Software report

Buciu Ștefan, ICA 256

1. Sudoku

* Problem definition

Sudoku is a puzzle where a 9x9 grid is given and the goal is to fill it with values from 1 to 9 such that there are no repeated values on rows, columns, and inside the 3x3 squares.

* CSP representation

We can represent the Sudoku problem as follows:

* Variable: every cell will have a variable assigned, thus where
* Domain: every variable can have values between 1 and 9, thus
* Constraints: the constraints are imposed by the puzzle rules
  + No value can be repeated on a row
  + No value can be repeated on a column
  + No value can be repeated in a 3x3 square
* Solution
* Backtracking:

Given a board, we recursively try all possible values for each empty cell such that no constraint is violated.

This method is implemented in the class *SudokuBacktracking* in the Java source code provided.

* Problem Reduction (node consistency):

Given a board, precompute the domains for each variable and we consider values only from the computed domains. For example, if a board is given with value at position , we then remove the value from all domains of the variables on the same row, column, and in the same 3x3 square. With this approach, when a value is chosen for an empty cell, it will not waste resources on computing a solution using the value .

This method is implemented in the class *SudokuPRNodeConsistency* in the Java source code provided.

* Basic Search Strategy (forward check):

This approach follows a similar idea with Node Consistency, by updating the domains once a variable is committed to. Thus, whenever we start committing a new value, the domain will be up to date with all the previous choices made.

This method is implemented in the class *SudokuSSForwardCheck* in the Java source code provided.

* Analysis

TODO

1. Maximum Cut

* Problem definition

The Maximum Cut problem is NP-complete, and it aims to partition an undirected graph into two sets such that the number of edges that cross the cut is maximized (i.e., one end of the edge in each set). We will be trying to solve this problem version, however there is a more general version where the edges have a positive weight assigned.

* CSP representation

We can represent the MaxCut problem as follows:

* Variable: given a graph with a set of vertices , a variable will be assigned for each vertex.
* Domain: after a cut, a vertex can be part of a set , thus the domain for each variable is .
* Constraints: after a cut, a set of edges will be selected . Given a selected edge , then the vertices connected by it and must have variables assigned such that .
* Solution
* Backtracking:

We solve MaxCut with backtracking by iterating over the given edges and keeping track of the partitioning. If we select edge , then each vertex is added to a set and . However, since the graph is undirected, we check both configurations (i.e., adding to S and to T).

A solution is correct if the constraints are satisfied (i.e., ). If we have a solution, we then compute the size of the cut by counting how many edges have the first vertex in a set, and the second in the other set.

This method is implemented in the class *MaxCutBacktracking* in the Java source code provided.

* Problem Reduction:

1. Pruning

To help reduce the problem space, we can use a pruning approach where given an intermediate cut such that , and an edge then .

This method is implemented in the class *MaxCutPRPruning* in the Java source code provided.

1. Arc consistency

Given a set of chosen edges, we build an agenda with both edge directions (i.e., ). Initially, all vertices can be part of either set. Until the agenda is empty, we select an edge (i.e., constraint) and check if the current domains of are violated. If the domain changes, we then add all arcs that have the vertex in question on the right side.

After that, we then check if the configuration is valid and try to update the max cut.

This method is implemented in the class *MaxCutPRArcConsistent* in the Java source code provided.

* Basic Search Strategy (forward check):

Initially, all edges are considered. During backtracking, when we commit to an edge, when then filter out all edges that would violate the committed edge. The resulting filtered collection is then passed to the next backtracking iteration.

This method is implemented in the class *MaxCutSSForwardCheck* in the Java source code provided.

* Analysis

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **G2 (8 nodes, 14 edges)** | **G3 (6 nodes, 9 edges)** |
| Backtracking | 26349 ms | 124 ms |
| Pruning | 5728 ms | 43 ms |
| Arc consistency | 219 ms | 18 ms |
| Forward check | 1016667 ms | 115 ms |

From the tests performed, the *arc consistency* variant is the fastest. However, as graphs grow larger, all algorithms have a long running time. This is due to the complexity of the problem and the backtracking approach. The *forward check* variant is the slowest and that’s because on every backtrack iteration, it has to update the domain of possible edges.