Complete search and greedy solutions

Atli FF

Árangursrík forritun og lausn verkefna

School of Computer Science

Reykjavík University

Complete search / Brute force

Solution space

- The set of all possible solutions to a problem is called the *solution space*
- Note that this solution space will then contain all the wrong solutions too
- Iterating over the entire solution space is called a complete search or brute force solution

Iterating over solution spaces

- Iterating over different solution spaces is key to being able to brute force problems.
- For the simplest cases, like ones you may have seen already in problems, we can just nest for/while loops.

Iterating over a variable number of loops

- A common somewhat harder thing is iterating over a sequence of integers, with variable bounds
- Say we want to test every vector of n integers where each value can be from 0 to $m-1$.
- We could do this usually with n for loops, but if n is an input this does not work
- In python we can use itertools, C++ needs something handmade

All vectors

```
int n, m; cin >> n >> m;
vector<int> counter(n, 0);
while(true) {
  process_solution(counter); // whatever the problem needs
  bool done = true;
  for(int i = 0; i < n; ++i) {
    counter[i]++;
    if(counter[i] == m) counter[i] = 0;
    else {
      done = false;
      break;
    }
  }
  if(done) { break; }
}
```

Iterating over subsets

- For subsets we could do the same where we have a `vector<bool>` and index i denotes whether we include the i-th element or not
- But for this specific case we can do something much faster
- `ints` consist of bits, so we can instead just have a number and let the i-th bit denote whether we include the i-th element or not
- This means the subsets will just be the numbers from 0 to $2^n - 1$ for n elements, which can be iterated over with a simple for loop

Bit masks

- These things are usually called bit masks
- If $A = \{1, 2, 3, 4, 5, 6\}$ and $B = \{1, 3, 5, 6\}$ then the corresponding numbers in binary would be 111111 and 110101 since the first bit is the one we write last. In decimal these are 63 and 53.
- This also gives us a whole load of useful operations that work on masks

Bit operations

<table>
<tr><th>Code</th><th>Meaning</th></tr>
<tr><td><code>0</code></td><td>The empty set</td></tr>
<tr><td><code>(1 << n) - 1</code></td><td>The set of the first n elements</td></tr>
<tr><td><code>1 << k</code></td><td>The set containing only the k-th element</td></tr>
<tr><td><code>A | B</code></td><td>The union of A and B</td></tr>
<tr><td><code>A & B</code></td><td>The intersection of A and B</td></tr>
<tr><td><code>A ^ B</code></td><td>The symmetric difference of A and B</td></tr>
<tr><td><code>~ A</code></td><td>The complement of A</td></tr>
</table>

Iterating over permutations

- Say we want to iterate over all permutations of some vector/list.
- Luckily this is built into a lot of languages:
  - `next_permutation(v.begin(), v.end())` in C++
  - `itertools.permutations` in Python

```
int n = 5; vector<int> perm(n);
for(int i = 0; i < n; ++i) perm[i] = i + 1;
do {
        for(int i = 0; i < n; ++i) cout << perm[i] << ' ';
        cout << '\n';
} while(next_permutation(perm.begin(), perm.end()));
```

Backtracking

- The methods to iterate over permutations and subsets were rather specialized
- Backtracking is a general framework to iterate over complex spaces
- Solves many classic problems like n-queens and sudoku

Backtracking

- Define some initial "empty" state and have some notion of partial or complete states
- For example in sudoku this is an empty grid, a partially filled grid and a fully numbered grid
- Then define transitions to further states
- In sudoku this would be filling in a number such that it doesn't create an immediate contradiction

Backtracking

- Now start with your empty state
- Use recursion to traverse all states by using the transitions
- If the current state is invalid, stop exploring this branch
- Process all complete states

Backtracking - pseudo code

```
state S;

void generate() {
        if(!is_valid(S)) return;

        if(is_complete(S)) print(S);

        for(each state P that S can transition to) {
                apply transition to P;
                generate();
                undo transition to P;
        }
}

S = empty state;
generate();
```

Backtracking - Subsets

- We can even replicate earlier functionality this way

```
const int n = 5; bool pick[n];

void generate(int index) {
  if(index == n) {
    for(int i = 0; i < n; ++i)
      if(pick[i]) cout << i << ' ';
    cout << '\n';
  } else {
    pick[index] = true;
    generate(index + 1); // pick element at index
    pick[index] = false;
    generate(index + 1); // don't pick element at index
  }
}
generate(0);
```

Backtracking - Permutations

```
const int n = 5; int perm[n]; bool used[n];

void generate(int index) {
  if(index == n) {
    for(int i = 0; i < n; ++i)
      cout << perm[i] + 1 << ' ';
    cout << '\n';
  } else {
    // decide what the element at index should be
    for(int i = 0; i < n; ++i) {
      if(!used[i]) {
        used[i] = true;
        perm[index] = i;
        generate(at + 1);
        used[i] = false; // remember to undo move!
      } } } }
memset(used, 0, n); generate(0);
```

Backtracking - N queens

- Another classic backtracking problem is n-queens
- We have a $n \times n$ chessboard and want to place n queens on it so no two of them can attack each other
- We could use bit tricks to iterate over all subsets of n pieces in the board, but that would be too slow
- Backtracking is much faster since we prune branches of computation early, it's almost universally good to do extra work earlier to prune branches when backtracking

Backtracking - N queens

- We go through the cells in order
- Our transition is placing a queen, or not placing a queen
- We don't place a queen if it would be able to attack another placed queen

```
const int n = 8;
bool has_queen[n][n];
int threatened[n][n];
int queens_left = n;

// generate function

memset(has_queen, 0, sizeof(has_queen));
memset(threatened, 0, sizeof(threatened));
generate(0, 0);
```

Backtracking - N queens

```
void generate(int x, int y) {
  if(y == n) generate(x + 1, 0); // move onto next column
  else if(x== n) { // we are at the end
    if(queens_left == 0) // this is a valid solution
      print(); // exact implementation not important
   } else {
     if(queens_left > 0 && threatened[x][y] == 0) {
       has_queen[x][y] = true;
       for(auto p : queen_threaten(x, y)) // good exercise to implement this!
         threatened[p.first][p.second]++;
       queens_left--;
       generate(x, y + 1);
       has_queen[x][y] = false;  // now to undo the move
       for(auto p : queen_threaten(x, y))
         threatened[p.first][p.second]--;
       queens_left++;
     }
     generate(x, y + 1); // also have to try leaving it empty
   }
}
```

Greedy algorithms

Greedy algorithms

- An algorithm that always makes *locally* optimal moves is called greedy
- For some kinds of problems this will give a *globally* optimal solution as well
- Seeing when this is the case can be very tricky, and if used in the wrong context the solution will get a WA verdict

Submitting greedy solutions

- The tricky thing with these solutions are that it's often hard to know if you've made a mistake and thus get WA or if there's some hole in the greedy algorithm
- It's often easy to think of all kinds of greedy solutions, but they are very often wrong
- Generally one would like to consider complete search or dynamic programming first, but of course some problems do require greedy solutions

Coin change

- A classical example is making change. Say you want to sum up n and have only denominations of $1, 5$ and 10, what's the least amount of coins you can give back?
- The greedy solution would be to just always give the biggest coin you can that's not too much. So for say 24 we'd do $10, 10, 1, 1, 1, 1$.
- Is this always optimal?

Coin change

- Well, it turns out to depend on the denominations. Say we have denominations of $1, 8$ and 20.
- For $n = 24$ we then give back $20, 1, 1, 1, 1$ instead of the optimal $8, 8, 8$.
- We will come back to this problem tomorrow when we solve the general case using dynamic programming.

Taxi assignment

- Let's consider another problem. You are managing a taxi company and today n drivers showed up and you have m cars.
- But not all drivers and cars are created equal. Car i has h_i horsepower and driver j can only handle at most g_j horsepower.
- What's the greatest number of drivers you can pair to cars such that they can handle their car?

The greedy step

- The greedy idea here is to simply pair each car to the worst driver that can still handle that car.
- Thus we start by sorting the drivers and cars and then simply linearly walk through each and pair them together.
- It might not be obvious, but this actually gives the best answer.

Implementation

```cpp
int main() {
    int n, m; cin >> n >> m;
    vi a(n), b(m);
    for(int i = 0; i < n; ++i) cin >> a[i];
    for(int i = 0; i < m; ++i) cin >> b[i];
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    int ans = 0;
    for(int i = 0, j = 0; i < m; ++i) {
        while(j < n && a[j] < b[i]) j++;
        if(j < n) ans++, j++;
    }
    cout << ans << '\n';
}
```

Sorting

- Greedy algorithms very often involve sorting
- More generally they often involve always picking the "extremal" option out of the local options, in some sense
- Biggest, shortest, cheapest, first, etc.

Job scheduling

- Say we have a list of jobs, each starting at some time s_j and finishing at some time f_j
- What's the largest amount of jobs we can complete if they can't overlap?

Solution

- The solution is shockingly simple, but not obviously correct
- Order the jobs by completion time f_j and then walk through them
- If you can complete a job in addition to the ones you've already picked, pick it
- The jobs you've picked by the end are the solution

Proof of correctness

- Why is this correct though? Let's prove it.
- Suppose the algorithm is not optimal. Say we pick jobs of indices $i_1, i_2, \ldots, i_k$ but a better solution picks $j_1, j_2, \ldots, j_l$.
- Say the solutions agree on the first r jobs (possibly 0).
- Now neither i_{r+1} nor j_{r+1} clash with the jobs $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$. But because we ordered things by end time, we must have that job i_{r+1} ends no later than j_{r+1}. But then we could just as well have picked i_{r+1}. But this holds for any r, so by induction we have that $i_1, \ldots, i_k$ is no worse than $j_1, \ldots, j_l$, which gives a contradiction.
- Thus the algorithm is optimal.