ICC4101 - Algorithms and Competitive Programing

Lecture 3 - Complete Search (Brute force)

Javier Correa

Today's contingency

- You can leave early and work from home
- We will end the lecture at most at 18:00
- You can send today's problems untill 23:59 without penalization

Complete search is a simple approach to solve any problem and in a programming contest, it should be the one of the first approaches to test.

If correctly implemented, you should never receive a Wrong Answer, maybe a Timeout, which would mean to explore more sophisticated solutions.

Iterative Complete Search

The state space of the problem can be generated by a number of `for` construct.

This means that the state `s` of a problem can be modeled as a vector with fix size:

$$\mathbf{s} = [s_1, s_2, \ldots, s_N]$$

A complete search solution would imply the creation of all possible combinations. Something like:

```
for s1 in valuesOfS(1):
    for s2 in valuesOfS(2):
        ...
            for sn in valuesOfS(n):
                s = [s1, s2, ..., sn]
                evaluate(s)
```

Remember, n is fixed.

Some important points:

- To avoid the exploration of impossible states, **you should prune as much as possible!**
- Be comfortable with nested `for` loops
- For some of the problems you can use `Python` tools to aid you, e.g. `itertools.product`, `itertools.permutations`, `itertools.combinations` and `itertools.combinations_with_replacement`.

Toy example

Find and display all pairs of 5-digit numbers that collectively use the digits 0 through 9 once each, such that the first number divided by the second is equal to an integer `N`, where 2 ≤ `N` ≤ 79.

That is, `abcde / fghij = N`, where each letter represents a different digit. The first digit of one of the numbers is allowed to be zero, e.g. for N = 62, we have 79546 / 01283 = 62; 94736 / 01528 = 62.

Idea: iterate over `fghij` and calculate `abcde`, check that between both numbers all digits are present.

Pruning: the minimum value for `fghij` is 1234, while the maximum is `98765 / N`.

In [5]:
```python
N = 46
# First for loop
for fghij in range(1234, (98765 // N) + 1):
    abcde = "{:05d}".format(N * fghij)
    fghij = "{:05d}".format(fghij)
    combined = fghij + abcde

    # Check that all digits (0-9) are present
    digits = set(combined)

    if len(digits) == 10:
        print(f"{abcde} / {fghij} = {N}")
```

```
58374 / 01269 = 46
```

Recursive Complete Search - Backtracking

Backtracking is a more general (meta-)algorithm to explore more complex state spaces.

Backtracking

General backtracking procedure, given a partial solution s:

- Verify if s is a solution. If s is a solution, process it (problem dependent).
- Create all extended solution starting at s.
- Verify border conditions.
- Recursively call this procedure for all extended solutions.

Backtracking

Backtracking can be used to explore all solution, find the first solution (with a property in particular) or find the optimal solution.

Solution space

Let's assume that a solution can be modeled with a vector $s = (s_1, s_2, \ldots s_n)$, where each s_i can take values from a *finite* set $\mathbf{S}_i$ and n can vary between different solutions.

- A candidate solution is of the form:

$$s_T = (s_1, \ldots, s_k)$$

- Extending the solution s_T is achieved by adding an element:

$$s_{T+1} = (s_1, \ldots, s_T, s_{T+1})$$

Generic Algorithm

```
def backtracking(s):
    if is_solution(s):
        process_solution(s)
        return True # Or continue exploring
    for s_p1 in extend_solution(s):
        if not test(s_p1):
            continue
        result = backtracking(s_p1)
        if result:
            return True # Or continue exploring
    return False
```

Similar, but using `filter` (should run a little bit faster)

```
def backtracking(s):
    if is_solution(s):
        process_solution(s)
        return True # Or continue exploring
    for s_p1 in filter(test, extend_solution(s)):
        result = backtracking(s_p1)
        if result:
            return True # Or continue exploring
    return False
```

Generic Algorithm

- `is_solution(·)` indicates if the argument if a complete solution to the problem.
- `process_solution(·, ·)` process a/the solution to the problem.
- `extend_solution(·)` given a partial solution, return/generates all solution one step larger.
- `test(.)` this function returns true if the extended solution is a valid solution.

Note that `test` could be optional if its logic is included within `extend_solution`.

Generic Algorithm

- How to model s?
- How to extend a solution?
- What to do to process the solution?

Toy example

- Find *all* subsets of size n of a total of m elements.

Toy example

- We model subsets as an binary array `s[]` in which if `s[i]==True` indicates that the ith element belongs to the subset.
- We extend a partial solution appending either a `True` or a `False` to the end of the partial solution.
- We print a solution once we found it
  - A partial solution is any such that $\sum_i s[i] < n$ and `len(s) < m`.
  - A solution to the problem is any such $\sum_i s[i] = n$ and `len(s) == m`.

Toy example

In [6]:
```python
def is_solution(a, n, m):
    """
    Check if a partial solution is a solution to the problem
    """
    if len(a) == m and sum(a) == n:
        return True
    else:
        return False

def extend_solution(a, m):
    """
    Extend a partial solution
    """
    if len(a) < m:
        for c in [True, False]:
            yield a + [c]
```

In [7]:
```python
def process_solution(a):
    """
    Process only prints a solution
    """
    friendly = map(lambda x: str(x[0]), filter(lambda x: x[1], enumerate(a)))
    #friendly = []
    #for i,ai in enumerate(a):
    #    if ai:
    #        friendly.append(str(i))
    print("Subset with the following elements " + ", ".join(friendly))

def backtracking(n, m, a=[]):
    """
    Main backtracking
    """
    if is_solution(a, n, m):
        process_solution(a)
        return
    for a_s in extend_solution(a, m):
        backtracking(n, m, a_s)
```

In [8]:
```python
# Call backtracking to find all subsetds of size 3 out of a set with 5 eleme
backtracking(3, 4)
```

```
Subset with the following elements 0, 1, 2
Subset with the following elements 0, 1, 3
Subset with the following elements 0, 2, 3
Subset with the following elements 1, 2, 3
```

Task assignment

Given n workers and n tasks and a cost matrix, e.g. `C[i,j]` represents how many man-hours it takes worker `i` to complete task `j`, you are to find the task assignment (assign each worker to fulfill a task) that minimizes the total cost.

Task assignment

- We model the assignment as a list of tuples `(w, t)` which assigns worker `w` to task `t`.
- We extend a solution by adding a new tuple to the list of an unassigned worker to an unfulfilled task.
- Once we find a valid solution, we calculate it's cost and keep the assignment with the lowest cost
  - A valid assignment is one that all workers have a task assigned and all task have a worker assigned to them

In [9]:

```python
def is_solution(assignment, n):
    """
    A solution is complete when all workers and all tasks are covered.
    """
    if len(assignment) == n:
        workers = set(w for (w,t) in assignment)
        tasks = set(t for (w,t) in assignment)
        if len(workers) != n or len(tasks) != n:
            return False
        return True
    else:
        return False
```

In [10]:

```python
def extend_solution(assignment, n):
    """
    Extend an assignment by assignming a free worker to an unfulfilled task
    """
    workers = sorted([t for (t,T) in assignment])
    tasks = sorted([T for (t,T) in assignment])

    free_workers = [x for x in range(n) if x not in workers]
    free_tasks = [x for x in range(n) if x not in tasks]

    # assignment = [(1,0)]
    for w in free_workers:
        for t in free_tasks:
            yield assignment + [(w, t)]
            # [(1,0), (2,1)]
            # [(1,0), (2,2)]
            # [(1,0), (2,3)]
```

In [11]:
```python
def process_solution(assignment, costs):
    """
    Once we find a solution, check if it is the best solution
    """
    global best_assignment

    cost = sum(costs[w][t] for (w,t) in assignment)
    if best_assignment is None or cost < best_assignment[1]:
        best_assignment = (assignment, cost)

def backtracking(costs, assignment=[]):
    """
    Main backtracking
    """
    if is_solution(assignment, len(costs)):
        process_solution(assignment, costs)
    else:
        for a_s in extend_solution(assignment, len(costs)):
            backtracking(costs, a_s)
```

In [12]:
```python
best_assignment = None
costs=[[4, 2, 3, 1],
       [9, 3, 4, 2],
       [2, 4, 6, 2],
       [7, 3, 1, 0]]

backtracking(costs)

print(u"The best assignment has a cost of {0}, corresponding to:".format(bes
for (i,j) in best_assignment[0]:
    print("assign worker {0} to task {1}".format(i, j))
```

```
The best assignment has a cost of 7, corresponding to:
assign worker 0 to task 1
assign worker 1 to task 3
assign worker 2 to task 0
assign worker 3 to task 2
```

Sudoku

- A Sudoku is a `9x9` board where no repeated values are not allowed by looking at each row, column and `3x3` block.

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Solving sudoku with backtracking

- How do we model a solution?
- How do we extend a solution?
- What do we do when we find a solution?

Solving sudoku

We model a solution as a list (or tuple) of 81 elements, each one representing a cell of the puzzle.

If a cell has a number, the same number is located in the corresponding position of the list, otherwise its a None.

In [13]:
```
N = None
sudoku = [N, N, 3, 9, N, N, N, 5, 1,
          5, 4, 6, N, 1, 8, 3, N, N,
          N, N, N, N, N, 7, 4, 2, N,
          N, N, 9, N, 5, N, N, 3, N,
          2, N, N, 6, N, 3, N, N, 4,
          N, 8, N, N, 7, N, 2, N, N,
          N, 9, 7, 3, N, N, N, N, N,
          N, N, 1, 8, 2, N, 9, 4, 7,
          8, 5, N, N, N, 4, 6, N, N]
```

In [14]:
```python
def is_solution(sol):
    """
    Check if a partial solution is a full solution
    """
    none_cells = sum(map(lambda x: x is None, sol))
    if none_cells == 0:
        return True
    else:
        return False
```

Solving sudoku

- We extend a partial solution by adding a new entry to the board

In [15]:
```python
def extend_solution(sol):
    """
    Extend a solution by adding a new element to the board
    """
    none_cells = sum(map(lambda x: x is None, sol))
    if none_cells > 0:
        indx = next(i for i in range(81) if sol[i] is None)
        for value in range(1,10):
            new_sol = list(sol)
            new_sol[indx] = value
            yield new_sol
```

Solving sudoku

- Use a `test` function to see if the solution we have created is still a valid solution. Check rows, cols and `3x3` blocks for repeated elements.

In [16]:
```python
def test(sol):
    """
    Check if the partial solution has valid entries
    """
    for i in range(9):
        partial = [[], [], []]
        for j in range(9):
            if sol[9*i + j] is not None:
                partial[0].append(sol[9*i + j])
            if sol[i + 9*j] is not None:
                partial[1].append(sol[i + 9*j])
            if sol[9*(j//3) + j%3 + (i//3)*27 + (i%3)*3] is not None:
                partial[2].append(sol[9*(j//3) + j%3 + (i//3)*27 + (i%3)*3])
        for k, partial_ in enumerate(partial):
            if len(list(set(partial_))) != len(partial_):
                return False
    return True
```

Solving sudoku

- Once we find a solution, print the board

In [17]:
```
def process_solution(sol):
    """
    Print solution
    """
    for i in range(9):
        if i % 3 == 0:
            print("+-------+-------+-------+")
        print("|", sol[9*i], sol[9*i+1], sol[9*i+2], "|", sol[9*i+3], sol[9*
    print("+-------+-------+-------+")
```

Solving sudoku

- Main backtracking

In [18]:
```
def backtracking(sol=[None]*81):
    """
    Sudoku backtracking
    """
    if is_solution(sol):
        process_solution(sol)
        return True
    for next_sol in extend_solution(sol):
        if test(next_sol):
            result = backtracking(next_sol)
            if result:
                return True
    return False
```

In [19]:
```
sudoku = [N, N, 3, 9, N, N, N, 5, 1,
          5, 4, 6, N, 1, 8, 3, N, N,
          N, N, N, N, N, 7, 4, 2, N,
          N, N, 9, N, 5, N, N, 3, N,
          2, N, N, 6, N, 3, N, N, 4,
          N, 8, N, N, 7, N, 2, N, N,
          N, 9, 7, 3, N, N, N, N, N,
          N, N, 1, 8, 2, N, 9, 4, 7,
          8, 5, N, N, N, 4, 6, N, N]
```

In [20]:
```
backtracking(sudoku)
```

```
+-------+-------+-------+
| 7 2 3 | 9 4 6 | 8 5 1 |
| 5 4 6 | 2 1 8 | 3 7 9 |
| 9 1 8 | 5 3 7 | 4 2 6 |
+-------+-------+-------+
| 1 6 9 | 4 5 2 | 7 3 8 |
| 2 7 5 | 6 8 3 | 1 9 4 |
| 3 8 4 | 1 7 9 | 2 6 5 |
+-------+-------+-------+
| 4 9 7 | 3 6 1 | 5 8 2 |
| 6 3 1 | 8 2 5 | 9 4 7 |
| 8 5 2 | 7 9 4 | 6 1 3 |
+-------+-------+-------+
```

Out[20]:

```
True
```

Problems:

Problem 1. Necklace

`https://www.udebug.com/UVa/11001`

Problem 2. Ant's Shopping Mall

`https://www.udebug.com/UVa/12498`

Problem 3. All Walks of length n from the first node

`https://www.udebug.com/UVa/677`

Problem 4. Movie Police

`https://www.udebug.com/UVa/12515`

Problem 5. Little Bishops

`https://www.udebug.com/UVa/861`

Problem 6. Integer Sequences from Addition of Terms

`https://www.udebug.com/UVa/927`

In [ ]: