Algorithm

Chun-Yen Ho

August 5, 2011

1 Basic Concept

We rewrite the DeltaBlue algorithm to increase efficiency and to reduce the complexity of the implementation. We can divide the differences between DeltaBlue and our Model into two part, simpler constraint hierarchy and a different API.

The original DeltaBlue solver supports different constraint strengths. But in our model, we only use two type of constraints, namely required constraints for specifying relationship of variables, and stay constraints to decide which variable to change.

Rather than the four kinds of basic operations used in the original algorithm, add/remove constraints and add/remove variables, The API of our model consists of two parts, changing priority and removing conditional constraint. The first one corresponds to user's operations, and the second one is for computing the desired constraint relationship.

To take advantage of these differences, we make the changes described in the following section.

2 Changing Priority

When user operates on the interface and causes the value of variable to change, the only effect it has on the underlying constraint graph is to raise of the strength of variable priority. In the original DeltaBlue algorithm, this should be done in two steps, remove the original stay constraint and then add a new stay constraint with higher strength. In our algorithm, we combine these two operations.

We can separate this operation into two cases, depending on whether the stay constraint is enforced or not. If the stay constraint is not enforced, then we directly add a new constraint and start propagating the new walkabout strength. If the stay constraint is enforced, by careful observations, we can prove that the only effect on the underlying constraint graph is to change the walkabout strength of variables.

Consider all downstream constraints of the modified variable. In the process of propagating walkabout strength, the input priority will be modified before the output priority. Thus, when choosing the output method, the original output variable, whose walkabout strength was the minimum, will still be the minimum one, and become the new output. Also notice that if a stay constraint is enforced then the variable will have no other upstream constraint. Since all possible constraint will have the same method before and after the operation, the underlying plan will not change.

Below is the pseudocode of our algorithm.

```
user change the state of $var_i$;
$pri_i++$;
addConstraint($var_i$'s stay constraint);

addConstraint(cn)
{
newMethod = $\arg\min_{method \in cn}$ method.output.strength;
cn.currentMethod = newMethod;
cn.currentMethod.output.strength = $\min_{method \in cn - \{currentMethod\}}$ method.output.strength;
foreach nextCn that connected to cn.currentMethod.output do
    if nextCn != cn then
        addConstraint(nextCn);
    end
end
}
```

Algorithm 1: Raising priority

Note that in addConstraint(), when there is no conflict, newMethod will be the same as the old one. Thus it will only propagate the variable strength.

3 Removing unused conditional constraint

The original DeltaBlue algorithm treats conditional constraints as constraints whose input is the union of all variables in the condition and input variable in the method. Though this can guarantee a correct fulfilment of constraints, the result may not be desired. Since when the condition of the conditional constraint is not true, conditional constraint would have no effect to the output variable. But in DeltaBlue algorithm, this is not reflected in the underlying graph. Since some unnecessary stay constraints are enforced, the number of stay constraints may be less than desired.

To address this problem, we remove the conditional constraint when the condition is not true. We accomplish this by calling the remove constraint method in DeltaBlue. But again to take advantage of the model we use, we implement it with a slightly different algorithm.

In DeltaBlue algorithm, removing a constraint is done in 3 steps, collecting all unenforced downstream constraint, removing the constraint and trying to enforce each of the downstream constraint by adding them consecutively. In our algorithm, we propagate the changes of walkabout strength through the downstream variables until finding a variable that does not change, and trying to enforce each stay constraint on the path. This is correct because that all the unenforced constraints in our algorithm will be stay constraint, and when there is a variable does not change, the walk about strength of that variable will be the same as the one before we remove the constraint. If any downstream stay constraint is enforceable, it should be already enforced.

Thus, the algorithm for removing conditional constraint is as follows.

```
The condition of a conditional constraint cn is not true;
if cn has not been enforced then
    return;
else
    oldOutput = cn.output;
    oldOutput.strength = oldOutput.stayConstraint.strength;
    foreach nextCn that connected to oldOutput do
        if nextCn != cn then
            addConstraint(nextCn);
        end
    end
    propagateChange(oldOutput);
end

propagateChange(var)
{
foreach downstream constraint cn of var do
    oldOutput = cn.output;
    oldStrength = cn.output.strength;
    oldOutput.strength = oldOutput.stayConstraint.strength;
    foreach nextCn that connected to oldOutput do
        if nextCn != cn then
            addConstraint(nextCn);
        end
    end
    if oldStrength != oldOutput.strength then
        propagateChange(oldOutput);
    end
end
}
```

Algorithm 2: Removing conditional constraint

The addConstraint() here is the same as the one in previous algorithm. Note that enforcing a stay constraint is equal to set the output strength to the strength of that stay constraint.