CP600 Assignment 13

Donna J. Harris
April 11, 2022

Problem:

1. Implement the generic ***Backtrack*** algorithm from the lesson note in Python.
2. Use your ***Backtrack*** algorithm to solve the n-queens problem in Python, by providing appropriate implementation for the ***IsSolution***, ***ConstructCandidates***, ***Process*** and ***IsFinished*** functions
3. Modify your callback functions to find the total number of solutions you found for 8-queens, 10-queens and 12-queens problem, as well as the number of recursive calls for each case. Report your findings in a table.
4. Put the result table generated in previous question in a13.pdf. The table has the result of solving n-queens problem for (n=8, 10, 12) as above.
5. Explain any insight that you can derive from the table.

Solution:

Pseudocode

```
def Backtrack(A, k, S):
      # input:   A, Current partial candidate solution;
      #          k, Number of queens placed;
      #          S, Set of input values (unused in this implementation)
      # output:  Returns True, if configured for a single solution

1.    IF IsSolution(A, k, S):
 a.         Process(A, k, S)
2.    ELSE:
 a.         L ← ConstructCandidates(A, k, S)
 b.         FOR c IN L:
  i.              A[k] ← c
 ii.              INCREMENT global recursive_call_count
iii.              Backtrack(A, k+1, S)
 iv.              IF IsFinished():
  1.                    RETURN True
```

```
def ConstructCandidates(A, k, S):
     # input:   A, Current partial candidate solution;
     #          k, Number of queens placed;
     #          S, Set of input values (unused in this implementation)
     # output:  Returns set of candidates

1.   Result ← []
2.   FOR i FROM 1 TO LENGTH(A):
 a.       hasThreat ← False
 b.       FOR j FROM 1 TO k:
  i.           IF A[j] EQUALS i OR |j-k| EQUALS |i-A[j]|:
  1.                hasThreat ← True
 c.       IF NOT hasThreat:
  i.           Result ← Result + i
3.   RETURN Result
```

```
def IsSolution(A, k, S):
     # input:   A, Current partial candidate solution;
     #          k, Number of queens placed;
     #          S, Set of input values (unused in this implementation)
     # output:  Returns True, if solution found; Otherwise, returns False

1.   RETURN k > LENGTH(A)-1
```

```
def Process(A, k, S):
     # input:   A, Current partial candidate solution;
     #          k, Number of queens placed;
     #          S, Set of input values (unused in this implementation)
     # output:  Returns nothing (in this implementation)

1.   INCREMENT global solution_count
2.   PRINT A[1:LENGTH(A)]   # Print the part that ignores index 0
```

```
def IsFinished(A, k, S):
     # input:   A, Current partial candidate solution;
     #          k, Number of queens placed;
     #          S, Set of input values (unused in this implementation)
     # output:  Returns False (not implemented for single-solution support)

1.   RETURN False
```

Algorithmic Analysis

IsSolution(), **Process**(), and **IsFinished**() are constant time. **ConstructCandidates**() is more complex, with a worst case of O(n^2). **Backtrack**() is the function that is called recursively, so it is the one of greatest interest and the aspect that dominates, as the others are all sequentially occurring.

Every time **Backtrack**() calls itself, it has one less thing to solve within the n-length candidate solution, which points towards n!. These calls are also happening for every one of the candidate solutions available at that level, which, in a worst-case situation, is n. This leads us to a time complexity that is greater than O(n!), and closer to an upper bound of ***O(n(n!))***.

Implementation Notes

Question 1 Python file: p1.py
Implementation of the initial Backtrack() algorithm, no executing code.

Question 2 Python file: p2.py
Implementation of the n-queens problem, with a main() that demonstrates for n=8.

Question 3 Python file: p3.py
Implementation of the n-queens problem, for n=8, 10, and 12 and presenting a results table.

Note: all progress output is silent by default. Only the results will be displayed after all calculations are complete.
To toggle this setting, change PRINT_ON to True at the top of the p3.py file.

Results:

n	**Number of Possible Solutions**	**Number of Recursive Calls Made**
8	92	2056
10	724	35538
12	14200	856188

Response:

The first thing I noticed when I ran p3.py during development (with many intermediate print statements still throughout) was how n=8 and n=10 returned results without a noticeable delay but n=12 took a noticeable length of time to complete. I even hesitated to remove print statements that showed progress, since it can look as though the program is hanging.

Getting into the numbers more deeply,

n	**Calls per Solution** *(calls / solution)*	**Calls per n** *(calls / n)*	**Growth of Calls per n** *(calls per n_2 / calls per n_1)*
8	22.34782609	257	-
10	49.08563536	3553.8	13.82801556
12	60.29492958	71349	20.07681918

We can see that as n increases, even in the range from 8 to 12, the recursive calls made per number of queens grows very quickly when compared to the number of solutions being generated. We see a 20 times increase in the number of calls just from increasing n=10 to n=12.

There is a Wikipedia page that features a table of the calculated number of solutions for up to 27 queens (https://en.wikipedia.org/wiki/Eight_queens_puzzle#Counting_solutions_for_other_sizes_n) and the number of solutions grows by full orders of magnitude and is in high exponential growth.

But what we're seeing in the assignment's limited problem space is how quickly the number of recursive calls grows to a tremendous number. The trend within our own data, with knowledge of the known results, shows that the number of recursive calls for this approach will become staggeringly large at an exceptionally quick rate and likely would not be reasonable to execute on an average machine for much beyond n=12. I was curious and tried n=14 for fun, only to be met with a very noisy, constantly whirring fan, using 99% of my CPU power, and a long wait before I manually cancelling it.

We find here that the recursion is a limiting factor as the number of queens increases. The number of solutions is only a very small subset of all the possibilities in the problem space. Even when strategically limiting the problem space, there is still a large and growing gap between the number of possibilities and actual solutions and an extreme number of recursive calls necessary to calculate all of those possibilities.