AI Programming: Assignment 2

Eirik Vågeskar

October 6, 2015

1 Generality of the A* Implementation

The implementation of A* used here is the exact same as in Assignment 1, and the arguments for this implementation's generality can be found that report. What follows is an explanation of how the domain specifics of a constraint satisfaction problem are handled through external methods:

- Search nodes: Search nodes are objects of the CSProblem class.
- *goal_test*: The argument passed to the parameter *goal_test* is the static method *all_domains_have_size_one* of the CSProblem class. It returns true if all domains of a CSProblem instance are of size one.
- *get_successors*: The generation of successors is handled by the method *get_successors* of the CSProblem instance.
- *move_cost*: Implemented as an anonymous function returning 0. The move cost is not very relevant for the CSPs, as the guiding heuristic overestimates the number of nodes to be generated and is thus not admissible.
- *heuristic_function*: The heuristic function is implemented as the method *domain_sizes_minus_one* of the CSProblem instance.

2 Generality of A*–General Arc Consistency Algorithm

This section explains the essentials of the CSProblem class and elaborates on the previous section.

The CSProblem class has three fields: *domains*, a Python dictionary whose keys and values are the variables and their domain respectively; *constraints*, containing instances of the Constraint class (see next section); and *queue*, a Python *deque* (FIFO queue) which contains *(focal variable, constraint)* tuples to be fed to the *revise* algoritm. The class also contains implementations of the *initialize*, *domain_filtering* and *rerun* algorithms described on p. 5 of the task description.

Elaboration of methods described in last section:

- *get_successors*: finds the variable with the smallest remaining domain, unless the domain size is 1. Generating successors for domains of size 1 leads to dead ends. Sizes above 1 are suitable for a guess. A domain with size 0 is a sign of a state with violated constraints, and will be pruned away because it can not generate any successors.

  The method generates successors by making a copy of the current CSProblem instance for each value of the selected variable's domain. Each copy is assigned one of the possible values as the only value of the selected variable's domain. Each copy's rerun method is run before it is returned.

- *heuristic_function*: This is the static method *domain_sizes_minus_one* of CSProblem, which returns the sum of the number of values in every variable's domain, minus one for every variable. This is based on the heuristic given in the task. There is a little tweak in the implementation, due to which the method will return $+\infty$ if any domain has a size of 0. This places the CSProblem instance last in the agenda. The consequences is that the algorithm spends less time removing states that are dead ends, but this also increases the size of the open set in the A* algorithm.

3 Constraint Generation and Evaluation

Note: The order of this section and the next is swapped when compared to the task description

A constraint is contained in an instance of the class Constraint. The constructor has three parameters: *func_var_names*, the variables that take part in *expression*; *expression*, an expression that can be used in an anonymous function; and the optional *actual_var_names*, which are the names of the variables.

These parameters are used to generate an anonymous function through Python's *eval*, which is stored in the Constraint's *constraint_formula* field. *actual_var_names* is stored in the Constraint's *variables* field. These can be used by the CSProblem for looking up the variable's domain in the CSProblem's *domains* dictionary. A constraint saying that nodes with keys '0' and '1' should not be of the same value could be created with the call *Constraint(['x', 'y'], 'x != y', [0, 1])*.

Exactly how such constraints are generated from a problem specification, is a matter that is handled through sub-classing of the CSProblem class. When the sub-class has been able to make a constraint specification that can be passed to the Constraint constructor, it calls the CSProblem superclass' *add_constraint* method. This constructs a Constraint and adds it to its own list of constraints.

As a fun addition, constraints can even be passed as a string in this format: `"x y; x==y; 0 1"` (0 and 1 are variable names). These are parsed by CSProblem's *create_constraint_from_text* method, which calls *add_constraint* when the string has been parsed. This makes it possible to add constraints via the command line.

4 Clean Separations of Constraints and Instances of Constraints and Variables

The separation of the constraints, constraint instances and variable instances is kept clean by, respectively: (1) A unique constraint object is created for each constraint in a problem and is stored in a Python list (which is a collection of pointers). (2) The constraint instance, the actual content of the constraint, is stored within this constraint object. (3) The variable instances, the current domains of the variables, are handled by only storing the variable name. This is used for look-up at the time of evaluation in the individual CSProblem instance.

One further note on the separation of variable instances: When a successor of a CSProblem instance is generated, a copy of the entire CSProblem instance is created through the use of the Python *copy* module's *deepcopy* function. The copying process makes a copy of the CSProblem and all its data structures recursively, ensuring that no manipulation of a copy affects any aspect of the original.