In-Class Assignment - CSP

Basil R. Yap
50.021 Artificial Intelligence - Term 8

July 26, 2018

1 - pure backtracking

Assume we use pure backtracking to search for a solution to this problem. We use a fixed variable ordering in the search (V1, V2, V3 and V4) and the values are considered in the order shown in the picture above. Then, show the order in which individual variable assignments are considered by backtracking (this is analogous to the order in which nodes are expanded by depth-first search).

Show the first 10 assignment even if it is immediately found to be inconsistent upon testing. Assume that the search continues even after finding a valid solution. Write each variable assignment per line as a number from 1 to 4 (indicating the variable) followed by a letter, drawn from R, G, B, e.g. 1 R.

Solution
The sequence of the pure backtracking search are as follows:

1. 1R
2. 1R2B
3. 1R2B3B [INVALID]
4. 1R2B3G
5. 1R2B3G4B [INVALID]

6. 1R2G
7. 1R2G3B
8. 1R2G3B4B [INVALID]
9. 1R2G3G
10. 1R2G3G4B [STOP / SOLVED]

2 - backtracking with forward checking

Repeat assuming we use backtracking with forward checking to search for a solution to this problem. We use the same variable ordering and value ordering as before. Show the order in which assignments are considered by BT-FC. Whenever propagating after an assignment causes a domain to become empty, that causes backup in the search. Assume that the search continues even after finding a valid solution. Write each assignment (up to 10) as before.

Solution
The sequence of the pure backtracking search are as follows:

1. 1R
2. 1R2B [INVALID]
3. 1R2G
4. 1R2G3B [INVALID]
5. 1R2G3G
6. 1R2G3G4B [STOP / SOLVED]
7. 1B
8. 1B2G
9. 1B2G3G
10. 1B2G3G4B [STOP / SOLVED]

3 - Arc consistency

Write the values in each of the indicated variable domains after any changes required to achieve arc consistency for just that arc. Then, assume that the following arcs are done sequentially, with the effects on the domains propagating. Write domains as a sequence of letters, for example, R B. If there are no values left in a domain, write None.

```
V1 - V2: D1=   D2=
V1 - V3: D1=   D3=
V2 - V4: D2=   D4=
V3 - V4: D3=   D4=
V1 - V2: D1=   D2=
V1 - V3: D1=   D3=
```

Solution

V1-V2	D1 = RGB	D2 = GB
V1-V3	D1 = RGB	D3 = GB
V2-V4	D2 = G	D4 = B
V3-V4	D3 = G	D4 = B
V1-V2	D1 = RB	D2 = G
V1-V3	D1 = RB	D3 = G

4 - Thinking of CSP

Indicate whether the following statements are true or false.

1. Constraint propagation without search is sufficient to find a satisfying set of assignments in most problems.

2. If constraint propagation leaves all variables with non-empty domains, there is at least one solution.

3. If constraint propagation leaves some variable with an empty domain, there is no solution.

4. If constraint propagation leaves all variables with singleton domains, there is a unique solution.

5. Constraint propagation takes on the order of ed^2 arc tests for one pass through all the arcs in a constraint graph with e edges and d values per domain.

6. When doing backtracking with Forward Checking (BT-FC), then Forward checking during backtracking requires verifying that the newly assigned value for a variable is consistent with all previously assigned values of variables.

7. Backtracking search with full constraint propagation (until arc-consistency is reached) generally results in an answer using fewer arc-tests than search with forward-checking only.

8. Backtracking with Forward Checking (BT-FC) never explores more possible variable values than pure backtracking.

9. BT-FC never tests more constraint arcs than pure backtracking.

10. Dynamic ordering of variables based on domain size (e.g. minimum remaining values) is generally a very effective strategy to speed up solving CSP (whether or not all the answers are desired).

11. Dynamic ordering of values based on impact on neighboring variables (e.g. least constraining value) is generally a very effective strategy to speed up solving CSP when all possible answers are desired and not just one.

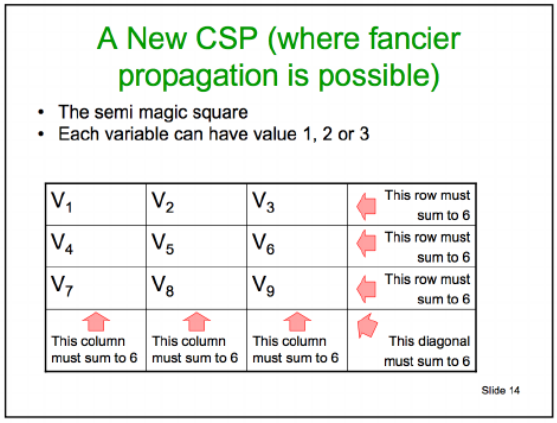
Solution

1. True
2. False
3. True
4. True
5. False
6. False
7. True
8. True
9. True
10. False
11. True

5 - *Semi-Magic Square*

Consider a 3×3 array, each of whose entries can be either 1, 2 or 3. We want to find an assignment to each of the entries so that the entries in each row, in each column and in one of the diagonals are different. Note that this will also ensure that these row, colum and diagonals add up to 6 ($1 + 2 + 3$). But, note that the "adding to 6" constraint is not a "binary constraint", that is, it involves more than three variables. However, the constraint that each pair of values in the row, column or diagonal be different is a "binary constraint".

You can find a description of the Semi-Magic Square problem in these slides `https://www.autonlab.org/_media/tutorials/constraint05.pdf`, in particular in Slide 14 (reproduced here).



Implement a CSP that captures this problem. Use the file `semi-magic.py` in the code distribution; it imports csp.py.

You should use the variable names in the image above in your CSP formulation.

Experiment solving this problem with the various solution methods, ranging from pure backtracking, variable and value orderings, and inference methods (forward_checking and mac). Look at the number of assignments attempted by each algorithm, that should give you some idea of the effectiveness of the methods on this problem. Make sure that you understand the results.

Solution

```
import pdb
from csp import *
from random import shuffle

def solve_semi_magic(algorithm=backtracking_search, **args):
    """ From CSP class in csp.py
        vars        A list of variables; each is atomic (e.g. int or string).
        domains     A dict of {var:[possible_value, ...]} entries.
        neighbors   A dict of {var:[var,...]} that for each variable lists
                    the other variables that participate in constraints.
        constraints A function f(A, a, B, b) that returns true if neighbors
                    A, B satisfy the constraint when they have values A=a, B=
                    """
    # Use the variable names in the figure
    csp_vars = ['V%d'%d for d in range(1,10)]

    ###############################################
    # Fill in these definitions

    csp_domains = {var:[1,2,3] for var in csp_vars}
    csp_neighbors = {
    "V1":["V2","V3","V4","V5","V7","V8"],
    "V2":["V1","V3","V5","V8"],
    "V3":["V1","V2","V6","V9"],
    "V4":["V1","V5","V6","V7"],
    "V5":["V1","V2","V4","V6","V8","V9"],
    "V6":["V3","V4","V5","V9"],
    "V7":["V1","V4","V8","V9"],
```

```
"V8" :["V2" ,"V5" ,"V7" ,"V9"],
"V9" :["V1" ,"V3" ,"V5" ,"V6" ,"V7" ,"V8"]
}
def csp_constraints(A, a, B, b):
    return (a != b)

########################################

# define the CSP instance
csp = CSP(csp_vars, csp_domains, csp_neighbors,
          csp_constraint)

# run the specified algorithm to get an answer (or None)
ans = algorithm(csp, **args)

print 'number_of_assignments', csp.nassigns
assign = csp.infer_assignment()
if assign:
    for x in sorted(assign.items()):
        print x
return csp
```

Sequence produced: 123231312

6 - Scheduling a Job Shop

You can find a description of Job Shop Scheduling in these slides `https://www.autonlab.org/_media/tutorials/constraint05.pdf`, in particular Slides 39-44.

Implement a CSP that captures this type of constraints. The file jobs.py has several sample problems. Note that, in general, we want the shortest possible schedule, that is, the smallest deadline for completion that is feasible. We have given you deadline values that work for each of the problems.

Experiment solving these problems with the various solution methods, ranging from pure backtracking, variable and value orderings, and inference methods (forward_checking and mac). What seems to matter most?

Try some infeasible values for the deadline and report what happens.

Solution
INCOMPLETE, TO BE COMPLETED.