Problem:
The goal of this problem is to simulate a “Cat and Mouse” search game. Think of it as something that a cat might use to design a collection of traps with the best chance of catching a mouse – or that a mouse might use to find a safe path through a cat's traps.
Two players are given starting positions on a Directed Graph and take turns moving; staying in place is also counted as a move. The mouse has the first move. To win, the mouse much reach a goal node, and the cat must reach the mouse – before a maximum number of moves. The cat can win by finding the mouse, or the mouse bumping into the cat.

There are many choices when implementing this problem. This code is a quickly, and relatively simple solution to the problem.

General Algorithm:
1. Enumerate all valid cat and mouse maths of maximum length
2. Take Cartesian Product (“Inner Join”) of the paths, search for and count win/loose/draw states. Capture a successful path
3. Display results

Serial Implementation:
• Graph is read from file
• For each Player: Follow edges to target nodes starting with player's start node
  • Handle dead-ends, goal nodes, and no-moves
  • Add path to all paths list
• Evaluate the Cartesian Product – all possible combinations of paths
  • Check wins at initial state
  • Check wins after mouse moves, and after cat moves; or declare draw.
• Display Results
**Objects**

- **Node**
  - Name – 3 character name of node
  - private Node::vEdgeOut vector<bool> of edges to next node in graph.
  - addEdge(NodeId) – update vEdgeOut
  - allEdges(vector<NodeId> out) – return all edges
  - addReverseEdge(NodeId) / countIn / countOut – unused, for future path optimization

- **GameState**
  - typedef vector<NodeId> PlayerPath
  - typedef tbb::enumerable_thread_specific<BigCount> CounterType;
  - typedef tbb::concurrent_vector<PlayerPath> PlayerPaths
  - GameState::Read(ifstream) – read initial state and graph from file
  - EnumeratePaths_Serial() – fills PlayerPaths
    calls EnumeratePathCat_Serial() EnumeratePathMouse_Serial
  - class EvalPaths
    - tbb::parallel_for( tbb::blocked_range2d< MousePaths, CatPaths> )
      - Mouse's Move
        if (pathMouse[k] == pathCat[k-1]) mouse bumps into cat
        if (pathMouse[k] == niGoal) mouse wins
      - Cat's Move
        if (pathMouse[k] == pathCat[k]) cat gets mouse
  - SumWinners() – combine the enumerable_thread_specific (CounterType) counts
  - Result() – print results and sample path

**Technical**

- Used Intel Threading Building Blocks library 3.0.056 with MSVC9 / 2008
- Does not use BigNum – will fail if sum of path count is greater than unsigned long can store.
  It should be simple to add by replacing “typedef unsigned long BigCount” with a BigNum
  implementation (the Cartesian Product is never stored anywhere) – but I have no time to test it.

**Parallel Implementation:**

- Encountered major problems with tbb::combinable. It appears to be unusable without Lambda
  syntax. Extremely difficult to find code samples that use it. The errors are difficult and unhelpful
  because templates are involved.
- Similar problems with tbb::enumerable_thread_specific – it breaks VS2008 intellisense, but I was
  able to determine how to use it from the sample.
Possible Further Optimizations:

- Parallelize EnumeratePaths
- Graph Checks:
  - if cat cannot reach mouse, cat cannot win
  - if mouse cannot reach goal, mouse cannot win
  - if both of the above, neither can win (results do not need to be processed, only counted)
- Parallel version of EvalPaths for a single, very long, path evaluation

Problems:

- Tbb::combinable seems unusable inside parallel_for without Lambda function syntax
  - unsigned long *x = winDraw.local();
    - error C2663: 'tbb::combinable<T>::local' : 2 overloads have no legal conversion for 'this' pointer
  - unsigned long x = countDraw.combine(std::plus<unsigned long>);
    - // Error C2275: 'std::plus<_Ty>' : illegal use of this type as an expression
  - unsigned long x = countDraw.combine(&GameState::sum);
    - // Error C2064: term does not evaluate to a function taking 2 arguments

Conclusion:

This is a problem with many choices to make, and possibilities to test. I was not able to get into it as much as I would have liked (Cold season), but I think the code produces correct results in a reasonable time, and the simplicity of the code makes it a good example.