Search and Optimization

Carlos Tavares

December 6, 2024

Search, Optimization and Counting i

Search and optimization problems are everywhere in computer science:

Search (S): which element of the search space constitutes the solution.

Optimization (O): which elements of the search space constitute the **best** solution.

Counting (C): How many elements of the search space are a solution to the problem.

Search, Optimization and Counting ii

Typically they can phrased as instances of one another, but the optimality problems are typically harder than searching ones, and counting are harder than optimality problems.

$$S \subseteq O \subseteq C$$

Example: finding an element on a list.

Search, Optimization and Counting iii

Meta-algorithms to a search problem, optimization and counting: **brute force** search, go through all the (search space) until finding (a solution/the best solution/counting all solutions)

Search problem:

```
def is_solution (possible_solution):
    # returns true if possible_solution is the solution of the
    problem

def search ():
    for i in solution_space:
        if is_solution (i) == True:
            return i
```

Search, Optimization and Counting iv

Optimization problem

```
def evaluate (possible_solution):
    #returns a score for possible_solution

def search ():
    for i in solution_space:
        best = o
        if ievaluate (i) > best
            best = i
    return best
```

Counting

```
def is_solution (possible_solution):
    # returns true if possible_solution is the solution of the
    problem

def count ():
    count = 0
    for i in solution_space:
        if is_solution (i):
            count = count + 1
    return count
```

Searching spaces i

So the first thing is to identify the search space and find a way of crawling it

Linear search: O(N)

Example: find a item on a collection of elements.

Quadratic search: O (N^2)

Example: find the pair of numbers that yield a certain sum, find a pair of points that have a certain distance.

Searching spaces ii

Solution:

```
for i in collection:
    for j in collection:
        verify_solution (i, j)
```

***Cubic* search: O (N^3)**

Example: Triangle finding.

Solution:

```
for i in collection:
    for j in collection:
        for h in collection:
            verify_solution (i, j,h)
```

Searching spaces iii

You can have as much higher-order as you want: N^4, N^5...

There are examples for every order... but cicles over cicles is not very interesting.

How can we build search spaces in an elegant way? Python **itertools**!

import itertools as it

#Similar to a nested cicles:
it.product (Collection, repeat = N)

Searching spaces iv

How to use this to *brute force search* (for *N* dimensions)?

```
for i in itertools.product (collection, repeat = N):
    verify_solution (i)
```

Example: find pairs of numbers that yield a certain sum

```
for i in itertools.product (collection, repeat = 2):
    if i [0] + i [1] == sum:
        return i
```

itertools.product is also a lazy constructor!

Other search spaces i

Combinations:

itertools.combinations (collection, length)

Brute force about all combinations:

```
for i in itertools.combinations (collection, 2):
    verify_solution (i)
```

Other search spaces ii

Permutations:

itertools.permutations (collection)

for i itertools.permutations (collection):
    verify_solution (i)

Some small exercises.

Exercise 1. Find a line with a certain norm from a set of points.

Exercise 2. Find all subsets of a set.

Exercise 3 Break a password of n ascii characters.

Exercise 4. Given a set of stocks with some risk and some advantage associated, determine what is the best set of n stocks with the greater advantage and risk not greater than x.

Exercise 5. Given a set of cities, verify what is the transversal that passes through all the cities with the lowest cost (Traveling salesman problem)

Sorting

Sorting i

Some times the search space possesses a structure. We know that the solution is not in some parts of the solution space.

One of those possible structures is if the search space is ordered. Example: find the maximum sum of three elements in a list, for instance

l = [2, 7, 5, 8, 9 , 3, 4, 1]

Brute force search: N^3

If it is ordered: (?)

l = [1, 2, 3,4,5, 7, 8, 9]

Sorting ii

The highest sum corresponds to the sum of the last three elements (24):

l = [1, 2, 3,4,5, **7, 8, 9**]

This is equivalent to the problem of finding the minimum (or largest) k elements.

We know that we can find the minimum in N, and we can find the k minimums in k.N time.

When should be use sorting?

What is the cost of sorting?

There are many algorithms for sorting:

- Bubble sort (Cost: N^2);
- Quicksort, Mergesort (Cost: $NLogN$)
- Many others... with slight advantages.

So, assuming quicksort, when it is advantageous to use it instead of linear search for $k - minimuns$?

Sorting using python: **l.sort()** where l is a list. Additionally **l.sort (reverse= True|False, key=myfunct)**

Search in ordered spaces i

Search in ordered spaces is generally easier than in non-ordered spaces: we can find an element in $logN$. Example:

Find 2 in $\{1, 2, 3, 4, 5\}$:
Pick the middle element: 3.
Element is lower than 3, then restart the algorithm with the left part of three

Find 2 in $\{1, 2\}$
Middle element: 1
...

Find 2 in $\{1\}$
Element found!

Search in ordered spaces ii

This kind of implementation can be done using binary search!

Search in ordered spaces iii

```
def binary_search_recursive(lst, target):
    if len(lst) == 0:
        return -1

    mid = len(lst) // 2

    # Check if the element is present at the middle
    if lst[mid] == target:
        return mid
```

Search in ordered spaces iv

```
# If the element is smaller than mid, then it's in the left subarray
elif lst[mid] > target:
    return binary_search_recursive(lst[:mid], target)

# Otherwise, the element is in the right subarray
else:
    return binary_search_recursive(lst[mid + 1:], target)
```

Exercise. Make use of the above algorithm to do binary search in a list of numbers.

Exercise. Using the fact that a set of points can be sorted by one their coordinates, provide an algorithm to find a line segment in a set of points.

Questions?