4-22-2022

Algoraph but in C++

Adam James Czeranko
Hope College

Andres Louis Solorzano
Hope College

Algoraph But In C/C++

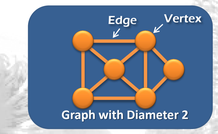
Andres Louis Solorzano, Adam James Czeranko, and Dr. Charles A. Cusack (Advisor)

For more information contact:
Dr. Charles A. Cusack
cusack@hope.edu
(616) 395-7271

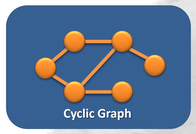
Graphs

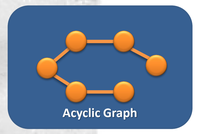
A *graph* (not to be confused with giraffes) is a set of *vertices* and set of *edges* connecting said vertices.

Diameter: The shortest greatest distance between any pair of vertices
Note that graphs with diam ≤ 2 are not Lemke graphs



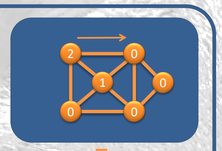
Cycle: A non-repeated sequence of connected vertices where the first vertex is also the last vertex
Note that acyclic graphs are not Lemke graphs



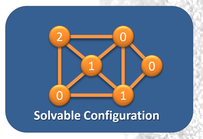


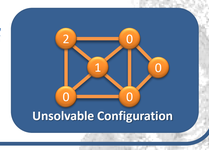
Pebbling

- In *graph pebbling*, each vertex is assigned a non-negative integer which represents the number of pebbles on that vertex.
- During a *pebbling move*, 2 pebbles are removed from a source vertex and 1 pebble is added to an adjacent vertex.
- A vertex is *reachable* if there is a sequence of pebbling moves that places 1 pebble on the vertex.
- Given a configuration, a graph is *solvable* if every vertex is reachable.
- The *pebbling number* of a graph G is the smallest integer $\pi(G)$, such that any configuration that uses $\pi(G)$ pebbles is solvable.
- A graph satisfies the *two-pebbling property* if for any configuration of more than $2\pi(G) - q$ pebbles, where q is the number of vertices in G with at least one pebble, two pebbles can be moved to any vertex.



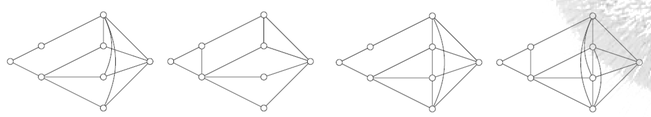
A legal pebbling move





Lemke Graphs

- *Lemke graphs* are graphs that do not satisfy the two-pebbling property.



The original Lemke graph, the two other Lemke graphs with minimum number of edges, and the Lemke graph with maximum number of edges on 8 vertices.

Number of Lemke Graphs for:
Less than 8 Vertices: 0 — 8 Vertices: 22
9 Vertices: 306 — 10 Vertices: ≥5957

Java vs. C/C++ Implementation

- For this project, our goal was to reimplement the Java Algoraph algorithms into C/C++ and improve efficiency.

Programming Language		Serial Time (sec) (1 core)	Parallel Time (sec) (24 cores)
1	Java	20850.19	Wall Clock Time: 1976.33 CPU Time: 24067.2
2	C/C++	18592.16	Wall Clock Time: 1526.16 CPU Time: 21083.68

* Time for running graphs of 8 vertices

Notice how the C/C++ version is ~20% faster than the Java Version in parallel and ~10% in serial

Differences In Implementation:

- The C/C++ version features just the algorithms to calculate Pebbling Number and Two-Pebbling Property whereas the Java version algorithms are implemented in the game framework which adds complexity to the code.
- Being that the code was written in C/C++, the new implementation required explicit manual memory management to ensure no memory leaks and no excess use of memory. In Java, these issues are handled automatically.
- Refactored the data structures in the C/C++ version to optimize algorithms to avoid recalculations.
- For parallelization, we used OpenMP for C/C++ and used the prebuilt concurrent package in java.util. for Java.
- In general, the compilation/interpretation of the code differ: Java is compiled to Java Bytecode and then is executed through the Java Virtual Machine on the computer whereas C/C++ is compiled to machine code and runs on the computer directly.

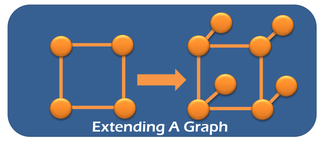
Algorithms

The Heuristics:

- **IsSolvableDistance**
  - For **every** vertex v with $\geq 2^k$ pebbles, mark every vertex of distance k or less from v as reachable. If all vertices are marked reachable, then the configuration is clearly **solvable**. Otherwise the solvability remains **unknown**.
- **IsSolvableShortestPath**
  - Uses the **Floyd–Warshall shortest-path** algorithm and makes all possible moves along the shortest-path tree toward the desired root.
- **IsSolvableShortestPebblePath**
  - Almost **identical** to the previous algorithm **except** that the distance from the root to a given node is the number of edges **minus** the number of vertices with pebbles on them.
- **IsUnsolvableWeightFunction**
  - Uses a weight function to determine if a configuration is unsolvable. Otherwise it is **unknown**.

Main Pebbling Algorithms:

- **Merge Pebbles**
  - Maintains a list of legal merged pebbles for each vertex and has **two** phases— the **distribute** phase where the initial pebble configuration distributes pebbles, followed by the **merge** phase where possible pebbling moves are made.
- **Pebbling Number**
  - Uses a **backtracking** algorithm to construct the unsolvable configurations on G with the maximum number of pebbles, backtracking when a solvable configuration is found. The algorithm adds pebbles to a configuration until it is **solvable**, at which point it removes the last pebble and places it on the next vertex and continues.
- **Two-Pebbling Property**
  - "Extends" the graph by adding a new vertex to each preexisting vertex with one edge between the two. Notice that if the extension nodes are **reachable**, this means that one is able to reach the original node with **two** pebbles. Thus, we can use our **solvability** algorithm to determine the **Two-Pebbling Property** after extending a graph.




```
111  #pragma omp parallel for schedule(dynamic,1) shared(finishedGraphs,graphs,
     currentlyRunningGraphs) num_threads(cores)
112  for(int i = 0; i < (int)graphs.size();i++) {
113      GraphData graphData = graphs[i];
114      if(finishedGraphs.find(graphData.graphID)==finishedGraphs.end()) {
115          #pragma omp critical
116              currentlyRunningGraphs.insert(graphData.graphID);
117          std::string result = runGraph(graphData.g6Form,graphData.graphID,
118          graphData.pebblingNum,minDiam,maxDiam);
119          #pragma omp critical
120          {
121              currentlyRunningGraphs.erase(graphData.graphID);
122              outFile<<result<<std::flush;
123          } } }
```

Parallelization

Parallelization is the process of adapting a program to run in parallel instead of serial (multiple cores vs single core).

Diagram of Fork-Join Parallelization Model

OpenMP follows the fork-join framework; the program begins within a single "master" thread that then creates more threads to be executed on separate cores.

Parallelization should not modify any functionality of the program; whether or not the red "#pragma omp" lines were included in the code to the left, the program still functions as intended.

Acknowledgements

- Hope College Computer Science Department
- Our awesome mentor, Charles A. Cusack
- All the previous faculty and students who worked on Algoraph

Hope COLLEGE

Algoraph