Kick Start 2018 - Round C

Analysis: Fairies and Witches

Fairies and Witches: Analysis

The problems asks us to choose a subset of edges of a graph such that no two edges share the same vertex and these edges can be used to form a convex polygon with non-zero area.

There are two key observations:

1. The number of ways to choose a subset of edges such that no two edges share same vertex is fairly small. In fact, for a complete graph with 15 nodes, the total number of such pairings is 10349536.
2. A convex polygon with non-zero area can be formed with sides of length $l_1, l_2, \ldots, l_k$ if and only if **k** ≥ 3 and 2 * max($l_1, l_2, \ldots, l_k$) < sum($l_1, l_2, \ldots, l_k$).

Let's try to prove the upper bound on the total number of different pairings. Let's consider the case in which the total number of nodes (**N**) is even and we have exactly **N**/2 pairings; let us denote the total number of ways to create these **N** / 2 pairings as f(**N**).

If we try to generate all possibilities via brute-force code, then one strategy can be to choose the minimum numbered (starting from 1) unused vertex and try to pair it with all the remaining vertices (see code in later section). We can say that at every level when we are forming a new pairing we are choosing one vertex of the pairing in exactly one way (i.e. minimum numbered (starting from 1) unused vertex) and the other vertex of pairing can be any of the remaining vertices. For example, if the unused vertices are (2,5,6,8,9,10), then we start by choosing 2 (the minimum number) as one vertex, and then we try to pair it with all 5 of the remaining vertices. Whenever we choose a particular pair, we mark both vertices as used, and then we recursively pair up the rest. In mathematical terms, we can represent the total number of ways to form such pairings as follows:

f(**N**) = (**N** - 1) * (**N** - 3) .... * 1

But in our problem there can be any number of edges in final polygon, not just **N**/2. It might be a good idea to try to derive a more general expression before you keep reading!

So we can calculate the total pairings by choosing any set of an even number of vertices and pairing them all up, which gives us the total number as:

g(**N**) = Σ C(**N**, i) * f(i), where i is even and C(**N**, i) is the number of ways of choosing i objects out of **N** different objects.

Small dataset

We can observe that for **N** = 6, we must use all 6 vertices so that we can have 3 edges, since a valid polygon must have at least 3 edges. All possibilities can be generated by enumerating all 6! permutations of arranging 6 numbers, and using the pairs of the first and second, third and fourth, and fifth and sixth numbers to get edges between those node numbers (if they exist).

Our C++ function for checking whether a polygon is valid can be something like this:

```
bool isGoodPolygon(const vector &edges) {
  if(edges.size() < 3) {
```

```
    return false;
  }
  int tot = 0;
  int mx = 0;
  for(int i = 0; i < edges.size(); i++) {
    tot += edges[i];
    mx = max(mx, edges[i]);
  }
  return tot - mx > mx;
}
```

The overall time complexity is O(N!).

Large dataset

The Large dataset uses the same overall strategy, but we need an efficient way to code the intended brute-force solution. One way of doing that is the following C++ code, which is using bit masks to store whether a vertex is used or not.

```
int n;
int L[N][N];
// Whether the i-th bit of mask is on.
int bit(int mask, int i) {
  return (mask >> i) & 1;
}

int solve(int mask, vector &edges) {
  int res = 0;
  if(mask + 1 == (1 << n)) {
    res = isGoodPolygon(edges);
    return res;
  }

  for(int i = 0; i < n; i++) if(!bit(mask, i)) {
    res += solve(mask | (1 << i), edges);
    for(int j = i + 1; j < n; j++) if(!bit(mask, j) && L[i][j]) {
      edges.push_back(L[i][j]);
      res += solve(mask | (1 << i) | (1 << j), edges);
      edges.pop_back();
    }
    break;
  }
  return res;
}
```

The overall time complexity is O(g(N) * N).