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, a 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 the current theoretical bounds can be significantly improved.

Contents

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 theorem[1].

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

A graph is said to be k-coloriable 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-coloriable 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.

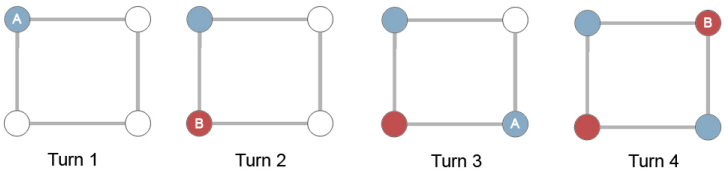
The first player tries to color the graph correctly. The second tries to hinder and stop him without overriding the coloring rule. The coloring rule stays the same: each vertex cannot have the same color as its neighbors. At the beginning of the game, we set a number N of colors than can be used in the 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.

The *game chromatic number* is defined 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.

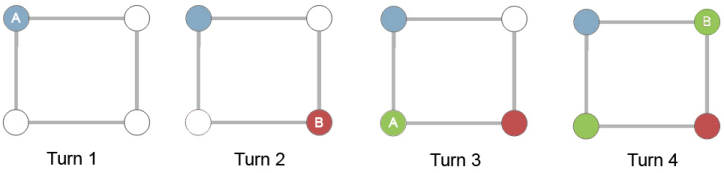
[1] He did not write any publication on this theorem.

1.3 Example of a game

Blablabla explication



Blabla explication



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

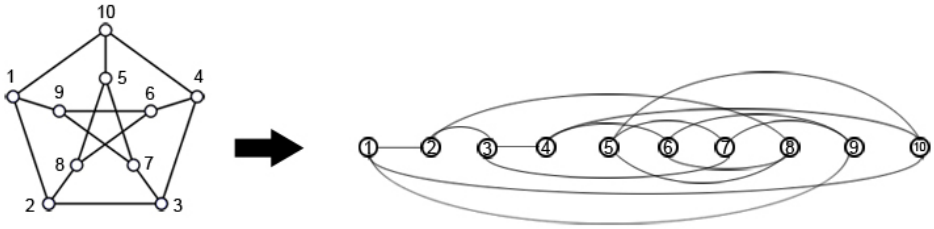
To understand the strategies that follow, we first introduce three important concepts:

- the order of a graph G
- the loose backward neighbors
- the number of 2-coloring of a graph G, denoted $col_2(G)$
- the chromatic number of a graph G, denoted $\chi(G)$
- the game chromatic number of a graph G, denoted $\chi_g(G)$

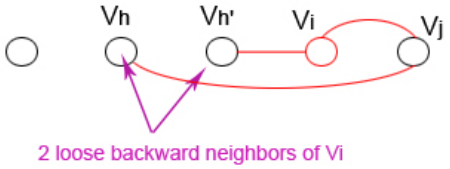
The chromatic number of a graph G is the minimum number of colors to make a good coloring of G.

The game chromatic number is the number of allowed colors for which Alice always has a winning strategy.

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.



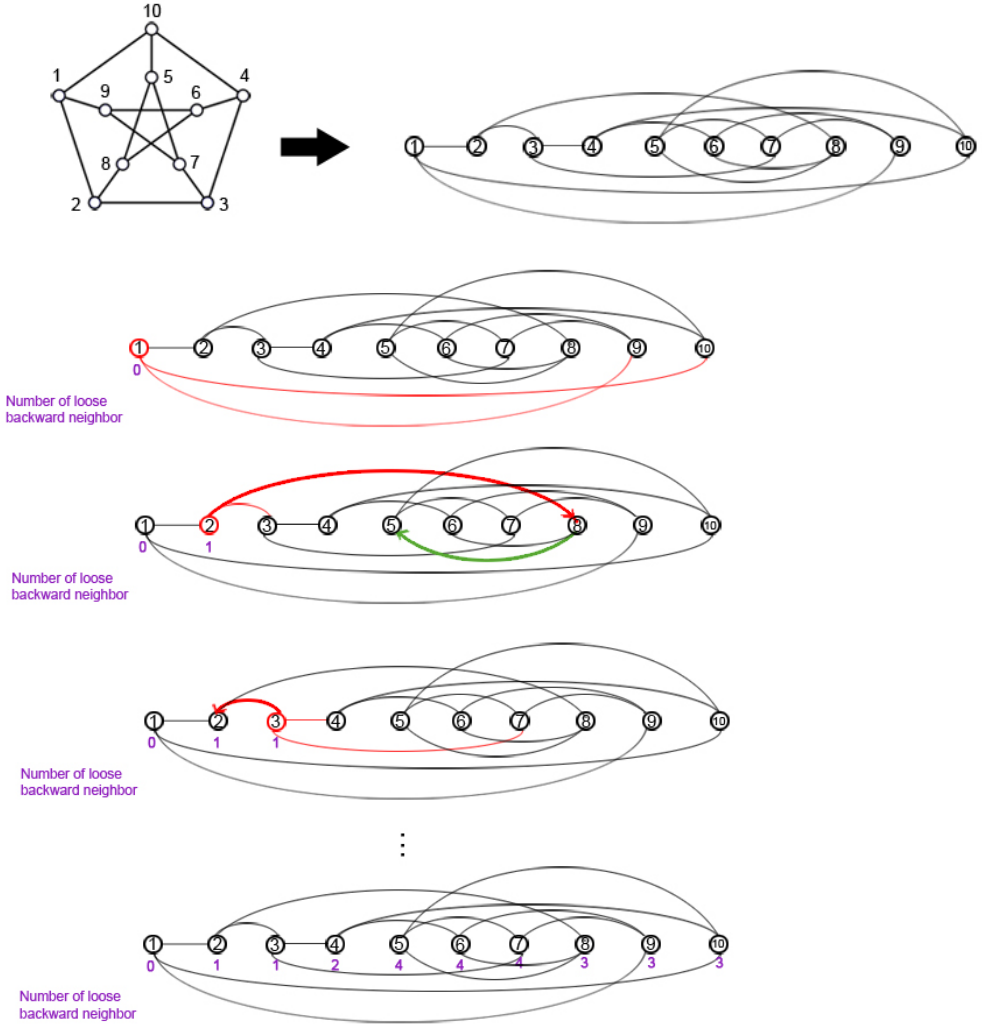
For the loose backward neighbor, we define an order of the graph G: $v_1v_2v_3...v_n$. A vertex v_h is a loose backward neighbor of v_i if $h < i$ and if v_h is in v_i neighborhood, or if v_h and v_i have a common neighbor v_j such as $h < i < j$.



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 a 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 $col_2(G)$.

2.1 First strategy

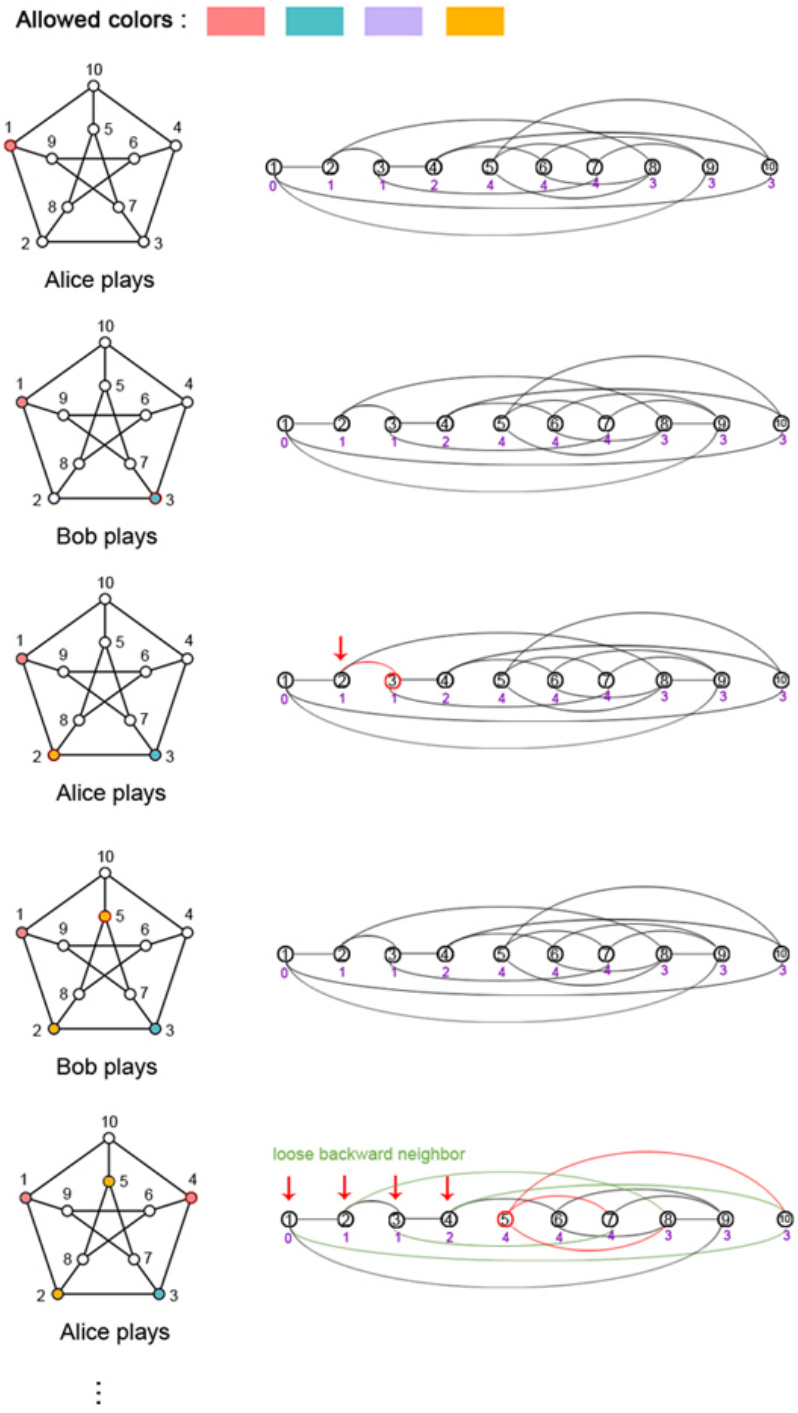
Alice determines a sequence of vertices of the graph G satisfying $col_2(G)$. The game begins only after.



Number of 2-coloring : k = 5

When Bob colors a vertex v_h, Alice's strategy is to seek in the order previously determined the loose backward neigbor of v_h not yet colored having the lowest index. Once found, it is colored. If Alice does not find such a vertex, she simply picks from the uncolored vertices set the one with the low-

est index in the order. According to this strategy, each graph G satisfies $\chi_g(G) \leq \chi(G) \times (1 + col_2(G))$.

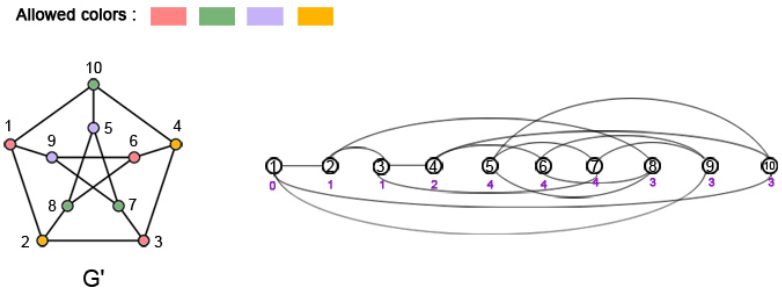


2.2 Second strategy

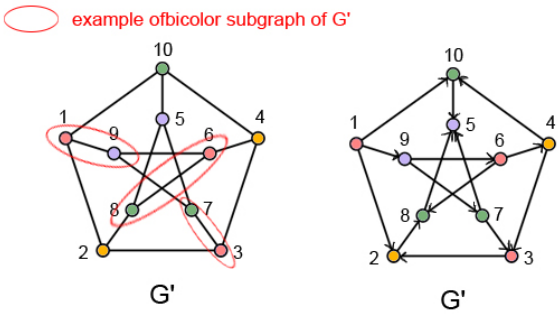
As in the previous strategy, Alice will once again need an order satisfying $col_2(G)$. Using this order, she will achieve her own coloring of the graph (without any intervention from Bob). This coloration must comply with the following rules:

- A vertex must not have the same color as its loose backward neighbors.
- Vertices forming a cycle must be colored with at least 3 colors.
- This coloring must use at most $col_2(G)$ colors.

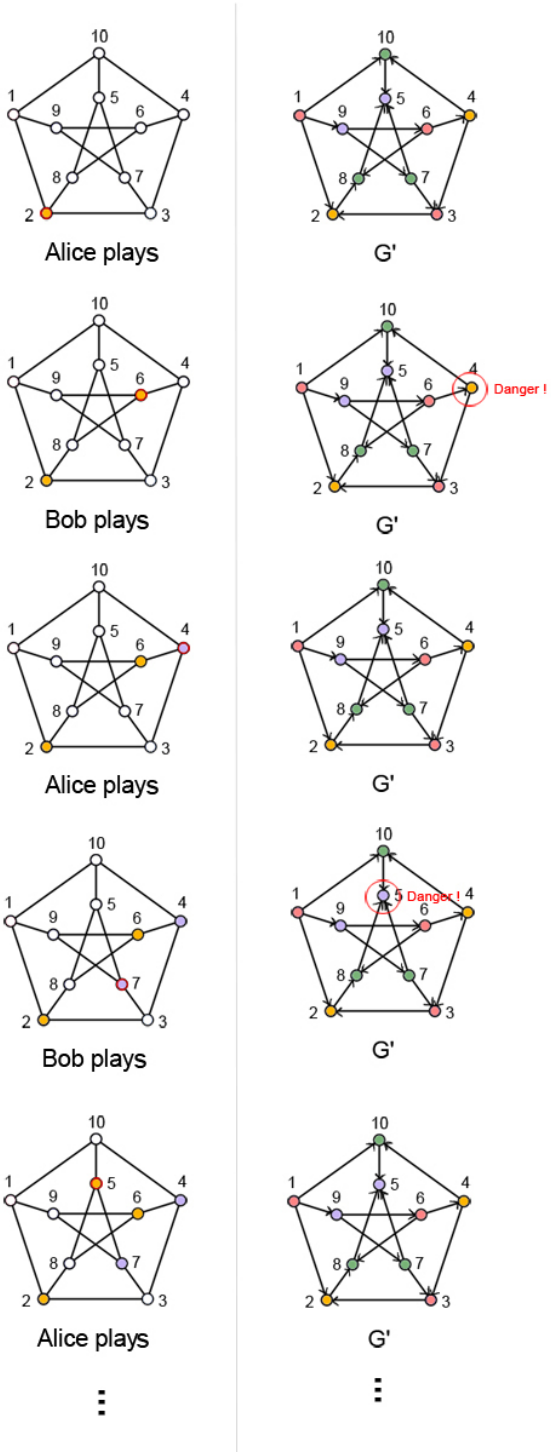
We will call the resulting graph G'.



Now Alice will use G' to provide guidance in the coloring of the graph G. For each bicolor subgraph of G', we direct the edges such that each vertex has an out-degree of no more than 1.



As each edge appears only in one subgraph, we know that we oriented the entire graph G. We can now begin the game. A vertex v (colored with a color c on the graph G') is endangered if one of its incoming neighbors is colored in a shade of c. Alice coloring the graph in accordance with the coloring of G', a vertex can only be endangered by Bob. In addition, the orientation of G' is such that there can be only one vertex in danger per round.

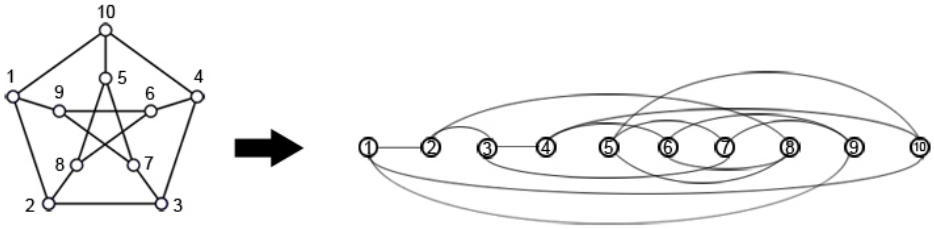


According to this strategy, each graph satisfies $X_g(G) \leq a(G) \times (a(G)+1)$, where $a(G)$, the acyclic chromatic number of G, is the minimum number of

colors used on G such that each cycle is colored with at least 3 colors.

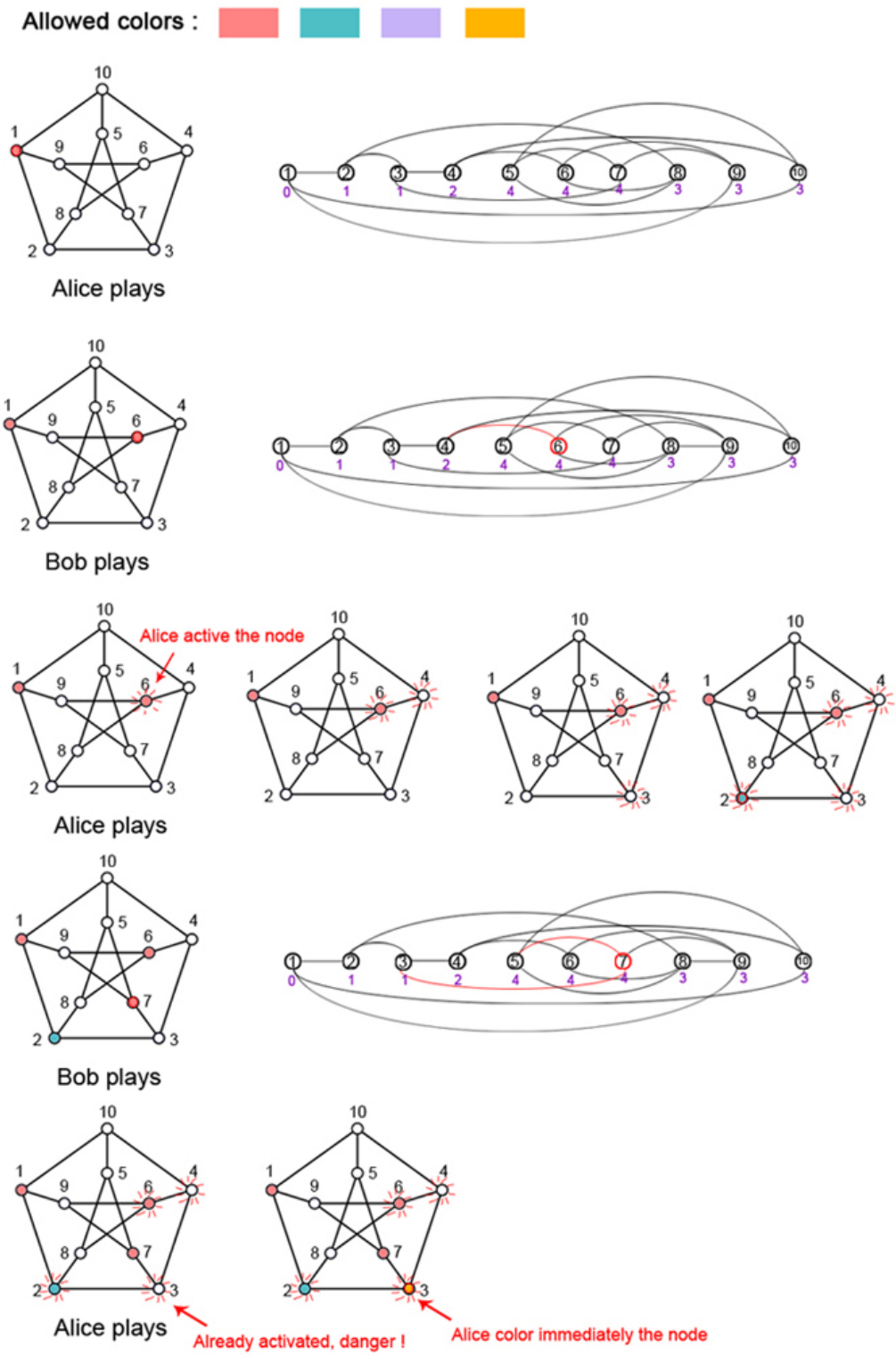
2.3 Third strategy - Daltonism may help

As with previous strategies, we order the vertices according to $col_2(G)$. We would like to draw your attention to the fact that we no longer use the loose backward neighbors, just the backward neighbors, i.e. neighbors with a lower index in the order.



In this case, we use a rather simple activation strategy. When Bob colors a vertex v, Alice begins by activating the vertex v, then she checks its backward neighbors. x is the backward neighbor of v having the lowest index in the order. If x is colored, we turn to the next backward neighbor of v. If x is activated, then it is in danger and we give it a color. If x is not activated, then we activate it and we check its backward neighbors. If x has no unshaded backward neighbor, then we color x. In the case where the vertex v (colored by Bob) has no unshaded backward neighbor, then we color the unshaded vertex having the lowest index in the order previously calculated.

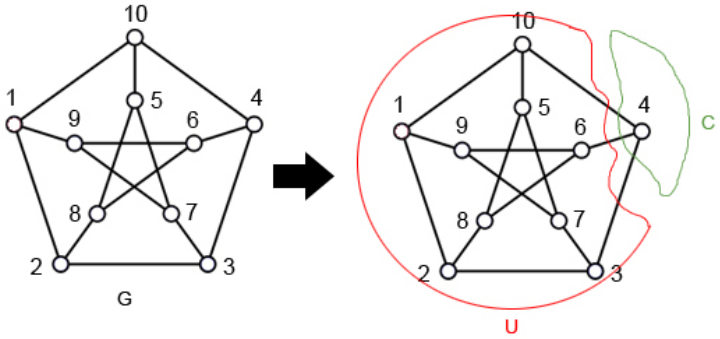
According to this strategy, each graph satisfies $\chi_g(G) \leq 3 \times col_2(G) - 1$.


Allowed colors :
Alice plays
Bob plays
Alice active the node
Alice plays
Bob plays
Alice plays
Already activated, danger !
Alice color immediately the node

2.4 Fourth strategy

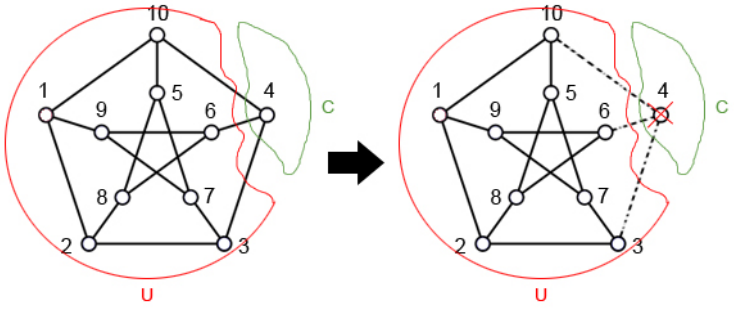
The purpose of this strategy is to find a better ordering of the graph G, on which we will use the activation strategy previously seen. We construct this order inductively (i.e. in the reverse order). The first selected node will be any vertex of G with a degree of at most 5. Suppose we have already taken the vertices $v_n, v_{n-1}, ..., v_{i+1}$ except v_i and we seek a candidate for v_i:

We split the vertices of G into two parts: $C = (v_n, v_{n-1}, ..., v_{i+1})$ and $U = V \setminus C$.

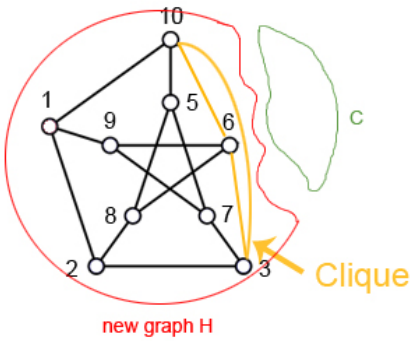


We construct a new graph H as follows:

- We remove the edges between the vertices of C.
- We remove each vertex v of C having at most three adjacent vertices in U.

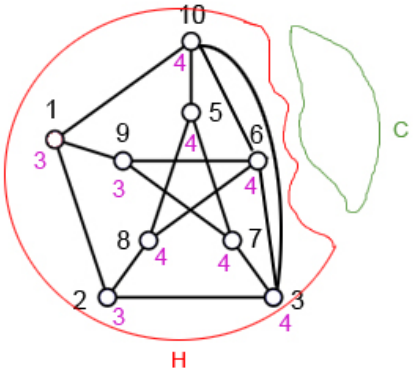


- For each vertex v removed, we add edges between its neighbors in U in order to form a clique.

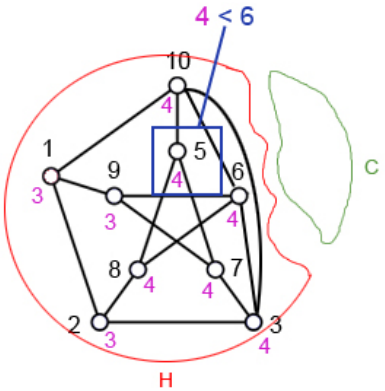


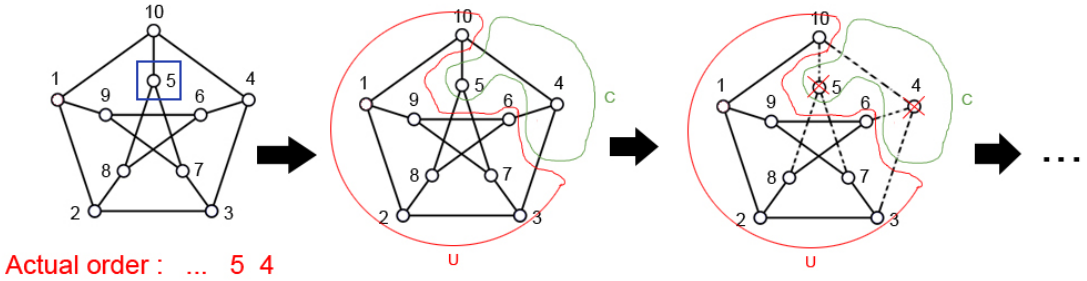
Now that we have our graph H, we assign to each edge e 2 dollars (USD) spread over its vertices as follows:

- If e connects two vertices of U, we give \$ 1 to each vertex of e.
- If e connects x in U and y in C, then we give \$ 0.50 to x and \$ 1.50 to y.



Once we have distributed the USD, we need to find a single vertex with a total value of less than \$ 6. It will be our v_i.





When we have our order, we use the color-blind activation strategy on it. According to this strategy, each graph satisfies $\chi_g(G) \leq 18$.

Chapter 3

Work

Contents

3.1 Objectives

After the study of the article, we agreed on implementing an AI to play the game. We wanted to know how would a computer program compete against the article's theoretical bounds. The first step was to choose the type of algorithmto use for this AI. Seeing how the move space was large (we were later comforted in this idea by our complexity study), Monte-Carlo Tree Search seemed to be the best choice for it allows good decisions without exploring the entire tree. Another important constraint was our inability to evaluate the profit of a game's position for a player before the game has ended. We could therefore not use more classical AI algorithms, where the tree search is stopped at a constant depth and the game state is evaluated before its end.

3.2 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 heuristic 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 (MCTS). It applies Monte-Carlo principles to tree 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 backpropagation. These 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 step produces children 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 by 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 nodes traversed. This is the crucial step to know what has been played and whether 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 selection algorithm can be one of the following (from [Cha10]) :

- 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 the highest number of games played.
- Secure Child : Choosing the node maximizing a lower confidence bound.

According to *Guillaume Chaslot* [Cha10] : "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 criteria 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.

3.3 How our AI works

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

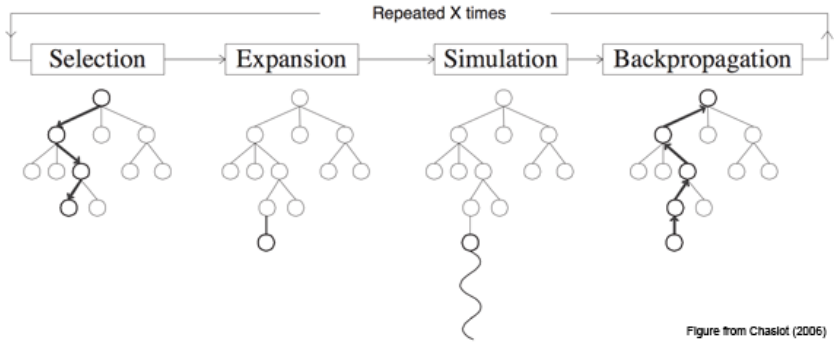


Figure 3.1: Steps of the Monte-Carlo tree search

- 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 : $\frac{games\ won}{games\ played}$.

The selection algorithm we used is based on multi-armed bandit algorithms as described in [Stu12]. 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 [Cha10]. 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 suits well 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-sum[1] game is called Minimax (established in [VN28]). When exploring a tree with

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

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 substracted 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 color 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.

3.4 Complexity

In this section, we present a short (and probably inaccurate) study of the space and time complexity of our implementation.

3.4.1 Space complexity of the search tree

We will first try to calculate how many different games are possible on a given graph. We will call v its number of vertices and c the number of colors used for the game. We have $c \leq v$.

Lets think about the way a map-coloring is played and the number of choices for each move. On the first move, v vertices can be colored with c colors, leading to vc choices. On the second move, $v-1$ vertices can be colored using c colors, for $c(v-1)$ choices. Up until the last move, where $(v-(v-1))=1$ can be colored with c colors. The total number of possible different games is :

$$\begin{aligned} cv * c(v-1) * c(v-2) * ... * c2 * c1 &= \prod_{i=0}^{i<v} c(v-i) \\ &= c^v \prod_{i=0}^{i<v} v-i \\ &= c^v * v! \end{aligned}$$

Considering the fact that introducing a new color in the game is isomorphic regardless of the color chosen, we can reduce the number of colors for a move to the number of colors already used plus one (the new color), up until c. This

leads to the following calculations :

$$\begin{aligned} 1v * 2(v-1) * ... * c2 * c1 &= \prod_{i=0}^{i<c}(i+1)(v-i) \prod_{i=c}^{i<v} c(v-i) \\ &= c!c^{v-c} \prod_{i=0}^{i<v}(v-i) \\ &= c!c^{v-c}v! \end{aligned}$$

Of course, not all of these games will lead to a proper coloring, so a certain amount will stop before coloring every vertex. We can use the assymptotic notation to describe that :

$$Number\ of\ possible\ games = O(c^v v!)^{2}$$

To calculate the number of nodes in the tree, we need to know the followings:

- The height of the tree is v.
- Its branching factor is at most vc

We can then define the number of nodes at a certain height as a geometric sequence whose common ratio is vc and start value is 1[3]. The total number of nodes in the tree is the associated geometric serie, defined as follow:

$$\sum_{i=0}^{i\leq v}(vc)^i = \frac{1-(vc)^{v+1}}{1-vc}$$

We can conclude that : $Number\ of\ nodes = O((vc)^v)$

3.4.2 Time complexity of a game

First of all, we will study the complexity of a single move. Playing a move consists of playing a certain number of simulations until the end of the game. Playing a simulation is composed of two parts : the selection algorithm and the moves themeselves. The selection algorithm complexity is $O(cv)$, because it selects a node from the possible moves by finding the maximum node from an unsorted list.

Now, let's take a look at the complexity of a simulated game. Until the end of the game, we select a node. The number of nodes uncolored after i turns is $v-i$. Thus, selecting a node at turn i is of complexity $O(c(v-i))$. This process is repeated until $i = v$, so the complexity of a simulation at turn i is:

$$\begin{aligned} \sum_{j=i}^{j<v}(v-j)c &= c \sum_{j=0}^{j\leq(v-i)} j \\ &= c\frac{(v-i)(v-i+1)}{2} \\ &= c\frac{(v-i)^2+(v-i)}{2} \\ &= O(c(v-i)^2) \end{aligned}$$

[2] We used this formula because it is simpler and gives the same order inside the O() notation
[3] The number of nodes on the level 0, where there is only the root

This simulation process is repeated n times before choosing a move to play. The complexity of a move on turn i is then:

$$C_{m_i} = O(nc(v-i)^2 + c(v-i)) = O(nc(v-i)^2)$$

Finally, the complexity of the entire game is:

$$\begin{aligned} C_g &= \sum_{i=0}^{i<v} nc(v-i)^2 \\ &= nc \sum_{i=0}^{i<v} (v-i)^2 \\ &= nc \frac{v(v+1)(2v+1)}{6} \quad \text{Remark : This is the sum of the } v \text{ first square numbers} \\ &= nc \frac{2v^3 + 3v^2 + v}{6} \\ &= O(ncv^3) \end{aligned}$$

We can see that this algorithm highly depends on the number of vertices in the graph, partly because of the selection algorithm. The total complexity could maybe drop to $O(ncv^2)$ if the selection algorithm was in $O(1)$. This could certainly be achieved by using the proper data structure to manage children of a node, like a maximum heap, or by sorting the children list. Those modifications would however introduce a small sorting time after each simulation, but far less than the linear selection algorithm.

3.5 Implementation of the experiments

In our test protocols, we used various graph classes that we chose for their particular topology or for difficulties encountered in their coloration or to represent common cases used in research. All these types of graphs combined cover most of the existing types of graphs.

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 many 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.

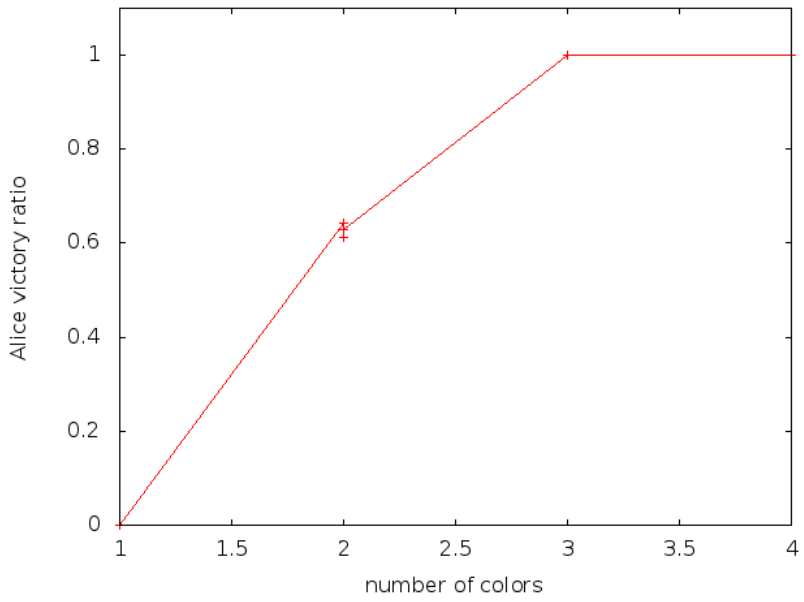
Here are the results:

3.5.1 Chains

The chains regularly appear in problems that involve calculations of paths. In addition, they provide interesting examples both for their topology and their simplicity.

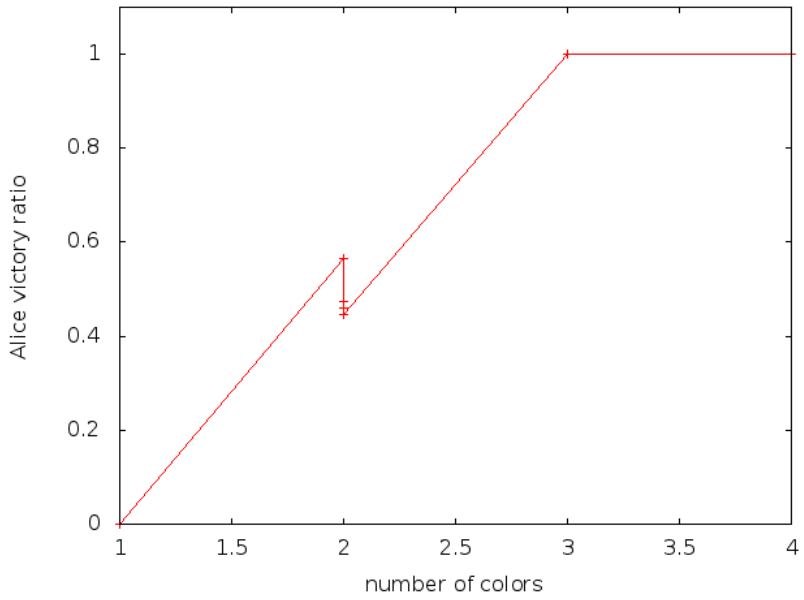
Odd number of vertices

Figure 3.2: Example of a chain with an odd number of vertices



Even number of vertices

Figure 3.3: Example of a chain with an even number of vertices



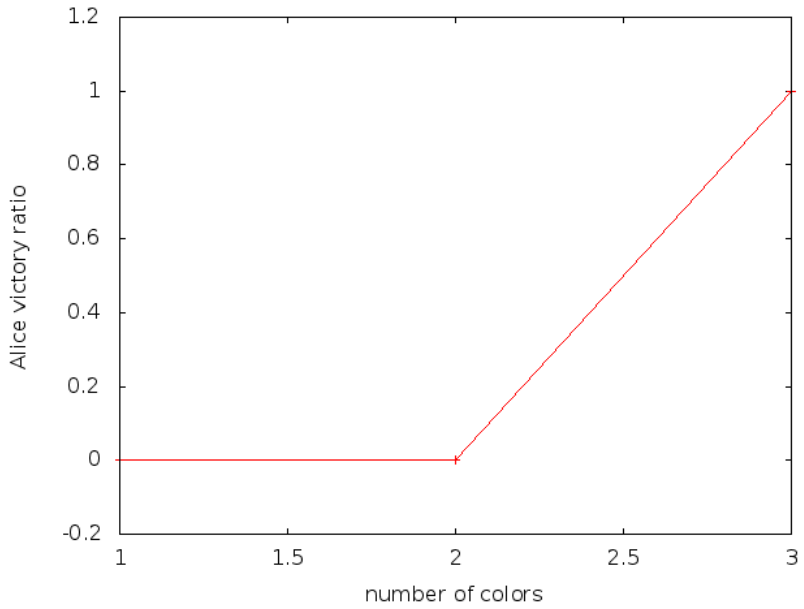
The chains are 2-coloriable graphs. However, we can see on those curves that 2 colors are not enough for Alice to win every time. Due to the simplicity of the graph, 3 colors seem to be sufficient regardless of the number of vertices in the graph. Moreover, on the two curves, we can clearly see that before the break point (3 colors) we have something relatively linear, meaning that Alice could hypothetically win every time with less than 3 colors.

3.5.2 Cycles

The cycles are used in famous problems such as the traveling salesman problem. They particularly interest us since they form a direct link with the practical applications of graph coloring problems.

Odd number of vertices

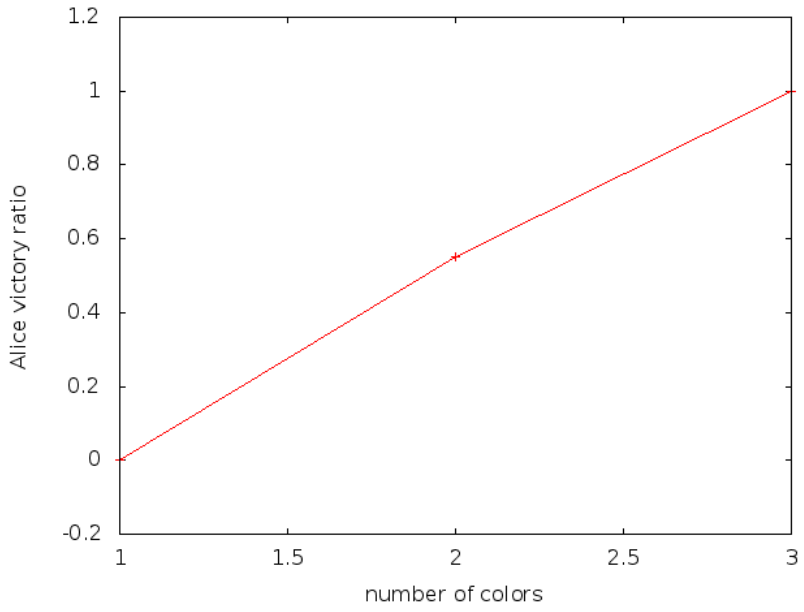
Figure 3.4: Example of a cycle with an odd number of vertices



An odd cycle is not 2-coloriable, but suprisingly Alice seems to win every time with only 3 colors, which is the minimal amount to achieve a proper coloring. This means that Bob cannot win on an odd cycle.

Figure 3.5: Example of a cycle with an odd number of vertices

Even number of vertices



An even cycle demands at least 2 colors to be properly colored. But this time, Alice only achieves a 0.5 ratio with 2 colors. She needs 3 colors to be able to beat Bob on every game. But seeing that there is no break point in the curve, we can not affirm that Alice can not win every time with 2 colors.

3.5.3 Grids

The grid is the type of graph which consumes the greatest amount of time in term of game simulation. This type of graph is mainly used in the fields of networking and communication. Toroidal grids are particularly relevant to the field of parallel computing.

2x5

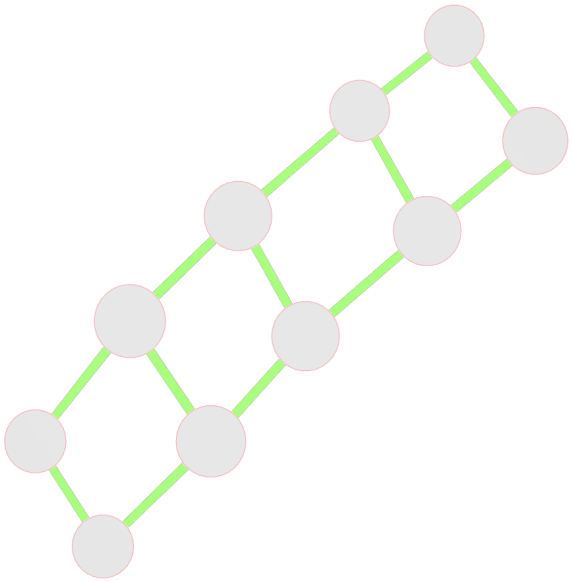
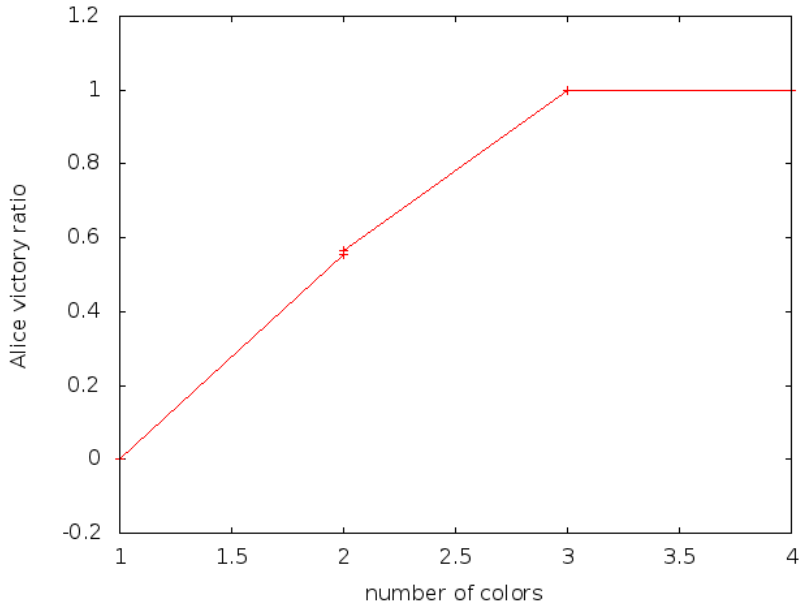


Figure 3.6: Example of a 2x5 grid



This little grid is quite similar to an odd cycle: 2-coloriable and Alice wins with 3 colors. This is propably because of the little size of the grid.

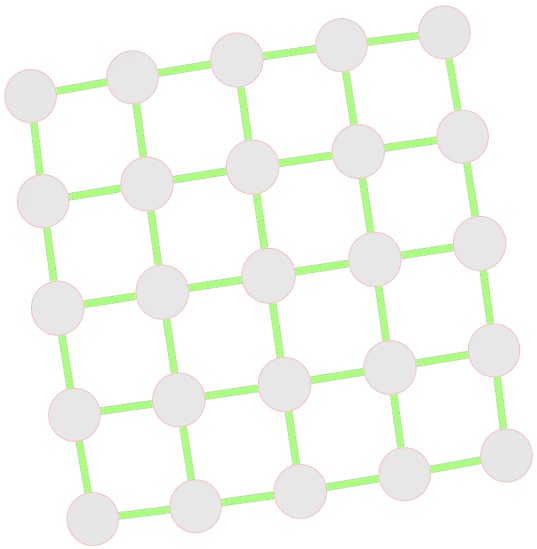
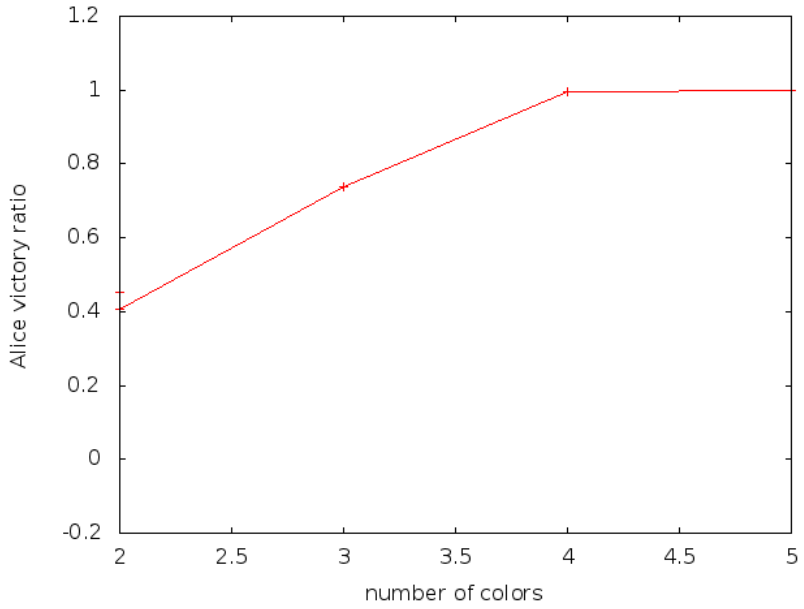


Figure 3.7: Example of a 5x5 grid

5x5



This grid is bigger than the previous one. This can be seen on the results : while still being 2-coloriable, our Alice needs at least 5 colors to be able to win every time. Moreover her ratio with 3 colors is only about 0.7 and with 4 colors it is very close but still under a perfect 1 ratio. From this we can conclude that the game chromatic number of this grid is certainly around 4 or 5.

5x5 toroidal

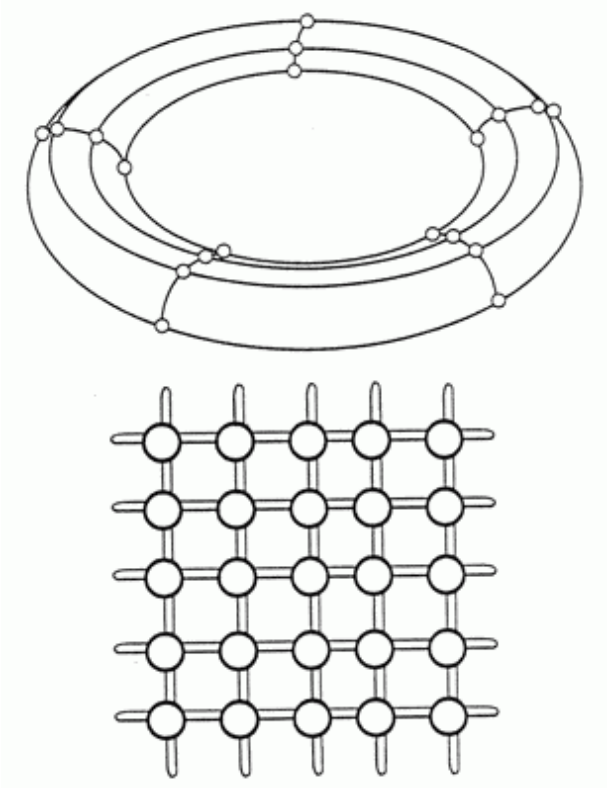
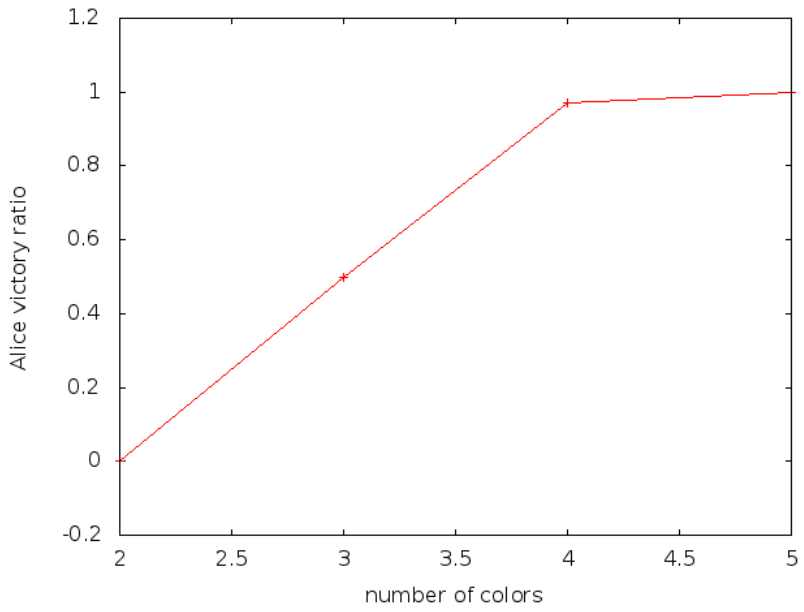


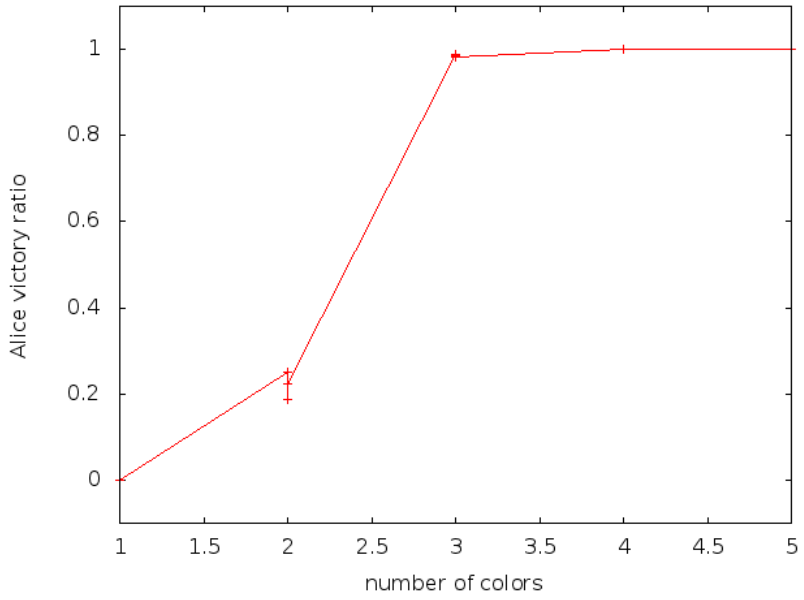
Figure 3.8: Example of a 5x5 toroidal grid



The toroidal nature of this grid implies that its chromatic number is 3. The game chromatic number of toroidal grids has been proved to be 5 in [RW09]. This result is confirmed by our data. Indeed, 5 is the least number of colors for wich Alice achieves a 1 ratio.

3.5.4 Binary Tree

Figure 3.9: Example of a binary tree



In the case of the binary tree, Alice obtains a ration of 1 with 4 colors, but the ratio is very close to 1 with 3 colors. In addition, we observe a break point on the curve at 3 colors. We can therefore assume that the game chromatic number of this graph could be between 3 and 4.

3.5.5 Non-Planar Graphs

Petersen Graph

The Petersen graph is named for *Julius Petersen*, a famous Danish mathematician. The Petersen graph is a special case which is reputed to constitute a good counterexample to many theories that work on other graphs.

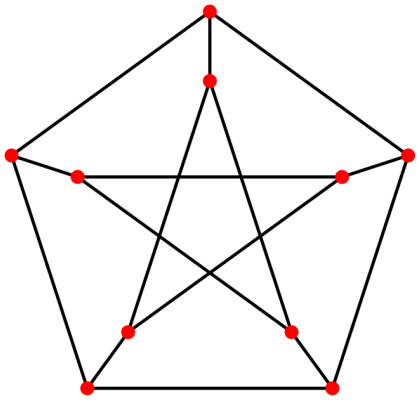
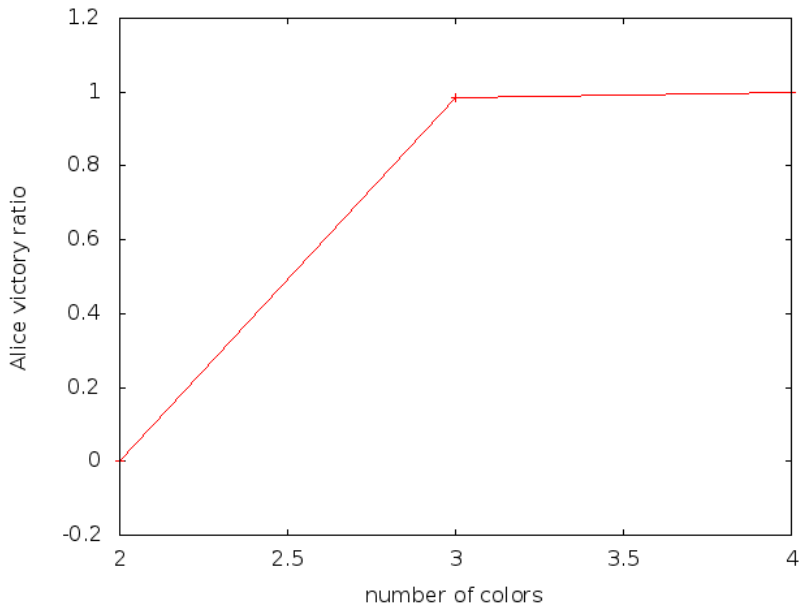


Figure 3.10: A Petersen graph



Alice needed 4 colors for the perfect ratio, but 3 is not far behind. However, 4 is still far under the bounds of the original article. In addition, we observe a steep curve between 1 and 3 colors. We can therefore assume that 3 colors would be a good candidate to define a new limit for the game chromatic number of this type of graph.

Icosahedron Graph

The Icosahedron is often found in biology in viral structures. We are interested in the particular topology.

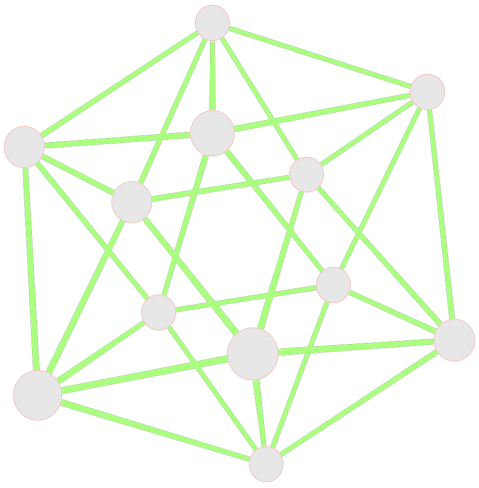
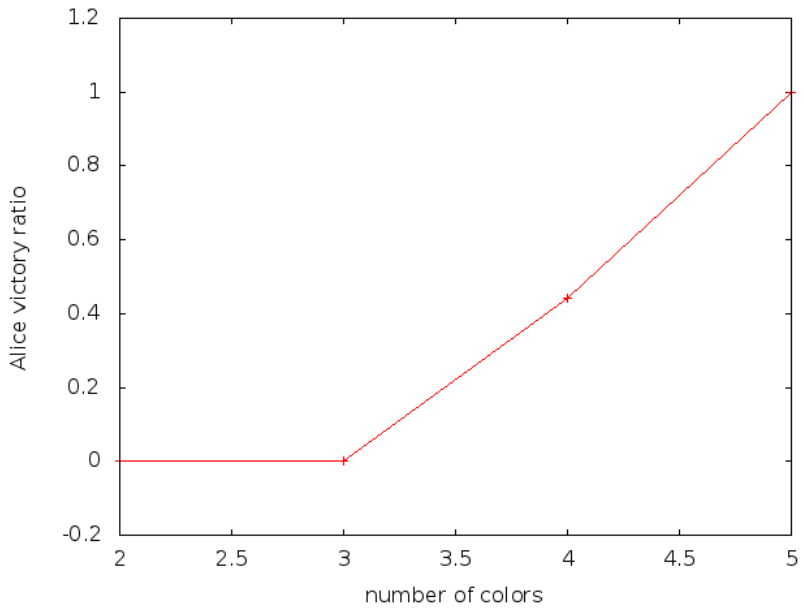


Figure 3.11: An Icosahedron graph



This graph seemed to be the most difficult for Alice. Her ratio with 4 colors is only 0.4. She wins every time with 5, but the bounds of the article are greater. Maybe the real game chromatic number is between 5 and the article's bounds.

Grotzsch Graph

The Grotzsch graph is a triangle-free graph with 11 vertices, 20 edges, and a chromatic number of 4. It is named for *Herbert Grotzsch*, a German mathematician.

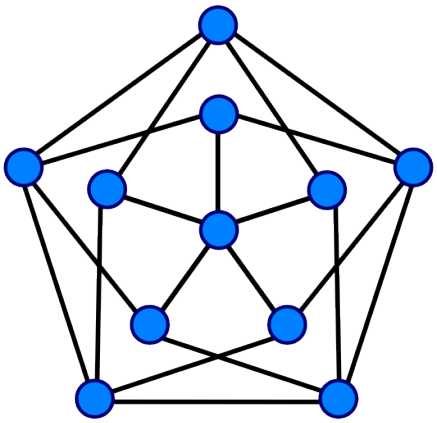
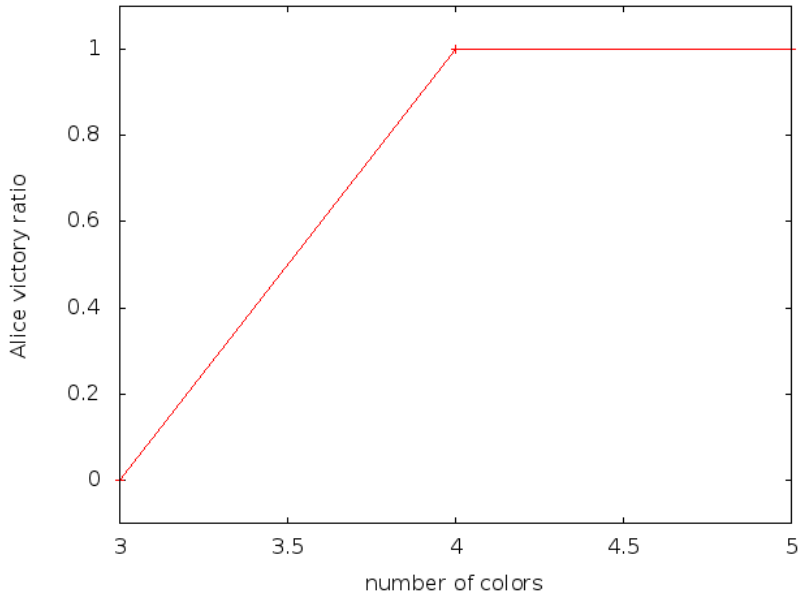


Figure 3.12: The grotzsch graph



We can see on the curve that Alice achieves a ration of 1 with 4 colors, which corresponds to the chromatic number of the graph. We would have an optimal game chromatic number.

3.6 Results summary

We recall that the game chromatic number of toroidal grids has been proved to be 5 in [RW09]. The Grotzsch graph was added later. We were therefore unable to calculate the theoretical values of each strategy in time.

Graph	Chromatic number	Strategy 1	Strategy 2	Strategy 3	Experimental results
Chain (odd)	2	6	6	5	3
Chain (even)	2	6	6	5	3
Cycle (odd)	3	12	12	8	3
Cycle (even)	2	8	12	8	3
Grid 2x5	2	6	12	5	3
Grid 5x5	2	10	12	11	5
Grid 5x5 tor	3	5	5	5	5
Binary tree	2	6	6	5	4
Petersen	3	18	12	14	4
Icosahedron	4	32	20	20	5
Grotzsch	4	?	?	?	4

We observe that the experimental values are far below the theoretical values, especially regarding the Petersen graph and the icosahedron.

Chapter 4

Conclusion

Contents

4.1 Assessment

Overall, we can say that the results are quite positive. Indeed, we were able to meet the original objectives and our results are better than what we expected.

We believe it is possible that there is a theoritical model which outperforms those presented in the article we studied.

4.1.1 Experimental biases

We have reason to believe that our protocols of experimentation can be improved to provide more reliable results:

- We can not make Alice or Bob stronger. If we increase the number of simulation performed by Alice, Bob will also increase the number of simulations that he performs, thus increasing its strength as much as that of Alice. Furthermore, the goal is to see whether Alice can win regardless of the strength of Bob, but our Bob is not infaillible.
- Unlike the strategies of the article, our AI adapts itself based on the simulations it performs. Where a theoretical model always redo the same thing which seems to be the best choice, our AI adapts itself and will *know* if a move leads to a victory or not, thus being able to out of an impasse.

4.2 Prospect

For a more complete study, it would be interesting to perform simulations on other graphs and to be able to generate the theoretical calculated values on the spot, instead of calculating by hand.

To gain in performances and to accelerate the experiments, we could parallelize the simulations.

It would be interesting to look a little more into Bob's strategies. We considered that Bob and Alice played symmetrically (if a move is bad for Alice, then it is good for Bob), but a bad move for Bob on the short term may be good for him in the end.

Bibliography

[BGKZ07] Tomasz Bartnicki, Jarosław Grytczuk, H. A. Kierstead, and Xuding Zhu. The map-coloring game, 2007.

[CBSS] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai.

[Cha10] Guillaume Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Université de Maastricht, 2010. http://www.unimaas.nl/games/files/phd/Chaslot_thesis.pdf.

[CM07] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithm for tree search. Technical report, INRIA, 2007.

[CSB+06] 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.

[CWB07] G. M. J. B. Chaslot, M. H. M. Win, and B. Bouzy. Progressive strategies for monte-carlo tree search, 2007.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[RW09] Andre Raspaud and Jiaojiao Wu. Game chromatic number of toroidal grids. *Information Processing Letters*, 109:1183 – 1186, 2009.

[Stu12] 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.

[VN28] J Von Neumann. *Zur Theorie der Gesellschaftsspiele*. 1928.

List of Figures

Studies and research project report.
The map coloring game.
Master 2, 2012

The sources of this report are freely available on http://github.com/AlexPerrot/map-coloring.