Fakulti Teknologi Maklumat dan Sains Kuantitatif

Jilid 3, Bil. 1, Sept. 2001

JURNAL TEKNOLOGI MAKLUMAT DAN SAINS KUANTITATIF

KANDUNGAN **Muka Surat**

Universiti Teknologi MARA

SOLVING A BACKTRACKING INTELLIGENTLY: A CASE STUDY

Muthukkaruppan Annamalai
Jabatan Sains Komputer
Fakulti Teknologi Maklumat dan Sains Kuantitatif,
Universiti Teknologi MARA (UiTM)
40450 Shah Alam, Selangor

ABSTRACT

This paper studies a special kind of Constraint Satisfaction Problem (CSP) related to a case of resource allocation. It attempts to study the procedure of allocating a collection of resources to a group of contenders, in a satisfactory manner. In this study, we have modeled the CSP as a constraint graph. This allows us to device an augmented backtracking algorithm that could be employed to find a solution for this CSP. There are three important features of this algorithm that we would like to highlight here, namely I) minimally backtracking to resolve the constraint violation, II) making use of the information associated with a failed search to prune the search space and III) not repeating a computation. This algorithm was tested on the New Vehicle Sales System application (Mohd Zamri, 1998) and has been found to consistently produce the desired result.

Keywords: Constraint satisfaction problem; Ordered intelligent graph; Intelligent backtracking

Tel: +603 5543 5470
e-mail: mk@tmsk.itm.edu.my

THE PROBLEM

During the recent boom period, the demand for the Malaysian-made new *Proton* cars invariably far outpaces its supply. In a typical situation, a buyer places an order with a car dealer by describing the car model, up to three preferred car colors, and additional accessories for his pick. In a day, the dealer receives tens of orders. The car dealer maintains separate waiting lists for the different types of the *Proton* car models. A new order is appended to one of those waiting lists. At the end of each day, the dealer tenders the new orders to the *Proton* Car Company.

Periodically, the *Proton* Car Company delivers a batch of new cars of a certain type of model to the car dealer. Each of these cars is fitted with standard accessories and has been given a designated body color coat in the factory. Unfortunately, the new cars cannot be assigned to the buyers' orders on a particular car model waiting list, in a First-Come-First-Serve basis because the body color of the cars received normally does not follow the sequence of the orders on that list. Clearly, a dealer's is mainly interested to accept all the new cars in the batch and allot them quickly to some ready buyers. So, the dealer somehow tries to link each of the new cars in the batch to some orders on the waiting list, based on the car color preference information alone. Upon finding a match, the selected buyer's choices of additional accessories are fitted to the car before it is delivered to him.

As business was done this way, many *Proton* car buyers complained about having to wait very long time before taking delivery of their new car. There were allegations of favoritism and assisted queue jumping on the dealer's waiting list. Intentionally done, or otherwise, queue jumping was hard to trace and controlled in a manually managed allocation system.

In order to restore the customer satisfaction, the *Proton* Car Company realized that there is a need for a control mechanism to ensure a fair allocation of new cars to buyers on the car dealer's waiting line. This study was initiated to find a solution to this problem.

THE CSP MODEL

A CSP is stated as follows. We are given a set of variables, a finite and discrete domain for each variable, and a set of constraints. Each constraint is defined over some subset of the original set of variables that limits the combinations of values that the variables in this subset can take. The goal is to find an assignment to the variables such that the assignment satisfies all the constraints.

The car allocation problem described in the previous section can be cast as a resource allocation CSP. The resource allocation process we see here is the task of assigning each of the cars in a batch of new cars of a specific model to orders on the car dealer's waiting list in a fair manner. Therefore, we will attempt to satisfy a subset of car orders on a particular waiting list, on a First-Come-First-Serve basis, by assigning a preferred car color to each of the orders in that subset, from a set of available car colors.

The CSP has a variable for each of the car orders on the dealer's waiting list. The domain of each variable is the choice of preferred car colors named by the buyer. A constraint between a pair of variables is said to exist when two buyers have booked for a particular car model with the dealer and each has requested for at least one common color as their preferred car color. In fact, there can be many such constraints in a CSP, each one defined over some subset of the original set of variables.

For example, Table 1 shows part of the entries in a *Proton Wira 1.5* car model waiting list. A buyer's first, second and third choice of preferred car color is represented by fields: Color 1, Color 2 and Color 3, respectively. Notice that buyers 001, 002 and 003 are vying for a green colored car; buyers 001 and 002 are also vying for a blue colored car; buyers 004 and 005 are vying for a white colored car.

Table 1: The First Five Entries In A Dealer's *Proton Wira 1.5* Car Model Waiting List

Buyer ID	Model	Color 1	Color 2	Color 3
001	Wira 1.5	green		
002	Wira 1.5	blue	green	
003	Wira 1.5	red	blue	green
004	Wira 1.5	white		
005	Wira 1.5	white		

A CSP can be depicted using a constraint graph, in which each node represents a variable, and each arc represents a constraint between variables. Further, a node contains the domain values of the variable it represents. The equivalent constraint graph for the CSP described above is shown in Figure 1.

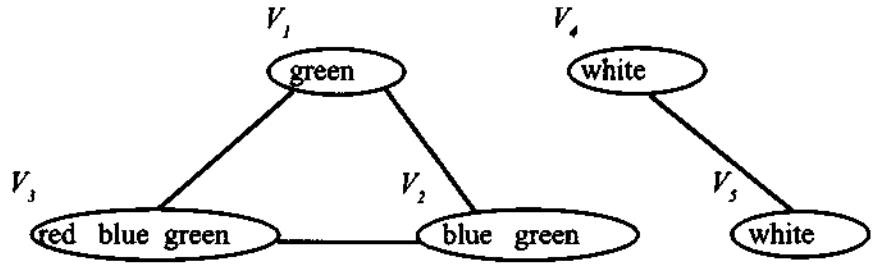

Figure 1: The Constraint Graph For The Five Entries In Table 1

An important point to note is that we have to allocate the cars to the buyers on a First-Come-First-Serve basis. Therefore, the nodes in the constraint graph should be ordered linearly, so that, the variables that appear earlier in the ordering could be instantiated first.

As a matter of fact, the constraint graph shown in Figure 1 should be redrawn as in Figure 2. This type of graph is called an ordered constraint graph.

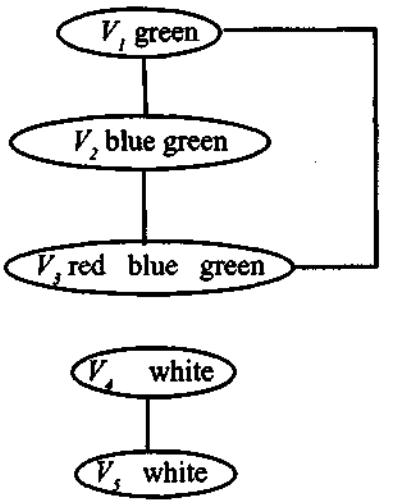

Figure 2: An Ordered Constraint Graph For The Five Entries In Table 1

The goal of the CSP is to find an assignment to the variables such that the assignments satisfy all those constraints. The goal should serve both the car dealer and the buyers in the waiting line, on even terms. Recall that the dealer's main interest is to accept all the new cars in the batch and allot them quickly to some ready buyers; and the buyers want their car to be delivered to them as quickly as possible.

Consequently, we will use the following constraints to realize our goal. They are attempting to:

- allocate ALL the cars in the batch of new cars to some orders on a particular waiting list.
- allocate the cars to the buyers according to the sequence of their orders on the waiting list.

The following distinctive examples will illustrate the solution(s) for the CSP represented in Figure 2.

Example 1: Supposing a batch of three new *Proton Wira 1.5* cars constituting of one blue, one green and one red car is delivered to the car dealer. The CSP has one solution: V_1 = green, V_2 = blue, and V_3 = red.

Example 2: Supposing a batch of three new *Proton Wira 1.5* cars constituting of one blue and two green cars are delivered to the dealer. In this case, we have two solutions. The

first solution: V_1 = green, V_2 = green, and V_3 = blue; and the second one: V_1 = green, V_2 = blue, and V_3 = green. We can choose either one of them to resolve the CSP.

Example 3: Supposing a batch of three new *Proton Wira 1.5* cars constituting of one green, one red and one white car is delivered to the car dealer. Here, the assignments will be done as follows: V_1 = green, V_3 = red, and V_4 = white. We could not assign a value to V_2, since the green color that could be matched with it has already been assigned to V_1, which appears before V_2 in the hierarchy. Furthermore, there is no blue color car in the batch of new cars. So, we have decided to overstep V_2 and assign the red color to V_3. By this way, we are able to allocate all the cars in the batch of new cars to some buyers on the waiting list, by compromising our stance to execute a strict sequential order of assignment. As a result, the overstepping of V_2 in this case is justified.

Example 4: Supposing a batch of two new *Proton Wira 1.5* cars constituting of one blue and one purple car is delivered to the car dealer. There is no green color car in the batch that could be assigned to V_1. The blue car will be assigned to V_2. We could not match the purple color with V_3, V_4 or V_5. If, assuming we could not assign the purple color to the remaining variables in the CSP, then we have only a partial solution to this problem. The solution is: V_2 = blue. In other words, we will assign the blue color to V_2, and the purple color will be left unassigned.

THE CHRONOLOGICAL BACKTRACKING METHOD

A CSP can be solved using the generate-and-test paradigm. In this brute-force method, each possible combination of the variables is systematically generated and then tested to see if it satisfies all the constraints. The number of combinations considered by this method is the size of the Cartesian product of all the variable domains. This is clearly not a feasible solution.

A more efficient method uses the backtracking paradigm (Kumar, 1992). This typical chronological backtracking method instantiate variables sequentially. As soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has other alternatives that can be examined.

The backtracking method essentially performs a depth-first search of the space of potential solutions of the CSP. Unfortunately, chronological backtracking suffers from thrashing, that is, a search in different parts of the space keeps failing for the same reason. For instance, supposing the variables are instantiated in the order $V_1, V_2, ..., V_i, ..., V_j, ..., V_n$. Further suppose there exist a constraint between V_i and V_j, such that for V_i = green, it disallows any value of V_j. In a backtrack search, whenever V_i is instantiated to 'green', the search will fail while trying to instantiate V_j, as no values for V_j would be found acceptable. This failure will be repeated for each possible combination that the variable V_k ($i < k < j$) can take. The cause of this kind of thrashing is referred to as arc inconsistency (Gaschnig, 1977; Mackworth, 1977).

Therefore, whenever a trashing situation as described above is encountered, we ought to devise a method to remove the culprit for the failure. In our case, the culprit is V_j, and not V_i. Why is this so? Recall that it is important to assign the values in the order the variables appear on the ordered constraint graph. So, V_i, which appears before V_j, ought to be instantiated and maintained. Moreover, we are not compelled to instantiate the variables in a strict order of their appearance in the constraint graph. Therefore, if V_j in the constraint graph is preventing us from further instantiating the variables beyond it, then the most logical thing to do is to skip it and carry on with the allocation process. Thus, we could continue to instantiate the remaining variables after V_j in the ordered constraint graph.

Another drawback with chronological backtracking is that it performs redundant work during the backtrack search after a constraint violation. While backtracking, it will undo the assignments to the variables after the most choice point is retracted, only to re-establish the assignments for some of those variables on the forward search. This kind of redundancy can be overcome by improvising the chronological backtracking method, that could check itself from repeating a computation. For instance, intelligent programs often maintain dependency records to store information about their inferences. These records aid in controlling the actions of the program (Barr and Feigenbaum, 1982).

Dependency records are first employed in robot problem solving programs, to clean up the consequential database entries following robot actions or failures. The effects of actions were represented in Add/ Delete lists. Fikes (1975) kept track of derivations, so that all consequences of a database entry could be erased when the entry was deleted by order of Delete list.

SOLVING THE CSP

Taking the cue from Fikes (1975) inferential recording approach, we have devised an intelligent backtracking method that records information leading to inconsistencies or fruitless path of investigation. However, in our case, this information will be used to remove the culprit for the failure from the ordered constraint graph. Moreover, it can be shown that the proposed method will also not repeat any computation.

This is how our method works. Whenever a variable is assigned some value, its assignment is noted. At some stage, say while attempting to assign variable V_j, a constraint violation was encountered. It would trigger a backtracking to be performed to check whether the violation could somehow be resolved by reassigning the previous variables with some alternate values from their respective domains. In any case, just before the backtracking is executed, the maximal set of assignments that has been concluded so far, as well as the information about the possible culprit (V_j), which is preventing the search from progressing, is stored. The latter information will be used to prune the CSP's subspace, before the rest of its space is searched.

It could be seen that the maximal set of assignments mentioned in the previous paragraph provides a partial solution to the CSP at that stage. Hence, if a search results in a failure,

then the culprit is removed from the constraint graph and a search for solution is re-initiated from the stage where the maximal set of assignments has been determined. Obviously, this method makes an effort to avoid redundant work by not repeating the computation for the assignment of variables from V_1 till V_f.

The algorithm to solve the CSP is illustrated by the procedure SOLVE. First, an ordered constraint graph (**OCG**) is constructed using car color preference information in the orders on the waiting list of a particular car model. Next, the color of cars in the batch of new cars for that model is copied into the **AvailList**. After that, procedure SOLVE is executed to generate **AssignList**, a solution for the CSP.

The values in the **AssignList** will be used to instantiate the variables in the order of their sequence on the pruned **OCG**.

```
procedure SOLVE( ref OCG, AvailList, ref AssignList )
    AttemptList ← ϕ        // stores the maximal set of assignments; initialized to empty set
    AssignList ← ϕ         // stores the partial/ full solution to the CSP; initialized to empty set
    StartNode ← V1↑        // we begin search from the first node. (↑ indicates reference to
                           the node)
    CulpritNode ← Nil      // there is no culprit node at the beginning
    Done ← False           // flag to indicate completion

        // search to see if the OCG has any culprits. (The underlined parameters are passed by
reference)
    repeat
        if SEARCH( OCG, AvailList, AttemptList, AssignList, StartNode,
                   CulpritNode )
            Done ← True
        else       // if the attempt failed to solve the CSP, perform this remedial action and
                   try again
            AttemptList ← AssignList              // update maximal set of assignments
            AvailList ← AvailList – AttemptList   // and the AvailList so that we can
                                                  // recommence the search from a
                                                  // choice point
            AssignList ← ϕ                  // AssignList is reset to empty set
            CulpritNode ← Nil          // assume that there is no culprit at the beginning
                                                  of the search
            StartNode ← NEXT( OCG, CulpritNode )   // the choice point to restart the
                                                   // search will be the node after the
                                                        // culprit node
             remove the CulpritNode from OCG          // the OCG will be modified
        endif
    until Done
end_SOLVE
```

The procedure SEARCH carries out depth first search to find a solution for the CSP. It reports back information about culprits of constraint violation to SOLVE. The latter performs some remedial actions before the search for solution could be furthered. On completion, the reference parameter, **AssignList**, holds a solution for the CSP.

```
procedure SEARCH( OCG, AvailList, AttemptList, ref AssignList, CurrentNode,
                  ref CulpritNode ) : boolean
❶   // the search is complete when either all the members in the AvailList has been assigned or
there
    // are no more variables left in the pruned OCG
  if AvailList = ϕ || CurrentNode = Nil
    AssignList ← AttemptList          //full solution for the CSP is copied into AssignList
    return True
  endif

❷   // match a member in the domain of the node with one in the AvailList
  for each x ∈ domain (CurrentNode) and x ∈ AvailList
    AvailList ← AvailList - {x}       // remove the selected member from AvailList
    AttemptList ← AttemptList + {x}   // and add that member to AvailList

    if SEARCH( OCG, AvailList, AttemptList, AssignList,
        NEXT( OCG, CurrentNode ), CulpritNode )
        return True                   // indicating a full solution has been found
    else
        AvailList ← AvailList + {x}   // undo recent changes to AvailList and
        AttemptList ← AttemptList - {x}  // AttemptList, so that we can try to
    endif                             // match another member in the domain of
                                      // the node with one other member in the
                                      // AvailList
  endfor

❸   // The search failed. In the present state, a more definite maximal set of assignments has been
    // found, which becomes a partial solution to the CSP
if count( AttemptList ) > count( AssignList )
    CulpritNode ← CurrentNode         // the current node is hindering the progression of the
                                      // search
    AssignList ← AttemptList          // the partial solution to the CSP is copied into
endif                                 // AssignList

❹   return False                      // indicating a failed attempt
end_SEARCH
```

The procedure NEXT is called within SOLVE and SEARCH to obtain the reference to a succeeding node in the OCG.

```
procedure NEXT( OCG, CurrentNode ) : Node↑
    if there is a node after CurrentNode in the OCG
        return reference to the node appearing after CurrentNode
    else
        return Nil
end_NEXT
```

The proof of the above algorithms is beyond the scope of this paper. However, for the benefit of the readers, we shall conduct a simple walkthrough. Let us consider Example 3 described in The CSP Model section. First, an OCG is constructed using the car color preference information on the waiting list of the *Proton Wira 1.5* model. Next, the color of cars in the batch of new Proton Wira 1.5 cars delivered to the dealer, is copied into the **AvailList**. Subsequently, the procedure SOLVE is invoked, which in turn invokes SEARCH to find for a solution for the CSP starting from V_1. The parameters for SEARCH are as follows: the constructed **OCG**; **AvailList** = {green, red, white}; **AttemptList** = {}; **AssignList** = {}; **CurrentNode** = V_1ü; and **CulpritNode** = Nil. Note that both the **AssignList** and **CulpritNode** are reference parameters.

In this SEARCH procedure, V_1 will be instantiated with green. Then, SEARCH is invoked recursively with some changed parameter values, namely **AvailList** = {red, white}; **AttemptList** = {green}; and **CurrentNode** = V_2ü. **AssignList** and **CulpritNode** values remain unchanged, {} and Nil, respectively.

In this recursively invoked SEARCH procedure, we could not find any valid assignments for V_2, as both the members in the domain of the V_2 = {blue, green}, cannot be matched with any of the members in the AvailList. Since the search cannot progress further, we skip block ≠ and jump to block ③. Here, the count(**AttemptList**) = 1, is indeed greater than count(**AssignList**) = 0; thus confirming a maximal set of assignments has been found. So, result of the first attempt is copied into **AssignList**. This becomes the partial solution to the CSP. Obviously, the culprit node that is preventing the advancement of the search is V_2. At ④, control returns to the previous SEARCH procedure after the **AssignList** and **CulpritNode** are updated.

In here, we undo assignments to **AvailList** and **AttemptList**. So, **AvailList** and **AttemptList** returns to its original form {green, red, white} and {}, respectively. We try to find another match for V_1 but fail, as 'green' is the only color in the domain of V_1. Consequently, we exit block ≠ and go to block ③. Now, the count(**AttemptList**) = 0, is less than count(**AssignList**) = 1. Note that **AssignList** is passed by reference. The control returns to the caller, procedure SOLVE.

Procedure SOLVE discards the culprit node, V_2 from the OCG and re-initiates the search from the following node on the OCG, V_3. Prior to that, it updates **AttemptList** to contain the partial solution that has been found this far. SEARCH will be invoked once again with

the following parameters: the modified **OCG**; **AvailList** = {red, white}; **AttemptList** = {green}; **AssignList** = {}; **CurrentNode** = V_3ü; and **CulpritNode** = Nil.

In the procedure SEARCH, V_3 will be instantiated with 'red', and SEARCH will be invoked recursively with parameter values: **AvailList** = {white}; **AttemptList** = {green, red}; and **CurrentNode** = V_4ü. **AssignList** and **CulpritNode** values remain unchanged, {} and Nil, respectively.

In the subsequent recursive SEARCH, we will instantiate V_4 with 'white', and SEARCH will be invoked once more with changed parameter values: **AvailList** = {}; **AttemptList** = {green, red, white}; and **CurrentNode** = V_5ü.

Note that **AvailList** has become empty. As a result, the recursion halting condition in block ① has become true. The generated solution in **AttemptList** is copied into **AssignList**, and the procedure backtracks all the way to SOLVE, carrying the solution in the reference parameter, **AssignList**.

The values in **AssignList** = {green, red, white}, will be assigned to the nodes in the order of their sequence on the pruned OCG. Recall that the pruned OCG has discarded V_2 and retained V_1, V_3, V_4, and V_5. Hence the values {green, red, white}, will be assigned to V_1, V_3, and V_4, respectively. Logically, this process establishes the color of the cars that will be allocated to a subset of buyers on the waiting line. Consequently, those successful buyers' orders will be removed from the waiting list. The remaining orders on the waiting list would be used to build a new OCG in the subsequent allocation process.

CONCLUSION

The goal of the CSP is to find a satisfactory set of assignments for its variables such that the assignment satisfy all the constraint. For this purpose, an intelligent backtracking scheme has been devised to find either a full or a partial solution for the CSP, if there is one. While carefully searching the search space, the augmented chronological backtracking method records information related to variable instantiation failure (due to constraint violations). Then, this recorded information is referred to prune that subspace of the CSP before the CSP's remaining space is searched. In particular, this innovative backtracking method overcomes trashing due to arc inconsistencies, and also avoids redundant work during the backtrack search.

ACKNOWLEDGMENT

The author would like to thank Mohd Zamri Abdul Aziz, who has implemented and tested the intelligent backtracking scheme in his New Vehicle Sales System application (Mohd Zamri, 1998).

An earlier version of this paper was presented at the Faculty's colloquium on 11 March

2000.

REFERENCES

Barr, A. and Feigenbaum, E.A. 1982. Dependencies And Assumptions, *The Handbook Of Artificial Intelligence*, Vol. 2, 72 – 76.

Fikes, R. E. 1975. Deductive Retrieval Mechanisms For State Description Models, *IJCAI*, 4, 99 – 106.

Gaschnig, J. 1977. A General Backtrack Algorithm That Eliminates Most Redundant Tests, *In Proceedings of the Fifth International Joint Conference on AI*, 457 – 467.

Kumar, V. 1992. Algorithms For Constraint Satisfaction Problems. *Artificial Intelligence*, 13 (1), 32 – 44.

MackWorth, A. K. 1977. Consistency In Networks Of Relations. *Artificial Intelligence*, 8 (1), 99 – 118.

Mohd Zamri Abdul Aziz. 1998. *New Vehicle Sales System*. B.Sc (IT) Final Year Project Report, UiTM.