Programming Challenges (GB21802)

Week 3 - Problem Solving Paradigms: Search

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: April 24, 2021)

Version 2021.1

Last Week's Review

- Linear Structures (Arrays, Vectors)
  - Simple, but effective, often used in programming challenges;
  - Learn array functions in the library: sorting, binary search, etc;
- Tree Structures (Map, Set)
  - Are fast for querying data;
  - Hard to implement by hand, but available on the library too;
- Union-Find Disjoint Set (UFDS)
  - Efficiently assign data items into different groups;
  - Implementation and algorithm is very simple;
- Segment Trees
  - Good for Max/Min Range Query in dynamic data
  - Hard to implement!

Outline for this week

This week, we study "Search-based" approaches to solving programming challenges:

- Complete Search (Brute Force)
- Divide and Conquer
- Greedy Search
- Dynamic Programming (next week!)

What is Search?

In daily life, we **search** for something when we are trying to find where this thing is located.

- Keys of your bycicle;
- Your wallet;
- Your cellphone;

The search has an **Objective** (what are we searching?), and the **search space** (where can we search?).

The thing you search is always in the last place you look
(*By definition!*)

What is a “Search Problem”?

A **search problem** is a problem that we can describe as: "check many **possible answers** until you find a **solution**".

- One answer to a search problem can be **correct** or **incorrect**;
- In some search problems, an answer can have a **quality score**;
- The **set of answers** to a search problem can be **sorted** by some criteria;

Many problems in programming challenges (and also in real life) can be described as search problems. We can apply very simple **search algorithms** to solve these problems.

Search Problem Examples (from past weeks)

We can **calculate** the answer, which is usually very fast,
or we can **search** the answer, which is usually slow.

Traffic Lights (Week 1)

Objective: FIND the time (in seconds) when all the different lights are synchronized at green.

Search Space: Time s from 1 to 18000.

File Fragmentation (Week 2)

Objective: Given a set of binary fragments, FIND the binary string that matches every pair.

Search Space: All binary strings from $x = 0$ to $x = 2^{n-1}$.

The Generic Search Algorithm

Any problem that can be described as a **Search Problem** can also be solved using a **Search Algorithm**.

1. Sort and order the search space; $n = 1$
2. Generate the n^{th} answer: a_n.
3. Test if a_n is a solution to the problem;
4. $n = n + 1$, go back to 2.

Some questions when you program the search algorithm:

- How to generate one answer? How many answers we store in memory?
- How many answers exist in total? How to order the search space?
- Is it possible to skip some answers? (pruning)
- How many solutions are necessary?

Search Problem Example

File Fragmentation

Let's use this algorithm to last week's "File Fragmentation".

We find the correct size of the answer, and we test each binary string of that size.
To do that, we check if it fits all the fragments.

Input – A Set of binary fragments

```
0011
100
1100
00
00
001
```

Output – one bin string that fit all

```
000000  ->  0011 + 00    X
000001  ->  0011 + 00    X
  ...
001100  ->  0011 + 00    OK
            00 + 1100    OK
            001 + 100    OK!
```

How long does would this algorithm take?

Search Problem Example

File Fragmentation – Considerations

Using this **Full Search** algorithm, we can find a solution to the File Fragmentation problem. However, how long does this algorithm take?

- What is the search space?
  The search space is every binary string B of max size n. ($O(2^n)$)
- How to check one answer?
  We test all combinations of 2 out of m fragments. $O(C_{m,2})$
- What is the expected running time?
  We need to check every solution: $2^n \times C_{m,2} = O(2^n C_{m,2})$

We calculate the running time of the algorithm using n and m

Search Problem Example

Let's practice some simple search problems!

Consider the same input for all problems

You have unsorted an array A with n integers ($n < 10.000$).
Each a_i is in $\{0 \leq a_i \leq 100.000\}$.

For each problem below, what is the expected running time of a full search?

1. Find the **largest** and the **smallest** element of A;
2. Find the k^{th} **smallest** element of A;
3. Find the **largest gap** G; where $x, y \in A$ and $G = |x - y|$;
4. Find the **longest increasing subsequence (LIS)** of A;

Example of LIS: $A = [3, 2, 5, \overline{1}, 4, \overline{2, 3, 5}, 7, \overline{6, 10}]$

Search Problem Example

Let's practice!

What is the expected running time of each problem?

1. **Find the Largest and smallest element of A:**
   - $O(n)$: a full pass over all numbers.
2. **Find the k^{th} smallest elements of A:**
   - Repeat the full pass k times, remove smallest each time. $O(nk)$, or $O(n^2)$ in worst case;
   - Sort A and list first k elements: $O(n \log n)$
3. **Fing the largest gap:**
   - Try all possible pairs: Two loops: $O(n^2)$
   - Find the smallest and largest numbers $O(n)$ (proof: largest gap = biggest - smallest)
4. **Longest increasing subsequence:**
   - Test all possible subsequences: $O(2^n)$
   - Dynamic programming: $O(n^2)$
   - Greedy search: $O(n \log k)$ – Look for this algorithm!

Part II: Search Algorithms

Search Algorithm Paradigms

There are many variations on Search Algorithm:

- Complete Search/Brute Force;
- Divide and Conquer;
- Greedy Approach;
- Dynamic Programming (Next week!)
- Heuristic Search (not in this course!)
- Meta-heuristic Search (not in undergraduate school!)
- ... and many others.

Depending on the problem, we can use different approaches. Which approach is better depends on the problem, so let's learn some of them.

Complete Search

Definition

The search algorithm we used until now is the **Complete Search.**
A Complete Search algorithm checks all the solutions in the search space of the problem.

Another name for Complete Search algorithms is **Brute Force.** Brute force sounds negative, but Complete Search algorithms have some uses, so we will use a "nicer" name.

Complete Search

Structure

The structure of a Complete Search algorithm is very simple:

- **Define or List all possible answers**
  Choose the right data structure that contains all answers.
- **Check each answer if it is a solution**
  Normally using a loop or recursive function;
- **Prune, Prune, Prune**
  They key to a fast Complete Search algorithms is to reduce the number of answers. In the program, we can do this using "break" in loops, or using early "return" clauses in recursive functions.

Complete Search

Example: UVA 725 – Division

Problem Summary

You receive an integer N. You have to find all pairs of numbers with 5 digits (*abcde* and *fghij*) that satisfy two conditions:

1. $fghij / abcde = N$
2. $a, b, c, d, e, f, g, h, i, j$ are all different.

Example: $N = 62$

79546 / 01283 = 62
94736 / 01528 = 62

QUIZ!

Think: How can you solve this problem using **Complete Search**?

Complete Search Example: UVA 725 Division

Naive Solution

Test all x where $0 \leq x \leq 99999$, calculate $y = x * n$, and test if all digits of x and y are different.

```
for (int x = 0; x <= 99999; x++) {
  y = x*n;
  digits = count_digits(x,y);
  if (digits == 1<<10 - 1) printf("%0.5d/%0.5d=%d\n",y,x,N);
}

int count_digits(int x, int y) {
  int used = (x < 10000); % bit array: each bit mark one digit
  int tmp;
  tmp = x; while (tmp) {used |= 1 << (tmp%10); tmp /= 10; }
  tmp = y; while (tmp) {used |= 1 << (tmp%10); tmp /= 10; }
  return used;
}
```

Complete Search Example: UVA 725 Division

Prunning the Loop

The algorithm in the previous slide is **very slow.**
The loop tests many numbers that will **never** be the right answer.

How can we prune (reduce) the number of answers that we test?

- What is the absolute minimum and maximum values of x?
  01234 to 98765 (different digits)
- Maximum for y is also 98765, so the real maximum for x is less:
  $x_{\text{max}} = 98765/N$, (remember: $x * y = N$)
- Can you think of other ways to prune?

Notes about Complete Search algorithms

- A Complete Search should always give the correct solution (if bug free)
    - It checks all answers, so it should always find the correct one;
    - Checking all answers usually takes too long;
- For some problems, the Complete Search is the right algorithm.
    - The problem is so small, so a Complete Search is the simplest solution.
    - Prune, prune, prune!
- For a very hard problem, a Complete Search can give you a starting point;
    - Maybe the Complete Search will be "time limited exceeded";
    - But you can use Complete Search to generate "test cases";

Complete Search Example 2: Simple Equations

Problem Summary – UVA 11565

Input: A, B, C, $1 \leq A, B, C \leq 10000$.

Find x, y, z so that:

- $x + y + z = A$,
- $x * y * z = B$,
- $x^2 + y^2 + z^2 = C$,

To solve this problem we can loop and test on of x, y, z (3-nested loop).

But what should be the minimum and maximum value of the loops?

Complete Search Example 2: Simple Equations

Initial Pruning

Consider $x^2 + y^2 + z^2 = C$.

Since $C \leq 10000$, and $x^2, y^2, z^2 \geq 0$, if $y = z = 0$ then the range for x must be $-100, 100$.

```
int x,y,z;
for (x = -100; x <= 100 && !sol; x++)
  for (y = -100; y <= 100 && !sol; y++)
    for (z = -100; z <= 100 && !sol; z++)
      if (y != x && z != x && z != y &&
          x + y + z == A && x * y * z == B &&
          x*x + y*y + z*z == C) {
             printf("%d %d %d\n", x,y,z);
             exit(0);
          }
```

QUIZ: How can we prune this loop even more?

Complete Search Example 2: Simple Equations

More Pruning

There are many other ways that we can prune the loop:

- We can change the range using the actual input values of A, B, C
- We only need one solution. We can break the loop once we find it.
- We can use equation 2 to think of other ways to prune.

Complete Search Tips

Main question about Complete Search: Is the program fast enough?

To make your program faster, find the **critical part** of the code, and improve that first!

Tip 1 – Filtering Vs Generating

Filter Programs examine all answers and remove incorrect ones. Generally iteractive. Generally easier to code. Example: Request for proposal.

Generating Programs gradually build answers and prune invalid partial answers. Generally recursive. Generally faster. Example: 8 queens.

Complete Search Tips

Tip 2 – Prune Early

In the N queen problem, if we imagine a recursive solution that places 1 queen per column, we can prune rows, columns and DIAGONALS.

Also remember to mark impossible places when you enter the recursion, and unmark when you leave, using bitmasks.

Tip 3 – Pre-computation

Sometimes it is possible to generate tables of partial solutions.

Load this data in your code to accelerate computation (at the expense of memory). The programming cost is high, since you have to output the tables in a way to facilitate putting it in the code.

Complete Search Tips

Tip 4 – Solve the problem backwards

Sometimes a less obvious angle of attack may be easier.

Example: "Rat Attack". A city with 1024 x 1024 blocks has $n \leq 20000$ rats in some of its blocks. You have a bomb with radius $d \leq 50$. Where do you place the bomb to kill most rats?

Obvious Approach: Check the radius of the bomb (50^2) at each of the 1024^2 blocks.
Cost: $1024^2 \times 50^2 = 2.62 \times 10^9$

Backwards Approach: Make a 1024x1024 matrix of “killed rats”. For each rat, add it to each cell in the bomb radius.
Cost: Number of rats (n) and bomb radius (d)
$n * d^2 = 20000 * 2500 = 5 \times 10^7 + 1.048 \times 10^6$.

Part III – Divide and Conquer, and Greedy Algorithms

Divide and Conquer

Divide and Conquer (D&C) is a problem-solving paradigm in which a problem is made simpler by ’dividing’ it into smaller parts.

- Divide the original problem into sub-problems;
- Find (sub)-solutions for each sub-problems;
- Combine sub-solutions to get a complete solution;

Examples

Quick Sort, Binary Search, etc...

Canonical Divide and Conquer

1. Sort an static array;
2. You want to find item *n*.
3. Test the middle of the array.
4. If *n* is smaller/bigger than the middle, throw away the second/first half.
5. Repeat

Search time: O(log n) plus sorting time if necessary.

Binary Search on Simulation Problems

Simulation problems usually require us to find a value that solves a complex simulation.

Problem Example: Paying the debt

You have to pay V dollars. You pay D dollars per month, in M months. Each month, before paying, your debt increases by i.

If we fix M,I and V, what is the minimal D?

$V = 1000$, $M = 2$, $i = 1.1$, what is the minimum D?

- $D = 500$:
  - $m_1 : V_0 * 1.1 - D = 600, m_2 : v_1 * 1.1 - D = 160$
- $D = 600$:
  - $m_1 : V_0 * 1.1 - D = 500, m_2 : v_1 * 1.1 - D = -50$

How to approach the Simulation Problem?

Possible Approaches:

- Complete Search (what about continuous search space?)
- Find the derivative of the simulation and solve it to zero;
- Start from the end state of the simulation and calculate back;
- These approaches can be hard for complex simulations!

Binary Search Approach:

1. Estimate a minimum and maximum possible answer (a, b)
2. Choose the middle value as an answer and simulate it;
3. Adjust the limits (a, b) based on simulation result;
4. Go back to 2.

Binary Search for Simulation

Input: $m = 2, v = 1000, i = 0.1$

- Choose initial range: (ex: [0.01 to 1100.00]); Initial d: 550.005
- $f(d, m, v, i) = \text{loop}(v * (1 + i) - d)$, m times
- Do binary search in this range;

a	b	d	simulation: f(d,m,v,i)	action:
0.01	1100.00	550.005	error: -54.98	increase a
550.005	1100.00	825.002	error: 522.50	decrease b
550.005	825.002	687.503	error: 233.75	decrease b
550.005	687.503	681.754	error: 89.38	decrease b
550.005	618.754	584.379	error: 17.19	decrease b
550.005	584.379	567.192	error: -18.89	increase a
567.192	584.379	575.786	error: -0.84	increase a
...	...	...	a few iterations later ...	...
...	...	576.190	error $< \epsilon$	stop: answer = 576.19

Total number of steps: $O(log_2((b - a)/\epsilon))$

The Greedy Search Algorithm

Definition

A **Greedy Search Algorithm** will make the locally optimal choice at each step of the program, with the hope that eventually it will reach the globally optimal solution.

Greedy can be very fast, or very wrong. For a greedy algorithm to work, a problem must show two properties:

- It has optimal sub-structures. In other words: an answer to the search problem must have **local steps that can be evaluated**;
- It has the **greedy property**: If you always choose the local step with the highest evaluation, you will "eventually" reach the optimal solution.

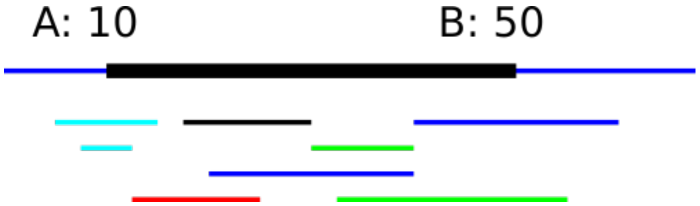
The first property is similar (but not identical) to dynamic programming. The second property is harder to prove.

Greedy Search Example: Minimal Coverage

Consider an interval [A,B], and a set of n intervals $S = [(a_1, b_1), (a_2, b_2), \ldots (a_n, b_n)]$.

Find the minimal subset of S which completely covers [A,B].

- A = 10, B = 50;
- S = [(5,15), (8,12), (40,60), (30,40), (20,40), (13,25), (33,55), (18,30)]



Greedy Search Example: Minimal Coverage

Search Algorithm

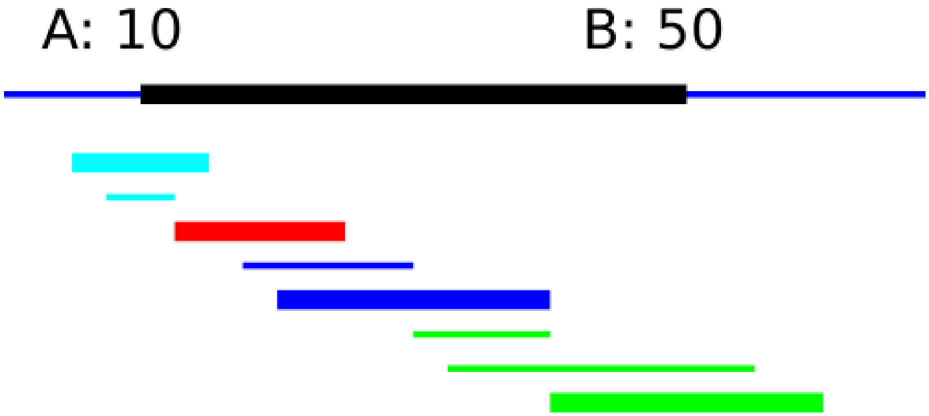
The algorithm that solves this problem progressively adds items to cover the area from left to right. It starts with a solution set $C = \emptyset$, and cover point $A_c = A$.

1. Find and remove subset $S_c \subset S$, of all intervals that cover A_c;
   (if S is sorted by left value, this is fast)
2. Choose the interval $s_i \in S_c$ with the **maximum** left value.
3. Update A_c with the left value of s_i
4. Go back to 1.

It is common to see Greed Algorithms in problems of the type "Find the best Subset";

Greedy Search Example: Minimal Coverage

Solution



Greedy Search Example: Coin Change

Given a target value V and a set of coin sizes S, select a group C of coins (with repetition) that adds to V, so that the $|C|$ is minimum.

Example Input: $V = 42, S = \{25, 10, 5, 1\}$

Example Output: $25 + 10 + 5 + 1 + 1$: Size 5

What is the Greedy algorithm for this problem?

Greedy Search Example: Coin Change

Wrong Answer!

A Greedy algorithm for Coin change would be to always take the largest coin, and reduce V by the value of that coin:

- $V = 42, S = \{25, 10, 5, 1\}, C = \emptyset$, Take 25
- $V = 17, S = \{25, 10, 5, 1\}, C = \{25\}$, Take 10
- $V = 7, S = \{25, 10, 5, 1\}, C = \{25, 10\}$, Take 5
- $V = 2, S = \{25, 10, 5, 1\}, C = \{25, 10, 5\}$, Take 1
- $V = 1, S = \{25, 10, 5, 1\}, C = \{25, 10, 5, 1\}$, Take 1

Wrong Answer!

- $V = 6, S = \{4, 3, 1\}, C = \emptyset$, take 4
- $V = 2, S = \{4, 3, 1\}, C = \{4\}$, take 1
- $V = 1, S = \{4, 3, 1\}, C = \{4, 1\}$, take 1

Greedy Example 2 – Load Balancing UVA 410

Problem Description

- There are C chambers, and $S < 2C$ items.
- Each item has a positive weight M_i.
- You need to assign each item to a chamber in order to minimize “imbalance”

$$A = \sum_{i=1}^{S} M_i / S$$

$$\text{Imbalance} = \sum_{i=1}^{C} |C_i - A|$$

Can you figure out a greedy search solution?

Greedy Example 2 – Load Balancing UVA 410

Problem Description

You have C chambers, and S < 2C specimens with different positive weights. You need to decide where each specimen should go to minimize “imbalance”.

Insights:

- A chamber with 1 individual is always better than a chamber with 0 individuals.
- Order of chambers does not matter.

Greedy Example 2 – Load Balancing UVA 410

Problem Description

You have C chambers, and S < 2C specimens with different positive weights. You need to decide where each specimen should go to minimize "imbalance".

Greedy algorithm: Order the individuals by weight, and put one in each chambers until the chambers are full, then add one in each chamber backwards.

A similar approach can be used to solve this week's problem "Dragon of LooWater".

Part iv – Outro

Summary

- Search Algorithms will check every possible answer in a problem, until they find the solution;
- The set of "every possible answer" (**the search space**) depends on the data structure, algorithm, and smart pruning;
- The time performance of Search Algorithms depends on the size of the search space;
- **Complete Search** takes a lot of time, but it will always find the correct answer (if the search space contains it);
- **Binary Search** and **Greedy Search** are much faster, because they discard a large part of the search space. They require special conditions for the problem;

Search Algorithms in CS Research

Search algortihms (including Greedy and Binary search) sound simple, but they have a very important place in CS research.

The key idea of search algorithms is central to the definition of NP-completeness: a solution to an NP-complete problem can be **checked** in polynomial time. This implies that the approach to solve many NP-Complete problems is to define the search space, and systematically check the answers in this space: Just what we're doing!

This also to other problems where we do not have complete information, and/or we do not know efficient algorithms.

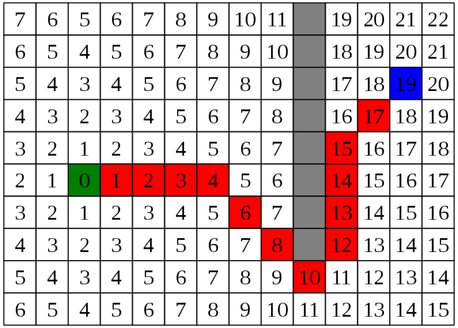
There are, of course, more complex approaches to search algorithms:

- Heuristic Search;
- Meta-heuristic Search;

Heuristic Search

Heuristic Search is a search algorithm guided by a **Heuristic function**, which is a function that **estimates** the distance of an answer to the optimal solution.

One famous example of a heuristic algorithm is A* search, which is often used in path-finding and game AI.



7	6	5	6	7	8	9	10	11	▨	19	20	21	22
6	5	4	5	6	7	8	9	10	▨	18	19	20	21
5	4	3	4	5	6	7	8	9	▨	17	18	19	20
4	3	2	3	4	5	6	7	8	▨	16	17	18	19
3	2	1	2	3	4	5	6	7	▨	15	16	17	18
2	1	0	1	2	3	4	5	6	▨	14	15	16	17
3	2	1	2	3	4	5	6	7	▨	13	14	15	16
4	3	2	3	4	5	6	7	8	▨	12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Meta-heuristic Search

Meta-heuristic search are a more general form of heuristic search. While a Heuristic search uses a function that guides the search for **one specific problem**, Meta-heuristic search guides the search towards an **entire category of search spaces**.

Meta-heuristic search are used in real world industrial optimization problems, and are an active area of research.

Some examples of meta-heuristics:

- Evolutionary Algorithms / Genetic Algorithms;
- Hill Climbing;
- Swarm Algorithms;
- Etc...

Problem Discussion

Quick Summary of the homework for this week

1- Dragon of Loowater

You need to select the knights that can defeat a dragon with many heads;

- You need one knight per head;
- A knight can only defeat the head if he is bigger than the head;
- A knight charges a reward equals to his size;

Input

- List of Dragon heads and their sizes
- List of Knights and their sizes

Output

- Find the minimum cost necessary to defeat the dragon;
- Or write if it is impossible to defeat the dragon;

Hint: Data pre-processing is important here.

2- Stern-Brocot Number

This problem describes a tree-structure that can generate all fractions of rational numbers. For any given fraction, you must find the path in this tree that leads to that fraction.

Input

- Two numbers that make a fraction. e.g.: 5, 7;

Output

- The path to that fraction in the tree;

Hint: Try to do it by hand a few times!

3- Bars

This is just the famous **knapsack problem.**

Input

Size of the knapsack, and size of the items.

Output

"YES" if you can solve the knapsack, "NO" if you can't.

Hint: The knapsack problem is a permutation problem, so pruning is important! (every item you prune, the search space is cut in half)

4- Rat Attack

- You have an $n \times n$ matrix (max 1024) with some rats in each cell;
- You have a rat trap (bomb) that kills all rats in a matrix $2d + 1 \times 2d + 1$
- Where can you put the trap to kill the largest number of rats?

Hint: Search all positions for the trap, and counting all the rats at that position will take too much time. But you can change the data structure to reduce this time.

5- Simple Equations

Given the numbers A, B, C, you need to find x, y, z that complete the following equations:

- $x + y + z = A$
- $xyz = B$
- $x^2 + y^2 + z^2 = C$

Hint: You need to search all values of x, y, z (triple loop). Pruning is important;
Hint: What are the maximum and minimum values of x, y, z? Equation "C" is a good place to start.

6- Through the Desert

- Simulate a car going through the desert;
- Find the least amount of starting fuel needed to win.

- You could try to calculate the amount of fuel needed in each section of the trip (between gas stations);
- Or you could just simulate the trip, and make a binary search based on the amount of fuel left;

7- Zones

A cellphone company has a plan for N towers, but will only build M of them. ($M \leq N$). You know how many people are served by each tower, and you have to choose the towers that **maximize** the number of people.

Hint: This is a "select the maximum" subset problem. Can it be solved by greedy seach?
Hint: One big problem is the overlap between the towers. Be careful!

8- Little Bishops

- Like 8 queens, but with bishops!
- The number of bishops and the size of the board can be very big. Be careful of TLE!

About these Slides

Image Credits I

[Page 44] A* pathing image by dbenzhuser, CC-BY-SA 2.5