

The map coloring game

Final report

Arnaud AUDOIN Clément BADIOLA Samuel DA SILVA
Alexandre PERROT Camille RITLEWSKI

Client : Paul DORBEC

24 janvier 2013

Abstract

This document is the final report of a research project carried out by a group of 5 students within their year of Master 2.

The project addresses the graph coloring game, variant of the graph coloring problem, by using Monte Carlo heuristics and statistical tools to determine whether the theoretical bounds calculated by the researchers can be improved.

The results, presented in detail later, suggest that...

Contents

1	Introduction	1
1.1	Graph coloring	1
1.2	Map-coloring game	1
1.3	Exemple of a game	2
1.4	Applications	2
2	Theory	3
3	Work	4
3.1	Monte Carlo	4
3.2	How our AI works	5
3.3	Implementation of the experiments	7
3.3.1	Chains	7
3.3.2	Cycles	8
3.3.3	Grids	9
3.3.4	Binary Tree	11
3.3.5	Non-Planar Graphs	12
4	Conclusion	13
4.1	Assessment	13
4.1.1	Failures	13
4.2	If we were to do it again...	13
4.3	Prospect	13
A	Diagrammes	14
B	Manuel de maintenance	15
C	Extraits de code	16

Chapter 1

Introduction

1.1 Graph coloring

Graph coloring is not a new problem in mathematics and computer science. The first results in this field date back to 1852, when *Francis Guthrie* postulated the famous four color theorem¹.

Coloring a graph consists of assigning a color to each vertex of the graph, such that two adjacent vertices are of different colors. A such coloring is called a proper coloring. There are other forms of coloring, such as edge or face coloring, but they always reduce to a vertex coloring on another graph.

A graph is said to be k -colorable if it admits a proper coloring with k colors. The minimal number of colors needed to achieve a proper coloring of a particular graph is called the *chromatic number* of that graph, noted $\chi(g)$. Deciding if a graph is k -colorable for a given value of k is NP-complete.

1.2 Map-coloring game

The studied article's subject defines a variant of the graph coloring involving two players.

The first player tries to color the graph correctly. The second tries to hinder and stop him without overriding the coloring rule. The coloring rule stays the same: each vertex cannot have the same color as its neighbors. At the beginning of the game, we set a number N of colors than can be used in the game. The game ends if all the vertices are colored or if it is not possible to color the graph without adding a color while exceeding N , which is contrary to the rules.

We defines the *game chromatic number* as the minimal number of colors with which the first player can achieve a correct coloring of the graph whatever the strategy of the second player.

¹He did not write any publication on this theorem.

1.3 Exemple of a game

1.4 Applications

The graph coloring game has no direct applications. It helps to get a better understanding of the field of graph coloring.

Graph coloring applications examples:

- memory allocation,
- management of a room's occupation,
- management of schedules etc.

Chapter 2

Theory

To understand the strategies that follow, we first introduce three important concepts:

- the order of a graph G
- the loose backward neighbors
- The number of 2-coloring of a graph G , denoted $col^2(G)$

The order of a graph G is a linear representation of the vertices of the graph where we associate an index to each vertex according to its position in the order.

For the loose backward neighbor, we define an order of the graph G : $v^1v^2v^3\dots v^n$. A vertex v^h is a loose backward neighbor of v^i if $h < i$ and if v^h is in v^i neighborhood, or if v^h and v^i have a common neighbor v^j such as $h < i < j$.

Now, let's review the whole range of orders for a graph G . For each order found, we determine the integer k such as each vertex of the order has an number of loose backward neighbors $<$ to k . We keep the order which has the smallest k . This k is the number of 2-coloring. In addition, the associated order satisfies $col^2(G)$.

Chapter 3

Work

Contents

3.1 Monte Carlo	4
3.2 How our AI works	5
3.3 Implementation of the experiments	7
3.3.1 Chains	7
3.3.2 Cycles	8
3.3.3 Grids	9
3.3.4 Binary Tree	11
3.3.5 Non-Planar Graphs	12

3.1 Monte Carlo

Monte-Carlo methods are a set of randomized algorithms. The general idea is to use random samplings rather than computing to help solve a problem. Such algorithms are mostly used when computing a solution to the problem is not feasible. Because of the use of random samplings, these are heuristics by nature : their results are not always guaranteed to be exact. Throughout the years, Monte-Carlo methods have been successfully used in physics, mathematics and computer sciences.

In this report, we will interest ourselves in a special case of Monte-Carlo methods : Monte-Carlo tree search, aka MCTS. It applies Monte-Carlo principles to trees exploration. The main field of application is artificial intelligence, especially games with very large move space where a complete tree exploration is not possible in reasonable time, such as Chess and Go.

MCTS is composed of four steps : selection, expansion, simulation and back-propagation. Those steps are repeated until time runs out, the maximum number of simulated games has been played, a satisfying result is found or whatever condition is relevant. Generally, the more iterations are run, the more accurate is the result.

The selection step chooses a node in order to play a simulation from it. Several algorithms exist for this stage, depending on the context. The expansion

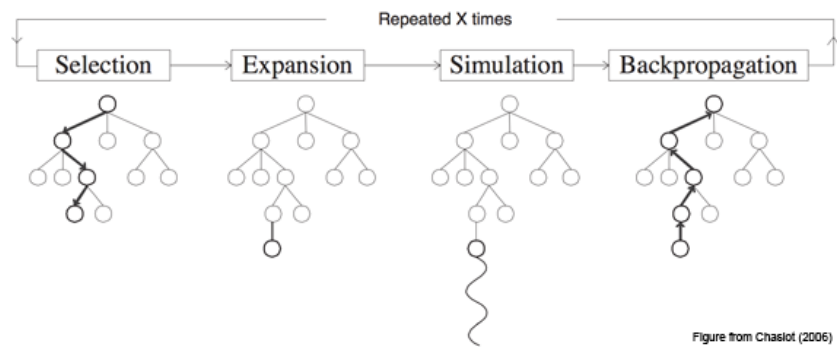


Figure 3.1: Etapes de Monte-Carlo tree search

step produces child of the selected node if not already generated. This step makes the search tree grow in height. The simulation step plays the game until the end following nodes selected with a selection algorithm which can be the same as the one used in the selection step or a different one. Finally, the backpropagation step uses the outcome of the simulated game to modify the estimated value of the traversed nodes. This is the crucial step to know what has been played and if the selected moves constitute good choices or not.

After playing enough simulated games, a player can then choose amongst the playable moves which one to play using the estimated value of the move. The choice algorithm can be one of the followings (from [3]) :

- Max Child : Choosing the node with the highest value.
- Robust Child : Choosing the node with the highest number of games played.
- Robust-Max Child : Choosing the node with both the best value and highest number of games played.
- Secure Child : Choosing the node maximizing a lower confidence bound.

According to *Guillaume Chaslot* [3] : "MCTS is applicable if at least the following three conditions are satisfied: (1) the payoffs are bounded (the game scores are bounded), (2) the rules are known (complete information), and (3) simulations terminate relatively fast (game length is limited)". Those three criterias are verified in the case of the map-coloring game. We will use Monte-Carlo tree search to implement an AI capable of playing Alice or Bob.

3.2 How our AI works

To modelise a map-coloring game, each node of the exploration tree will contain:

- The move played when choosing that node, which is simply the node chosen in the game graph and the color used.
- The total number of games played through that node.

- The number of games with a favorable outcome for Alice while playing through that node.

For each node in the search tree, its value can be seen as the victory ratio for Alice : $\frac{\text{games won}}{\text{games played}}$.

The selection algorithm we used is based on multi-armed bandit algorithms as described in [7]. Indeed, selecting a node amongst the children of the current node in the tree can be seen as a multi-armed bandit problem, where each node is a bandit, its profit being its value. We chose the UCT algorithm, based on the UCB1 multi-armed bandit algorithm, described in [3]. This algorithm is called best-first because it always selects the node which seems to be the best. This estimation is based on the value of the node and an upper confidence bound defined as : $\sqrt{\frac{2 \ln n}{n_j}}$, n being the total number of games played on the parent and n_j being the number of games played on this node.

For every game played through a node, n and n_j grow, thus the upper confidence bound of this node is reduced. This is a formal way of describing an intuitive idea : "The more I play this move, the more I am confident in the value I have calculated". The best node according to the algorithm is the node whose value plus its upper confidence bound is the highest. By using this algorithm, we can quickly stop exploring the least interesting nodes to concentrate on better ones, thus maximizing the profit from a fixed number of simulations.

Nevertheless, this algorithm is adapted to a general Monte-Carlo tree search. In our particular case, we deal with a game between two opponents whose objectives are opposite. Using this algorithm directly would bring us to consider that the opponent is playing toward the same goal, thus helping us, which is of course false. The canonic way of dealing with that problem in a two-player zero-sum¹ game is called Minimax (established in [8]). When exploring a tree with Minimax, the values are maximized when playing with player 1 and minimized when playing player 2, to represent the opposite goal. We reproduced a similar process in the UCT algorithm. If Alice is playing, the confidence interval of the node is added to its value, and the chosen node is the one with the highest resulting total. On the contrary, if Bob is playing, the confidence interval is subtracted from the value, resulting in a lower confidence bound. The selected node is then the one with the lowest lower confidence bound. Using this minimax UCT algorithm, our AI implementation will be the same both for Alice and Bob.

The other steps of the Monte-Carlo tree search are now straightforward. The expansion step just generates every legal move on the graph from the current position with the currently used colors, and moves with a new colors and adds a node for each one. The simulation step reuses the selection algorithm to play until the game is finished. Finally, the backpropagating step increments the number of games played and won according to the outcome of the game.

¹A game is said to be zero-sum if the winner wins as much as the loser is losing.

3.3 Implementation of the experiments

In our program, many options are parametrizable. Those can be set using program arguments :

- The graph to play on (graphviz format supported)
- The number of colors to play with
- The number of games to play
- The number of simulations to be run by Monte-Carlo before selecting a move

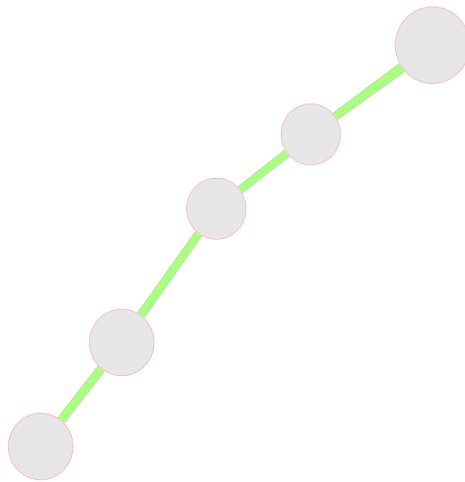
After some testing, we settled on the following experimental conditions :

- A number of simulations of 1000. This seemed to be a good compromise between speed and AI accuracy.
- A number of games played of 1000. It seemed again to be a good compromise between computation time and statistical confidence. However, we could not play as much games on the 5x5 grids with a high number of colors (typically 4 or 5). We then tuned down the games to 100 for those cases.

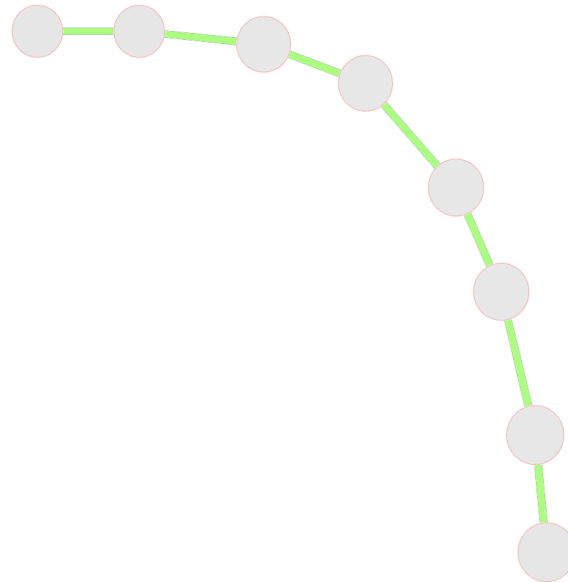
Here are the results :

3.3.1 Chains

Odd number of vertices

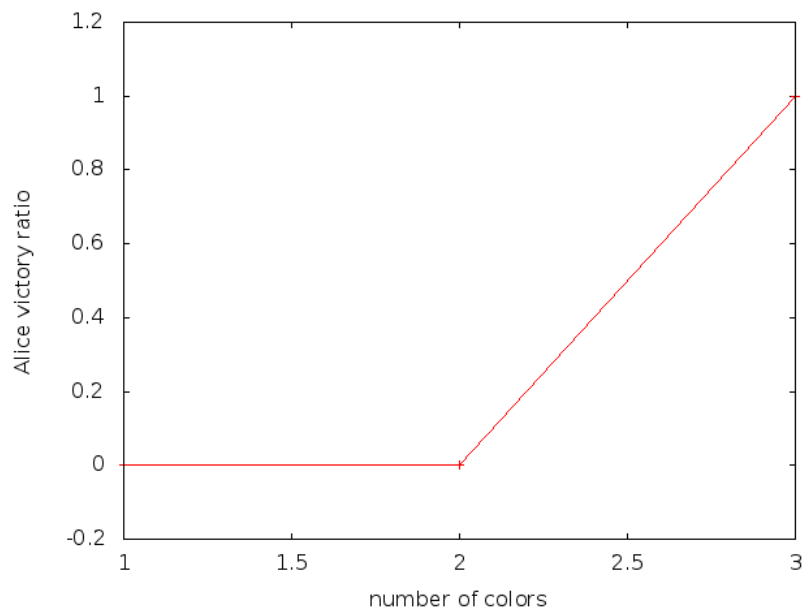


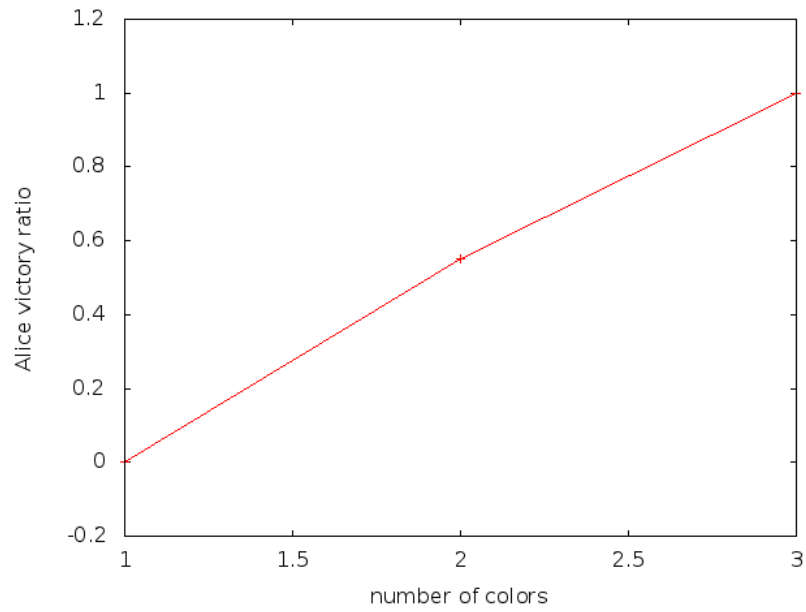
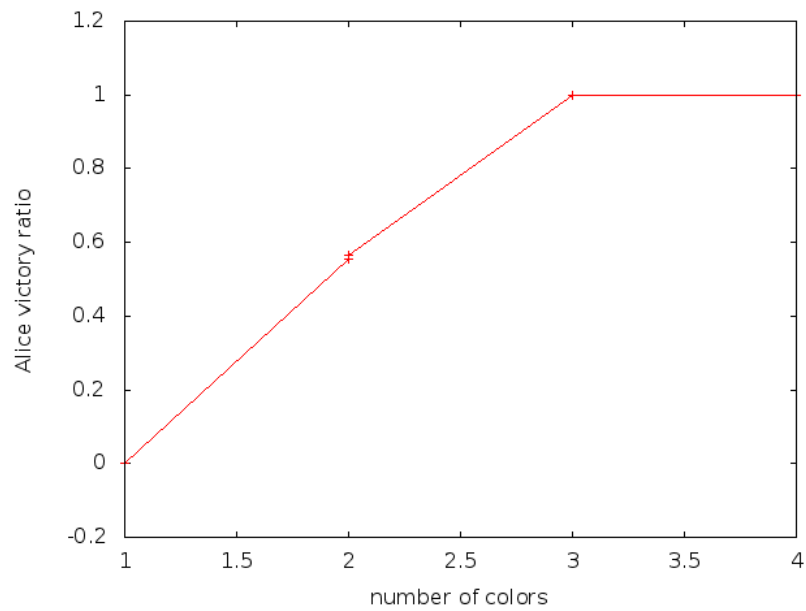
Even number of vertices

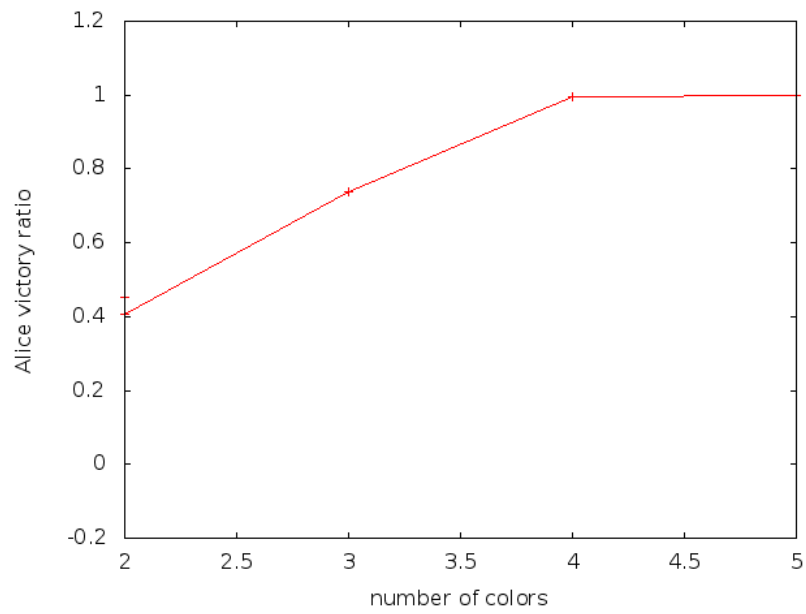
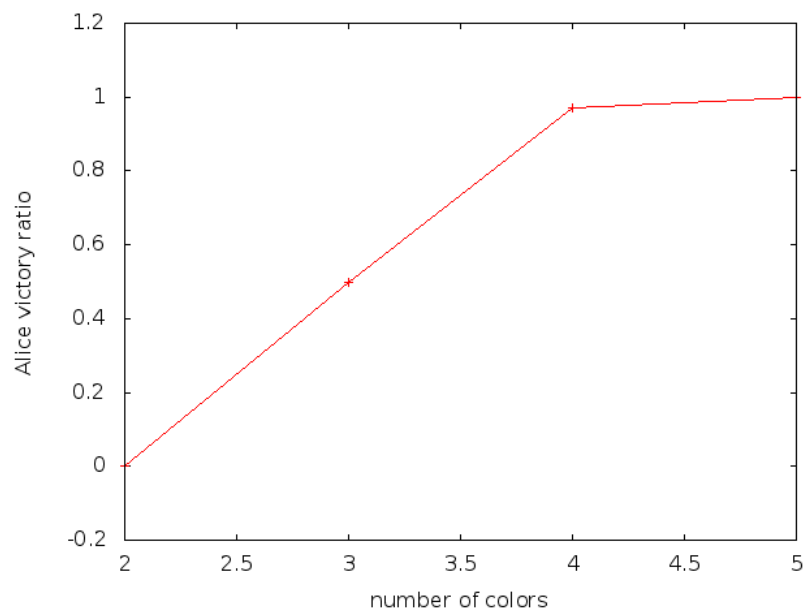


3.3.2 Cycles

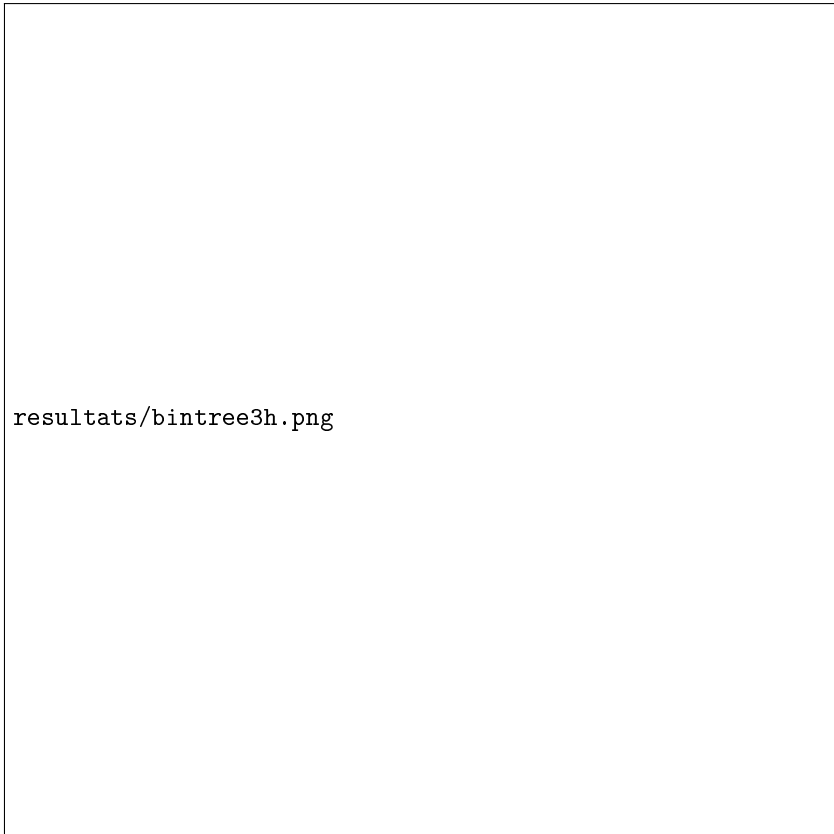
Odd number of vertices



Even number of vertices**3.3.3 Grids****2x5**

5x5**5x5 toroidal**

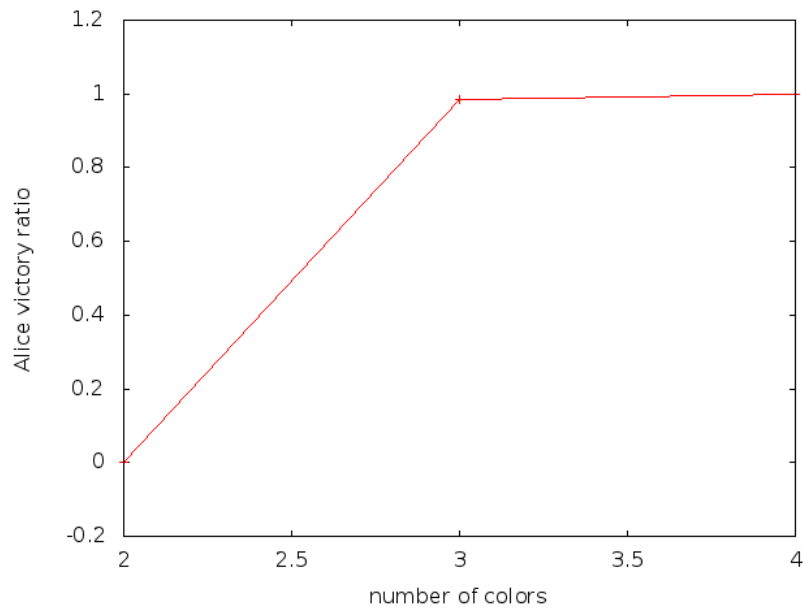
3.3.4 Binary Tree



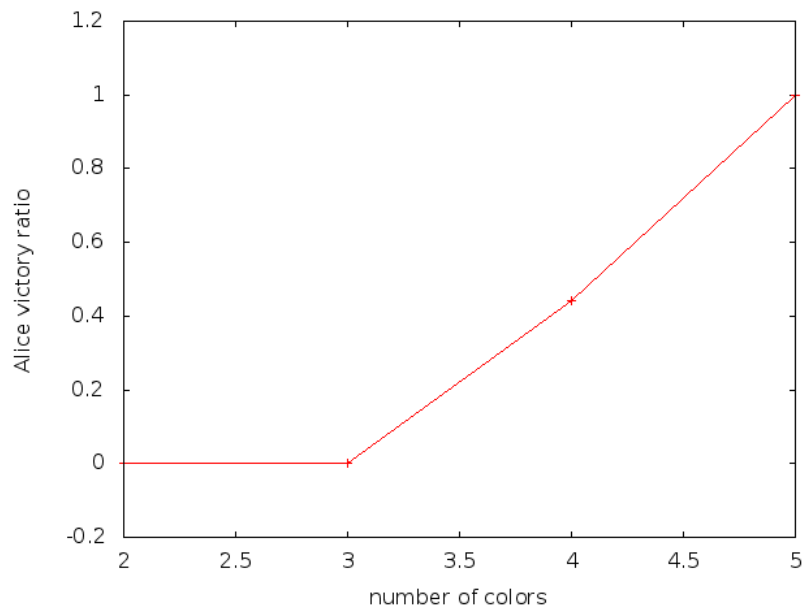
resultats/bintree3h.png

3.3.5 Non-Planar Graphs

Petersen Graph



Icosaedron Graph



Chapter 4

Conclusion

Contents

4.1	Assessment	13
4.1.1	Failures	13
4.2	If we were to do it again...	13
4.3	Prospect	13

4.1 Assessment

Texte ici.

4.1.1 Failures

Texte ici.

4.2 If we were to do it again...

4.3 Prospect

Appendix A

Diagrammes

Appendix B

Manuel de maintenance

Appendix C

Extraits de code

Bibliography

- [1] Pierre arnaud Coquelin and Rémi Munos. Bandit algorithm for tree search. Technical report, INRIA, 2007.
- [2] G. Chaslot, J. Saito, B. Bouzy, J. Uiterwijk, and H. Van Den Herik. Monte-carlo strategies for computer go. *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91+, 2006.
- [3] G. M. J. B. Chaslot, M. H. M. Win, and B. Bouzy. Progressive strategies for monte-carlo tree search, 2007.
- [4] Guillaume Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Université de Maastricht, 2010. http://www.unimaas.nl/games/files/phd/Chaslot_thesis.pdf.
- [5] Istvan Szita Guillaume Chaslot, Sander Bakkes and Pieter Spronck. Monte-carlo tree search: A new framework for game ai.
- [6] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.
- [7] Chris Stucchio. Why multi-armed bandit algorithms are superior to a/b testing. http://www.chrisstucchio.com/blog/2012/bandit_algorithms_vs_ab.html, 2012.
- [8] J Von Neumann. *Zur Theorie der Gesellschaftsspiele*. 1928.

List of Figures

3.1 Etapes de Monte-Carlo tree search	5
---	---

List of Tables

Rapport de projet d'études et de recherche.
The map coloring game.
Master 2, 2012

Les sources de ce rapport sont disponibles librement sur