Code Jam 2012 - Round 3

Analysis: Havannah

This problem probably looked a little scary to many competitors at first. (I, for one, remember some nasty problems involving hexagons in other contests). However, this one wasn't as bad as it looked.

Let's first see what kind of cells there are and how to distinguish between them:

- **Inner Cells**: Those were the cells that are not part of the border of the board. They had 6 neighboring cells.
- **Corner Cells**: Those were the cells where two edges of the board overlap. They had 3 neighboring cells.
- **Edge Cells**: Those were the cells that were neither inner cells, nor corner cells. They had 4 neighboring cells.

Note that it was a bit counter-intuitive that the corner cells were *not* edge cells (as they, in fact, lie on the edges), however that was clearly defined in the statement: "Corners do not count as edges.".

We can enumerate the edges and the corners of the board. Each edge will have a distinct integer index between 0 and 5, inclusive, and each corner will also have a distinct index between 0 and 5, inclusive. This is okay, but for simplicity later we will give the indices from 0 to 5 to the corners and add 6 to the indices of the edges, to obtain 6 to 11 for the edges. If a cell neither is a corner, nor belongs to any of the edges, we can mark it with -1. Thus, we create a function `int checkType(row, col)` that checks if the cell at (row, col) is a corner (and which one if it is), part of an edge (and which edge, if it is) or neither.

- Corners are the cells: {(1, 1), (1, S), (S, S * 2 - 1), (S * 2 - 1, S * 2 - 1), (S * 2 - 1, S), (S, 1)}
- Edges are the cells: {(1, X), (X, 1), (X, S * 2 - 1), (S * 2 - 1, X)} where X is any integer, together with all cells for which |row - col| == S - 1.

Don't forget that corners are not considered edges, so exclude them when checking for edges.

Now let's examine in more detail the winning figures:

- **Bridge**: This was probably the simplest of the three: just a set of stones that connects any two corner cells. The minimal number of stones to achieve it was 2 on a board with S = 2 (which was the minimal possible board).
- **Fork**: This was a bit more complicated. We need a set of stones, that connects 3 distinct edges (i.e. cells from three distinct edges). The minimal number of stones to achieve it was 5 on a board with S = 3 (note that the board with S = 2 does not have any edge cells).
- **Ring**: This was the most complicated figure. We need a set of stones that encircles an empty cell. As it might have been a bit unclear what "encircles" in this case means, the problem statement contained a detailed explanation and an example. The minimal number of stones to achieve it was 6 on a board with S = 3. Note that a ring on a board with S = 2 is impossible, as it inevitably would lead to a bridge first.

As many of the competitors might have realized, the Bridge and Fork cases are relatively simple to detect. One way to solve them would be to use binary search over the move in which they are created and check if a Bridge and/or a Fork exists. That was okay, however it didn't work for Rings. Some competitors used this solution for Forks and Bridges, and another one for Rings. However, this required some extra code, which is rarely a good idea in a speed contest.

Some competitors probably realized that even an $O(M^2)$ solution would pass. However, we will describe an O(M) solution, which was very fast and not much harder to implement.

First, it's a good idea to store the board in a both memory and time efficient way. The thing to notice was that when the board is really big, then it is also really sparse (only 10000 stones in a board with over 9000000 cells). So the way to go was to use any kind of set data structure our programming language of choice provided. In C++, a hashset is fastest, but tree sets are also fine. Putting a stone is a constant operation if we are using a hashset and logarithmic one if we're using balanced tree set (for example C++ STL set).

Now we start putting stones on the board in the order they are given. We need a fast way to check if putting a stone created some of the winning figures. The fork and the bridge needed some cells to be connected (i.e. to belong to the same connected component). In fact, the ring also requires the same thing. As many competitors in this round probably know, these are usually handled using the union find algorithm. As a reminder union-find supports the following basic operationss:

- Create a new node, belonging to a group of size 1.
- Merge two new groups into one.
- For a given node, find the group it is.

Implemented correctly, all of these operations are blazingly fast.

We will use union-find here with a minor addition. After putting a stone, we see if some of the neighboring cells contains a stone already. If none do, we add the new stone to its own connected component. If there are stones in adjacent squares, we merge them all to a single connected component that also contains the new stone. Additionally to the straight-forward union-find algorithm, we will store what type of cells each component contains. Since the number of different types of cells we are interested in is relatively small (only 12 types) we can use a mask with 12 bits (corresponding to the indices from 0 to 11 we mentioned earlier). We do this in order to have an easy way to see if a component contains corner or edge cells. When merging the component A to component B, we change the mask of B to contain the bits of the mask of A as well (since now they are the same component). This can be done really easily with the bitwise operation OR. If, after merging, a component ends up with 2 set bits in the first 6, or 3 set bits in the second 6 positions, then we've just created a bridge or a fork, respectively.

Everything's fine with that, but we still haven't answered the question how do we handle rings. Well, having the stones and their respective components makes it easy to check for them. In order for a ring to be created, we must have just added a stone that connects a component to itself. But if it does, it still does not necessarily create a ring. What we can do is check all neighbors of the cell where we are putting the stone, and search for two cells with stones with the same component.

We can represent the neighbors of the given cell as the following:

```
# 1 2
6 * 3
5 4 #
```

Going clockwise twice (or counter-clockwise, if you prefer, it doesn't matter) and looking at the neighbors there should be one of the following sequences (as a subsequence of the created sequence of 12):

1. {C, X1, C, Y1, Y2, Y3}
2. {C, X1, X2, C, Y1, Y2}
3. {C, X1, X2, X3, C, Y1}

where the cells C belong to the same component, and each of the cells X and Y are either empty or belong to some component (not necessarily the same as C, and not necessarily the same as other Xs and Ys).

After this move a ring is formed if and only if:

- At least one of the Xs is an empty cell
- At least one of the Ys is an empty cell

(Note that if there is a {C, X1, C, Y1, Y2, Y3} sub-sequence, then there will be a {C, X1, X2, X3, C, Y1} one, however we've included them both for clarity).

Why is this true? Well, if none of the Xs or none of the Ys is an empty cell, then the two Cs were already connected "from this side" and adding the stone doesn't change it into a ring (obviously). If both some of the X cells and some of the Y cells contain an empty cell, then we've just encircled at least one of them! Imagine it this way - a circle has two sides - inside and outside. We know that we created a circle (since we are connecting a component to itself), but we don't know which side is the inner one and which side is the outer one. Thus, if both contain an empty cell, then we have an empty cell in the inner side for sure.

What is the time complexity of the given algorithm? Since OR is a constant operation, we have a constant number of bits to check when looking for a Fork or a Bridge, and checking for a ring involves also a constant number of operations (equal to the number of neighbors), the complexity is dominated by the speed of the union-find operations. Thus, the total complexity of the algorithm is O(M * RACK(M)), where RACK() is the inverse of the Ackerman function. However, RACK() is so slowly growing, that you can assume the algorithm will run in nearly O(M) for the given constraints.

Remark: Another nice trick for dealing with rings is to start with all pieces played on the board and work backwards, always considering which empty squares are connected to each other. Removing a stone possibly connects two of these components, and so we can again use union-find to track the state quickly. This is conceptually simpler, but it is slower because we need to first find all the connected components after the pieces are played.