Constraint Satisfaction Problem

Constraint Satisfaction Problem AI solver for CS-UY 4613

By Kevin Lee (KL3642)

This program uses Backtracking Search to search with heuristics and inference to solve the map coloring problem.

How to use

1. Create project directory that includes `app.py`
2. Open shell or terminal in project dir (`C:\\...\\Constraint-Satisfaction-Problem` or however you named it).
3. Use the following syntax:
    1. `python app.py 'input_file.txt'`
        1. Note: All files used must be present in project dir including input files.
    2. For example: `python app.py Input1.txt`
4. The program will create an `output.txt` file in the same directory
    1. Note: Repeated usage of the program will overwrite existing `output.txt` files

Source Code

app.py

```
 """
CS-UY 4613
Source Code for Project 2: Map Coloring Problem

Author: Kevin Lee (KL3642)

Description: Implement the Backtracking algorithm
for solving the Map Coloring Problem.
"""

# Standard Libraries
import sys
import copy

# Problem Class
class Problem:
    def __init__(self, adj, domain, var_names):
        self.adj = adj
        self.domain = domain
        self.var_names = var_names

# Check if assignment is complete
def complete(csp, assignment):
    # Check if all variables are assigned a value
    for var in assignment:
        if len(assignment[var]) > 1 or len(assignment[var]) == 0:
            return False

    # Check if all assignments are legal
    for var in assignment:
        for adj in csp.adj[var]:
            if assignment[var] == assignment[adj]:
                return False

    return True

# Select unassigned variable using MRV and Degree Heuristics
# Returns next variable to use
def selectUnassigned(csp, assignment):
    # MRV Heuristic
    # Number for minimum remaining value
    min_rem = len(csp.domain) + 1

    for elem in assignment:
        if len(assignment[elem]) > 1:
            min_rem = min(min_rem, len(assignment[elem]))

    # Find variables with minimum remaining value
    MRV_candidate = []

    for elem in assignment:
        if len(assignment[elem]) == min_rem:
            MRV_candidate.append(elem)

    # Degree Heuristic
    final_candidate = []
    degree = ('var',0)

    for elem in MRV_candidate:
        neighbors = csp.adj[elem]
        count = 0
        # Count number of unassigned neighbors
        for neighbor in neighbors:
            if len(assignment[neighbor]) != 1:
                count += 1

        if count >= degree[1]:
            degree = (elem, count)

    return degree[0]

# Inference function
# Returns True if forward checking passes
# False otherwise
def inference(csp, var, assignment):
    neighbors = csp.adj[var]

    for neighbor in neighbors:
        if assignment[var][0] in assignment[neighbor]:
            assignment[neighbor].remove(assignment[var][0])
        if len(assignment[neighbor]) == 0:
            return False

    return True

# Backtracking search algorithm
def backtrack(csp, assignment):
    # Check if assignment is complete
    if complete(csp, assignment):
        return assignment
    # Select variable
    var = selectUnassigned(csp, assignment)
    for val in csp.domain:
        # If value is consistent with assignment
        if val in assignment[var]:
            backup = assignment
            assignment[var] = [val]

            inferences = inference(csp, var, assignment)

            if inferences:
                result = backtrack(csp, assignment)
                if result:
                    return result

            assignment = backup
    return False

# Search
def search(csp):
```

```
def search(csp):
    # Initialize assignment set
    assignment = {}
    for var in csp.var_names:
        assignment[var] = copy.copy(csp.domain)

    return backtrack(csp, assignment)

# Create and write to output file
def output(res):
    f = open("output.txt", 'w')

    if res:
        for var in res:
            f.write(var + ' = ' + res[var][0] + '\n')

    else:
        f.write('FAIL')

def main():
    # Grab filename and w from stdin
    filename = sys.argv[1]

    # Open File and parse input
    f = open(filename)
    lines = f.readlines()

    # Remove whitespace
    for ind in range(len(lines)):
        lines[ind] = lines[ind].strip()
        if len(lines[ind]) == 0:
            lines.remove(lines[ind])

    # Init problem variables
    var_names = []
    adj_list = {}

    # Init node variables
    domain = []

    # Assign values
    for name in lines[1].strip().split(' '):
        var_names.append(name)

    for domain_val in lines[2].strip().split(' '):
        domain.append(domain_val)

    # Parsing adjacency list
    ind = 0
    for line in lines[3:]:
        new_adj_list = []
        adj_ind = 0
        for adj in line.strip().split(' '):
            if int(adj) == 1:
                new_adj_list.append(var_names[adj_ind])

            adj_ind += 1
        adj_list[var_names[ind]] = new_adj_list
        ind += 1

    # Create problem
    problem = Problem(adj_list, domain, var_names)

    res = search(problem)

    # Create and write to output
    output(res)

if __name__=='__main__':
    main()
```

Output Files

Output1.txt

```
 NSW = B
NT = B
Q = G
SA = R
WA = G
V = G
```

Output2.txt

```
 R1 = G
R2 = B
R3 = Y
R4 = R
R5 = Y
R6 = B
R7 = G
R8 = R
```