of our testing server, a few other virtual servers were running on the host server with a light consumption of resources but taking into account that we were leveraging at most 4 cores at a time, remaining cores could easily handle the traffic.

Java runtime used for experiments is `1.7.0_09-icedtea`.

Simple bash scripts were used for purposes of launching of individual games. Our Java application contains uniform entry point allowing setting of necessary parameters for running of an experiment. Each experiment were performed 100 times and average values were then used. Data gathered during experiments were then processed in R environment. Graphs contained in this work are also generated with R.

4.4 MCTS Tuning

Before we started any of our experiments, we wanted to be sure that our MCT algorithm has appropriate setting of parameters. We did three tests for approximate tuning of three parameters - C (UCT coefficient), simulation depth and

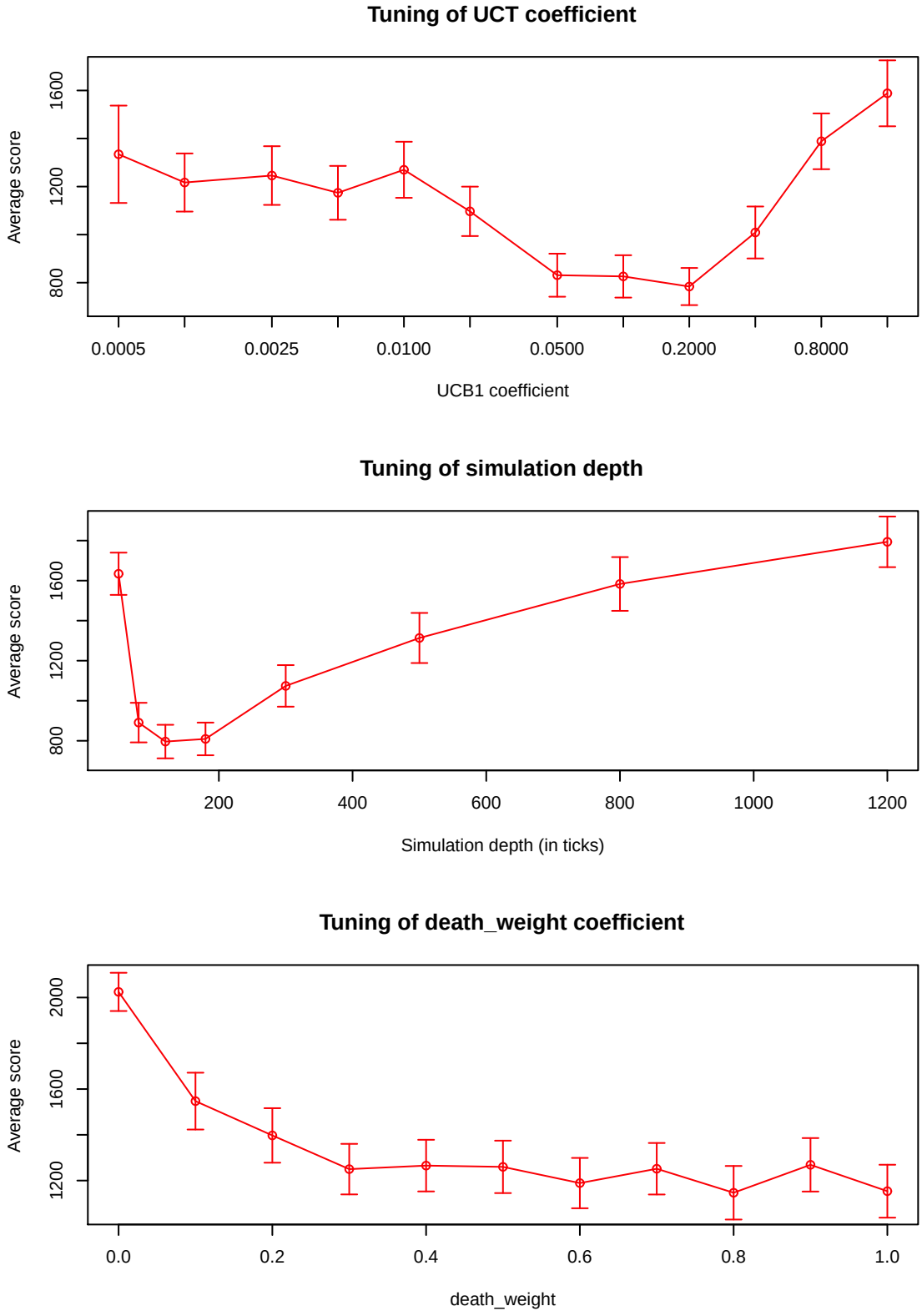


Figure 4.2: Tuning of MCTS parameters (bars show 95% confidence intervals). Top - tuning of UCT coefficient, middle - tuning of maximum simulation depth, bottom - death_weight tuning.

death_weight (parameter used in evaluation function, see Section 4.2.2). Graphical results of these tests are depicted by Figure 4.2. All tunings were performed with time ticks of 40 ms.

We first tuned C, with initial setting of simulation depth 200 and death weight 0.25. Test showed us significant improvements with C between 0.05 and 0.2 so we decided to use $C = 0.1$.

Secondly, we tuned maximum depth of simulations, with already tuned $C = 0.1$ and initial $death\_weight = 0.25$. According to the test, we set simulation depth to 120.

Finally, we tuned *death_weight* parameter used in evaluation formula with already tuned MCTS parameters C and simulation depth. According to the test, best values of the parameter are between 0.3 and 1.0. During the evaluation of the test, some experiments with $death\_weight = 0.25$ were performed and a score with this *death_weight* didn't differ much, we decided not to repeat the experiment with better setting of the parameter and run all tests with $death\_weight = 0.25$.

4.5 Comparison of the Algorithms

For distributed MCTS, three basic tests were performed. At first, strength of an algorithm depending on computational time (length of a single tick) with times from 10 ms to 200 ms, 100% reliable channel and channel speed fixed to a value of expected optimal communication requirements. Second test was strength in dependence on channel speed with fixed time set to 40 ms. Range of speeds used depends on expected requirements on the communication; therefore values vary between tests. When we talk about average strength-speedup of some algorithm, we mean the average of strength-speedups measures for all measured computational times (10 ms-200 ms, stepping by 10 ms). Channel speeds are scaled exponentially with values from 2^{lb} to 2^{ub} for some algorithm-dependent values $lb, ub \in \mathbb{R}$.

We recall that the aim of the ghosts is minimization of Pac-Man's score so the lower score indicates better performance of the ghosts.

Besides the final score, we measured various other characteristics, such as average number of simulations performed per second, average size of MCTS tree at moments requiring a decision or real ammount of communication. These characteristics simplified the analysis of behaviour of the algorithms.

4.5.1 Centralized Monte-Carlo Tree Search

In Section 3.2, we defined the strength-speedup measure which is used for comparison of distributed algorithms with plain (centralized) algorithm. So before we perform experiments with distributed algorithm, we run experiments with plain MCTS. During these tests, we also compare approaches to expansion of simultaneous nodes in Ms Pac-Man game. Average number of simulations calculated per second by plain MCTS algorithm was 11002. This number will be also used for comparison of algorithms.

Figure 4.3 depicts results of plain MCTS running on single processor. We tested both optimistic and pessimistic expansion of simultaneous nodes. Results

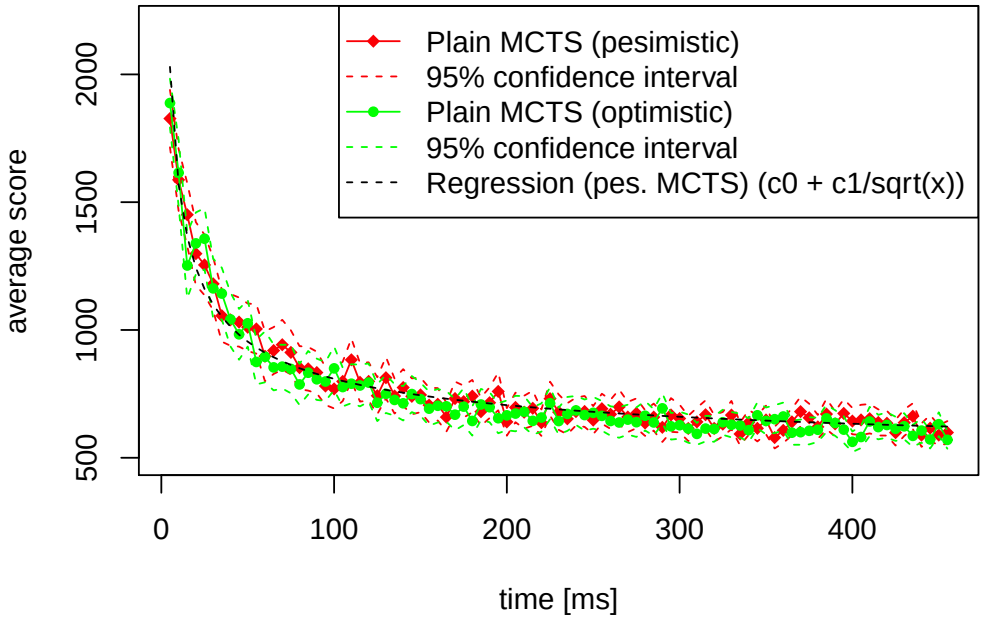


Figure 4.3: Plain MCTS strength.

Solid red and green lines show strength of centralized MCTS with pessimistic and optimistic simultaneous nodes expansion, dashed lines of corresponding colours are 95% confidence intervals. Black dashed line is a fit of data of pessimistic MCTS (red lines) to $c_0 + \frac{c_1}{\sqrt{(t)}}$. This fit is used for calculation of strength-speedup of distributed algorithms.

haven't showed a difference in performance of MCTS using these expansion. We have finally chosen the pessimistic expansion for further experiments.

A shape of the curve connecting measured points in the figure does not follow decreasing progression which is, of course, caused by measurement error and local minima overcame by the convergence of MCTS. However, for purposes of the strength-speedup measure, we need to have smooth and decreasing function of strength of plain MCTS. For the sake of the measure, we calculate such a function by polynomial regression. We fitted measured data of plain MCTS with pessimistic expansion on polynomial function $S(t) = c_0 + \frac{c_1}{\sqrt{t}}$, where S stands for strength (or score), c_0, c_1 are fitted coefficients and t is length of a tick (time for computation). Form of $S(t)$ was suggested according to shape of plotted data. The fitted function is drawn in the figure as a dashed curve. Once we know the regression of the plain MCTS strength, we can simply calculate strength-speedup with usage of inverse function of $S(t)$.

$$strength\text{-}speedup(score, time) = \frac{S^{-1}(score)}{time} \tag{4.3}$$

$$\text{where } S^{-1}(score) = \left(\frac{c_1}{score - c_0}\right)^2 \tag{4.4}$$

4.5.2 Independent Agents

Independent agents does not perform any communication. Each of them builds an independent MCTS tree. The trees are synchronized by having the same random seed for a construction of trees and so trees are, after same number of interations, equal. For this algorithm, we expected strength speedup approximately 1.0 or a bit lower (each agent usually manages to calculate a slightly different number of simulations what may lead to a different calculated action and thus to an invalid synchronization).

For comparison, we tested also the independent agents algorithm with distinct random seeds. Results are depicted by Figure 4.4. Average strength speedups are 0.967 for equal random seeds and 0.941 for distinct random seeds what corresponds with expectations.

Since there is no communication in algorithms, no further tests with altering communication channel speed and channel reliability were done.

4.5.3 Joint-Action Exchanging Agents

Joint-action exchanging agents serve for comparing more complex coordination methods with the simplest imaginable one. Particular agents use only simulations calculated by themselves and the only way how the algorithm can outperform plain MCTS is choosing the best of local minima calculated by agents. Surprisingly, the algorithm is quite successful and the mechanism of voting for the best action reached a strength speedup of 1.73. This result overcame our expectations. Tests with various channel speeds (with computation time 40 ms) showed that even for a little communication at the rate of 1 kilobyte per second, the algorithm performs as measured. We did not perform tests with different channel speeds

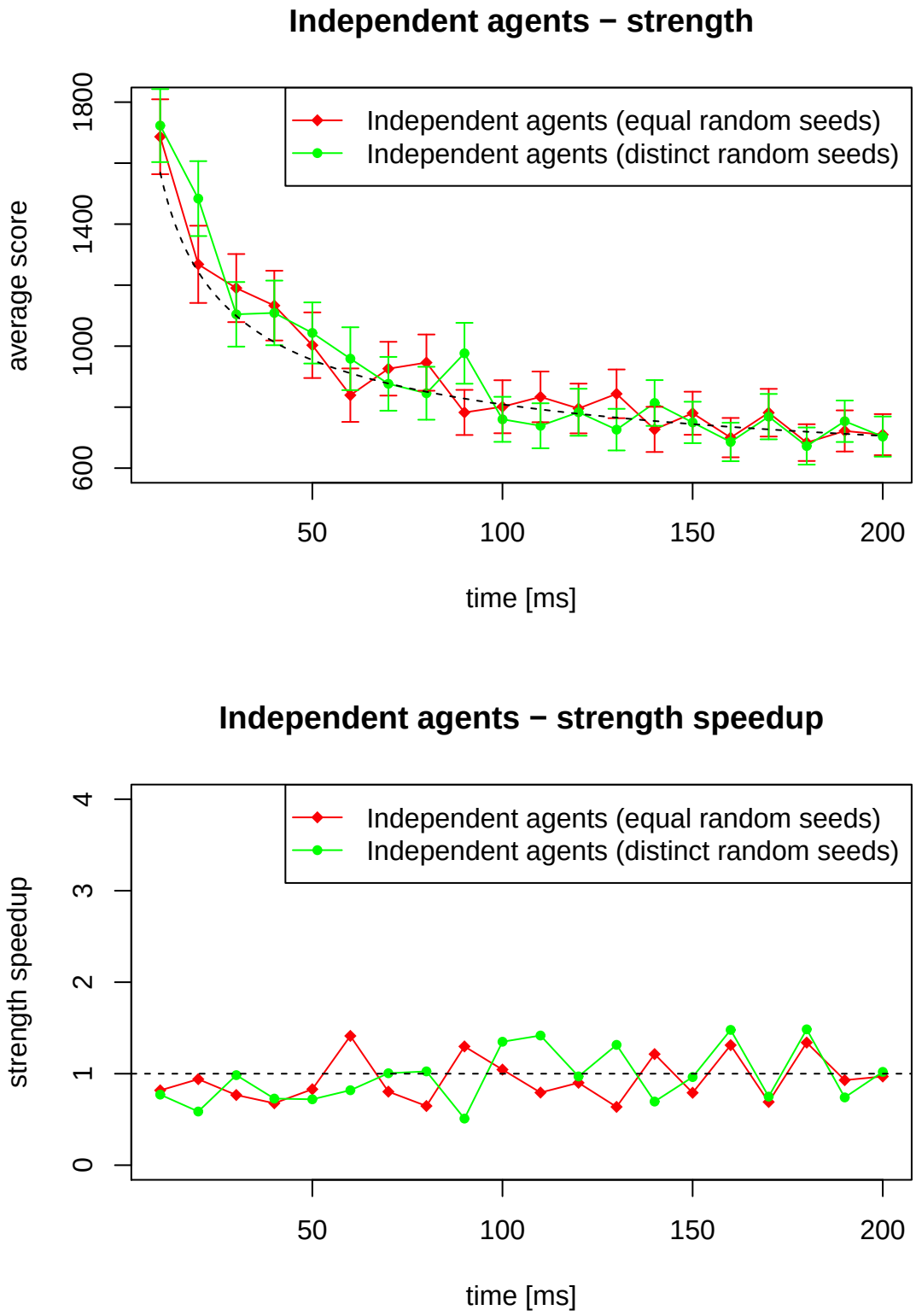
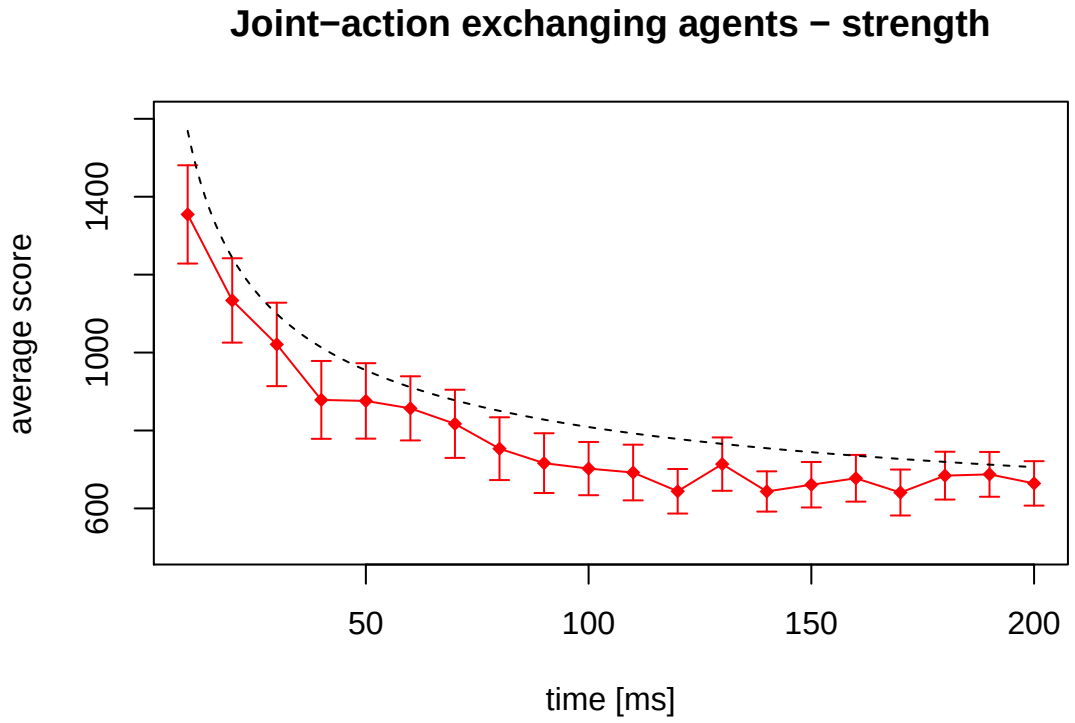


Figure 4.4: Independent agents strength.
Top - strength of independent agents (with and without equal random-seed synchronization) depending on computational time, 95% confidence intervals. Bottom - strength-speedup of independent agents.



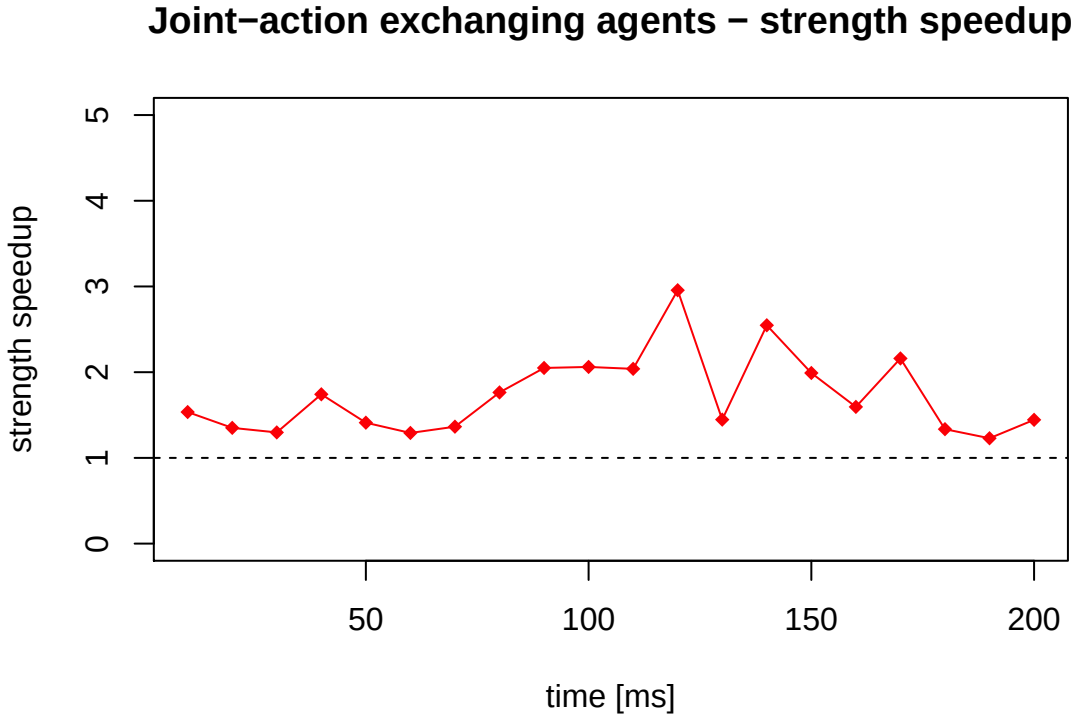


Figure 4.5: Joint-action exchanging agents strength.
Top - strength of joint-action exchanging agents depending on computational time, 95% confidence intervals. Bottom - strength-speedup of joint-action exchanging agents.

since the difference between strength speedup measured with unlimited communication and no communication is not so high in comparison with measure error. Even though, we suppose that if the communication successfully transmitted retains at average rate of 1 kBps, the algorithm's performance will be very similar to values measured in previous tests.

4.5.4 Root Exchanging Agents

According to good results of root parallelization algorithm referred in Section 2.3.6, we expected good behaviour of an algorithm based on this parallelization. Root exchanging agents reached average strength speedup 2.48 (average strength efficiency 0.62)) what is below our expectation (see Figure 4.6). We performed further analysis of behaviour of the algorithm and found a bug causing decrease of number of simulations computed per second by 15%. We didn't have time to run all experiments again but we were able to estimate the behaviour of the algorithm after bug removal using regression of performance of plain MCTS. Recalculated strength speedup reached a value of 2.92 (efficiency 0.73). With computational time of 40 ms, the algorithm performs with speedup of 2 (no recalculation) once the amount of communication is at least 4 kBps. Performance of the algorithm in dependence on channel speed is depicted by Figure 4.7.

4.5.5 Simulation Results Passing Agents

The agents share full information about the performed simulations. In our implementation, the ghosts need 256kB/s to reach a full performance of the algorithm, which is, expressed as strength speedup, 1.60 in average. The algorithm performs worse than joint-action exchanging agents. Expenses of such massive sharing among all trees are too high not only for requirements on the width of communication channels but also for the amount of calculated simulations which is reduced by approximately 1/3. Experimental results confirmed that this approach is robust against communication failures. According to the experiments, the strength speedup remains over 1.5 for the channel reliability 0.6 and higher. Results are ilustrated by Figures 4.8 and 4.9.

4.5.6 Tree-Cut Exchanging Agents

Since one simulation may be transmitted more than once, the amount of computations unlimitedly grows with amount of communication. To find suitable amount of communication, we performed tests with varying of channel speed first. Figure 4.10 shows results of these tests. According to them, with computational time of 40 ms, communication of approximately 64kB per second showed best results of speedup 2.19. Then tests with fixed channel speed of 64kBps were performed (Figure 4.11. Average strength-speedup measured is 2.07 which is quite good result in comparison with other algorithms. The amount of communication required by the algorithm is the main disadvantage. On the other hand, if a wide channel is available, the algorithm could keep amount of communication on certain optimal level and remaining capacity may be used to resist communication failures.

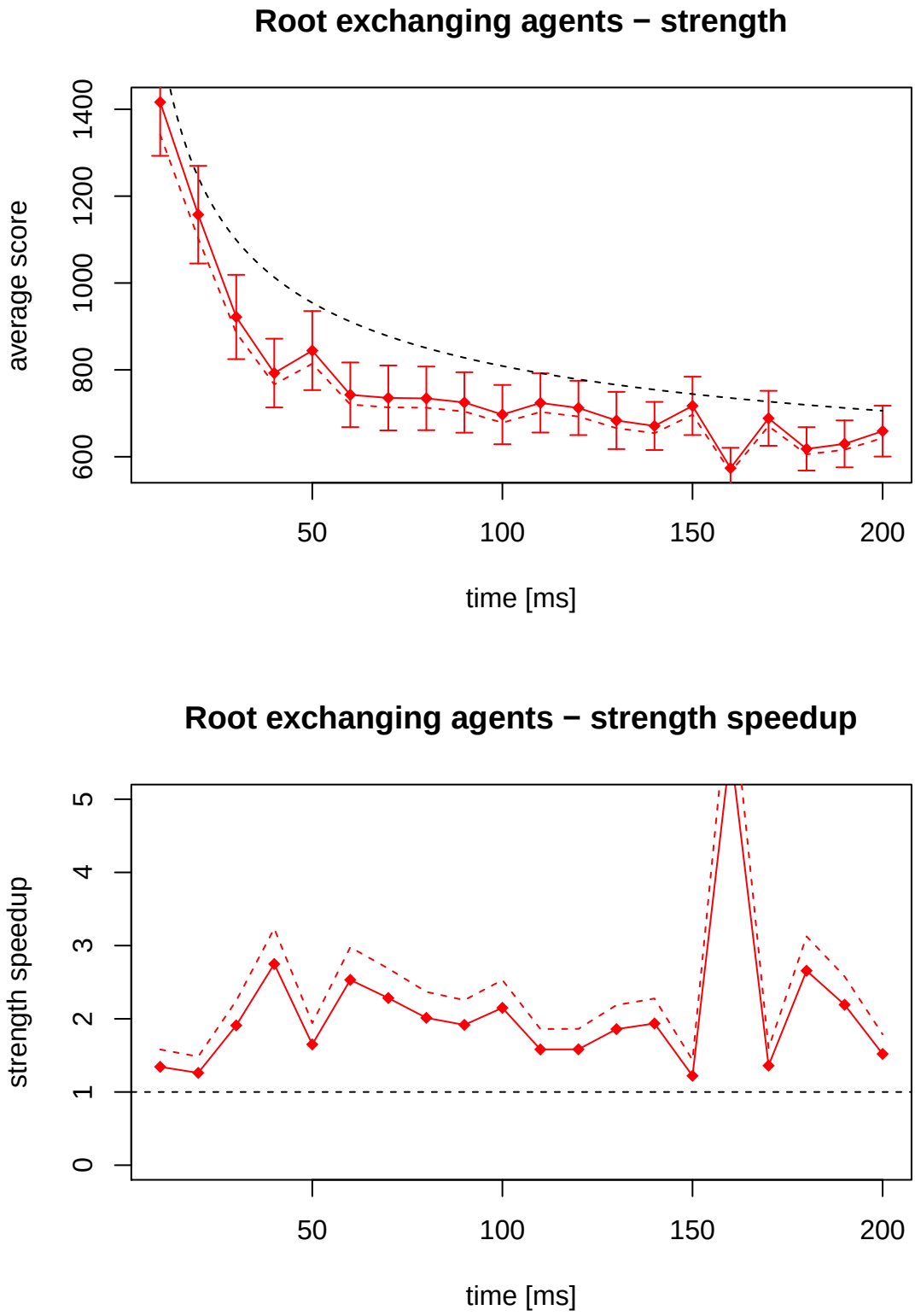
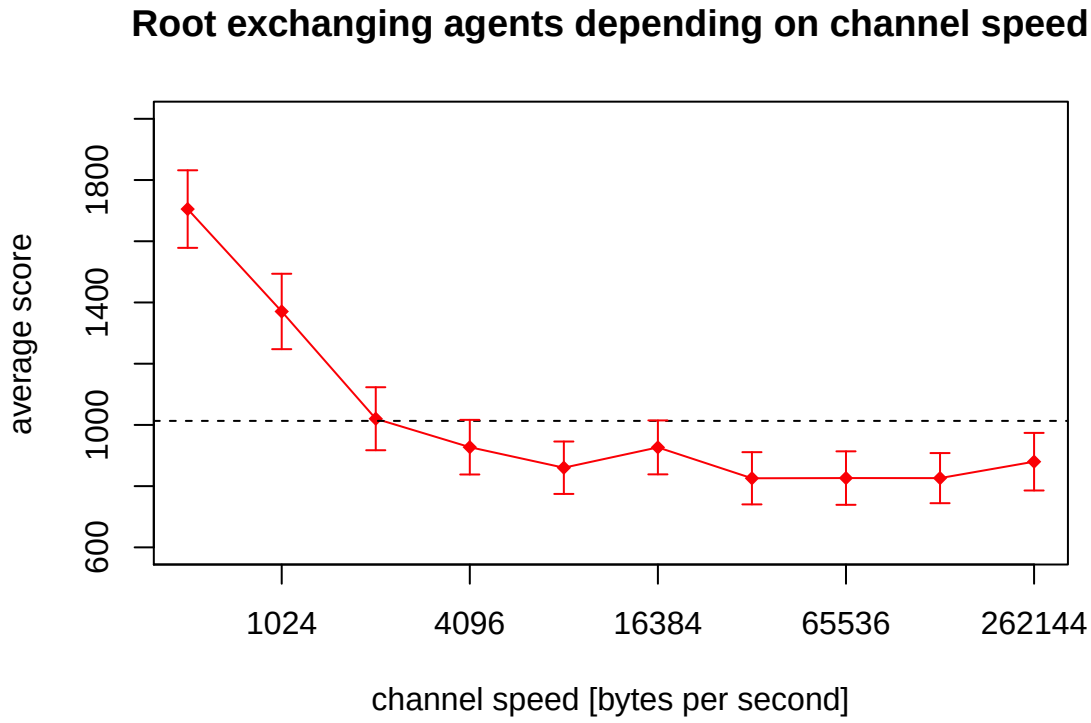


Figure 4.6: Root exchanging agents strength.
Top - strength of root exchanging agents depending on computational time, 95% confidence interval. Bottom - strength-speedup of root exchanging agents. Red dashed line is an estimate of behaviour of the algorithm after removing performance bug found after the experiments.



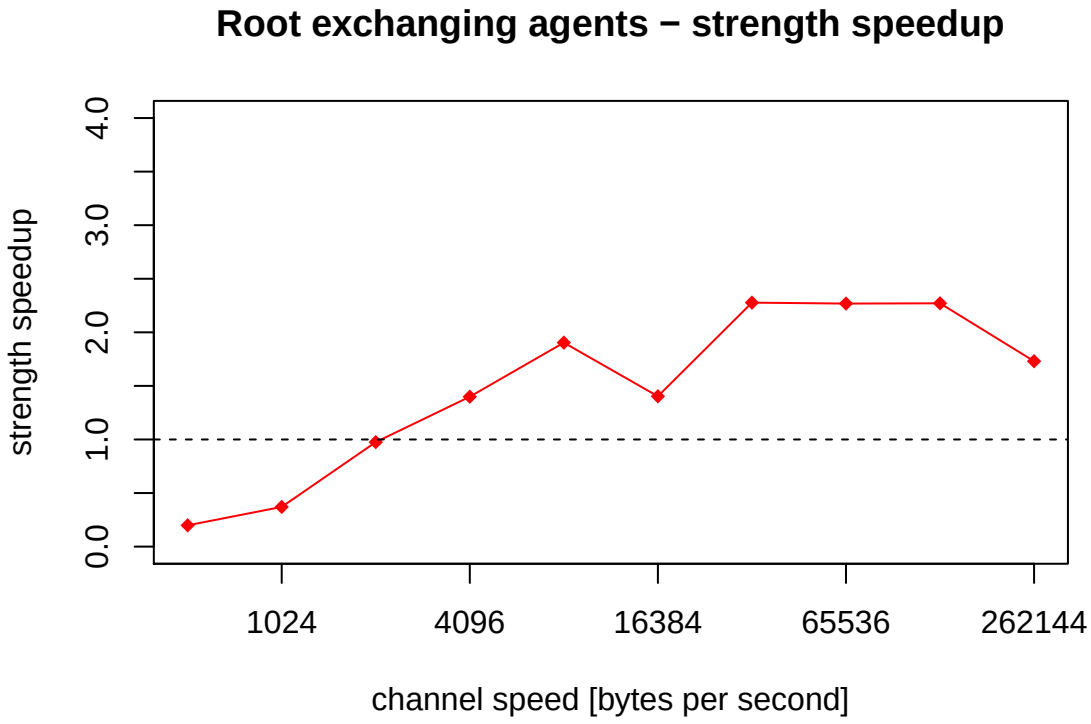
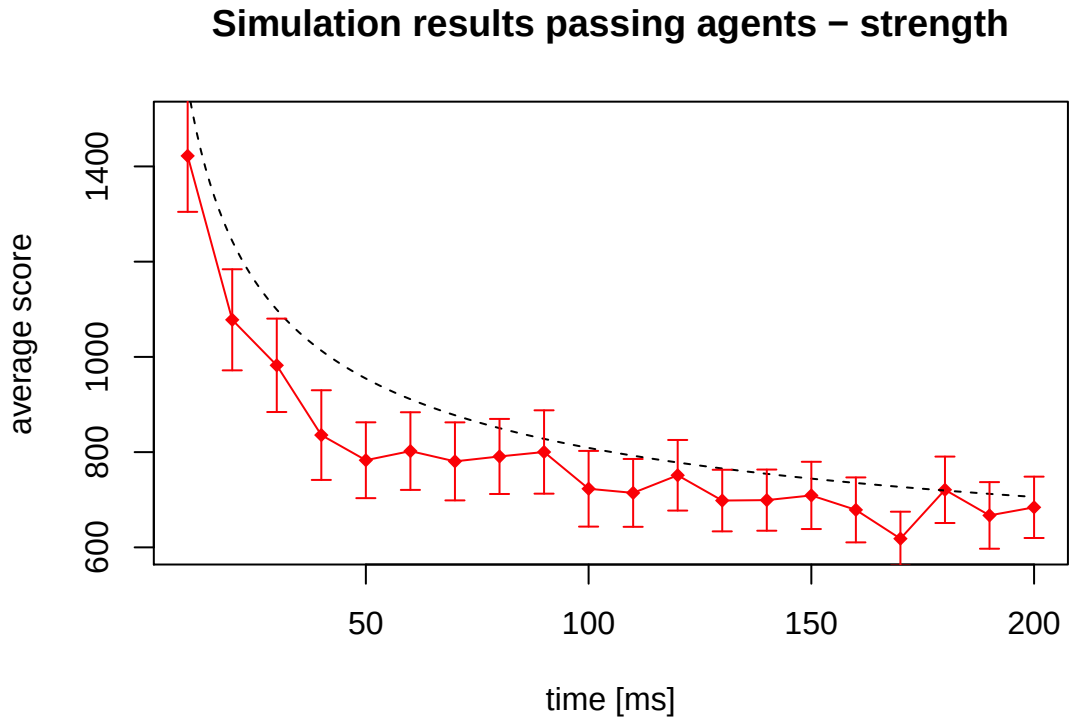


Figure 4.7: Root exchanging agents strength depending on channel speed.
Top - strength of root exchanging agents depending on channel speed, 95% confidence interval.
Bottom - strength-speedup of root exchanging agents depending on channel speed.



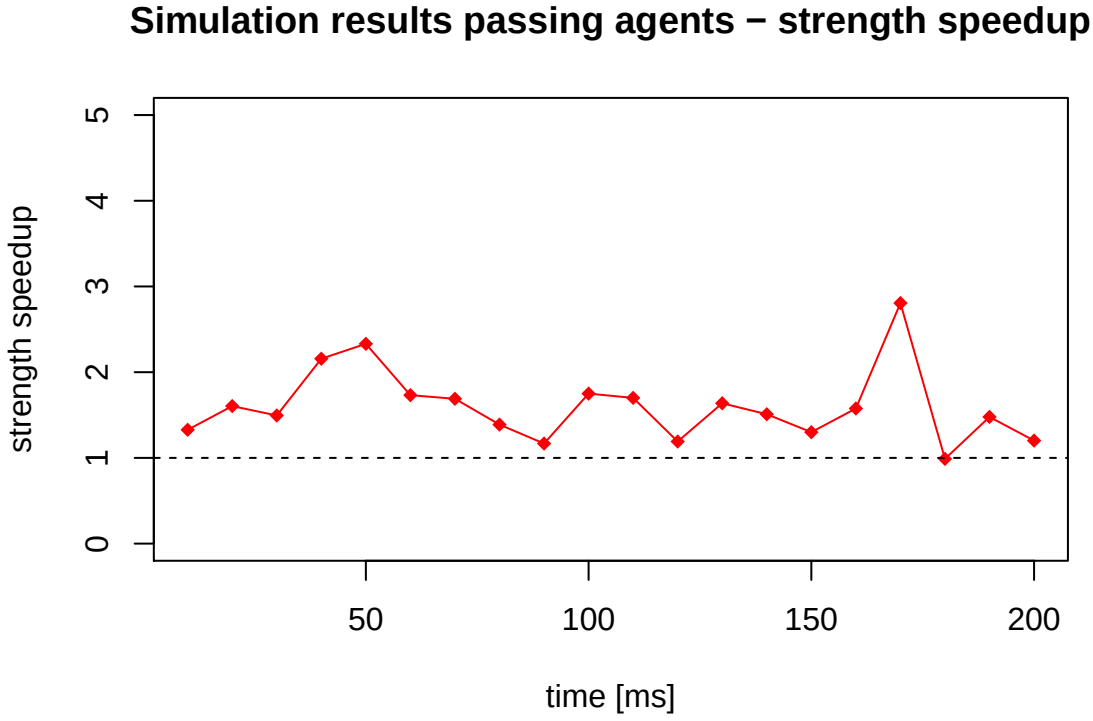
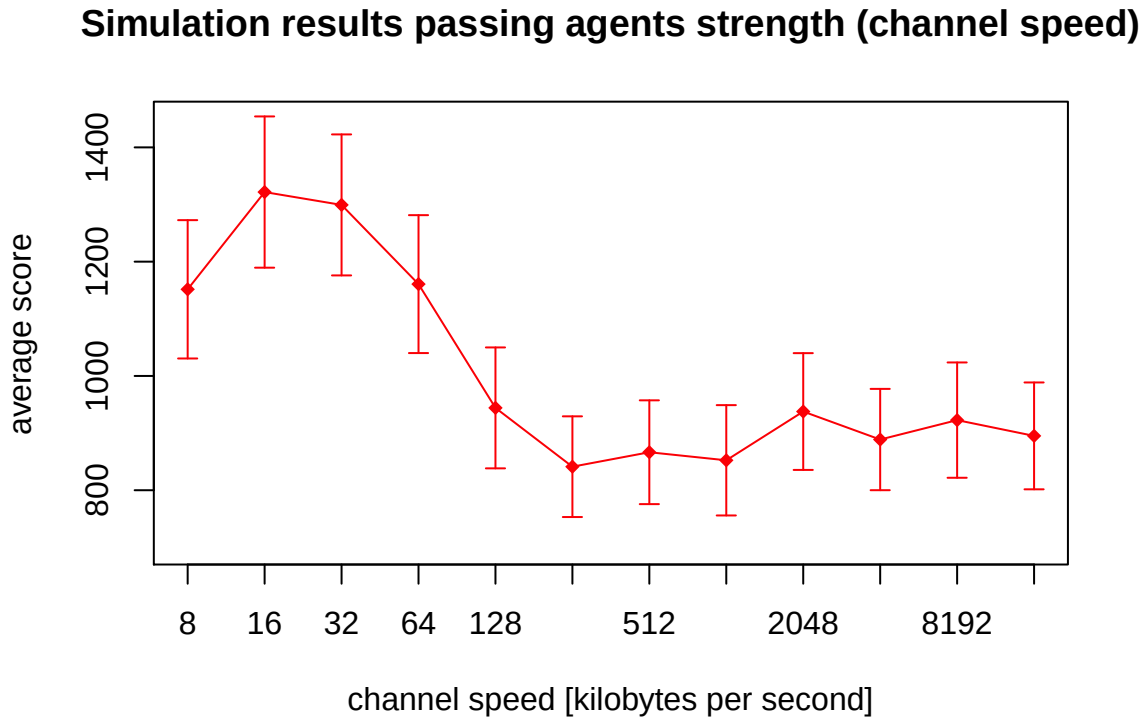


Figure 4.8: Simulation results passing agents strength.
Top - strength of simulation results passing agents depending on computational time, 95% confidence intervals. Bottom - strength speedup of simulation results passing agents.



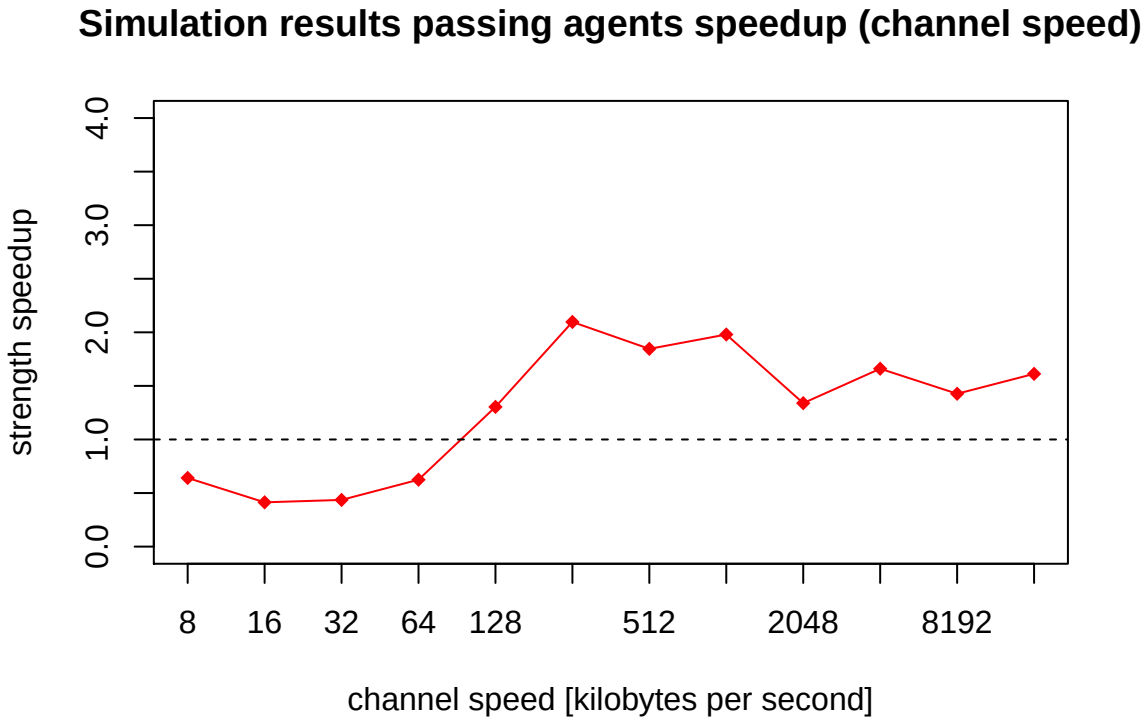


Figure 4.9: Simulation results passing agents strength depending on channel speed. Top - strength of simulation results passing agents depending on channel speed. Bottom - strength-speedup of simulation results passing agents depending on channel speed.

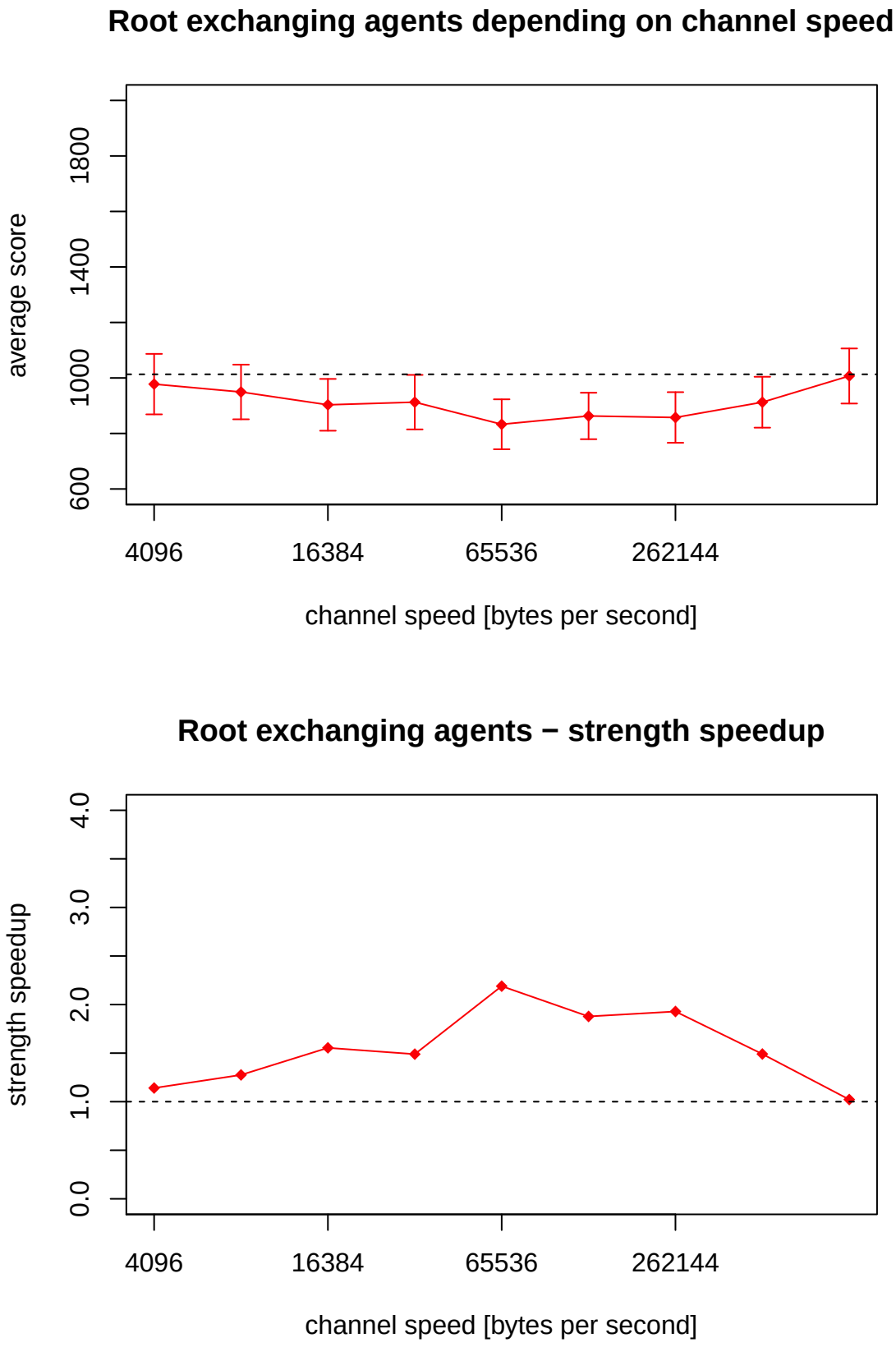
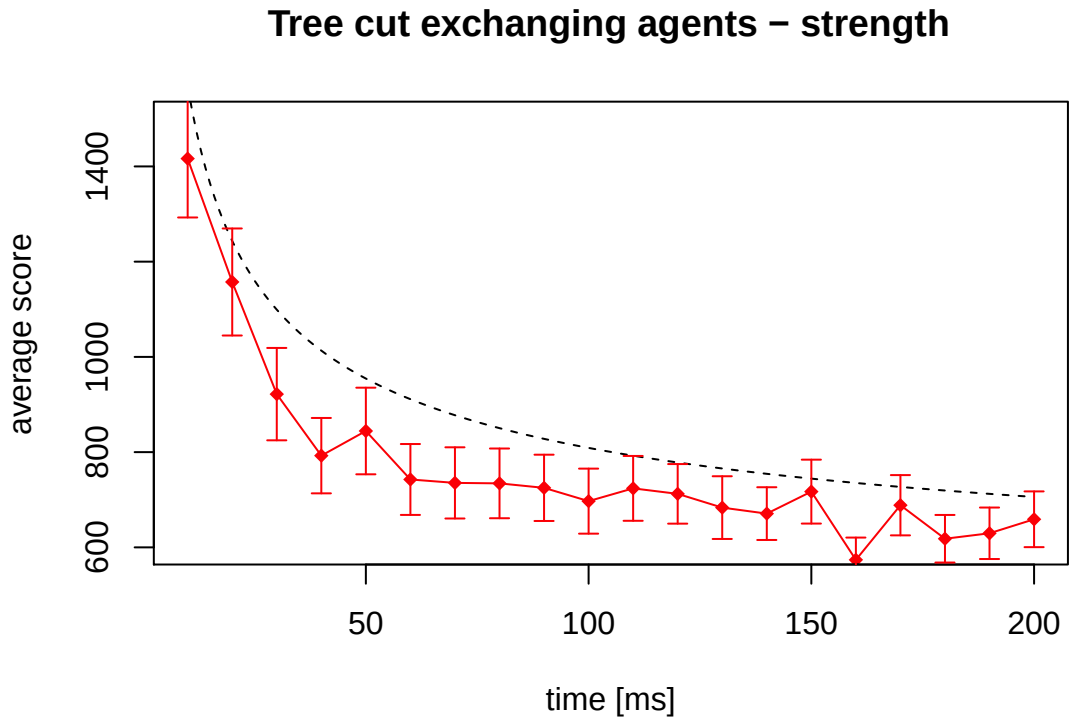


Figure 4.10: Tree cut exchanging agents strength depending on channel speed. Top - strength of tree cut exchanging agents depending on channel speed. Bottom - strength-speedup of tree cut exchanging agents depending on channel speed.



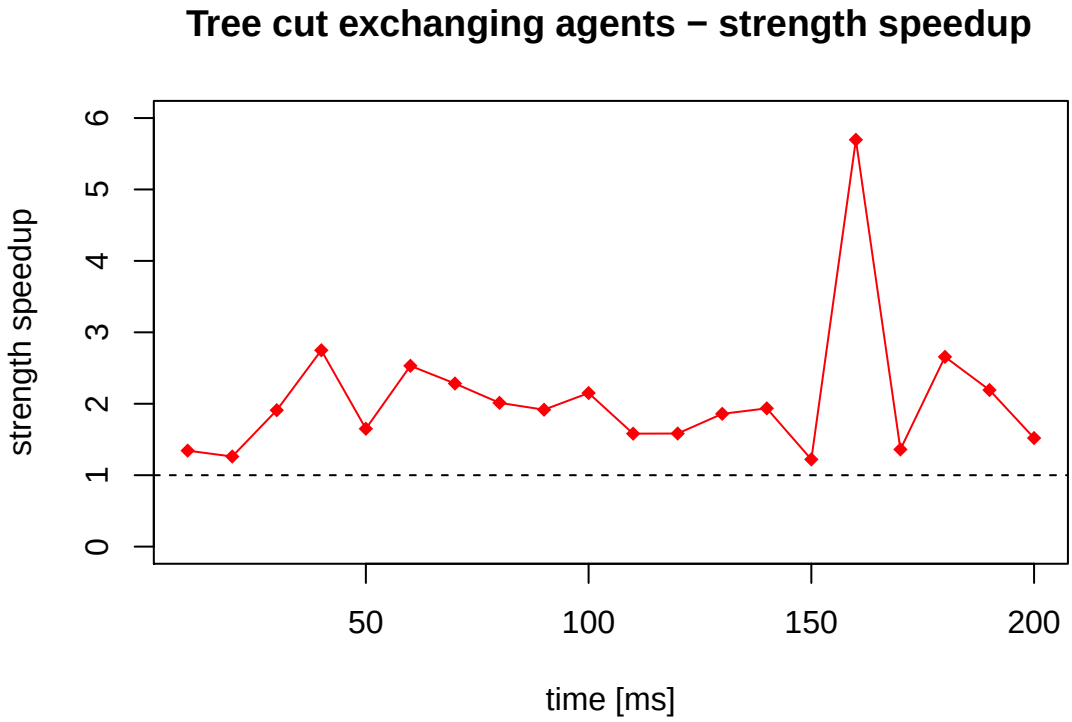


Figure 4.11: Tree cut exchanging agents strength.
Top - strength of tree cut exchanging agents depending on computation time. Bottom - strength-speedup of tree cut exchanging agents depending on computation time.

4.5.7 Comparison and Conclusion

In Table 4.5.7, we provide a synoptic comparison of the algorithms. All algorithms have good robustness against communication failures. What differs is the strength of the algorithms (compared according to the results of the tests on Ms Pac-Man game) and communication requirements. We consider root exchanging agents algorithm as the winner of tests since it reached the best value of strength-speedup together with low requirements of the amount of communication.

	strength (speedup)	communication requirements	failure robustness
Independent agents	very bad (0.97)	no	-
Joint-action ex. ag.	bad (1.73)	very low	very good
Root ex. agents	very good (2.92)	low	good
Sim. results passing ag.	bad (1.60)	very high	good
Tree-cut ex. agents	medium (2.07)	medium (scalable)	good (scalable)

Table 4.1: Comparison of distributed MCTS algorithm.

The table compares properties of proposed distributed MCTS algorithms used for multi-agent coordination. All algorithms are designed to be robust against communication failures and such robustness grow with decreasing communication requirements.

Conclusion

Summary

This work investigates possibilities of usage of Monte-Carlo tree search for distributed multi-agent coordination. Especially we have considered the problems of inter-agent information transmission, such as communication failures.

We reviewed the perfect-information game and proposed the definition enriching such game with teams. Additionally, we reviewed general coordination and communication issues together with approaches to the distributed team coordination inspired with distributed constraints optimization problems. One chapter is devoted to a description of Monte-Carlo tree search and its variant (UCT) and to existing MCTS parallelization algorithms.

The main result is a set of distributed algorithms based on MCTS dedicated to action planning in the environment of perfect-information games with teams. We have designed several algorithms of various levels of complexity, communication requirements and scalability trade-offs. We put detailed description of the design of the proposed algorithms.

For the evaluation, we used the domain of a simplified game of Ms Pac-Man. It provided us a testing environment with four cooperative agents. The quality of particular algorithms was measured in dependence on computational time, amount of communication and communication failures. In terms of the properties, we compiled a comparison of the algorithms and suggested the best candidates suitable for the solution of team coordination problem. *Root exchanging agents* algorithm, based on existing parallel MCTS algorithm, showed up as the best of such candidates. It reached the overall best performance, with small communication requirements and good robustness against communication failures. The second adaptation of existing parallel algorithms is *simulation results passing agents* which, in opposite to root exchanging agents, performed worst (if we don't consider *independent agents* algorithm proposed for comparison purposes) and requires the highest amount of communication. The algorithm has still some good general properties such as deeper insight of the tree so we developed third algorithm as a trial of integration of good properties of previous two algorithms - *tree-cut exchanging agents*. The algorithm requires more communication than root exchanging agents and performs a bit worse, yet some open questions of the algorithm design retain (see Future Work). The last algorithm, *joint-action exchanging agents*, showed unexpectedly good results. With its simplicity and very low communication requirements and good robustness, it outperformed simulation results passing algorithm.

A part of this thesis is also DVD containing source codes of implemented algorithms, experimental data, source codes of this thesis and other tools used for performing experiments (see Attachment 1).

Future Work

1. The algorithms can be used for coordination of large-scale teams. Scalability of algorithm proposed in our thesis is limited since agents' trees

contain information. If the tree would contain information reduced to some neighbourhood of the agent, it would lead to better scalability. Algorithms solving distributed constraints optimization problems could also bring new ideas to improve the scalability.

2. The algorithm of *tree-cut exchanging agents* can be further enquired. Especially, (1) tuning of number of cuts transmitted per a game step may be tuned and (2) the way of transmission of the cuts may be adjusted. In the original algorithm, cuts are transmitted in one message so if the communication failure occurs in the middle of the message, the whole cut is lost. But cuts may be transmitted node by node what would lead to better robustness of the algorithm. (3) The algorithm has also a special property: its performance starts decreasing with certain amount of communication (it is caused by too much redundant communication leading to unadequate repetitiveness of applying of the almost same tree cuts). This property may be also further enquired and some dynamic balancing of the communication may be performed by the algorithm to avoid decrease of the performance.

Bibliography

[1] P Auer, N Cesa-Bianchi, and P Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2 (2002). Ed. by Jyrki Kivinen, pp. 235–256. ISSN: 08856125.

[2] Bruno Bouzy. "Progressive Strategies for Monte-Carlo Tree Search". In: *Computer* 4.3 (2007), p. 343. ISSN: 17930057.

[3] Tristan Cazenave and Nicolas Jouandeau. "On the Parallelization of UCT". In: *Proceedings of the Computer Games*. Citeseer, 2007, pp. 93–101.

[4] Guillaume M Chaslot, Mark H Winands, and Jaap H Van Den Herik. "Parallel Monte-Carlo Tree Search". In: *Computers and Games* 5131 (2008), pp. 60–71. ISSN: 15302075.

[5] Guillaume Maurice Jean-Bernard Chaslot et al. "Monte-Carlo Strategies for Computer Go". In: *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence* (2006). Ed. by P Y Schobbens, W Vanhoof, and G Schwanen, pp. 83–90.

[6] Guillaume Chaslot. "Monte-Carlo Tree Search". PhD thesis. Maastricht University, 2010.

[7] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *Search*. Lecture Notes in Computer Science 4630 (2006), pp. 72–83.

[8] Nozomu Ikehata and Takeshi Ito. *Monte-Carlo tree search in Ms. Pac-Man*. 2011.

[9] Levente Kocsis and Csaba Szepesvári. "Bandit based Monte-Carlo Planning". In: *Machine Learning ECML 2006*. Lecture Notes in Computer Science 4212.1 (2006). Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, pp. 282–293. ISSN: 03029743.

[10] Kien Quang Nguyen and Ruck Thawonmas. *Applying Monte-Carlo Tree Search to collaboratively controlling of a Ghost Team in Ms Pac-Man*. 2011.

[11] Philipp Rohlfshagen, David Robles, and Simon M. Lucas. *Ms Pac-Man vs Ghosts Competition*. URL: `http://www.pacman-vs-ghosts.net/`.

[12] Mohammad Shafiei, Nathan Sturtevant, and Jonathan Schaeffer. "Comparing UCT versus CFR in Simultaneous Games". In: *Citeseer* (2009).

[13] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Revision 1.1. Cambridge University Press, 2009.

[14] Ruck Thawonmas Tetsuo Shirakawa Masahiro Nakamura. *ICEP_IDDFS*. URL: `http://www.pacman-vs-ghosts.net/users/268`.

[15] Olivier Teytaud et al. "THE PARALLELIZATION OF MONTE-CARLO PLANNING". In: *ICINCO* (2008), pp. 198–203.

[16] Roie Zivan, Robin Glinton, and Katia Sycara. *Distributed Constraint Optimization for Large Teams of Mobile Sensing Agents*. 2009.

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

MCTS	Monte-Carlo tree search
UCB	Upper-confidence bound
UCT	UCB for trees
DCOP	Distributed constraints optimization problem

Attachment 1 – DVD Contents

DVD attached to this work contains following files and directories:

- `README` - file containing description of the directories, specific `README`s included also in selected subdirectories;
- `text` - contains PDF version of this thesis (`thesis.pdf`) and LaTeX source files of the thesis in `tex` subdirectory;
- `src` - Netbeans project containing Java sources of implemented algorithms;
- `data` - contains data files obtained by evaluation tests (refer to `README` for description of particular files,
- `bin` - contains compiled `experiment.jar` file. The file provides possibility to run a single Ms Pac-Man game with ghosts controlled by selected algorithm running against Pac-Man controlled by ICEP_IDDFS. Usage can be found in `README`.