AI programming: Module 2
Constraint-Satisfaction Problem and Best-First search

Bjørnar Walle Alvestad

2015

Norwegian University of Science and Technology

NTNU

Introduction

This report will describe essential details of the A*-GAC algorithm implemented in module 2 of the AI programming course. This module consists of combining constraint satisfaction problems with best-first search, using the A* algorithm implemented in module 1 of this course. The produced algorithm will be used to solve a variety of vertex coloring problems.

This module is implemented using the Java programming language. It uses code from module 1 of this programming course.

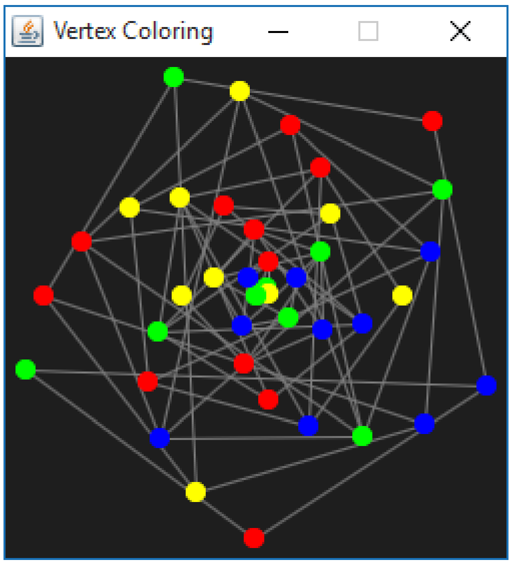

Figure 1: A fully colored graph.

Generality of A*

The core code of the A* algorithm is left unchanged in this module. All code geared towards CSP and GAC are implemented in the *CspState* class, which is a subclass of the abstract class *AstarState* defined in module 1. *CspState* implements the following methods which will be used by the A* algorithm:

- `float heuristic()`
  The heuristic is defined as the sum of the domain size of every variable in the CSP, minus 1. That means the heuristic will be calculated to 0 when all variables are singleton sets.
- `boolean isSolution()`
  A state is considered a solution if all the variables are singleton sets, and the number of unsatisfied constraints equals 0.
- `boolean isContradictory()`
  Method for checking is this state is contradictory. A state is defined as contradictory if any one domain is empty, meaning there is no valid value to be assigned for a given variable.
- `AstarState[] generateChildren()`
  This method generates successor states by performing assumptions matching the value of a given variable. The algorithm selects only one variable to generate assumptions, starting with the variable associated with the first node in the input graph. If a variable has more than one value in its domain, a successor state will be generated for each value, and the assumption forced by setting the domain to only contain the assumed value. GAC-rerun is then invoked on the newly generated state.
  Successor generating is problem specific. The mechanism described above is implemented in the class *VertexColorState*, which is a subclass of *CspState*. A default method for successor generating is defined in *CspState*, which will generate successors for every variable in the current CSP.
- `int id()`
  This method will produce a unique integer identifier for a given state. The id is determined by a series of hash functions used on the constraints, variable names and the content of their domains. It is guaranteed that two equal states will produce the same identifier.
- `float arc_cost()`
  The arc cost is set to 1 in all situations. This reflects that one guess is needed to advance to the next state.

Generality of A*-GAC

The A*-GAC algorithm is completely separated from the problem specific code. The required functionality from A*-GAC is imported to the vertex coloring problem, and is not modified in the problem specific context.

This implementation of A*-GAC operates on integer domains, meaning no objects can be inserted in a variable domain. The GAC filtering algorithms reside in the class *Gac*. It takes a CSP state object as argument, and offers methods for solving the CSP using various calls to the domain filtering loop. Domains of the provided CSP state object may be reduced by the GAC algorithms.

Constraints are specified as strings, and can be defined dynamically at application runtime. The constraint system supports the following operations within a constraint:

- Additive **+ -**, multiplicative *** / %**
- Relational **< > <= >=**, equality **== !=**, ternary operator **?:**
- Bitwise operations: AND **&**, OR **|**, XOR **^**, NOT **~**
- Logical AND **&&**, logical OR **||**, logical NOT **!**

Additional parenthesis and integer constants can be specified within the constraint string. Variable names are denoted by characters, and are case sensitive. Variable names may not contain numbers, as they will be interpreted as integer literals.

Code chunks

This module utilizes a third party library called *Compilation Toolbox*[1] for compiling and loading java classes at runtime. When a constraint is created, the constraint string is inserted into a class template string which implements the interface *CodeChunkCore*.

```
public interface CodeChunkCore {
    public boolean exec(int… args);
}
```

Figure 2: The interface CodeChunkCore.

Consider the constraint `C > A + B`. When entered into the code chunk generator, it will produce java source code equal to what is shown in figure 3. Values from the argument list are assigned to the variables in alphabetical order.

The code is dynamically compiled and loaded back into the application. Compiling each constraint into java bytecode means significant performance improvements over interpreters and parsers. The constraint can be tested by invoking the `exec` method in the newly compiled *CodeChunkCore*. It has return type boolean.

```
package astargac.codechunk;
public class CodeChunkCoreWrapper
implements CodeChunkCore {
   public boolean exec(int... args) {
      int A=args[0], B=args[1], C=args[2];
      return C > A + B;
   }
}
```

Figure 3: Java code produced for the constraint C > A + B.

[1] Compilation toolbox: https://code.google.com/p/compilation-toolbox/

Constraint network

The constraint network contains all original constraints, variables, domains and the relation between them. The constraint network is implemented in the class *Cnet*. It has functionality for adding new constraints, and assigning values to the variable domains. Variables are created automatically when a new constraint is referencing them.

The A* search algorithm will work with *CspState* objects, not constraint networks. Once the constraint network is created, and all constraints and values specified, the initial *CspState* object can be created using the *generateCspState()* method. This state will be passed to the A* algorithm. CSP-states contains *VariableInstances* and *ConstraintInstances*. These are specialized versions of the regular *Variable* and *Constraint* classes, as their domain values will be altered by the GAC filtering. Variables and constraints contained in the constraint network will never be modified. Because the CSP-state regularly will be used to generate successor states, they are easily cloned using the appropriate copy constructor. Variable instances, their domains, the constraint instances and the relation between VIs and CIs are properly copied when generating successor states.

References

Downing, Keith L., *AI Programming (IT-3105) Project Module #2: Combining Constraint-Satisfaction Problem-Solving with Best-First Search.* [Online].

Downing, Keith L., *Essentials of the A* Algorithm.* [Online].