Backtracking & Brute Force Optimization

Intro2CS – weeks 12-13
Backtracking

• A general way (using recursion) to find a solution to a problem.
• Given a partial solution, we have several options in extending it.
• Try one option, and if that doesn’t work, backtrack (undo the change) and try another

• Often used for “constraint satisfaction problems”
Sudoku

```
 8  9  3  
 5  5  5  
 1  2  7  
 6  7  2  
 1  7  8  
 3  5  9  
 6  9  2  
```


Sudoku Solving

BOARD_SIZE = 9
QUADRANT_SIZE = 3

def solve_sudoku(board):
    return _sudoku_helper(board, 0)

def _sudoku_helper(board, ind):
    ...

def is_legal(board, row, col, num):
    ...

def print_board(board):
    ...
def print_board(board):
    for row in board:
        for num in row:
            if(num == 0):
                print(' ', end=' ')
            else:
                print(num, end=' ')
    print()
def is_legal(board, row, col, num):
    for x in range(BOARD_SIZE):
        if board[x][col] == num and x != row:
            return False
    if board[row][x] == num and x != col:
        return False

    quad_row = row - row % QUADRANT_SIZE
    quad_col = col - col % QUADRANT_SIZE

    for ind in range(BOARD_SIZE):
        x = quad_row + ind // QUADRANT_SIZE
        y = quad_col + ind % QUADRANT_SIZE
        if board[x][y] == num and (x != row and y != col):
            return False
    return True
def _sudoku_helper(board, ind):
    if ind >= BOARD_SIZE ** 2:
        print_board(board)
        return True
    row = ind // BOARD_SIZE
    col = ind % BOARD_SIZE

    if board[row][col]:
        return _sudoku_helper(board, ind + 1)

    for num in range(1, BOARD_SIZE + 1):
        if is_legal(board, row, col, num):
            board[row][col] = num
            if _sudoku_helper(board, ind + 1):
                return True
            board[row][col] = 0
    return False
board = [[0, 0, 0, 0, 0, 0, 0, 0, 0],
         [8, 0, 0, 0, 9, 1, 3, 0, 5],
         [0, 0, 0, 5, 0, 7, 0, 6, 9],
         [0, 3, 0, 0, 5, 0, 9, 0, 4],
         [0, 0, 0, 0, 0, 0, 0, 0, 0],
         [1, 0, 5, 0, 2, 0, 0, 7, 0],
         [6, 1, 0, 7, 0, 9, 0, 0, 0],
         [0, 7, 4, 8, 6, 0, 0, 0, 2],
         [0, 0, 0, 0, 0, 0, 0, 0, 0]]

solve_sudoku(board)
Questions

• Can you estimate the number of boards we need to explore?
• Provide a bound?

• How many of these will yield legal Sudoku puzzles?
Number of valid Sudoku boards

• There are around $6.67 \times 10^{21}$ legal Sudoku boards. This was estimated via search.

• Notice that a board can be relabeled to obtain a different legal board.

• Some permutations on the columns / rows also yield new boards.
The 8 Queen Problem

• Find a way to place all 8 queens on a chess board so that:
  • No two queens are on the same row
  • No two queens are on the same column
  • No two queens are on the same diagonal

• Again, how can we estimate how many board configurations there are (incl. illegal ones)?
Counting solutions

The following table gives the number of solutions for placing $n$ queens on an $n \times n$ board, both fundamental and total.

<table>
<thead>
<tr>
<th>$n$:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>..</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fundamental:</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>12</td>
<td>46</td>
<td>92</td>
<td>341</td>
<td>1,787</td>
<td>9,233</td>
<td>45,752</td>
<td>..</td>
</tr>
<tr>
<td><strong>all:</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>4</td>
<td>40</td>
<td>92</td>
<td>352</td>
<td>724</td>
<td>2,680</td>
<td>14,200</td>
<td>73,712</td>
<td>365,596</td>
<td>..</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24</th>
<th>25</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>28,439,272,956,934</td>
<td>275,986,683,743,434</td>
<td>2,789,712,466,510,289</td>
</tr>
</tbody>
</table>

(Fundamental solutions can be rotated / reflected to obtain other solutions)
• We use an 8x8 Boolean list as a board.
• A cell is True if a queen is placed there there.

BOARD_SIZE = 8
QUEEN = 'Q'
EMPTY = '_'

def print_board(board):
    print("=="*BOARD_SIZE)
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col]:
                print(QUEEN, end=' ')
            else:
                print(EMPTY, end=' ')
    print()
def can_place_queen_at(board, row, column):

def _queen_helper(board, column):

def find_solutions():
    board = [[False]*BOARD_SIZE for _ in range(BOARD_SIZE)]
    _queen_helper(board, 0)
def can_place_queen_at(board, row, column):
    pass

def _queen_helper(board, column):
    if column == BOARD_SIZE:
        print_board(board)
        return True
    for row in range(BOARD_SIZE):
        if can_place_queen_at(board, row, column):
            board[row][column] = True
            if _queen_helper(board, column+1):
                return True
            board[row][column] = False
    return False
def can_place_queen_at(board, row, col):
    for ind in range(1, BOARD_SIZE):
        if board[(row+ind) % BOARD_SIZE][col]:
            return False
        if board[row][(col+ind) % BOARD_SIZE]:
            return False

    # diagonals:
    if row+ind<BOARD_SIZE:
        if col+ind<BOARD_SIZE:
            if board[row+ind][col+ind]:
                return False
        if col-ind>=0:
            if board[row+ind][col-ind]:
                return False
    if row-ind>=0:
        if col+ind<BOARD_SIZE:
            if board[row-ind][col+ind]:
                return False
        if col-ind>=0:
            if board[row-ind][col-ind]:
                return False
    return True
Optimization

• Given some possible (large) set: $U$
• A value function $V: U \rightarrow \mathbb{R}$

• The task: to find $x \in U$ that has maximal value.
  
  – (minimization problems are equivalent to maximizing the negation of the value)
Solving this sort of basic task is very beneficial.

The simplest algorithm:
• go over all $x \in U$.
• Evaluate $V(x)$.
• Keep the maximal.

Unfortunately, most interesting problems take a long time to solve.

(U is really large in most interesting cases).
Example

• The Knapsack problem

• A thief breaks into a house and wants to steal items.
• There are $n$ items.
• Item $i$ has a size $s_i$, and a value $v_i$.
• The thief wants the set of items with the highest total value that can fit into the bag.
Knapsack

- To define as an optimization problem:
  - \( U = \{ I : I \subseteq \{1,2,\ldots,n\}, \sum_{i \in I} s_i \leq BagSize \} \)
  - \( V(I) = \sum_{i \in I} v_i \)

- What is the size of \( U \)?
- How do we solve the problem?
• Another variant: each item has a weight, a size, and a value. The thief is limited both in weight and size.

• In python...

```python
class Item:
    def __init__(self, weight, size, value):
        self.weight = weight
        self.size = size
        self.value = value

    def __repr__(self):
        return str((self.weight, self.size, self.value))
```
class Knapsack:
    def __init__(self, max_weight, max_size):
        self.items = []
        self.max_weight = max_weight
        self.max_size = max_size
        self.total_weight = 0
        self.total_size = 0
        self.total_value = 0

    def put(self, item):

    def can_hold(self, item):

    def remove(self, item):

    def copy(self, other):
def put(self, item):
    if self.can_hold(item):
        self.items.append(item)
        self.total_size += item.size
        self.total_weight += item.weight
        self.total_value += item.value

def can_hold(self, item):
    return item not in self.items and \
    item.weight + self.total_weight < self.max_weight and \
    item.size + self.total_size < self.max_size
def remove(self, item):
    if item in self.items:
        self.items.remove(item)
        self.total_size -= item.size
        self.total_weight -= item.weight
        self.total_value -= item.value

def copy(self, other):
    self.max_weight = other.max_weight
    self.max_size = other.max_size
    self.total_weight = other.total_weight
    self.total_size = other.total_size
    self.total_value = other.total_value
    self.items = other.items[:]
Idea for the algorithm

• Use recursion (backtracking).

• Given a partially filled bag, and a set of items

• Try with the first item (& make recursive call)
  – If the first item doesn’t fit, skip it.

• Try without the first item (& make recursive call)
```python
def find_best(items):
    bag = Knapsack(MAX_WEIGHT, MAX_SIZE)
    best_bag = Knapsack(MAX_WEIGHT, MAX_SIZE)
    return _find_best_helper(items, bag, best_bag, 0)

items = [Item(random.random(), random.random(), random.random())
         for _ in range(NUM_ITEMS)]
print("The best result: ", find_best(items))
```
def _find_best_helper(items, knapsack, best_bag, index):
    max_value = knapsack.total_value

    if index == len(items):
        if (max_value > best_bag.total_value):
            best_bag.copy(knapsack)
            print("found", best_bag.total_value, best_bag.items)
        return max_value

    if knapsack.can_hold(items[index]):
        knapsack.put(items[index])
        max_value = max(max_value,
                        _find_best_helper(items, knapsack, best_bag, index + 1))
        knapsack.remove(items[index])

    max_value = max(max_value,
                    _find_best_helper(items, knapsack, best_bag, index + 1))
    return max_value
Back to N-Queens

We can phrase the N-Queen problem as an optimization problem:

- Find the board setting that has the fewest violated constraints.

- If we find the minimal number of violated constraints, and it happens to be 0, we have a legal solution.
Are there alternatives to brute force search?

• Yes!

• We can try guessing: start with random assignments until we find a good solution.

• We can try to have a solution with conflicts and try to move only conflicted pieces.

• Unfortunately, they are not much more efficient.
```python
def set_random(board):
    conflicts = 0
    for col in range(BOARD_SIZE):
        row = random.randint(BOARD_SIZE)
        if not can_place_queen_at(board, row, col):
            conflicts += 1
            board[row][col] = True
    return conflicts
```
```python
def try_random_solutions():
    board = [[False]*BOARD_SIZE for _ in range(BOARD_SIZE)]
    counter = 0
    best = BOARD_SIZE+1
    while best:
        counter+=1
        clear_board(board)
        num_conflicts = set_random(board)
        if num_conflicts < best:
            best = num_conflicts
            print("round:",counter," found:", num_conflicts, "conflicts")
    print_board(board)
```
```python
def try_changing_conflicted():
    board = [[False]*BOARD_SIZE for _ in range(BOARD_SIZE)]
    num_conflicts = set_random(board)
    best = num_conflicts
    counter = 0
    while best:
        col = random.randrange(BOARD_SIZE)
        for row in range(BOARD_SIZE):
            if board[row][col]:
                if not can_place_queen_at(board, row, col):
                    counter+=1
                    board[row][col] = False
                    board[random.randrange(BOARD_SIZE)][col] = True
                if not counter%1000:
                    print("round:", counter,
                          " found:", num_conflicts, "conflicts")
                    print_board(board)
                    break
        num_conflicts = count_conflicts(board)
    if num_conflicts < best:
        best = num_conflicts
        print_board(board)
```