CSC 480: Artificial Intelligence I: 2024 Spring, Assignment #2

Last Modified: May 18, 2024

Overview:

- Student: Ayden Deng
- ID: 2121072
- E-mail: ydeng24@depaul.edu

Purpose:

To consider:

- Constraint Satisfaction Problems
- Logic
- Logic Programming

I used:

- **Python3 for Problem 1**
- **Prolog for Problem 3**

Assignment:

0. Free points! (10 Points)

(Because I am such a *nice guy!* :)

1. Constraint Satisfaction Problems (30 Points)

- Write a program to solve the 8-queens problem systematically, with backtracking.

- Now, write another program to solve the 8-queens problem by placing a queen down in her own row, but in a random column. Find a queen with the most conflicts, and randomly assign her a different column.
- Which program did *less* work to solve the 8-queens problem?

Answer

- ***The random column placement method usually does less work.***
- *Time Complexity:*
  - *Backtracking: **O(N!)***
  - *Random: This complexity is typically polynomial - In the worst case, it might need to run infinite adjustments, but in general, it is usually faster than **O(N!)**. The specific complexity is hard to define strictly but is typically polynomial in nature.*
- *Space Complexity:*
  - *Backtracking: **O(N^2)***
  - *Random: **O(N)***
- *In conclusion, **the random column placement method (`problem1_random.py`) usually does less work to solve the 8-queens problem.***

Code & Output

```python
# problem1_backtracking.py
# python3 ./src/problem1_backtracking.py
# Time complexity: O(N!)
# Speed Complexity: O(N^2)

def print_board(board):
    for row in board:
        print(" ".join(str(col) for col in row))
    print("\n")

def is_safe(board, row, col):
    for i in range(row):
        if board[i][col] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, len(board))):
```

```python
        if board[i][j] == 1:
            return False
    return True

def solve_n_queens(board, row):
    if row >= len(board):
        return True
    for col in range(len(board)):
        if is_safe(board, row, col):
            board[row][col] = 1
            if solve_n_queens(board, row + 1):
                return True
            board[row][col] = 0
    return False

def main():
    n = 8
    board = [[0 for _ in range(n)] for _ in range(n)]
    if solve_n_queens(board, 0):
        print_board(board)
    else:
        print("No solution found")

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

```
aydendeng@loop-eduroam-236-181 assignment2 % python3 ./src/problem1_backtracking.py
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

Python ▾ Cancel wrap Copy

```python
# problem1_random.py
# python3 ./src/problem1_random.py
# Time complexity: This complexity is typically polynomial - In the worst case, it might need to run infinite adjustments, but in general, it is usually faster than O(N!). The specific complexity is hard to define strictly but is typically polynomial in nature.
# Speed Complexity: O(N)

import random
```

```python

def print_board(board):
    for row in board:
        print(" ".join(str(col) for col in row))
    print("\n")

def calculate_conflicts(board):
    conflicts = [0] * len(board)
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts[i] += 1
                conflicts[j] += 1
    return conflicts

def solve_n_queens_random(n):
    board = [random.randint(0, n - 1) for _ in range(n)]
    while True:
        conflicts = calculate_conflicts(board)
        if max(conflicts) == 0:
            return board
        max_conflict = max(conflicts)
        max_conflict_queens = [i for i, x in enumerate(conflicts) if x == max_conflict]
        queen = random.choice(max_conflict_queens)
        new_position = random.randint(0, n - 1)
        while new_position == board[queen]:
            new_position = random.randint(0, n - 1)
        board[queen] = new_position

def main():
    n = 8
    board = solve_n_queens_random(n)
    solution = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        solution[i][board[i]] = 1
    print_board(solution)

if __name__ == "__main__":
    main()

```

```
aydendeng@loop-eduroam-236-181 assignment2 % python3 ./src/problem1_random.py
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

---

2. Logic representation and reasoning (30 Points)

(**NOTE:** Below we are using mathematical notation. Variables are `lowercase`. Constants are `Uppercase`.)

Questions:

a. Translate the sentences above into First Order Logic. This will be your knowledge base.
   - Every runner is health-conscious
   - Someone is health conscious if and only if that person sees the doctor regularly
   - Anyone who sees the doctor regularly and who has a health problem has that problem found early
   - Anyone who has a health problem and who has that problem found early gets that problem successfully treated
   - `mike` is a person and is a runner.
   - `mike` has the health problem `diabetes`.

b. Translate your sentences to disjunctive form for resolution.

c. *Use resolution to show that `diabetes` problem of `mike` will be successfully treated.*

Answer:

a. Translate the sentences above into First Order Logic. This will be your knowledge base.
   - Every runner is health-conscious:
     - $\forall x(Runner(x) \rightarrow HealthConscious(x))$
   - Someone is health conscious if and only if that person sees the doctor regularly
     - $\forall x(HealthConscious(x) \leftrightarrow SeesDoctorRegularly(x))$
   - Anyone who sees the doctor regularly and who has a health problem has that problem found early

        - $\forall x \forall y(SeesDoctorRegularly(x) \land HasHealthProblem(x,y) \rightarrow ProblemFoundEarly(x,y))$
    - Anyone who has a health problem and who has that problem found early gets that problem successfully treated
        - $\forall x \forall y(HasHealthProblem(x,y) \land ProblemFoundEarly(x,y) \rightarrow SuccessfullyTreated(x,y))$
    - `mike` is a person and is a runner.
        - $Person(mike) \land Runner(mike)$
    - `mike` has the health problem `diabetes`.
        - $HasHealthProblem(mike,diabetes)$

b. Translate your sentences to disjunctive form for resolution.

    - Every runner is health-conscious:
        - $\forall x(Runner(x) \rightarrow HealthConscious(x))$
            - Since
                - $P \rightarrow Q = \neg P \lor Q$
            - Then
                - $\forall x(Runner(x) \rightarrow HealthConscious(x)) = \boldsymbol{\forall x(\neg Runner(x) \lor HealthConscious(x))}$
    - Someone is health conscious if and only if that person sees the doctor regularly
        - $\forall x(HealthConscious(x) \leftrightarrow SeesDoctorRegularly(x))$
            - Since
                - $P \leftrightarrow Q = (P \rightarrow Q) \land (Q \rightarrow P),\ P \rightarrow Q = \neg P \lor Q$
            - Then
                - $\forall x(HealthConscious(x) \leftrightarrow SeesDoctorRegularly(x))$
                    - $= \forall x((HealthConscious(x) \rightarrow SeesDoctorRegularly(x)) \land (SeesDoctorRegularly(x) \rightarrow HealthConscious(x)))$
                    - $= \forall x((\neg HealthConscious(x) \lor SeesDoctorRegularly(x)) \land (\neg SeesDoctorRegularly(x) \lor HealthConscious(x)))$
                    - $= (\neg HealthConscious(x) \lor SeesDoctorRegularly(x)) \land (\neg SeesDoctorRegularly(x) \lor HealthConscious(x))$
            - Hence
                - $\forall x(HealthConscious(x) \leftrightarrow SeesDoctorRegularly(x)) = \boldsymbol{(\neg HealthConscious(x) \lor SeesDoctorRegularly(x)) \land (\neg SeesDoctorRegularly(x)}$

$\lor HealthConscious(x))$

- Anyone who sees the doctor regularly and who has a health problem has that problem found early
  - $\forall x \forall y(SeesDoctorRegularly(x) \land HasHealthProblem(x,y) \rightarrow ProblemFoundEarly(x,y))$
    - Since
      - $P \rightarrow Q = \neg P \lor Q$
      - De Morgan's Laws: $\neg(A \land B) \equiv \neg A \lor \neg B$
    - Then
      - $\neg(SeesDoctorRegularly(x) \land HasHealthProblem(x,y)) \equiv \neg SeesDoctorRegularly(x) \lor \neg HasHealthProblem(x,y)$
      - $\forall x \forall y(\neg SeesDoctorRegularly(x) \lor \neg HasHealthProblem(x,y) \lor ProblemFoundEarly(x,y))$
    - Hence
      - $\neg SeesDoctorRegularly(x) \lor \neg HasHealthProblem(x,y) \lor ProblemFoundEarly(x,y)$
- Anyone who has a health problem and who has that problem found early gets that problem successfully treated
  - $\forall x \forall y(HasHealthProblem(x,y) \land ProblemFoundEarly(x,y) \rightarrow SuccessfullyTreated(x,y))$
    - Since
      - $P \rightarrow Q = \neg P \lor Q$
      - De Morgan's Laws: $\neg(A \land B) \equiv \neg A \lor \neg B$
    - Then
      - $\neg(HasHealthProblem(x,y) \land ProblemFoundEarly(x,y)) \equiv \neg HasHealthProblem(x,y) \lor \neg ProblemFoundEarly(x,y)$
      - $\forall x \forall y(\neg HasHealthProblem(x,y) \lor \neg ProblemFoundEarly(x,y) \lor SuccessfullyTreated(x,y))$
    - Hence
      - $\neg HasHealthProblem(x,y) \lor \neg ProblemFoundEarly(x,y) \lor SuccessfullyTreated(x,y)$
- `mike` is a person and is a runner.
  - $Person(mike) \land Runner(mike)$
    - $Person(mike)$
    - $Runner(mike)$

- `mike` has the health problem `diabetes`.
  - HasHealthProblem(mike,diabetes)

c. *Use resolution to show that `diabetes` problem of `mike` will be successfully treated.*

   i. Goal: SuccessfullyTreated(mike,diabetes)

      1. Since
         a. $\forall x(Runner(x) \to HealthConscious(x))$
         b. Then
            i. $Runner(mike) \to HealthConscious(mike)$
            ii. $\neg Runner(mike) \lor HealthConscious(mike)$
      2. Since
         a. $\forall x(HealthConscious(x) \leftrightarrow SeesDoctorRegularly(x))$
         b. Then
            i. $(HealthConscious(mike) \to SeesDoctorRegularly(mike)) \land (SeesDoctorRegularly(mike) \to HealthConscious(mike))$
            ii. $(\neg HealthConscious(mike) \lor SeesDoctorRegularly(mike)) \land (\neg SeesDoctorRegularly(mike) \lor HealthConscious(mike))$
      3. Since
         a. $\forall x \forall y(SeesDoctorRegularly(x) \land HasHealthProblem(x,y) \to ProblemFoundEarly(x,y))$
         b. Then
            i. $SeesDoctorRegularly(mike) \land HasHealthProblem(mike,diabetes) \to ProblemFoundEarly(mike,diabetes)$
            ii. $\neg SeesDoctorRegularly(mike) \lor \neg HasHealthProblem(mike,diabetes) \lor ProblemFoundEarly(mike,diabetes)$
      4. Since
         a. $\forall x \forall y(HasHealthProblem(x,y) \land ProblemFoundEarly(x,y) \to SuccessfullyTreated(x,y))$
         b. Then
            i. $HasHealthProblem(mike,diabetes) \land ProblemFoundEarly(mike,diabetes) \to SuccessfullyTreated(mike,diabetes)$
            ii. $\neg HasHealthProblem(mike,diabetes) \lor \neg ProblemFoundEarly(mike,diabetes) \lor SuccessfullyTreated(mike,diabetes)$
      5. We knew

a. *Runner(mike)*

b. *HasHealthProblem(mike,diabetes)*

ii. *Let's proof*

1. *Since*

   a. *Runner(mike) & ¬Runner(mike) VHealthConscious(mike)*

   b. *Then*

      i. *HealthConscious(mike)*

2. *Since*

   a. *HealthConscious(mike)*

   b. *¬HealthConscious(mike) VSeesDoctorRegularly(mike)*

   c. *Then*

      i. *SeesDoctorRegularly(mike)*

3. *Since*

   a. *SeesDoctorRegularly(mike)*

   b. *HasHealthProblem(mike,diabetes)*

   c. *¬SeesDoctorRegularly(mike) V¬HasHealthProblem(mike,diabetes) VProblemFoundEarly(mike,diabetes)*

   d. *Then*

      i. *ProblemFoundEarly(mike,diabetes)*

4. *Since*

   a. *HasHealthProblem(mike,diabetes)*

   b. *ProblemFoundEarly(mike,diabetes)*

   c. *¬HasHealthProblem(mike,diabetes) V¬ProblemFoundEarly(mike,diabetes) VSuccessfullyTreated(mike,diabetes)*

      i. *Then*

         1. ***SuccessfullyTreated(mike,diabetes)***

5. ***Hence, the `diabetes` problem of `mike` will be successfully treated.***

---

3. Prolog (30 Points)

(**NOTE:** Below we are using Prolog notation. Variables are `Uppercase`. Constants are `lowercase`.)

- In class I showed how 2 sorted-lists can be merged in a sorted fashion. That means we are *so close* to implementing `merge-sort`! We will implement `merge-sort` below.

Example output:

```
mergeSort(\[\],L).
L = \[\]
mergeSort(\[2\],L).
L = \[2\]
mergeSort(\[4,2,3,1\],L).
L = \[1, 2, 3, 4\]
mergeSort(\[5,4,6,2,7,3,1,8\],L).
L = \[1, 2, 3, 4, 5, 6, 7, 8\]
```

Our program will be implemented with 5 predicates:

- `mergeSort/2`: which in comes an unsorted list, out goes the sorted list.
- `merge/3`: which merges the
- `split2ways/3`:
- `insertIn1/5` and `insertIn2/5`:

So let us go!

Steps:

1. Finish `mergeSort/2`:

```
mergeSort(\[\],\[\])        :- !.        # Base case when list is empty
mergeSort(\[X\],\[X\])        :- !.        # Base case when list has only 1

# YOUR CODE TO HANDLE GENERAL CASE
# (1) Split the in coming list into 2 sublists
# (2) Recursively sort both lists
# (3) Merge both lists into the output list
```

2. Here is all the code for `merge/3`.

(This is all of it, there is nothing else to add.)

```
merge(\[\],L,L).
merge(L,\[\],L).
merge(\[F0|R0\],\[F1|R1\],\[F0|L\])        :-
    (F0 < F1),
merge(R0,\[F1|R1\],L).
merge(\[F0|R0\],\[F1|R1\],\[F1|L\])        :-
    (F0 >= F1),
    merge(\[F0|R0\],R1,L).
```

3. Here is all the code for `split2ways/3`.

(This is all of it, there is nothing else to add.)

```
split2ways(Original,R1,R2)        :-
insertIn1(Original,\[\],\[\],R1,R2).
```

4. You must finish the predicates

   - `insertIn1()` (`insertIn1/5`),
   - `insertIn2()` (`insertIn2/5`),

   Both have 5 arguments:

   - The incoming list
   - 2 lists which hold temporary lists that build the answers
   - 2 lists which will hold the answers

   Both predicates will have the same base case (_facts_) corresponding to when the list to split is empty. It is:

```
insertIn1(\[\],R1,R2,R1,R2).
```

It means "*When there are no items to split, the temporary lists `R1` and `R2` are also the answer lists.*"

Your job is to write the 2 *rules* that build the temporary lists:

```
# YOUR insertIn1 RULE HERE

insertIn1(\[\],R1,R2,R1,R2).          # This is the base case

# YOUR insertIn2 RULE HERE

insertIn2(\[\],R1,R2,R1,R2).          # This is the base case
```

What are the remaining 2 rules?

- Hints:
  - Both should separate the list to split into a head and a tail (`[H|T]`).
  - Both should build the a list a their respective position
    - `insertIn1/5` should call `insertIn2/5`
    - `insertIn2/5` should call `insertIn1/5`

Answer:

Code

```
mergeSort([], []) :- !.
mergeSort([X], [X]) :- !.
mergeSort(List, Sorted) :-
    split2ways(List, Left, Right),
    mergeSort(Left, SortedLeft),
    mergeSort(Right, SortedRight),
    merge(SortedLeft, SortedRight, Sorted).

merge([], L, L).
merge(L, [], L).
merge([F0|R0], [F1|R1], [F0|L]) :-
    F0 < F1,
    merge(R0, [F1|R1], L).
merge([F0|R0], [F1|R1], [F1|L]) :-
```

```
    F0 >= F1,
    merge([F0|R0], R1, L).

split2ways(Original, R1, R2) :-
    insertIn1(Original, [], [], R1, R2).

insertIn1([], R1, R2, R1, R2).
insertIn1([H|T], R1, R2, FinalR1, FinalR2) :-
    insertIn2(T, [H|R1], R2, FinalR1, FinalR2).

insertIn2([], R1, R2, R1, R2).
insertIn2([H|T], R1, R2, FinalR1, FinalR2) :-
    insertIn1(T, R1, [H|R2], FinalR1, FinalR2).

```

Output

```
?- [src/problem3_merge_sort].
true.

?- mergeSort([], L).
L = [].

?- mergeSort([2], L).
L = [2].

?- mergeSort([4, 2, 3, 1], L).
L = [1, 2, 3, 4] .

?- mergeSort([5, 4, 6, 2, 7, 3, 1, 8], L).
L = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.5)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- [src/problem3_merge_sort].
true.

?- mergeSort([], L).
L = [].

?- mergeSort([2], L).
L = [2].

?- mergeSort([4, 2, 3, 1], L).
L = [1, 2, 3, 4] .

?- mergeSort([5, 4, 6, 2, 7, 3, 1, 8], L).
L = [1, 2, 3, 4, 5, 6, 7, 8]
```