Introduction

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

- Iterating over different solution spaces is key to being able to brute force problems.

Iterating over subsets

- Iterating over different solution spaces is key to being able to brute force problems.

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 repluicate 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
   }
}
```