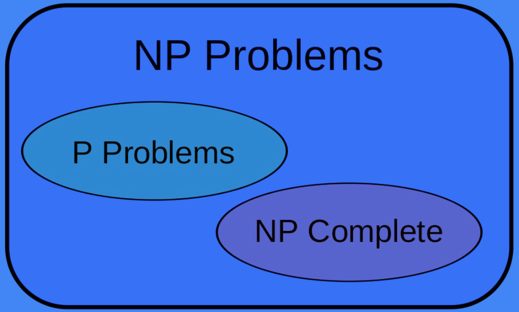
Intractable Problems and DP with Bitmask

Problem Solving Club
March 1, 2017



Agenda

- **Intractable problems**
  - Complexity classes P, NP, co-NP, #P, completeness
  - How to identify common intractable problems
- **Dynamic programming with bitmask**
  - How DP with bitmask helps solve intractable problems
  - Intractable problems that benefit from DP with bitmask
  - Examples of programming contest problems involving DP with bitmask

Intractable problems

- **Intractable** problems can be solved in theory (e.g., given large but finite time), but which in practice take too long for their solutions to be useful.
- This differs from **undecidable** problems, which cannot even be solved in theory (given any finite amount of time).
- A commonly cited undecidable problem is the **halting problem**:
    - Given the description of an arbitrary program and a finite input, decide whether the program finishes running or will run forever.
- Alan Turing famously proved the halting problem undecidable.

Intractable problems

- Which problems are intractable? Nobody really knows.
- **Cobham–Edmonds thesis:** Intractable problems are those that can be cannot be computed in polynomial time, i.e., in the complexity class P.
- This is the commonly used definition of intractability.

Complexity classes

Problems in computer science are divided into complexity classes.

- **P** Problems that can be solved in polynomial time (tractable).
  - Given a graph, what is the shortest path between two vertices?
- **NP** Problems where the solution can be verified in polynomial time.
  - Given a set of integers, is there any subset whose sum is zero?
- **co-NP** Complement of problems in NP.
  - Given a set of integers, is there **no** subset whose sum is zero?
- **#P** Counting problems associated with problems in NP.
  - Given a set of integers, **how many subsets** sum to zero?

Note: $P \subseteq NP$ and $P \subseteq \text{co-}NP$. It is not known if $P = NP$, $NP = \text{co-}NP$, $P = \text{co-}NP$

Complexity classes - completeness

- A problem is **complete** for a complexity class if it is among the "hardest" problems in the complexity class.
- NP-complete problems are the “hardest” problems in NP.
- If an NP-complete problem can be solved in polynomial time, all problems in NP can be solved in polynomial time.
- co-NP-complete and #P-complete are similarly defined.

Common intractable problems

NP-complete (solution can be verified in polynomial time)

- **Subset sum:** Given a set of integers, is there any subset whose sum is 0?
- **Hamiltonian path:** Given a graph, does a Hamiltonian path exist?
- **Travelling salesman:** Given a graph, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- **Satisfiability:** Given a boolean formula, is there any assignment of variables that will make it true? (Special case of 2-SAT is in P.)
- The complements of these problems are co-NP-complete.
- Counting versions of these problems are #P-complete.

Why do we care about intractable problems?

- The best known solutions for intractable problems generally run in exponential or subexponential time.
- For decades, people have tried to find polynomial time solutions to intractable problems, but have not succeeded.
- By recognizing intractable problems, we can avoid wasting time trying to find an efficient solution.
- Common techniques used to solve intractable problems are **complete search** and **DP with bitmask**.
- If exact solution is not needed, efficient approximation algorithms exist.

Dynamic programming with bitmask

- DP with bitmask is a technique usually used to solve intractable problems.
- It generally improves an $O(n!)$ solution to $O(2^n)$.
- While still intractable, the runtime is significantly better.
- Contest problems with $10 \leq n \leq 20$ can indicate DP with bitmask

n	2^n	n!
1	2	2
10	1,024	3,628,800
20	1,048,576	2,432,902,008,176,640,000

Travelling salesman problem

Given a graph, what is the shortest possible route that visits each city exactly once and returns to the origin city?

- What is an obvious greedy solution? Does it work?
- How would we solve this by complete search?
- What is the runtime?

Travelling salesman problem: Overlapping subproblems

Let's say we have 7 vertices. Consider these routes:

- $1 \to 2 \to 3 \to 4 \to \{5,6,7\} \to 1$
- $1 \to 3 \to 2 \to 4 \to \{5,6,7\} \to 1$

What can we say about the best order in which to visit 5,6,7 in these two cases?

Travelling salesman problem: Overlapping subproblems

Let's say we have 7 vertices. Consider these routes:

- $1 \to 2 \to 3 \to 4 \to \{5,6,7\} \to 1$
- $1 \to 3 \to 2 \to 4 \to \{5,6,7\} \to 1$

The best order to visit remaining vertices depends **only** on:

- The **set of vertices visited**
- The **current vertex**

Travelling salesman problem: Dynamic programming solution

Without loss of generality, assume that the cycle starts and ends at vertex 1.

If we have 7 vertices, we can use the following DP solution:

f(v_1, v_2, v_3, v_4, v_5, v_6, v_7, cur) = Assuming we've visited a certain set of vertices, and we are at "cur" vertex, the minimum distance to visit remaining vertices and return to vertex 1.

- v_i = 1 if vertex i has been visited, else 0
- cur = current vertex number

How big is the DP array?

Travelling salesman problem: Dynamic programming solution

DP function **f**$(v_1, v_2, v_3, v_4, v_5, v_6, v_7, cur)$

- Base case: **f**(1, 1, 1, 1, 1, 1, 1, cur) = dist[cur][1]
  - If we've visited all vertices, need to return to vertex 1
- General case: **f**$(v_1, v_2, v_3, v_4, v_5, v_6, v_7, cur)$ = $\min_{(j \text{ where } vj=0)}$ (dist[cur][j] + **f**(<set v_j=true>, j))
  - If we haven't visited all vertices, try all next vertices and choose the best one.
- The final answer is **f**(0, 0, 0, 0, 0, 0, 0, 1)
- What is the runtime of of this algorithm?

Where is the bitmask?

To implement **f**(v_1, v_2, v_3, v_4, v_5, v_6, v_7, cur), a **bitmask** is usually used to represent the set of visited vertices. Top-down DP is almost always used.

```
const int N = 20;
const int INF = 100000000;
int c[N][N]; // adjacency matrix
int mem[N][1<<N]; // DP memoize array
memset(mem, -1, sizeof(mem));
int tsp(int i, int S) {
    if (S == ((1 << N) - 1)) {
        return c[i][0];
    }
    if (mem[i][S] != -1) {
        return mem[i][S];
    }
    int res = INF;
    for (int j = 0; j < N; j++) {
        if (S & (1 << j))
            continue;
        res = min(res, c[i][j] +
            tsp(j, S | (1 << j)));
    }
    mem[i][S] = res;
    return res;
}

// tsp(0, 0) is the answer
```

Secret Santa (CCPC 2016)

- A **secret santa** is where n ($2 \leq n \leq 15$) people are each assigned another person to buy a gift for.
- There may also be some restrictions. For example, Jack (person 1) is not allowed to be assign Jane (person 2).
- Given n and a list of restrictions, how many ways can we assign people?
- Note: This is also known as the **permanent** of a matrix, and its calculation is #P-complete.
- How would we solve this by complete search? What is the runtime? What do you notice about the limits on n?

Secret Santa: Overlapping subproblems

Let's say we have 7 people. Consider these partial assignments:

- $(1\rightarrow3)(2\rightarrow5)$
- $(1\rightarrow5)(2\rightarrow3)$

What can we say about the number of ways to complete the remaining assignments in these two cases?

Secret Santa: Overlapping subproblems

Let's say we have 7 people. Consider these partial assignments:

- $(1 \to 3)(2 \to 5)$
- $(1 \to 5)(2 \to 3)$

The number of ways to complete the remaining assignments depends **only** on **the set of people who have already been assigned to** (here, 3 and 5).

Note: We have to assign people in a fixed order.

Secret Santa: DP solution

f(assigned) = # of ways to assign remaining people, where assigned is a **bitmask** of the people who have already been assigned to.

- Base case: **f**(everyone assigned) = 1
- General case: **f**(assigned) = $\text{sum}_{\text{(persons who have not been assigned to j)}}$(assign current person to j if it is allowed)
- The current person is an **implicit** DP parameter - it is the number of persons assigned (number of ones in the bitmask).
- What is the runtime of this algorithm?

Secret Santa: DP solution

```
def sv(bs):
      if bs == (1<<N)-1: return 1 # Base case
      if bs in dp: return dp[bs]
      ans = 0

      curPerson = 0 # Figure out current person by counting bits in bs
      for n in range(N):
            if bs & 1<<n:
                  curPerson += 1

      for n in range(N): # Try to assign curPerson to every possible other person
            if (not (bs & 1<<n)) and (not rst[curPerson][n]):
                  ans += sv(bs | 1<<n)
      dp[bs] = ans
      return ans

# answer is sv(0)
```

Summary

- **Intractable** problems can be solved in theory (e.g., given large but finite time), but which in practice take too long for their solutions to be useful.
- **DP with bitmask** is a problem solving technique for intractable problems, that usually improves an $O(n!)$ solution to $O(2^n)$.
- The **travelling salesman problem** is a common NP-complete problem. DP with bitmask reduces its $O(n!)$ solution to $O(n^2 2^n)$. This makes the problem feasible for a larger range of n.
- The **secret santa problem (permanent of a matrix)** is a #P-complete problem. DP with bitmask reduces its $O(n!)$ solution to $O(n2^n)$.