Lab 6 - Hamiltonian cycle detection

Şut George-Mihai

I. Assignment task

Given a directed graph, find a Hamiltonean cycle, if one exists. Use multiple threads to parallelize the search.

II. Implementations

A. Algorithm

The implemented algorithm is a basic backtracking algorithm which tries each possible Hamiltonian path by picking a specific node as the starting point and building the path gradually. At each step, we check if the current node is not in the path and if there is an edge between the previous node in the path and the current node. If the condition is satisfied, the current node is added to the container used for storing the path. At last, if the size of the constructed path is equal to the number of nodes in the graph, we check if there is an edge between the last node in the path and the first node. If so, we set a boolean $solutionFound$ to signify that the search is over and a solution has been found.

B. Parallelizing the solution

The parallel implementation uses the $std :: thread :: hardware\_concurrency$ function to obtain the number of concurrent threads supported (the C++ documentation suggests that this should only be considered as a hint, but in our case the CPU contains 6 cores with 2 threads each, so 12 threads in total, exactly like the function indicates). Knowing the optimal number of threads, we split the work by dividing the number of nodes to the number of threads. Each thread will take this amount of nodes as starting nodes for the searched paths and the solution will be the one found by the quickest thread.

C. Synchronization

A simple atomic variable is used to manage a counter for each thread (this is done in a method of the $HamiltonianCycleDetector$ class called $parallelHamiltonianCycleDetection$). The counter helps at representing the thread index when accessing the thread's own $std :: vector$ (which represents the solution). Before having this atomic variable, the program crashed with several errors such as $SIGSEGV$.

Additionally, when setting the result of the computation (in the $parallelTryFromNode$ method of the aforementioned class), we use a mutex for protecting the result variable from other threads while doing the write operation. In this way, we ensure that the thread which finishes the task most quickly doesn't have its result replaced by another thread.

III. Hardware specifications

Laptop: ASUS Strix 15 GL503GE

OS: PopOS 21.04

Kernel: x86_64 Linux 5.13.0-7614-generic

CPU: Intel i7-8750H (12) @ 4.100GHz

GPU: NVIDIA GeForce GTX 1050 Ti

IV. Results

Number of nodes	Sequential	Parallel
25	0.00105	0.00260
50	0.00539	0.01222
100	0.02049	0.04311
250	11.4615	5.79233

TABLE I: Table with execution times (seconds)