**Artificial Intelligence and Knowledge Engineering Laboratory**

***Vladyslav Gavryliuk 245603***

**Assignment 2**

**CSP (Constraint Satisfaction Problem)**

***Theory***

In general, a CSP is a problem composed of a finite set of variables, each of which has a finite domain has a finite domain of values, and a set of constraints. Each constraint is defined over some subset of the original set of variables and restricts the values these variables can simultaneously take. The task is to find an assignment of a value for each variable such that the assignments satisfy all the constraints. In some problems, the goal is to find all such assignmets.

1. **V** = { v ,..., v } is the set of **variables**.
2. **D** = { D ,..., D } is the set of **domains**. Each domain is a finite set containing the possible values for the corresponding variable.
3. **C** = {C ,..., C } is the set of **constraints**. A constraint C is a relation defined on a subset {v ,..., v }of all the variables, that is D v...v D Ê C .

\* [chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/http://www.cs.toronto.edu/~fbacchus/Papers/liu.pdf](chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/http:/www.cs.toronto.edu/~fbacchus/Papers/liu.pdf)

\*[chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/https://core.ac.uk/download/pdf/301666759.pdf](chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/https:/core.ac.uk/download/pdf/301666759.pdf)

***The aim :***

To satisfy all constraints (in our case that neighbours cannot be the same color), all points have to be connected (at most Variables – 1, at least 1 if size less than 2, otherwise at least 2)

For, example red can have connections with green, blue, yellow but cannot be merged with red.

***Comparing the algorithms.***

**Backtracking :**

The backtracking algorithm enumerates a set of *partial candidates* that, in principle, could be *completed* in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of *candidate extension steps.*

Conceptually, the partial candidates are represented as the nodes of a [tree structure](https://en.wikipedia.org/wiki/Tree_structure), the *potential search tree.* Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.

The backtracking algorithm traverses this search tree [recursively](https://en.wikipedia.org/wiki/Recursion_(computer_science)), from the root down, in [depth-first order](https://en.wikipedia.org/wiki/Depth-first_search). At each node *c*, the algorithm checks whether *c* can be completed to a valid solution. If it cannot, the whole sub-tree rooted at *c* is skipped (*pruned*). Otherwise, the algorithm (1) checks whether *c* itself is a valid solution, and if so reports it to the user; and (2) recursively enumerates all sub-trees of *c*. The two tests and the children of each node are defined by user-given procedures.

Therefore, the *actual search tree* that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is the number of nodes of the actual tree times the cost of obtaining and processing each node.

\* <https://en.wikipedia.org/wiki/Backtracking>

**Forward Checking**:

Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables. We speak about restricted arc consistency because forward checking checks only the constraints between the current variable and the future variables. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

**AC:**

Forward checking checks only the constraints between the current variable and the future variables. So why not to perform full arc consistency that will further reduces the domains and removes possible conflicts? This approach is called **(full) look ahead** or **maintaining arc consistency** (MAC).

The advantage of look ahead is that it detects also the conflicts between future variables and therefore allows branches of the search tree that will lead to failure to be pruned earlier than with forward checking. Also as with forward checking, whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no necessary.

\* <https://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html>

***Task1***

***Calculations***

Variables = (red, green, blue, yellow)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| BT | 13 | 18 | 17 | 20 | 26 | 37 | 47 | 43 | 58 | 122 | 146 |
| FC | 5 | 14 | 15 | 17 | 24 | 29 | 35 | 38 | 59 | 107 | 134 |
| AC | 1 | 4 | 12 | 16 | 23 | 27 | 32 | 34 | 36 | 43 | 55 |

To have more precisely better calculations we could add 1 additional color to variables. So, Let’s append this one as “black”.

Variables = (red, green, blue, yellow, black)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| BT | 21 | 48 | 61 | 119 | 368 | 833 | 1888 | 3307 | 4517 | 6403 | 10410 |
| FC | 6 | 21 | 53 | 97 | 341 | 496 | 1023 | 2518 | 4128 | 5619 | 6567 |
| AC | 1 | 6 | 15 | 22 | 30 | 44 | 49 | 50 | 56 | 51 | 57 |

Maximum time of execution of the program is :

****

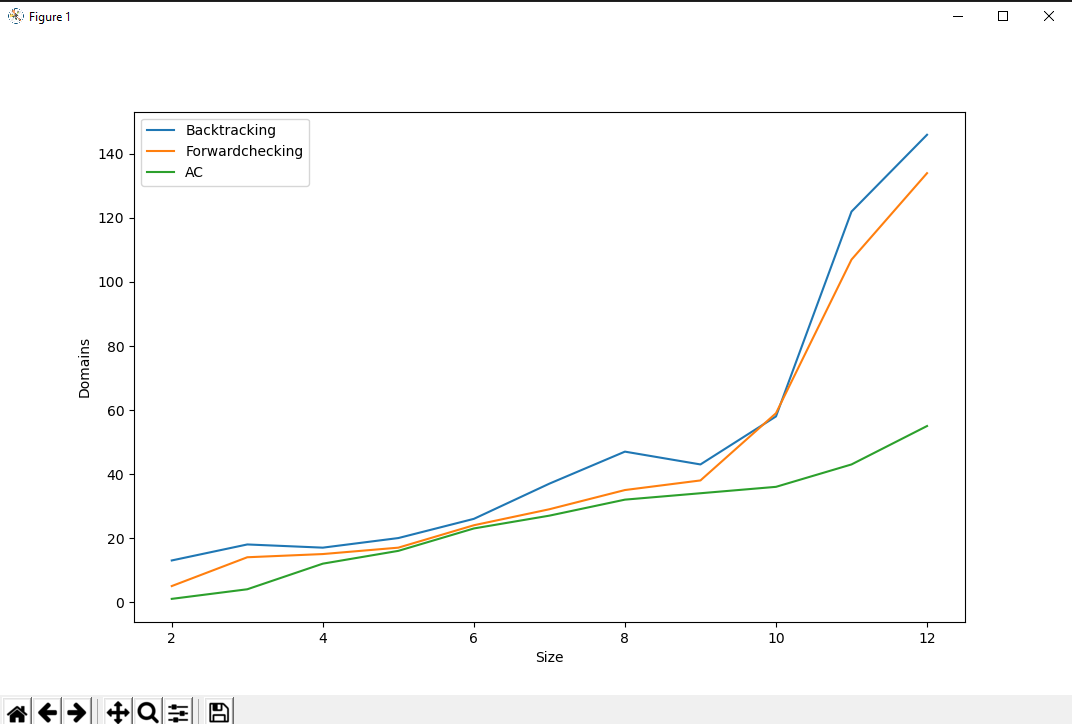
****

Measured by seconds.

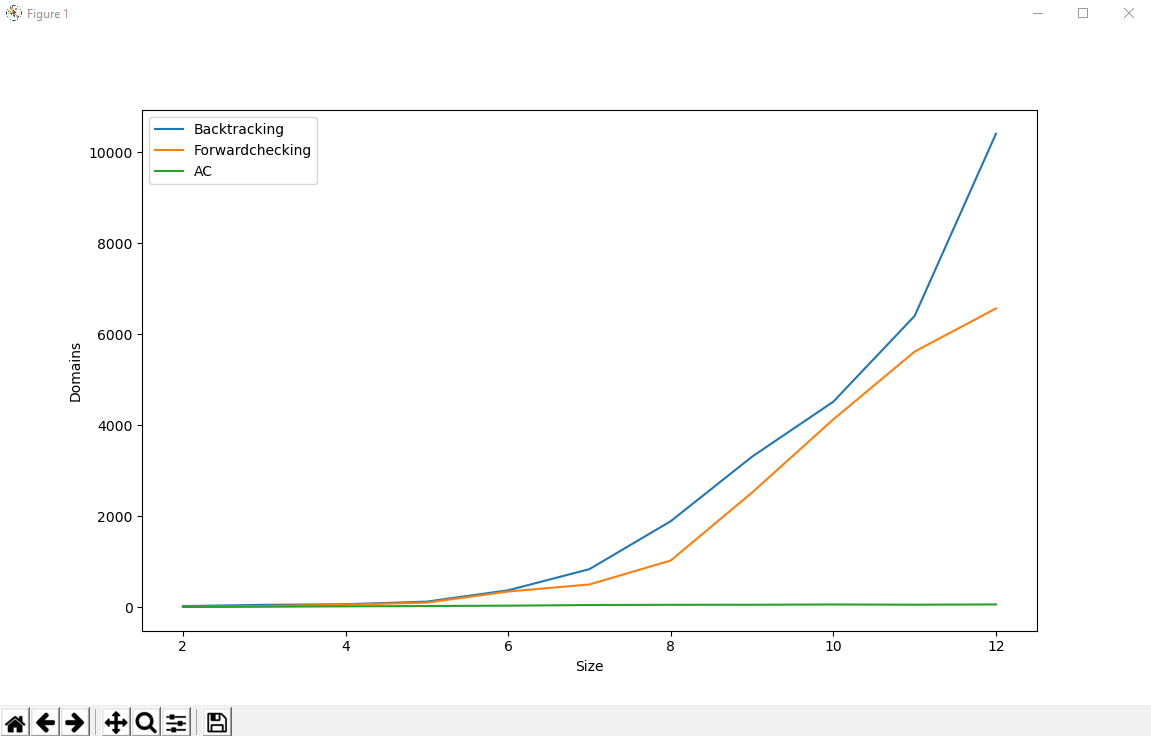
The screenshots above describes the results of backtracking algorithm, as it is the slowest one.

***Plots***

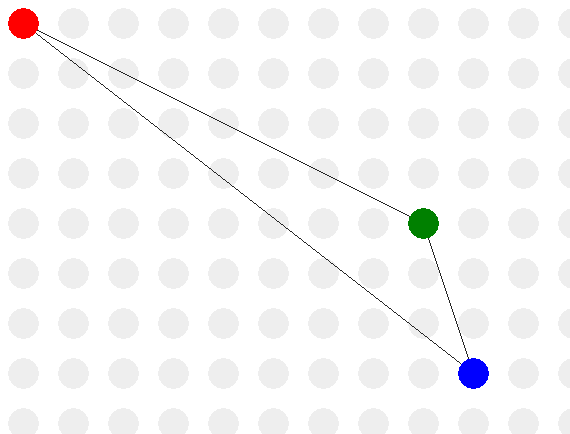
Variables = (red, green, blue, yellow)



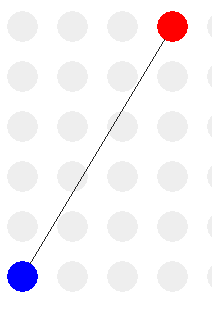
Variables = (red, green, blue, yellow)



***Results***



Size = 2

 Size = 3

From those images, we can be sure that algorithm takes firstly the color in the array with index 0 and increase it by 1 to selecting further colors (index += 1).

Array = [red, blue, green, yellow, black]

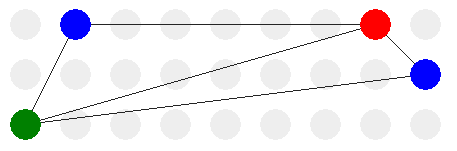
1 -> red

2 -> blue

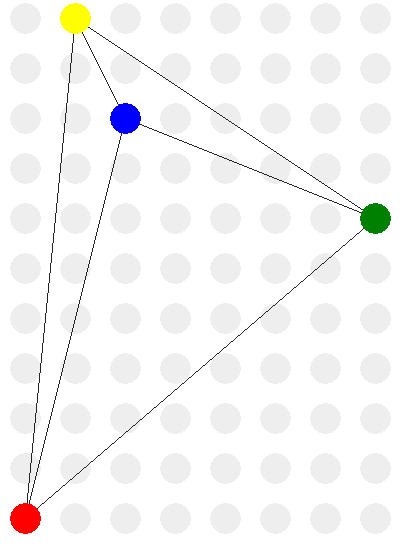
3 -> green

Now, I would like to compare Backtracking and Ac algorithms for the size equals 4.

**Ac**



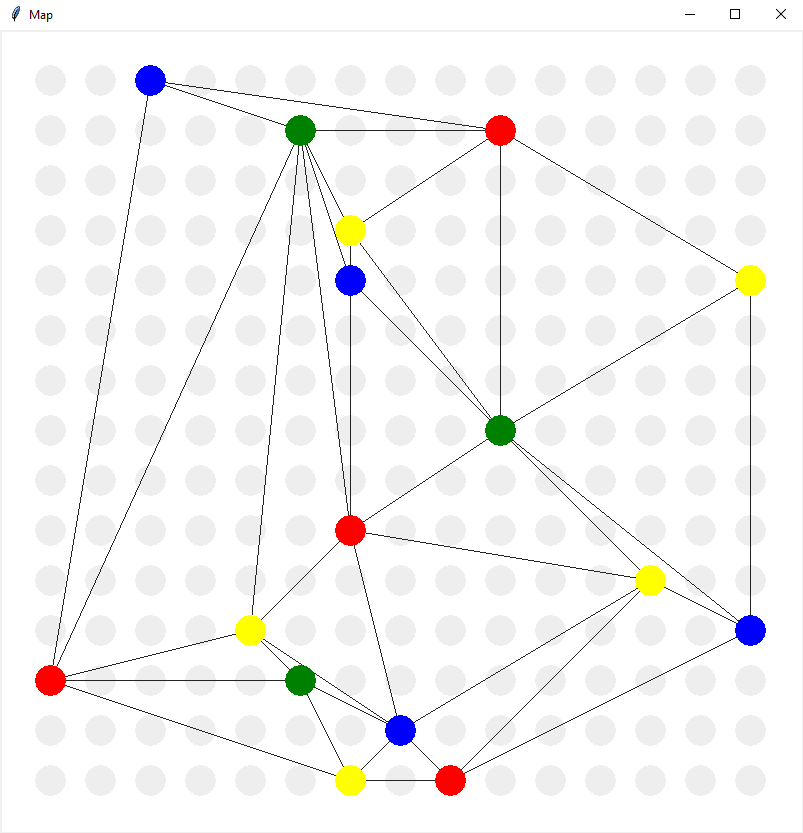
**Backtracking**



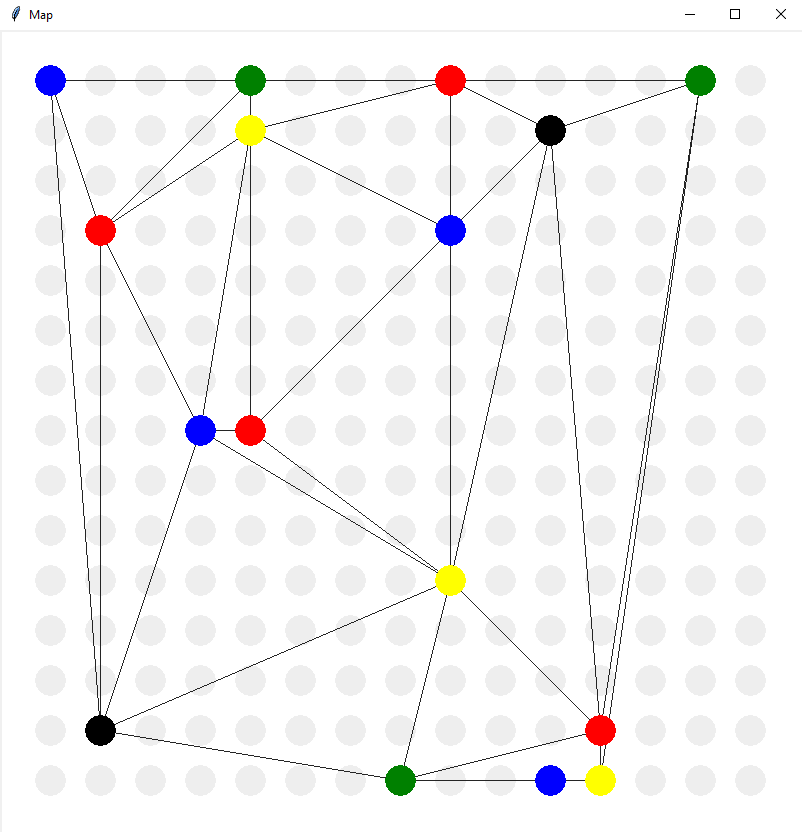
As we can see the major difference which appears is that in Ac algorithm it looks in such way, that if there is possibility to add already used color it will be considered, however the backtracking adds those colors which were not used (in most cases). That’s one of the reasons why we speed up the program. Moreover, the most important reason of speed up of execution is that we omit inconsistent solution immediately in Ac comparing with other ones.

The same performance we can obtain for size = 16.

**Ac**



**Backtracking**



Regarding some comments due to the largest size. For n = 16 it works fine, but if we want to increase the amount It can leads to some problems, the most popular I’ve received is “list index out of range”.

It means that the algorithm cannot find the solution, as the reason of random connections between the points (the lack of free colors which may be used to solve the problem). The distance between points is measured by **euclidean distance.**

* <https://en.wikipedia.org/wiki/Euclidean_distance>

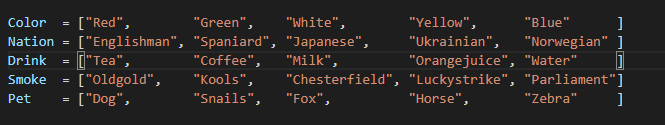
***Task 2***

By Einstein’s problem we have to find who drinks ***water*** and who owns the ***zebra***

Zebra puzzle

\* <https://en.wikipedia.org/wiki/Zebra_Puzzle>

To seed data :



To add and get variable

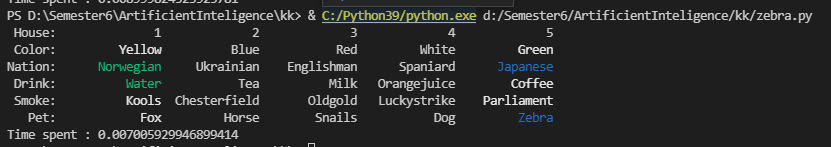


To add constraint (a and b describes the houses)

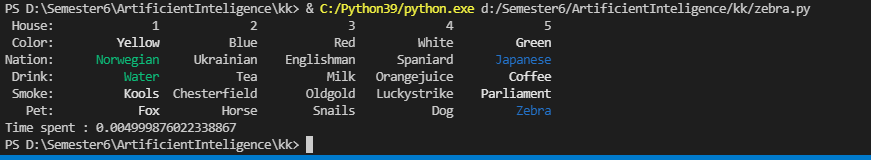


The result for backtracking :

Norwegian drinks water, Japanese owns the zebra.



The result for forward checking:



Let’s create a table with 5 repeats of each algorithm to see more precisely time measurements

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 1 | 2 | 3 | 4 | 5 |
| Backtracking | 0.0089 | 0.0099 | 0.0079 | 0.0084 | 0.0073 |
| Forward check | 0.0051 | 0.0049 | 0.0039 | 0.005 | 0.004 |

\* <https://labix.org/python-constraint>

\* <https://labix.org/doc/constraint/>

***Smile, because this is the end 😊***

***With best wishes,***

***Vladyslav Gavryliuk.***