Review Article

Available Online at: www.ijphr.com

International Journal of Pharmaceuticals and Health care Research

ISSN: - 2306 – 6091

ROLE OF ARTIFICIAL INTELLIGENCE CONSTRAINTS IN INDUSTRIAL HEALTH CARE ROBOTS – AN ANALYSIS

[1]M Mohamed Sirajudeen, [*2]M Abubacker Siddiq

[1]Department of Information Technology, University of Gondar, Ethiopia

[2]Department of Industrial Engineering, P.S.N Institute of Technology and Science, Palayankottai, Tirunelveli, Tamil Nadu, India

Abstract

In general, all kind of industrial health care robots solving problems by searching constraints. In these searches the structure of each state was not considered – just whether or not it was a goal state. To look at a class of problems called constraint satisfaction problems (CSP). In CSPs the goal or goals are stated implicitly, by defining constraints on the allowable structure of the state. A constraint is a limitation on what type of solution it will allow. In this research paper focus mainly "how to formulate CSPs". There are two alternative formulations: incremental and complete-state. Depending on which formulation to use, it leads to a different algorithm for solving the CSP. It covers two major techniques: backtracking search and local search for CSPs to be discuss in this paper.

Keywords: Search, Constraints, Problem, State and Back tracking.

Received on- 20.07.2015 ; Revised and accepted on- 31.07.2015; Available online- 10.08.2015

Introduction

Recall that until now formulated problems by stating four things:

- Initial state
- Actions (successor function)
- Goal test
- Path cost

It does not so far made any restrictions on allowable solutions, apart from an explicit goal test. The states of such problems can be thought of as being data structures. For example, in the *n*-queens problem each state could be an array of integers, indicating the position of the queen on each column of the board. In a CSP, the problem is formulated by stating:

- States defined by a set of variables X_j, each of which can take values from a domain D_j
- A goal test consisting of a set of constraints specifying allowable combinations of values for subsets of X_j

In effect, limited the domain of each variable specifies a form of constraint on the allowable states. Also, the goal test defines more constraints, limiting the possible combinations of values of the variables.

It turns out that a whole class of problems can be expressed in this way, allowing us to use some powerful general-purpose algorithms for all of the class of problems: let us consider an example of Map-Colouring. As an example will consider the *map-colouring problem*. This is a famous example in AI and is often used to illustrate CSPs. Suppose to have a map of a country, like the map of

Author for Correspondence:

M Abubacker Siddiq

Email: mdsirajudeen1@gmail.com

Australia shown in Figure 1. The country is divided into a number of regions. The problem is how it can colour each region of the country using a limited number of colours so that no two adjacent regions have the same colour.

Fig. 01: A map showing the regions of Australia

It expresses this problem as a CSP as follows. First to define 7 variables, one for each region on the map. It will call these *WA*, *NT*, *SA*, *Q*, *NSW*, *V*, and *T*, after the initial letters of the region names.[1] The domains of each of these variables are defined to be {*red*, *green*, *blue*}. In additions, it specify the constraints that adjacent regions must have different colours.[2,3] It can specify this for the regions *WA* and *NT* by stating *WA* ≠ *NT*, or alternatively:
(*WA*, *NT*) in {(*red*, *green*), (*red*,*blue*), (*green*,*blue*), (*green*,*red*), (*blue*,*red*), (*blue*,*green*)}.

To specify the constraints for other neighbouring regions in a similar way. Now it need to find a solution that satisfies all of the stated constraints.[4] Here to introduce two pieces of important terminology:

- *Complete*: a complete state is one in which all variables have been assigned a value
- *Consistent*: a consistent state is one that does not violate any of the specified constraints

A *solution* to a CSP is a complete and consistent assignment of values to variables.

Related work

Another way of visualising CSPs is by using a *constraint graph*. Fig. 02 shows the constraint graph for the Australia map-colouring problem. In this graph each node represents a variable in the CSP, whilst each arc represents a constraint.[5,6] In this constraint graph each constraint links only two variables. Therefore it refers to this problem as a *binary CSP*.

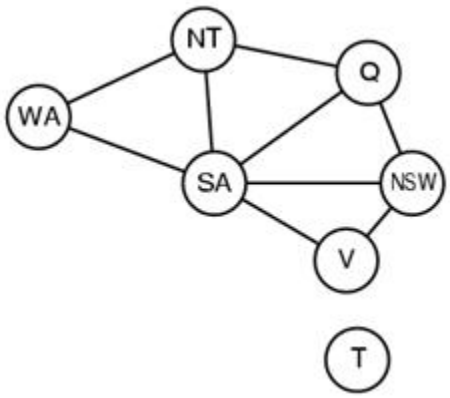

Fig. 02: A constraint graph of the map-colouring problem for Australia

It is possible to categorise CSPs based on the type of variable they use and on the size of their domains:

- *Discrete CSPs*: In a discrete CSP the variables take on discrete values. There are two types of discrete CSP:
  - *Finite domain*: If a discrete variable has a finite domain it means can take on a limited number of discrete values. For example, the {red, green, blue} domain in the map-colouring problem was a finite domain.
  - *Infinite domain*: Some discrete variables can have infinite domains. For example, integer values are discrete (e.g. there is no integer between 1 and 2), but there

are infinitely many integers. Strings are also infinite domain discrete variables.

- *Continuous CSPs*: In a continuous CSP variables can take one from a continuous range of values. For example, real numbers are continuous.

The type of variable determines how difficult the problem is to solve. Finite domain discrete variables are the easiest type: if it has *n* variables and a domain size of *d*, then there are only d^n complete assignments. If the domains are infinite or the variables continuous,[7,8] then cannot enumerate all allowable values for the variables. In these cases a special *constraint language* is required to specify the constraints.[9,10] This is beyond the scope of this course.

It also categorize CSPs based on the type of their constraints. There are three types of constraint it can have:

- *Unary* constraints (e.g. SA ≠ green)
- *Binary* constraints (e.g. (SA ≠ WA)
- *Higher-order* constraints (involve 3 or more variables)

The more variables that are involved in a constraint, the harder the problem is to solve. Problems with higher-order constraints are the hardest class of problems.

In an incremental formulation of CSPs the initial state has no variables assigned. The successor function assigns a (consistent) value to one unassigned variable. It never enter inconsistent states in principle only need to make *n* assignments for *n* variables. In a complete-state formulation for CSPs the initial state is complete, i.e. it has values assigned for all variables. However, it will almost certainly not be consistent.[11,12] The successor function modifies one variable to try to reduce the number of conflicts being violated.

Observations

The search technique that it uses for CSPs depends on what formulation it use:

- Incremental formulation → *Backtracking search*
- Complete-state formulation → *Local search*

A. Backtracking Search

Tree search algorithms are common for solving incremental formulation of CSPs. It must assign a value to one variable at each node of our search tree. However, there is a potential problem with computational complexity here. If it has "n" variables and *d* possible values for each variable, then the branching factor of the search tree is *nd*. For the Australia map-colouring example this is equal to 7*3=21, but for more complex problems the branching factor can become prohibitively large. How can it reduce this branching factor? The answer is that take advantage of the *commutatively* of variable assignments. Commutatively simply means that the order of operations does not matter. For example, the result of applying the operation *WA=red* followed by *SA=green* is the same as the result of applying *SA=green* followed by *WA=red*. Therefore it can reduce the branching factor by just selecting one variable assignment to perform at each tree level. If to do this the branching factor is only *d* instead of *nd*.

The most common technique for solving incremental formulations of CSPs is known as *backtracking search*. Backtracking search is basically a depth-first search with a single variable assigned at each level of the tree. If no legal assignments are possible then it backtrack to try other branches of the tree. This simple algorithm can achieve impressive results: for example, the 25-queens problem can be solved in a reasonable amount of time. However, it can improve the efficiency of the search by introducing a number of simple heuristics.

In the description of backtracking search, it did not address the following questions:

- Which variable to pick at each level?
- In what order should it consider child nodes?
- What are implications of current assignment for other unassigned variables?

It turns out that these factors all have a big influence on the efficiency of backtracking search, and by making the right choices of variable/value it can make dramatic improvements in efficiency. The rules can be summarized as follows:

- When choosing which variable to assign at each level, always choose the most constrained variable (the one with fewest remaining legal values). This is known as the *MRV* (*Minimum Remaining Values*) heuristic.
- If the MRV values are equal, use the *Most Constraining Variable* heuristic as a tie-

breaker. This means that it should choose the variable with the most constraints on remaining unassigned variables.

- When choosing an order to try to different values for each variable, always choose the *Least Constraining Value* first. This means that it should try the value that rules out the fewest values in the remaining unassigned variables.

As an example, let us consider the problem of colouring the map of Ethiopia shown in Fig. 3. Try drawing the map out for yourself and stepping through this example to make sure you understand how the algorithm works.

Step 1:
- Minimum Remaining Values: all equal
- Most Constraining Variable: Oromia=6
- Least Constraining Value: equal
- Therefore Oromia=Red

Step 2:
- Minimum Remaining Values: any apart from Tigray=2
- Most Constraining Variable: Amhara/Afar=3 (choose Amhara)
- Least Constraining Value:equal
- Therefore Amhara=Green

Step 3:
- Minimum Remaining Values: Afar/Benshangul=1
- Most Constraining Variable: Afar=2
- Least Constraining Value: *only 1 value possible*
- Therefore Afar=Blue

Step 4:
- Minimum Remaining Values: Tigray/Benshangul/Somali=1
- Most Constraining Variable: Benshangul=1
- Least Constraining Value: *only 1 value possible*
- Therefore Benshangul=Blue

Step 5:
- Minimum Remaining Values: Tigray/Somali/Gambella=1
- Most Constraining Variable: Gambella=1
- Least Constraining Value: *only 1 value possible*
- Therefore Gambella=Green

Step 6:
- Minimum Remaining Values: Tigray/Somali/Southern=1
- Most Constraining Variable: all zero (choose Tigray)
- Least Constraining Value: *only 1 value possible*
- Therefore Tigray=Red

Step 7:
- Minimum Remaining Values: Somali/Southern
- Most Constraining Variable: all zero (choose Southern)
- Least Constraining Value: *only 1 value possible*
- Therefore Southern=Blue

Step 8:
- Minimum Remaining Values: Somali
- Most Constraining Variable: n/a
- Least Constraining Value: *only 1 value possible*
- Therefore Somali=Green

Fig. 03: A map of Ethiopia

It can see that in this case the algorithm did not have to backtrack at all, instead going straight towards a valid solution.

Remember that this algorithm can be applied to any problem, so long as it can be formulated as a constraint satisfaction problem. For example, it can express the *n*-queens problem in the same way. In this case, the variables will be the locations of each queen: each queen is on one column only, so for *n* queens it needs *n* variables to specify the row number for each queen. The constraints will be that the queens should not be attacking each other. In fact, these simple heuristics can lead to dramatic improvements over the standard backtracking search. Whereas the 25-queens problem is solvable by the standard algorithm, with the addition of these heuristics, the 1000-queens problem is solvable.

B. Local Search

In a complete-state formulation it start with a complete (but not consistent) assignment of values to variables, and make modifications to values to try to achieve consistency. For a complete-state formulation of a CSP, it is common to use a *local search* technique. The algorithm can be summarised as follows:

- Make a random assignment of values to variables
- Loop until it reach a solution:
  - Pick a random conflicted variable (i.e. one that is violating a constraint)
  - Use the *min-conflicts* heuristic to choose new value

The min-conflicts heuristic simply states that it choose the value that will violate the fewest constraints. It can see this as a slightly modified version of *hill-climbing*: it always move to the state that will minimise the number of conflicts.

As an example, consider the Ethiopia map-colouring problem. Suppose it start off with the following random assignment of colours to regions:

- Tigray: green
- Amhara: green
- Afar: blue
- Benshangul: blue
- Gambella: green
- Oromia: blue
- Southern: red
- Somali: green

Now it follow the algorithm and choose random conflicted variables as follows:

Step 1:
- Tigray, Amhara, Afar, Oromia, Benshangul are all conflicting
- Randomly choose Oromia
- Current value=blue (2 conflicts); if it change to green there will be 2 conflicts; if it change to red there will be 1 conflict
- Therefore change Oromia to red

Step 2:
- Tigray, Amhara, Oromia, Southern are all conflicting
- Randomly choose Southern
- Current value=red (1 conflict); if it change to green there will be 1 conflict; if it change to blue there will be 0 conflicts
- Therefore change Southern to blue

Step 3:
- Tigray, Amhara are conflicting
- Randomly choose Tigray
- Current value=green (1 conflict); if it change to blue there will be 1 conflict; if it change to red there will be 0 conflicts
- Therefore change Tigray to red

Step through this example to make sure you understand it. As you can see it reached a solution in 3 steps. Generally, local search with *min-conflicts* can be very effective. Using this technique it becomes possible to solve the 1 million-queens problem. This algorithm has also been used to schedule observations on the Hubble Space Telescope.

Conclusion and future work

The above analytical survey makes a clear road map of the research avenue of industrial research in the field of robotics in all applications in the basis of constraints based solution. It creates awareness among the researchers to make an effective and efficient problem solving techniques in the robots architectural design and implementation.

References

1. Andrea Selinger and Randal C. Nelson, ``A Perceptual Grouping Hierarchy for Appearance-Based 3D Object Recognition", Computer Vision and Image Understanding, vol. 76, no. 1, October 1999, pp.83- 92. Abstract, gzipped postscript (preprint).
2. Randal C. Nelson and Andrea Selinger ``Large-Scale Tests of a Keyed, Appearance-Based 3-D

Object Recognition System", Vision Research Special issue on computational vision, Vol. 38, 15-16, Aug. 1998.

3. Abstract, gzipped postscript (preprint) Randal C. Nelson and Andrea Selinger ``A Cubist Approach to Object Recognition", International Conference on Computer Vision (ICCV98), Bombay, India, January 1998, 614-621.
4. Abstract, gzipped postscript, also in an extended version with more complete description of the all.
5. Randal C. Nelson, Visual Learning and the Development of Intelligence, In Early Visual Learning, Shree K. Nayar and Tomaso Poggio, Editors, Oxford University Press, 1996, 215-236.
6. Abstract, Randal C. Nelson, ``From Visual Homing to Object Recognition'', in Visual Navigation, Yiannis Aloimonos, Editor, Lawrence Earlbaum Inc, 1996, 218-250.
7. Abstract, Randal C. Nelson, ``Memory-Based Recognition for 3-D Objects", Proc. ARPA algorithms, and additional experiments. Image Understanding Workshop, Palm Springs CA.
8. The Architecture of Brain and Mind Integrating Low-Level Neuronal Brain Processes with High-Level Cognitive Behaviours, in a Functioning Robot2.
9. Diaconis, Persi; Shahshahani, Mehrdad (1987), "The subgroup algorithm for generating uniform random variables", Prob. in Eng. and Info. Sci. 1: 15–32, ISSN 0269-9648.
10. Dubrulle, Augustin A. (1999), "An Optimum Iteration for the MatrixPolar Decomposition", lect. Trans. Num. Anal. 8: 21–25, http://etna.mcs.kent.edu/
11. Golub, Gene H.; Van Loan, Charles F. (1996), Matrix Computations (3/e ed.), Baltimore: Johns Hopkins University Press, ISBN 978-0-8018-5414-9.
12. Higham, Nicholas (1986), "Computing the Polar Decomposition—with Applications", SIAM J. Sci. Stat. Comput. 7 (4): 1160–1174, doi:10.1137/0907079, ISSN 0196-5204,http://locus.siam.org/SISC/volume-07/art_0907079.html.