CS560 Fall 2019 | Assignment 1 | Kenken

Group members -

1. Matthew Afsahi

Introduction

In this assignment, the following algorithms were used to solve a kenken puzzle -

1. A basic backtracking algorithm to have a baseline performance.
2. An improved version of the backtracking algorithm in (1).
3. Min-conflicts: A variant of the hill climbing algorithm - which is a local search algorithm.

For each implementation, the final assignment is printed along with the number of assignments that occurred in the particular algorithm. In case of the local search algorithm, the number of iterations used are printed.

Kenken representation as a constraint satisfaction problem (CSP)

A generic CSP consists of -

1. X - A list of ***variables***
2. D - For each variable, a ***domain*** of its possible values.
3. C - Constraints on subsets of X that must be satisfied.

Once this representation is made, one could proceed to applying algorithms for solving the same.

Now in the specific case of Kenken puzzle of size N x N -

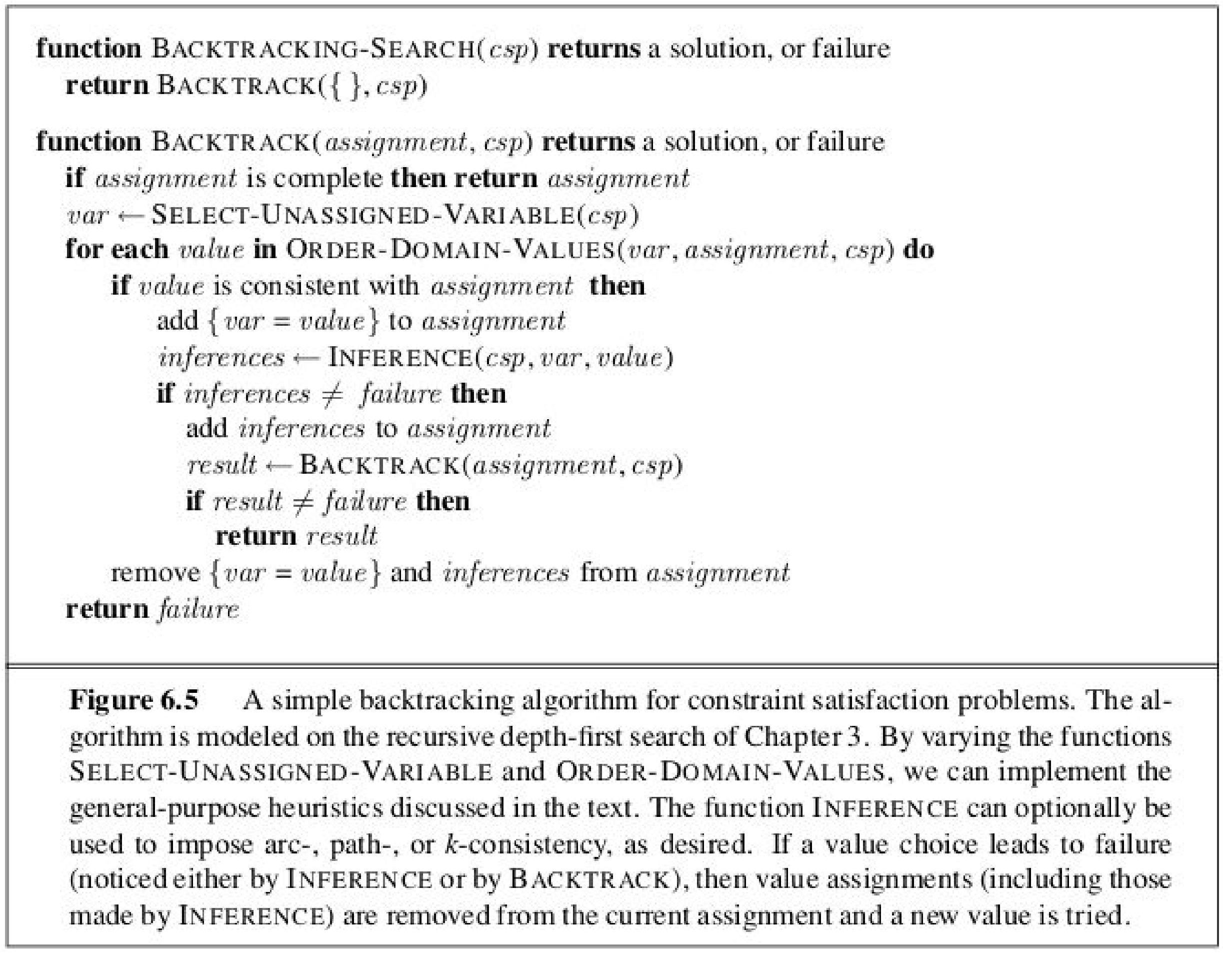
1. X - A list of variables [0, 1, … N*N - 1]. Each 2D location can be mapped to a single location.
2. D - For each variable V, a naive set of possible values are [1, 2, .., N]. However, this list can be pruned by some algorithms such as AC3 [1].
3. C - A kenken board has 3 constraints.
    a. Each row must have numbers from 1 to N, none repeated.
    b. Each column must have numbers from 1 to N, none repeated.
    c. The numbers in each “cage” must evaluate to the goal of that cage, using the values in that cage. Constraints are represented as a function f(A, a, B, b) which returns True if assignments A=a and B=b doesn’t violate any of the above constraints.

4. To enforce the constraints, the following were also added -
    a. Cage data structure - For each cage (represented by a capital letter in assignment specs), the Cage is a list of cells belonging to it, the operation of the cage (one of “+ - / * =”) and the goal that the cage must evaluate to.
    b. An inverse mapping of the above - for each variable to the cage it belongs to.
    c. Neighbors - For each location, its neighbors are all other locations in the same row and the same column.

With the above modeling of a kenken puzzle as a CSP in place, the generic methods to solve a CSP can be applied. These generic methods are available in the open source git repository that accompanies the book.
HW1 solution was built on top of the open source python implementations [3].

(1) Basic backtracking algorithm -

A basic algorithm is as follows [2] -



```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, value)
          if inferences ≠ failure then
              add inferences to assignment
              result ← BACKTRACK(assignment, csp)
              if result ≠ failure then
                  return result
      remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

In the basic version, there is no intelligence in SELECT-UNASSIGNED-VARIABLE() and INFERENCE() methods. Always the first unassigned variable found is chosen.

The implemented method has modified the above algorithm to also return the number of assignments that happened. (*add {var = value} to assignment* step). This is done by simply accumulating and returning the number of assignments.

(2) Improved backtracking -

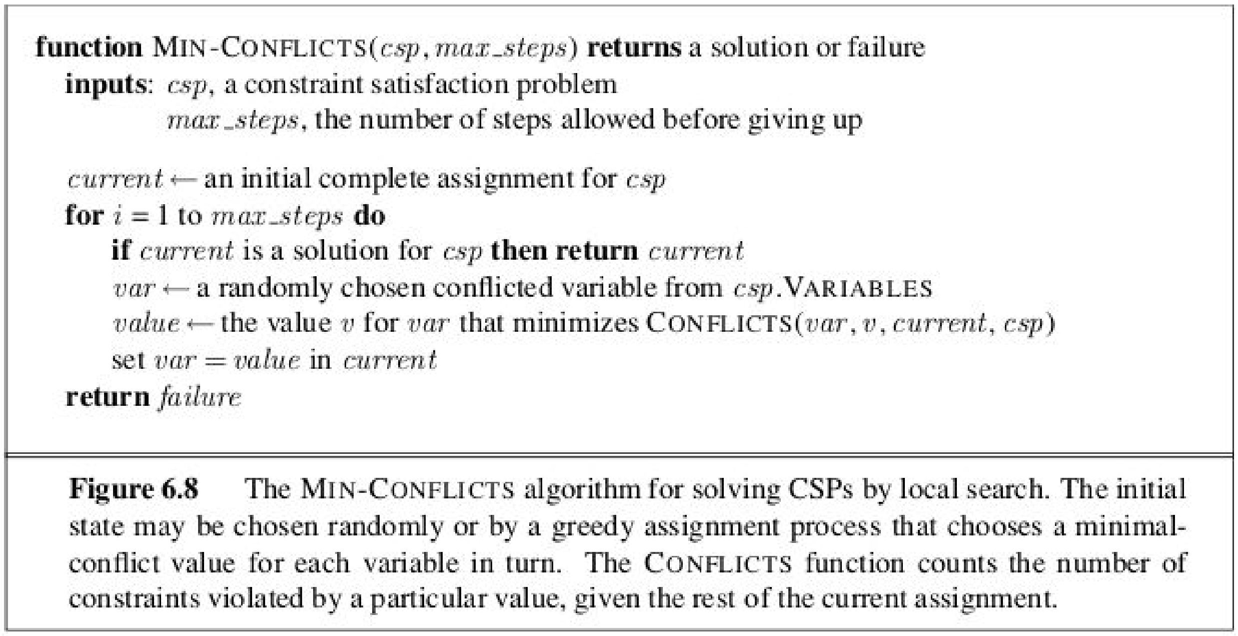
The following improvements were added to the above basic solution -

1. A preprocessing of arc-consistency with AC3 algorithm [1]. This reduces the domain size of a few variables. For e.g, A “2-” cage in a board of size 4 x 4 can only hold (1, 3) (3, 1) (2, 4) or (4, 2).
2. For SELECT-UNASSIGNED-VARIABLE(), the “minimum remaining values (mrv)” heuristic is chosen. Amongst the remaining unassigned variables, the one with the least remaining values in its domain is chosen.

As in (1), the implemented method has modified the open source algorithm [3] to also count and return the number of variable assignments that takes place.

(3) Min conflict (Hill climbing local search)

The basic idea of a local search is to start with a complete assignment of all variables that may be violating some constraints. In every iteration, an attempt is made to improve the constraint violation. A random conflicted variable is chosen and a value is assigned to it which minimises the number of conflicts.
In this way, either the actual solution is achieved or a very close actual solution is achieved at the end of the last iteration. This works very well especially when there are multiple solutions to a given CSP. However, it may not return a solution because of being stuck around a local maxima, a shoulder, a plateau etc. The algorithm is described in Fig 6.8 of the book [4].



```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for $i = 1$ to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen conflicted variable from csp.VARIABLES
      value ← the value $v$ for var that minimizes CONFLICTS(var, $v$, current, csp)
      set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

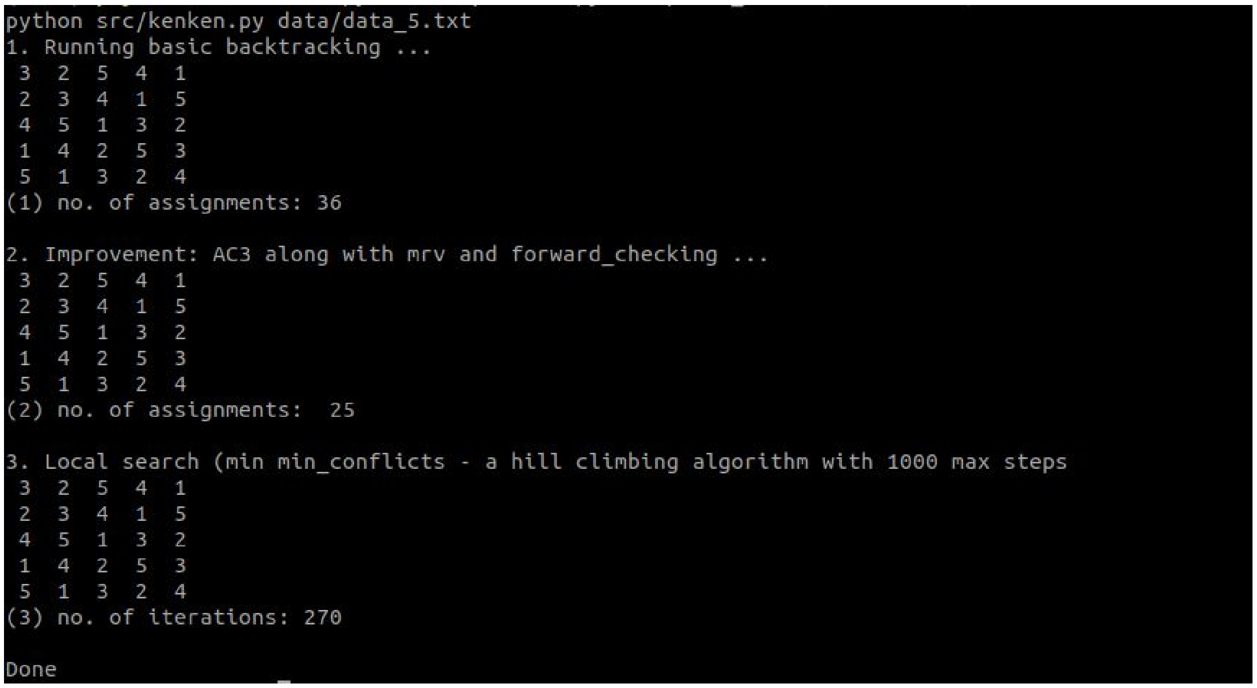
Again, in implementation, this was modified to also return i (iteration counter) in which either the solution or failure is returned.

Constraint function f(A, a, B, b) in the context of Kenken puzzle

As mentioned before, f(A, a, B, b) returns True or False, indicating if there is any constraint violation or not. Once we check the row and column constraints for A and B, there are 2 cases for checking cage constraints -

1. *A and B belong to the same cage:*
   a. For add and mul cage, we return True only if A=a, B=b along with the current assignments of the same cage satisfy the cage goal, or if they are less than the goal and not all cage members have been assigned. A future assignment may satisfy the cage goal.
   b. For sub and division cage - we only have 2 locations for subtraction or division cage. Cage rule is only satisfied when |a-b| = goal or max(a, b) / min(a, b) = goal. In all other cases, False is returned.
2. *A and B belong to different cages:*
   a. For add and mul cage: We need to check for validation of both A's and B's cage. For both, either the current assignment should be satisfying the goal, or it must be less than the goal and not all cage members must have been assigned. A future assignment may satisfy the cage goal.
   b. For sub and division cage - For each cage, we need to check if the current assignment satisfies the cage goal, or if there is still a future possibility of goal satisfaction.

Following is a screenshot of a run on a 5x5 board -



```
python src/kenken.py data/data_5.txt
1. Running basic backtracking ...
 3  2  5  4  1
 2  3  4  1  5
 4  5  1  3  2
 1  4  2  5  3
 5  1  3  2  4
(1) no. of assignments: 36

2. Improvement: AC3 along with mrv and forward_checking ...
 3  2  5  4  1
 2  3  4  1  5
 4  5  1  3  2
 1  4  2  5  3
 5  1  3  2  4
(2) no. of assignments:  25

3. Local search (min min_conflicts - a hill climbing algorithm with 1000 max steps
 3  2  5  4  1
 2  3  4  1  5
 4  5  1  3  2
 1  4  2  5  3
 5  1  3  2  4
(3) no. of iterations: 270

Done
```

It can be seen that the number of assignments got reduced from 36 to 25, due to the application of AC3 and forward checking.

References:

1. Artificial Intelligence: A modern approach, 3rd edition - Chapter 6 - Fig. 6.3.
2. Artificial Intelligence: A modern approach, 3rd edition - Chapter 6 - Fig. 6.5.
3. https://github.com/aimacode/aima-python
4. Artificial Intelligence: A modern approach, 3rd edition - Chapter 6 - Fig. 6.8.

Additional reference -
https://github.com/chanioxaris/kenken-solver