Code Jam 2011 - Round 2

Analysis: A.I. War

The computer game A.I. War hides at least a few good algorithmic problems. The author is anything but an expert, but musing about the game brought both this problem and Space Emergency (which originally took place on a graph like this one) to life. The game presents a trickier version of this problem: there are two A.I. homeworlds, multiple other planets that you might want to visit, and you don't know where to find any of them at the start of the game. Fortunately, here we're dealing with a simplified version of the problem and you didn't have to know anything about the game.

Let's first state the problem in graph theory terms. We are given an undirected graph with **P** vertices and **W** edges. We are looking for a sequence of vertices, starting with vertex 0 (our home planet) and ending with a neighbor of 1 (A.I.'s home planet), such that:

- every vertex in the sequence is adjacent to one of the previous vertices
- the sequence is as short as possible
- given the above, the number of distinct neighbors of the vertices in the sequence, but outside the sequence, must be as large as possible

Let **D** be the distance from vertex 0 to vertex 1. It's clear that every such sequence must have at least **D** elements. We also note that we will achieve exactly **D** elements if and only if the sequence forms a shortest path from vertex 0 to vertex 1. Hence we're looking for a shortest path. Unfortunately, there may be many shortest paths and we have to pick the one that optimizes the last requirement.

It simplifies things to include among the "threatened" planets those planets that we do conquer. Given that we already know that we have to conquer exactly **D** planets, we just have to subtract **D** at the end.

Crucial observation

The crucial observation for solving the problem is the following: if a vertex is at distance **d** from 0, it can only be threatened by a vertex at distance **d** - 1, **d** or **d** + 1. This is true because the distances of two adjacent vertices differ by at most 1. Therefore every vertex in the graph is only threatened (or conquered) within at most three consecutive vertices in the path. As we will see, this observation allows us to use dynamic programming to compute best paths of larger and larger lengths.

The algorithm

The first step in our algorithm is to run breadth-first search to compute for every vertex **v** its distance from 0: dist[**v**]. dist[0] = 0 and dist[1] = **D**. Every shortest path will start with vertex 0, then continue with vertices at distance 1, distance 2, ..., distance **D**-1.

For any two adjacent vertices **a**, **b** such that dist[**b**] = dist[**a**] + 1, define **F**(**a**, **b**) = the maximum number of planets threatened or conquered by a shortest path 0→...→**a**→**b**. We will compute this using dynamic programming for increasing distances from 0. The answer to the problem is the maximum value of **F**(**a**, **b**) - **D**, where **a**, **b** are adjacent, dist[**a**] = **D**-2, dist[**b**] = **D**-1, and **b** is adjacent to 1.

F(0, **a**) can be computed directly (there is only one path possible). It remains to compute the values of **F** for a given distance, given the values of **F** at a previous distance. To compute **F**(**b**,

c), dist[**b**]=**d**, dist[**c**]=**d** + 1, try all vertices **a** adjacent to **b** such that dist[**a**] = **d** - 1. In other words, we are looking for paths ending with **a**→**b**→**c**. We have already computed the value of **F**(**a**, **b**) in the previous iteration, the question is: how many new unique threatened vertices does extending the path to **c** add?

This is where our "crucial observation" becomes useful. If a neighbor of **c** was already threatened (or conquered) before, it must have been a neighbor of either **a** or **b**. Therefore we must add the number of neighbors of **c** that are not neighbors of either **a** or **b**. Let's call this value **G**(**a**, **b**, **c**). So we have the recursive formula that we can use for dynamic programming: **F**(**b**, **c**) = $\max_{\mathbf{a}}$ **F**(**a**, **b**) + **G**(**a**, **b**, **c**).

Computing G: four algorithms

We are almost done, but how do we compute the values of **G**, and what is the total run-time of the algorithm? There are O(**W**) values of **F** to compute (at most one for each edge), and for each one we look at O(**P**) other values of **F** (one for each **a**). So if we already knew all the values of **G**, it would take O(**PW**) time to compute all the values of **F** and solve the problem. Computing the values of **G** turns out to be the most time-consuming part though.

We may have to compute **G**(**a**, **b**, **c**) for every sub-path **a**→**b**→**c** of increasing dist. How do we compute these values? We have thought of at least 4 ways. In the actual contest it didn't really matter which you chose, all would easily run in time, but we will mention them just for fun.

- **Approach 1.** We need to compute the number of vertices **v** such that **v** is a neighbor of **c**, but not of **a** or **b**. The simplest approach here is to just check this condition for every **v**. This computation takes O(**P**) time. How many values do we need to compute? At most O($\mathbf{P}^3$), which gives a total run-time of O($\mathbf{P}^4$) for the whole algorithm. We can give a better estimate. Since **a** and **b** are neighbors, there are at most **W** such pairs. This gives the total run-time of O($\mathbf{P}^2$ **W**).
- **Approach 2.** Modify approach 1 slightly. Instead of checking every vertex **v**, we only need to check every neighbor of **c**. This way there are O(**W**) pairs **a**, **b** and O(**W**) pairs **c**, **v**, which gives the total run-time of O($\mathbf{W}^2$).
- **Approach 3.** Modify approach 1 in another way. Precompute the set of neighbors of each vertex as a bitmask neighbors[**v**]. Now we are simply interested in the number of bits set in (neighbors[**c**] **and not** (neighbors[**a**] **or** neighbors[**b**]). This is a speedy computation due to bit-level parallelism. If our computer has machine words of size **w** (this is typically 32 or 64), we cut our work by a factor of **w**. This assumes that we can count the number of bits set in a word in a single step, which modern processors support in a single machine instruction. The final runtime: O($\mathbf{P}^2$ **W** / **w**).
- **Approach 4.** Finally, we come to a theoretically interesting, but rather complex to implement aproach. Define two matrices, **A** of size **W** x **P**, and **B** of size **P** x **P**, as follows. $\mathbf{A}_{\mathbf{ij}}$ = 1 if vertex number **j** does not neighbor any of the two endpoints of the edge number **i**, 0 otherwise. $\mathbf{B}_{\mathbf{ij}}$ = 1 if vertices number **i**, **j** are adjacent (or equal), 0 otherwise. Now compute the matrix product **C** = **A** * **B**. If we apply the definition of matrix product and do the math, we will see that **C** is a **W** x **P** matrix and that the entries are exactly the values of **G**. Specifically, $\mathbf{C}_{\mathbf{ij}}$ = **G**(**a**, **b**, **j**) if the **i**-th edge is **a**↔**b**.
  If we evaluate the matrix product in the natural way, we get the same complexity as in approach 1: O($\mathbf{P}^2$ **W**). We have basically expressed approach 1 in matrix notation.
  The trick is that there are faster matrix multiplication algorithms. The fastest currently known is the Coppersmith-Winograd algorithm. To make a long story short, it gives us a theoretical asymptotic run-time of O($\mathbf{P}^{1.376}$ **W**).

We do not recommend the last approach in the contest. Not only is this unnecessarily complicated, it also wouldn't actually run any faster for the graph sizes we are considering. The

asymptotic advantage would only materialize for impractically enormous, dense graphs.