Exploration of Parallel Single Source Shortest Path Algorithms

Shane Loretz
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
shane.loretz@students.sjsu.edu

ABSTRACT
Five Parallel variants of the Bellman-Ford algorithm were developed and compared with a sequential implementation as a control. Tests were done on a typical quad-core desktop processor using random weakly connected directed graphs with positive edge weights. All but one of the parallel variants were found to be significantly slower than the sequential implementation, and their time to execute increased with each additional thread. The reason for this could be that the overhead of the OS scheduling threads was longer than one iteration of Bellman-Ford. The final variant was faster when run sequentially than the control algorithm. All variants were several orders of magnitude slower than a binary heap implementation of Dijkstra's algorithm.

1. INTRODUCTION
The single source shortest path problem is well understood sequentially. Binary and Fibonacci heap implementations of Dijkstra's algorithm can find solutions to large graphs with positive edge weights quickly, but Dijkstra's algorithm is inherently sequential. At every iteration it relies processing the nodes in order of their current tentative distance. Bellman-Ford is much slower than Dijkstra, but it works on all graph types and can consider all edges in parallel. Delta-stepping is a parallel algorithm developed recently that Combines elements of Dijkstra and Bellman-Ford which can be described as running Dijkstra using a priority queue of buckets [4].

Attempts were made to make parallel implementations of Bellman-Ford which resulted in five working variants. Each variant built off of lessons learned from the previous. Because only positive edge weights were used, the negative cycle check was left out of the variants source code. This means the results generated during testing are faster than a full implementation of Bellman-Ford.

2. TESTING

2.1 Graph Generation
The graphs used to test the different variants and control were generated to be weakly connected directed graphs with positive edge weights. The graph object instance contained an array of all nodes and an array of all edges. The nodes also contained an array of all edges exiting them. This is an adjacency list representation. Dynamically sized arrays were used for all containers.

2.2 Testing Methodology
A set of tests were run for each combination of the following: number of threads in {1,2,3,4}, number of vertexes in {1000,2500,5000,7500}, and average number of edges per vertex in {10, 20, 30, 40}. Three tests were run for each combination and averaged. The tests were started and the results were recorded using a Python script. The main test program is written in C++. It includes the source for all variants, and it measures their execution time. The results are output on STDOUT where the Python script records the results to a CSV file. The total testing time took 34 minutes in clock time and 70 minutes in CPU time.

The algorithms were tested for correctness by comparing their results to an implementation of Dijkstra's algorithm using a binary heap as a priority queue. All variants are assumed to be correct because no case was observed during testing where the answer from the variants was different from the answer produced by the Dijkstra implementation.

3. SEQUENTIAL BELLMAN-FORD
A sequential variant of the Bellman-Ford algorithm was developed as a control for testing.

Table 1: Sequential Bellman-Ford Pseudo code

	SEQ BELLMAN-FORD:
1	for each vertex N in G.V:
2	N.distance = infinity
3	vertex startVert.distance = 0
4	For each n in [1,|G.V|):
5	for each edge(vertex S,vertex T,weight W) in G.E:
6	alt = S.distance + W
7	if alt < T.distance:
8	T.distance = alt
9	shortestDistance = vertex endVert.distance

This algorithm runs in V + V*E time where V is the number of vertexes and E is the number of edges. This runs in O(VE) time as expected of Bellman-Ford.

Only the shortest distance is returned. The actual shortest path was not recorded by any of the algorithms, because the actual path is not needed when comparing the run times of the algorithms. Omitting the code to record the path does not change the algorithm's run time.

4. PARALLEL BELLMAN-FORD #1

The first attempt at parallelizing bellman-ford introduces a mutex on every vertex, it divides the work by giving each worker a range of vertexes to operate on, and it synchronizes threads using a thread barrier.

Table 2: Variant #1 Pseudo Code

	PAR BELLMAN-FORD #1:
1	for each vertex N in G.V:
2	N.distance = infinity
3	vertex startVert.distance = 0
4	for each K in [0,numThreads)
5	start worker thread with |G.V| / numThreads + 1 nodes assigned to it
6	wait for all threads to complete
7	shortestDistance = vertex endVert.distance
	WORKER THREAD:
1	For each n in [1,|G.V|):
2	for each vertex S in thread specific range:
3	S.mutex.lock()
4	baseDistance = S.distance
5	S.mutex.unlock()
6	for each outgoing edge(vertex S, vertex T, vertex W):
7	alt = baseDistance + w
8	T.mutex.lock()
9	if (alt < T.distance):
10	T.distance = alt
11	T.mutex.unlock()
12	synchronize with thread barrier

The algorithm acquires the vertex specific mutex before reading and writing its tentative distance. The worker threads are synchronized |V|-1 times to ensure that the distances propagate across the graph.

4.1 Performance

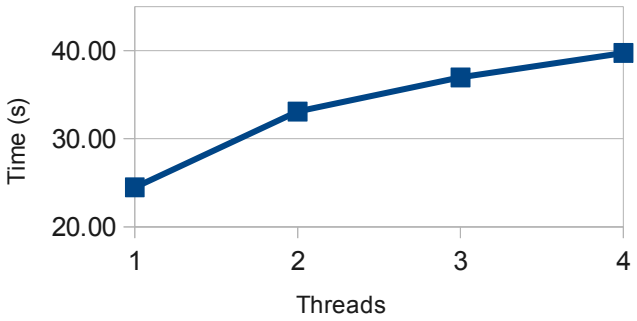
The expected performance of this variant is V + V*E / N where N is the number of worker threads. On paper this looks like an improvement over the sequential variant, but in practice it proved to be much slower.

Table 3: Average running times with 7.5k vertexes

Seq BF	PBF 1 worker	PBF 2 worker	PBF 3 worker	PBF 4 worker
3.20	15.73	21.19	23.74	25.53

The locking overhead is much greater than the actual work done. The parallel algorithm run with 1 worker is effectively the same as sequential Bellman-Ford, but it runs close to five times slower. Adding more threads only makes the problem worse. The threads slow each other down as they must acquire the mutex on both vertexes.

Figure 1: PBF #1 vs number of threads with 300k edges



With 300k edges and 4 workers this variant takes 39.74 seconds to complete, while the sequential version takes only 5.25 seconds. Increasing the parallelism actually results in worse performance because of the locking overhead.

5. PARALLEL BELLMAN-FORD #2

The second attempt eliminates the need to acquire a mutex on the source vertex of an edge by storing two tentative distances. One distance is used for reading, and the other distance stores the next tentative distance. The algorithm acquires only the mutex on the destination vertex and modifies the next distance. A cleanup loop sets the reading distance to the next distance after all edges have been relaxed.

Table 4: Variant #2 Pseudo Code

	PAR BELLMAN-FORD #2:
1	for each vertex N in G.V:

2	N.distance = infinity
3	N.nextDistance = infinity
4	vertex startVert.distance = 0
5	vertex startVert.nextDistance = 0
6	for each K in [0,numThreads)
7	start worker thread with |G.V| / numThreads + 1 nodes assigned to it
8	wait for all threads to complete
9	shortestDistance = vertex endVert.distance
	WORKER THREAD:
1	For each n in [1,|G.V|):
2	for each vertex S in thread specific range:
3	baseDistance = S.distance
4	for each outgoing edge(vertex S, vertex T, vertex W):
5	alt = baseDistance + w
6	T.mutex.lock()
7	if (alt < T.nextDistance):
8	T.nextDistance = alt
9	T.mutex.unlock()
10	synchronize with thread barrier
11	for each vertex S in thread specific range:
12	S.distance = S.nextDistace
13	synchronize with thread barrier

This variant trades the overhead of acquiring a mutex on the source vertex for the overhead of an extra cleanup loop on all owned nodes.

5.1 Performance

The expected performance of this variant is V + V*E/N + V/N. In practice it is only a minor improvement over the previous variant.

Table 5: Average running times with 300k edges

Threads	PBF #1	PBF #2
1	24.48	23.97
2	33.08	32.52
3	36.96	36.71
4	39.74	39.55

This variant's performance still continues to decrease when as the number of workers increases. It shows very minor improvement in run time. It appears that trading the overhead of locking the source mutex with an extra loop on vertices is close to even.

6. PARALLEL BELLMAN-FORD #2.5

A minor change to PBF #2 makes for a big improvement in performance. PBF#2.5 does an extra comparison to check if the new distance is less than the old distance, and it only acquires the destination mutex if it is. This removes the locking overhead from all edges that would not change the tentative distance.

Table 6: Variant #2.5 Pseudo Code

	WORKER THREAD:
1	For each n in [1,|G.V|):
2	for each vertex S in thread specific range:
3	baseDistance = S.distance
4	for each outgoing edge(vertex S, vertex T, vertex W):
5	alt = baseDistance + w
	if (alt < T.distance):
6	T.mutex.lock()
7	if (alt < T.nextDistance):
8	T.nextDistance = alt
9	T.mutex.unlock()
10	synchronize with thread barrier
11	for each vertex S in thread specific range:
12	S.distance = S.nextDistace
13	synchronize with thread barrier

6.1 Performance

The expected performance of this variant is the same as PBF#2 since the relaxation step happens in constant time: V + V*E/N +V/N, but in practice it is much faster.

Table 7: Average running times with 300k edges

Threads	Seq BF	PBF #2	PBF #2.5
1	5.25	23.97	6.34
2	x	32.52	6.76

3	x	36.71	7.01
4	x	39.55	7.21

As expected PBF #2.5 is slower than Sequential Bellman-Ford with just one thread, but it still gets even slower as more threads are added. It could be that waiting on a mutex per destination node is causing more slowdown than speedup.

7. PARALLEL BELLMAN-FORD #3

This variant is similar to PBF#2.5, but it makes use of a queue to allow trying to acquire a lock without blocking. PBF#3 workers do not suspend when they fail to acquire a lock, instead they try to acquire the next lock in their queue.

Table 8: Variant #3 Pseudo Code

	WORKER THREAD:
1	For each n in [1,|G.V|):
	create empty relaxRequestQ
2	for each vertex S in thread specific range:
3	baseDistance = S.distance
4	for each outgoing edge(vertex S, vertex T, vertex W):
5	alt = baseDistance + w
	if (alt < T.distance):
	relaxRequestQ.add(alt,T)
	while relaxRequestQ.isNotEmpty()
	alt,T = relaxRequestQ.pop_front()
6	if (T.mutex.try_lock()):
7	if (alt < T.nextDistance):
8	T.nextDistance = alt
9	T.mutex.unlock()
	else: //Add request to back of Q
	relaxRequestQ.add(alt,T)
10	synchronize with thread barrier
11	for each vertex S in thread specific range:
12	S.distance = S.nextDistace
13	synchronize with thread barrier

PBF#3 eliminates all waiting time on vertex mutexes. Mutexes that are not acquired are put back into the relax queue instead of blocking on them.

7.1 Performance

This variant showed only minor performance gains. Analyzing the time complexity of this algorithm is difficult and depends on how many relaxation requests are expected. The extra time serving requests off of a queue is about equal to the time gained from not blocking on a mutex lock operation.

Table 9: Average running times with 300k edges

Threads	Seq BF	PBF #2.3	PBF #3
1	5.25	6.34	6.32
2	x	6.76	6.79
3	x	7.01	7.01
4	x	7.21	7.21

The algorithm still shows increased time to execute when adding more threads. The reasons for this are not known. It is possible that the threads are being scheduled at different times by the operating system, and that the amount of work to do between iterations is much less than the time between threads being scheduled.

8. PARALLEL BELLMAN-FORD #4

This final variant removes the need for each vertex to have its own mutex. It does this by having the vertex object have an array of nextDistances, where each thread has its own. The minimum of these tentative distances is chosen to replace distance attribute on the node. There is no need to have vertex level locks because the thread was assigned a specific range of vertexes. Each thread does the clean up on only their vertexes.

Table 10: Variant #4 Pseudo Code

	PAR BELLMAN-FORD #4:
1	for each vertex N in G.V:
2	N.distance = infinity
3	N.nextDistance[numWorkers] = {infinity,...}
4	vertex startVert.distance = 0
5	vertex startVert.nextDistance[0:numWorkers-1] = {0,...}
6	for each K in [0,numThreads)
7	start worker thread with |G.V| / numThreads + 1 nodes assigned to it
8	wait for all threads to complete
9	shortestDistance = vertex endVert.distance

	WORKER THREAD:
1	For each n in [1,|G.V|):
2	for each vertex S in thread specific range:
3	baseDistance = S.distance
4	for each outgoing edge(vertex S, vertex T, vertex W):
5	alt = baseDistance + w
7	if (alt < T.nextDistance[worker_id]):
8	T.nextDistance[worker_id] = alt
10	synchronize with thread barrier
11	for each vertex S in thread specific range:
12	S.distance = min(S.nextDistance[])
13	synchronize with thread barrier

8.1 Performance

PBF #4 is an anomaly. Its algorithm suggests a time complexity of V*N + V*E/N + V/N which for N = 1 should be slower than the control sequential Bellman-Ford implementation. It turns out that this implementation runs significantly faster with just one thread than the sequential implementation.

Table 11: Average running times with 7.5k vertexes

Seq BF	PBF4 1 worker	PBF4 2 workers	PBF4 3 workers	PBF4 4 workers
3.20	3.14	3.38	3.48	3.60

The reason for this unexpected result is not known. PBF#4 should be slower than the control. Its time complexity is more than the control implementation and it also has locking overhead. The variant still takes longer with more workers available.

9. ANALYSIS

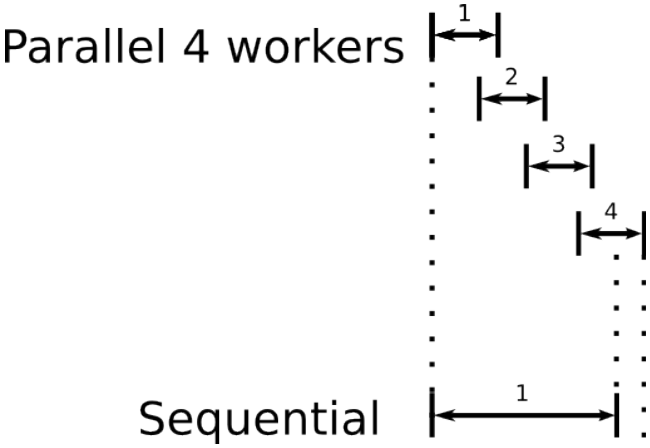
Every variant showed the same behavior: increasing the number of worker threads causes the total execution time to increase. A first thought might be that the threads are somehow running sequentially, but the CPU time of the test is more than double the wall clock time. This proves that multiple cores were being utilized. There may be an explanation for the behavior, but it requires several assumptions.

The first assumption is that the OS scheduler interrupts the CPU at fixed intervals. Analysis of the output from clock() in <ctime> shows that the number of clock always ends in four 0s, and the smallest time between two calls is 0.01 seconds. If it is assumed that the clock value returned bock clock() comes from a register that is queried every time the OS scheduler runs, then from this information it seems the first assumption is valid, and that the OS scheduler is called ever 0.01 seconds.

The second assumption is that each iteration in Bellman-Ford's outer loop takes much less time to complete than the interval the OS scheduler runs at. The average run time for sequential Bellman-Ford is 3.20 seconds with 7.5k vertexes. This suggests each iteration of the loop takes 3.20 s / 7500 = 0.0004 seconds. This is much less than 0.01 seconds and it suggests the assumption is true.

With these two assumptions the following scenario could occur. Say there are 4 worker threads and each thread is initially created and placed into a wait queue. The next time the OS scheduler runs it schedules all threads, but because it takes time to schedule each thread starts slightly after the previous thread. The first thread completes its work and encounters the thread barrier at the end of the loop. This puts the thread back into the wait queue. The second and third threads behave the same. The fourth thread completes last, but when it encounters the barrier it wake up all threads. The CPU scheduler is woken to schedule these threads and it schedules all of the threads again, repeating the cycle. The result is that each iteration effectively takes longer since the threads must all wait until the final thread reaches the barrier.

Figure 2: Possible explanation for execution time increasing with more threads



10. CONCLUSION

If the analysis is correct then it appears that parallelizing Bellman-Ford is not a worthwhile endeavor on consumer desktop machines. The variants presented here would only result in a speed increase when the amount of work they do each iteration is much greater than the time it takes the OS scheduler to schedule new threads. This means running the variants with more nodes and edges. It is not unheard of for parallel algorithms needing large inputs to become practical. An implementation of Delta-stepping was found to only be 3.96 times faster with 2^{21} nodes versus 30 times faster with 2^{28} nodes [2].

Implementations of Dijkstra run several orders of magnitude quicker. There exists a hybrid of Dijkstra and Bellman-Ford that can be run in parallel called delta-stepping[3]. Unfortunately attempts at getting a working implementation of this algorithm failed, so it's viability on desktop machines cannot be determined.

11. REFERENCES

[1] Cormen, Leiserson, Rivest, Stein. Introduction to Algorithms. 3ed. The MIT Press (2009)

[2] Madduri, Bader, Berry, Croback. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. Georiga Institute of Technology (2007)

[3] Meyer, Sanders. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. Max-Planck-Institut fur Informatik (2003)

[4] TeamTiffany. Writing a (spinning) thread barrier using c++11 atomics (2013) http://stackoverflow.com/questions/8115267/writing-a-spinning-thread-barrier-using-c11-atomics

12. APPENDIX

12.1 Averaged Results

12.1.1 Sequential Binary Dijkstra and Bellman-Ford

avgEdges	numVerts	sbd	sbf
10	1000	0.00	0.01
20	1000	0.00	0.02
30	1000	0.00	0.03
40	1000	0.00	0.04
10	2500	0.00	0.09
20	2500	0.00	0.20
30	2500	0.00	0.30
40	2500	0.00	0.41
10	5000	0.00	0.47
20	5000	0.00	1.00
30	5000	0.00	1.56
40	5000	0.01	2.09
10	7500	0.00	1.17
20	7500	0.00	2.50
30	7500	0.01	3.86
40	7500	0.01	5.25

12.1.2 Parallel Bellman-Ford #1 with 1 to 4 threads

avgEdges	numVerts	pbf_v1_1	pbf_v1_2	pbf_v1_3	pbf_v1_4
10	1000	0.12	0.17	0.19	0.20
20	1000	0.22	0.32	0.36	0.40
30	1000	0.31	0.48	0.56	0.60
40	1000	0.40	0.64	0.74	0.81
10	2500	0.74	1.04	1.17	1.25
20	2500	1.33	2.05	2.34	2.52
30	2500	1.96	2.93	3.29	3.51
40	2500	2.42	4.16	4.87	5.29
10	5000	3.06	4.05	4.51	4.82
20	5000	5.55	7.92	9.06	9.77
30	5000	8.08	11.77	13.53	14.68
40	5000	10.76	14.92	16.86	18.11
10	7500	6.93	9.03	10.06	10.78
20	7500	12.93	17.56	19.86	21.44
30	7500	18.59	25.08	28.09	30.18
40	7500	24.48	33.08	36.96	39.74

12.1.3 Parallel Bellman-Ford #2 with 1 to 4 threads

avgEdges	numVerts	pbf_v2_1	pbf_v2_2	pbf_v2_3	pbf_v2_4
10	1000	0.11	0.15	0.18	0.20
20	1000	0.21	0.31	0.35	0.39
30	1000	0.29	0.47	0.55	0.60
40	1000	0.39	0.63	0.73	0.81
10	2500	0.66	0.97	1.09	1.17
20	2500	1.26	2.00	2.28	2.45
30	2500	1.90	2.90	3.23	3.45
40	2500	2.40	4.20	4.90	5.31
10	5000	2.75	3.73	4.17	4.47
20	5000	5.33	7.75	8.80	9.50
30	5000	7.88	11.73	13.40	14.51
40	5000	10.52	14.91	16.79	17.97
10	7500	6.26	8.30	9.25	9.92
20	7500	12.31	16.79	19.12	20.82
30	7500	18.16	24.57	27.70	29.78
40	7500	23.97	32.52	36.71	39.55

12.1.4 Parallel Bellman-Ford #2.5 with 1 to 4 threads

avgEdges	numVerts	pbf_v2p5_1	pbf_v2p5_2	pbf_v2p5_3	pbf_v2p5_4
10	1000	0.02	0.03	0.03	0.04
20	1000	0.04	0.05	0.04	0.06
30	1000	0.05	0.06	0.07	0.07
40	1000	0.07	0.08	0.08	0.08
10	2500	0.18	0.21	0.22	0.23
20	2500	0.30	0.34	0.35	0.36
30	2500	0.40	0.45	0.47	0.48
40	2500	0.53	0.58	0.60	0.62
10	5000	0.80	0.92	0.96	0.99
20	5000	1.40	1.56	1.62	1.67
30	5000	2.01	2.18	2.25	2.31
40	5000	2.59	2.78	2.86	2.95
10	7500	1.88	2.15	2.22	2.29
20	7500	3.48	3.79	3.94	4.06
30	7500	4.90	5.28	5.41	5.56
40	7500	6.34	6.76	7.01	7.21

12.1.5 Parallel Bellman-Ford #3 with 1 to 4 threads

avgEdges	numVerts	pbf_v3_1	pbf_v3_2	pbf_v3_3	pbf_v3_4
10	1000	0.02	0.03	0.03	0.04
20	1000	0.04	0.04	0.05	0.05
30	1000	0.05	0.06	0.06	0.07
40	1000	0.07	0.08	0.08	0.08
10	2500	0.17	0.21	0.22	0.23
20	2500	0.30	0.34	0.35	0.36
30	2500	0.40	0.45	0.46	0.48
40	2500	0.52	0.58	0.60	0.62
10	5000	0.79	0.92	0.96	0.99
20	5000	1.40	1.56	1.62	1.66
30	5000	2.00	2.18	2.25	2.32
40	5000	2.57	2.77	2.86	2.95
10	7500	1.86	2.14	2.21	2.28
20	7500	3.47	3.81	3.95	4.06
30	7500	4.88	5.26	5.40	5.56
40	7500	6.32	6.79	7.01	7.21

12.1.6 Parallel Bellman-Ford #4 with 1 to 4 threads

avgEdges	numVerts	pbf_v4_1	pbf_v4_2	pbf_v4_3	pbf_v4_4
10	1000	0.02	0.02	0.02	0.03
20	1000	0.03	0.03	0.04	0.04
30	1000	0.04	0.04	0.05	0.05
40	1000	0.05	0.05	0.06	0.06
10	2500	0.14	0.16	0.17	0.17
20	2500	0.22	0.25	0.26	0.27
30	2500	0.30	0.33	0.35	0.35
40	2500	0.37	0.41	0.43	0.45
10	5000	0.62	0.71	0.73	0.75
20	5000	1.03	1.14	1.18	1.24
30	5000	1.51	1.61	1.67	1.72
40	5000	1.88	2.02	2.08	2.15
10	7500	1.45	1.64	1.69	1.75
20	7500	2.64	2.86	2.96	3.09
30	7500	3.68	3.93	4.04	4.17
40	7500	4.80	5.08	5.24	5.39

12.2 Source code

12.2.1 main.cpp

```
#include <iostream>
#include <sstream>
#include <ctime>
#include <cstdlib>
#include <limits>
#include <thread>

#include "graph.hpp"
#include "sssp.hpp"
```

```
#include "timer.hpp"

using std::cout;
using std::cerr;
using std::endl;

template <typename T>
T StringToNumber ( const std::string &Text )
{
   std::stringstream ss(Text);
   T result;
   return ss >> result ? result : 0;
}

void run_test(Graph &g, sssp_func_t searchFunc, const char *funcName, uint32_t startNode, uint32_t endNode)
{
   Timer timer;
   timer.begin();
   uint32_t distance = searchFunc(g,0,g.nodes.size()-1);
   double runtime = timer.stop();
   cout << "result:" << funcName<< " " << runtime << " distance " << distance << endl;
}

#define RUN_TEST(function) run_test(g,function,#function,0,g.nodes.size()-1)

int main(int argc, char *argv[])
{
   unsigned int seed;
   unsigned int numVerts;
   unsigned int avgEdges;

   if (argc != 4)
   {
      cerr << "Invalid arguments" << endl;
      return -1;
   }

   seed = StringToNumber<unsigned int>(argv[1]);
   numVerts = StringToNumber<unsigned int>(argv[2]);
   avgEdges = StringToNumber<unsigned int>(argv[3]);

   srand(seed);

   cout << "Max concurrent threads " << std::thread::hardware_concurrency() << endl;
```

```
    //generate a new graph
    cout << "Creating new graph object..." << endl;
    Graph g(numVerts,avgEdges,1,10);
    cout << "Created " << g.nodes.size() << " nodes" << endl;

    if (numVerts <= 10)
    {
        g.print_dot();
    }

    //RUN_TEST(sssp_sequential_fibonacci_dijkstra);
    RUN_TEST(sssp_sequential_binary_dijkstra);
    RUN_TEST(sssp_sequential_bellman_ford);
    RUN_TEST(sssp_parallel_bellman_ford_1);
    RUN_TEST(sssp_parallel_bellman_ford_2);
    RUN_TEST(sssp_parallel_bellman_ford_3);
    RUN_TEST(sssp_parallel_bellman_ford_4);
    RUN_TEST(sssp_parallel_bellman_ford_v2_1);
    RUN_TEST(sssp_parallel_bellman_ford_v2_2);
    RUN_TEST(sssp_parallel_bellman_ford_v2_3);
    RUN_TEST(sssp_parallel_bellman_ford_v2_4);
    RUN_TEST(sssp_parallel_bellman_ford_v2p5_1);
    RUN_TEST(sssp_parallel_bellman_ford_v2p5_2);
    RUN_TEST(sssp_parallel_bellman_ford_v2p5_3);
    RUN_TEST(sssp_parallel_bellman_ford_v2p5_4);
    RUN_TEST(sssp_parallel_bellman_ford_v3_1);
    RUN_TEST(sssp_parallel_bellman_ford_v3_2);
    RUN_TEST(sssp_parallel_bellman_ford_v3_3);
    RUN_TEST(sssp_parallel_bellman_ford_v3_4);

    RUN_TEST(sssp_parallel_bellman_ford_v4_1);
    RUN_TEST(sssp_parallel_bellman_ford_v4_2);
    RUN_TEST(sssp_parallel_bellman_ford_v4_3);
    RUN_TEST(sssp_parallel_bellman_ford_v4_4);

    //uint32_t bindist = sssp_sequential_dijkstra(g,0,g.nodes.size()-1);
    //cout << "Shortest path distance binary-heap: " << bindist << endl;

    return 0;
}
```

12.2.2 thread_barrier.hpp

```
#include <atomic>
```

```
#include <condition_variable>
#include <mutex>

//credit http://stackoverflow.com/questions/8115267/writing-a-spinning-thread-barrier-using-c11-atomics

class SpinningBarrier
{
   public:
      SpinningBarrier (unsigned int threadCount) :
         threadCnt(threadCount),
         step(0),
         waitCnt(0)
      {}

      bool wait()
      {
         if(waitCnt.fetch_add(1) >= threadCnt - 1)
         {
            std::lock_guard<std::mutex> lock(mutex);
            step += 1;

            condVar.notify_all();
            waitCnt.store(0);
            return true;
         }
         else
         {
            std::unique_lock<std::mutex> lock(mutex);
            unsigned char s = step;

            condVar.wait(lock, [&]{return step == s;});
            return false;
         }
      }
   private:
      const unsigned int threadCnt;
      unsigned char step;

      std::atomic<unsigned int> waitCnt;
      std::condition_variable condVar;
      std::mutex mutex;
};
```

12.2.3 parallel_bellman_ford.cpp

```
#include "sssp.hpp"
#include "graph.hpp"
#include "thread_barrier.hpp"
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>
#include <chrono>

using std::cout;
using std::cerr;
using std::endl;
//Paralell implementation of bellman ford

typedef struct {
   uint32_t distance;
   std::mutex mtx;
} user_data_t;

#define UD(node) ((user_data_t*)(node).userData)


void _init_bellman_ford(Graph &graph, uint32_t startNode)
{
   int n;

   //initialize node userData
   for (n = 0; n < graph.nodes.size(); n++)
   {
      GraphNode_t &node = graph.nodes.at(n);
      node.userData = new user_data_t();
      UD(node)->distance = c_infinite_distance;
   }
   UD(graph.nodes.at(startNode))->distance = 0;
}

void _bellman_ford_worker(Graph *pGraph, SpinningBarrier *barrier, uint32_t rangeBegin, uint32_t rangeEnd)
{
   int n, node, e;
   uint64_t baseDistance, newDistance, oldDistance; //64 bits to avoid overflow in new distance calc
   Graph &graph = *pGraph;

   for (n = 1; n < graph.nodes.size(); n++)
```

```
    {
        //Consider the edges on nodesin only our designated range
        for (node = rangeBegin; node < rangeEnd; node++)
        {
            GraphNode_t &source = graph.nodes.at(node);

            UD(source)->mtx.lock();
            baseDistance = UD(source)->distance;
            UD(source)->mtx.unlock();


            for (e = 0; e < source.edges.size(); e++)
            {
                GraphEdge_t &edge = source.edges.at(e);
                GraphNode_t &destination = graph.nodes.at(edge.destination);
                newDistance = baseDistance;
                newDistance += edge.weight;
                //relax edge
                UD(destination)->mtx.lock();
                oldDistance = UD(destination)->distance;
                if (newDistance < oldDistance)
                {
                    UD(destination)->distance = newDistance;
                }
                UD(destination)->mtx.unlock();
            }
        }
        //All threads go at the same time
        barrier->wait();
    }
}

uint32_t _bellman_ford(Graph &graph, uint32_t startNode, uint32_t endNode, int numWorkers)
{
    uint32_t shortestDistance = c_infinite_distance;
    int n, e;

    _init_bellman_ford(graph, startNode);

    std::vector<std::thread> threads;
    SpinningBarrier barrier(numWorkers);

    //Start each thread with their own range
    uint32_t rangeBegin = 0;
```

```
    uint32_t rangeSize = graph.nodes.size() / numWorkers + 1;
    for (int i = 0; i < numWorkers; i++)
    {
        uint32_t rangeEnd = rangeBegin + rangeSize;
        if (rangeEnd > graph.nodes.size())
            rangeEnd = graph.nodes.size();
        //create our worker thread
        threads.push_back(std::thread(_bellman_ford_worker, &graph, &barrier, rangeBegin, rangeEnd));
        //cerr<< "created thread [" << rangeBegin  << ", " << rangeEnd  << ")" << endl;
        rangeBegin = rangeEnd;
    }

    for (int i = 0; i < numWorkers; i++)
    {
        threads.at(i).join();
    }

    //return the distance we found
    shortestDistance = UD(graph.nodes.at(endNode))->distance;

    //delete userData
    for (int n = 0; n < graph.nodes.size(); n++)
    {
        GraphNode_t &node = graph.nodes.at(n);
        delete (user_data_t*)node.userData;
        node.userData = 0;
    }
    return shortestDistance;
}

uint32_t sssp_parallel_bellman_ford_1(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford(graph,startNode,endNode,1);
}

uint32_t sssp_parallel_bellman_ford_2(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford(graph,startNode,endNode,2);
}

uint32_t sssp_parallel_bellman_ford_3(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford(graph,startNode,endNode,3);
}
```

```
uint32_t sssp_parallel_bellman_ford_4(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   return _bellman_ford(graph,startNode,endNode,4);
}
```

12.2.4 parallel_bellman_ford_v2.cpp

```
#include "sssp.hpp"
#include "graph.hpp"
#include "thread_barrier.hpp"
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;


typedef struct {
   uint32_t distance;
   uint32_t nextDistance;
   std::mutex mtx;
} user_data_t;

#define UD(node) ((user_data_t*)(node).userData)


void _init_bellman_ford_v2(Graph &graph, uint32_t startNode)
{
   int n;

   //initialize node userData
   for (n = 0; n < graph.nodes.size(); n++)
   {
      GraphNode_t &node = graph.nodes.at(n);
      node.userData = new user_data_t();
      UD(node)->distance = c_infinite_distance;
      UD(node)->nextDistance = c_infinite_distance;
   }
   UD(graph.nodes.at(startNode))->distance = 0;
   UD(graph.nodes.at(startNode))->nextDistance = 0;
}
```

```
void _bellman_ford_worker_v2(Graph *pGraph, SpinningBarrier *barrier, uint32_t rangeBegin, uint32_t rangeEnd)
{
   int n, node, e;
   uint64_t baseDistance, newDistance, oldDistance; //64 bits to avoid overflow in new distance calc
   Graph &graph = *pGraph;

   for (n = 1; n < graph.nodes.size(); n++)
   {
      //Consider the edges on nodesin only our designated range
      for (node = rangeBegin; node < rangeEnd; node++)
      {
         GraphNode_t &source = graph.nodes.at(node);

         baseDistance = UD(source)->distance;

         for (e = 0; e < source.edges.size(); e++)
         {
            GraphEdge_t &edge = source.edges.at(e);
            GraphNode_t &destination = graph.nodes.at(edge.destination);
            newDistance = baseDistance + edge.weight;
            //relax edge
            std::unique_lock<std::mutex> lock(UD(destination)->mtx);
            oldDistance = UD(destination)->nextDistance;
            if (newDistance < oldDistance)
            {
               UD(destination)->nextDistance = newDistance;
            }
         }
      }
      //must wait for all threads to finish relaxation step
      barrier->wait();

      //update node distances for our nodes
      for (node = rangeBegin; node < rangeEnd; node++)
      {
         GraphNode_t &ourNode = graph.nodes.at(node);
         UD(ourNode)->distance = UD(ourNode)->nextDistance;
      }

      barrier->wait();
   }
}
```

```
uint32_t _bellman_ford_v2(Graph &graph, uint32_t startNode, uint32_t endNode, int numWorkers)
{
   uint32_t shortestDistance = c_infinite_distance;
   int n, e;

   _init_bellman_ford_v2(graph, startNode);

   std::vector<std::thread> threads;
   SpinningBarrier barrier(numWorkers);

   //Start each thread with their own range
   uint32_t rangeBegin = 0;
   uint32_t rangeSize = graph.nodes.size() / numWorkers + 1;
   for (int i = 0; i < numWorkers; i++)
   {
      uint32_t rangeEnd = rangeBegin + rangeSize;
      if (rangeEnd > graph.nodes.size())
         rangeEnd = graph.nodes.size();
      //create our worker thread
      threads.push_back(std::thread(_bellman_ford_worker_v2, &graph, &barrier, rangeBegin, rangeEnd));
      rangeBegin = rangeEnd;
   }

   for (int i = 0; i < numWorkers; i++)
   {
      threads.at(i).join();
   }

   //return the distance we found
   shortestDistance = UD(graph.nodes.at(endNode))->distance;

   //delete userData
   for (int n = 0; n < graph.nodes.size(); n++)
   {
      GraphNode_t &node = graph.nodes.at(n);
      delete (user_data_t*)node.userData;
      node.userData = 0;
   }
   return shortestDistance;
}

uint32_t sssp_parallel_bellman_ford_v2_1(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   return _bellman_ford_v2(graph,startNode,endNode,1);
```

```
}

uint32_t sssp_parallel_bellman_ford_v2_2(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   return _bellman_ford_v2(graph,startNode,endNode,2);
}

uint32_t sssp_parallel_bellman_ford_v2_3(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   return _bellman_ford_v2(graph,startNode,endNode,3);
}

uint32_t sssp_parallel_bellman_ford_v2_4(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   return _bellman_ford_v2(graph,startNode,endNode,4);
}
```

12.2.5 parallel_bellman_ford_v2p5.cpp

```
#include "sssp.hpp"
#include "graph.hpp"
#include "thread_barrier.hpp"
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;


typedef struct {
   uint32_t distance;
   uint32_t nextDistance;
   std::mutex mtx;
} user_data_t;

#define UD(node) ((user_data_t*)(node).userData)


void _init_bellman_ford_v2p5(Graph &graph, uint32_t startNode)
{
   int n;
```

```
    //initialize node userData
    for (n = 0; n < graph.nodes.size(); n++)
    {
        GraphNode_t &node = graph.nodes.at(n);
        node.userData = new user_data_t();
        UD(node)->distance = c_infinite_distance;
        UD(node)->nextDistance = c_infinite_distance;
    }
    UD(graph.nodes.at(startNode))->distance = 0;
    UD(graph.nodes.at(startNode))->nextDistance = 0;
}

void _bellman_ford_worker_v2p5(Graph *pGraph, SpinningBarrier *barrier, uint32_t rangeBegin, uint32_t rangeEnd)
{
    int n, node, e;
    uint64_t baseDistance, newDistance, oldDistance; //64 bits to avoid overflow in new distance calc
    Graph &graph = *pGraph;

    for (n = 1; n < graph.nodes.size(); n++)
    {
        //Consider the edges on nodesin only our designated range
        for (node = rangeBegin; node < rangeEnd; node++)
        {
            GraphNode_t &source = graph.nodes.at(node);

            baseDistance = UD(source)->distance;

            for (e = 0; e < source.edges.size(); e++)
            {
                GraphEdge_t &edge = source.edges.at(e);
                GraphNode_t &destination = graph.nodes.at(edge.destination);
                newDistance = baseDistance + edge.weight;
                //relax edge
                // only acquire lock if we could be better than the new distance
                if (newDistance < UD(destination)->distance)
                {
                    UD(destination)->mtx.lock();
                    if (newDistance < UD(destination)->nextDistance)
                    {
                        UD(destination)->nextDistance = newDistance;
                    }
                    UD(destination)->mtx.unlock();
                }
            }
```

```
        }
        //must wait for all threads to finish relaxation step
        barrier->wait();

        //update node distances for our nodes
        for (node = rangeBegin; node < rangeEnd; node++)
        {
            GraphNode_t &ourNode = graph.nodes.at(node);
            UD(ourNode)->distance = UD(ourNode)->nextDistance;
        }

        barrier->wait();
    }
}

uint32_t _bellman_ford_v2p5(Graph &graph, uint32_t startNode, uint32_t endNode, int numWorkers)
{
    uint32_t shortestDistance = c_infinite_distance;
    int n, e;

    _init_bellman_ford_v2p5(graph, startNode);

    std::vector<std::thread> threads;
    SpinningBarrier barrier(numWorkers);

    //Start each thread with their own range
    uint32_t rangeBegin = 0;
    uint32_t rangeSize = graph.nodes.size() / numWorkers + 1;
    for (int i = 0; i < numWorkers; i++)
    {
        uint32_t rangeEnd = rangeBegin + rangeSize;
        if (rangeEnd > graph.nodes.size())
            rangeEnd = graph.nodes.size();
        //create our worker thread
        threads.push_back(std::thread(_bellman_ford_worker_v2p5, &graph, &barrier, rangeBegin, rangeEnd));
        rangeBegin = rangeEnd;
    }

    for (int i = 0; i < numWorkers; i++)
    {
        threads.at(i).join();
    }

    //return the distance we found
```

```
    shortestDistance = UD(graph.nodes.at(endNode))->distance;

    //delete userData
    for (int n = 0; n < graph.nodes.size(); n++)
    {
        GraphNode_t &node = graph.nodes.at(n);
        delete (user_data_t*)node.userData;
        node.userData = 0;
    }
    return shortestDistance;
}

uint32_t sssp_parallel_bellman_ford_v2p5_1(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v2p5(graph,startNode,endNode,1);
}

uint32_t sssp_parallel_bellman_ford_v2p5_2(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v2p5(graph,startNode,endNode,2);
}

uint32_t sssp_parallel_bellman_ford_v2p5_3(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v2p5(graph,startNode,endNode,3);
}

uint32_t sssp_parallel_bellman_ford_v2p5_4(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v2p5(graph,startNode,endNode,4);
}
```

12.2.6 parallel_bellman_ford_v3.cpp

```
#include "sssp.hpp"
#include "graph.hpp"
#include "thread_barrier.hpp"
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>
#include <deque>
#include <utility>

using std::cout;
```

```
using std::cerr;
using std::endl;


typedef struct {
   uint32_t distance;
   uint32_t nextDistance;
   std::mutex mtx;
} user_data_t;

#define UD(node) ((user_data_t*)(node).userData)


void _init_bellman_ford_v3(Graph &graph, uint32_t startNode)
{
   int n;

   //initialize node userData
   for (n = 0; n < graph.nodes.size(); n++)
   {
      GraphNode_t &node = graph.nodes.at(n);
      node.userData = new user_data_t();
      UD(node)->distance = c_infinite_distance;
      UD(node)->nextDistance = c_infinite_distance;
   }
   UD(graph.nodes.at(startNode))->distance = 0;
   UD(graph.nodes.at(startNode))->nextDistance = 0;
}

void _bellman_ford_worker_v3(Graph *pGraph, SpinningBarrier *barrier, uint32_t rangeBegin, uint32_t rangeEnd)
{
   int n, node, e;
   uint64_t baseDistance, newDistance, oldDistance; //64 bits to avoid overflow in new distance calc
   Graph &graph = *pGraph;

   for (n = 1; n < graph.nodes.size(); n++)
   {
      std::deque<std::pair<uint64_t,user_data_t*>> relaxReqQ;

      for (node = rangeBegin; node < rangeEnd; node++)
      {
         GraphNode_t &source = graph.nodes.at(node);

         baseDistance = UD(source)->distance;
```

```
    for (e = 0; e < source.edges.size(); e++)
    {
        GraphEdge_t &edge = source.edges.at(e);
        GraphNode_t &destination = graph.nodes.at(edge.destination);
        newDistance = baseDistance + edge.weight;
        //Make note to relax this edge, only if it could have an effect
        if (newDistance < UD(destination)->distance)
            relaxReqQ.push_back(std::make_pair(newDistance, UD(destination)));
    }
}

//empty out our relax request Q
while (relaxReqQ.size())
{
    std::pair<uint64_t, user_data_t*> request = relaxReqQ.front();
    relaxReqQ.pop_front(); //remove this request from the Q

    //try relax
    user_data_t *pDestData = request.second;
    if (pDestData->mtx.try_lock())
    {
        uint64_t newDistance = request.first;
        if (newDistance < pDestData->nextDistance)
        {
            pDestData->nextDistance = newDistance;
        }
        pDestData->mtx.unlock();
    }
    else
    {
        //Didn't get the lock, try again later
        relaxReqQ.push_back(request);
    }
}

//must wait for all threads to finish relaxation step
barrier->wait();

//update node distances for our nodes
for (node = rangeBegin; node < rangeEnd; node++)
{
    GraphNode_t &ourNode = graph.nodes.at(node);
    UD(ourNode)->distance = UD(ourNode)->nextDistance;
```

```
        }

        barrier->wait();
    }
}

uint32_t _bellman_ford_v3(Graph &graph, uint32_t startNode, uint32_t endNode, int numWorkers)
{
    uint32_t shortestDistance = c_infinite_distance;
    int n, e;

    _init_bellman_ford_v3(graph, startNode);

    std::vector<std::thread> threads;
    SpinningBarrier barrier(numWorkers);

    //Start each thread with their own range
    uint32_t rangeBegin = 0;
    uint32_t rangeSize = graph.nodes.size() / numWorkers + 1;
    for (int i = 0; i < numWorkers; i++)
    {
        uint32_t rangeEnd = rangeBegin + rangeSize;
        if (rangeEnd > graph.nodes.size())
            rangeEnd = graph.nodes.size();
        //create our worker thread
        threads.push_back(std::thread(_bellman_ford_worker_v3, &graph, &barrier, rangeBegin, rangeEnd));
        rangeBegin = rangeEnd;
    }

    for (int i = 0; i < numWorkers; i++)
    {
        threads.at(i).join();
    }

    //return the distance we found
    shortestDistance = UD(graph.nodes.at(endNode))->distance;

    //delete userData
    for (int n = 0; n < graph.nodes.size(); n++)
    {
        GraphNode_t &node = graph.nodes.at(n);
        delete (user_data_t*)node.userData;
        node.userData = 0;
    }
```

```
    return shortestDistance;
}

uint32_t sssp_parallel_bellman_ford_v3_1(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v3(graph,startNode,endNode,1);
}

uint32_t sssp_parallel_bellman_ford_v3_2(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v3(graph,startNode,endNode,2);
}

uint32_t sssp_parallel_bellman_ford_v3_3(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v3(graph,startNode,endNode,3);
}

uint32_t sssp_parallel_bellman_ford_v3_4(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v3(graph,startNode,endNode,4);
}
```

12.2.7 parallel_bellman_ford_v4.cpp

```
#include "sssp.hpp"
#include "graph.hpp"
#include "thread_barrier.hpp"
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>
#include <utility>

using std::cout;
using std::cerr;
using std::endl;


typedef struct {
    uint32_t distance;
    uint32_t nextDistance[4];
    std::mutex mtx;
} user_data_t;
```

```
#define UD(node) ((user_data_t*)(node).userData)


void _init_bellman_ford_v4(Graph &graph, uint32_t startNode)
{
   int n;

   //initialize node userData
   for (n = 0; n < graph.nodes.size(); n++)
   {
      GraphNode_t &node = graph.nodes.at(n);
      node.userData = new user_data_t();
      UD(node)->distance = c_infinite_distance;
      UD(node)->nextDistance[0] = c_infinite_distance;
      UD(node)->nextDistance[1] = c_infinite_distance;
      UD(node)->nextDistance[2] = c_infinite_distance;
      UD(node)->nextDistance[3] = c_infinite_distance;
   }
   UD(graph.nodes.at(startNode))->distance = 0;
   UD(graph.nodes.at(startNode))->nextDistance[0] = c_infinite_distance;
   UD(graph.nodes.at(startNode))->nextDistance[1] = c_infinite_distance;
   UD(graph.nodes.at(startNode))->nextDistance[2] = c_infinite_distance;
   UD(graph.nodes.at(startNode))->nextDistance[3] = c_infinite_distance;
}

void _bellman_ford_worker_v4(Graph *pGraph, SpinningBarrier *barrier, uint32_t rangeBegin, uint32_t rangeEnd, int worker_id)
{
   int n, node, e, d;
   uint32_t minDistance;
   uint32_t *distances;
   uint64_t baseDistance, newDistance, oldDistance; //64 bits to avoid overflow in new distance calc
   Graph &graph = *pGraph;

   for (n = 1; n < graph.nodes.size(); n++)
   {
      for (node = rangeBegin; node < rangeEnd; node++)
      {
         GraphNode_t &source = graph.nodes.at(node);

         baseDistance = UD(source)->distance;

         for (e = 0; e < source.edges.size(); e++)
         {
            GraphEdge_t &edge = source.edges.at(e);
```

```
            GraphNode_t &destination = graph.nodes.at(edge.destination);
            newDistance = baseDistance + edge.weight;
            //Relax if this is smaller than the smallest distance we know
            if (newDistance < UD(destination)->nextDistance[worker_id])
            {
               UD(destination)->nextDistance[worker_id] = newDistance;
            }
         }
      }

      //must wait for all threads to finish relaxation step
      barrier->wait();

      //update node distances for our nodes
      for (node = rangeBegin; node < rangeEnd; node++)
      {
         GraphNode_t &ourNode = graph.nodes.at(node);
         //Find minumum of four
         distances = UD(ourNode)->nextDistance;
         minDistance = distances[0];
         if (distances[1] < minDistance) minDistance = distances[1];
         if (distances[2] < minDistance) minDistance = distances[2];
         if (distances[3] < minDistance) minDistance = distances[3];

         //write the new smallest distance
         UD(ourNode)->distance = minDistance;

         distances[0] = minDistance;
         distances[1] = minDistance;
         distances[2] = minDistance;
         distances[3] = minDistance;
      }

      barrier->wait();
   }
}

uint32_t _bellman_ford_v4(Graph &graph, uint32_t startNode, uint32_t endNode, int numWorkers)
{
   uint32_t shortestDistance = c_infinite_distance;
   int n, e;

   _init_bellman_ford_v4(graph, startNode);
```

```
    std::vector<std::thread> threads;
    SpinningBarrier barrier(numWorkers);

    //Start each thread with their own range
    uint32_t rangeBegin = 0;
    uint32_t rangeSize = graph.nodes.size() / numWorkers + 1;
    for (int i = 0; i < numWorkers; i++)
    {
        uint32_t rangeEnd = rangeBegin + rangeSize;
        if (rangeEnd > graph.nodes.size())
            rangeEnd = graph.nodes.size();
        //create our worker thread
        threads.push_back(std::thread(_bellman_ford_worker_v4, &graph, &barrier, rangeBegin, rangeEnd, i));
        rangeBegin = rangeEnd;
    }

    for (int i = 0; i < numWorkers; i++)
    {
        threads.at(i).join();
    }

    //return the distance we found
    shortestDistance = UD(graph.nodes.at(endNode))->distance;

    //delete userData
    for (int n = 0; n < graph.nodes.size(); n++)
    {
        GraphNode_t &node = graph.nodes.at(n);
        delete (user_data_t*)node.userData;
        node.userData = 0;
    }
    return shortestDistance;
}

uint32_t sssp_parallel_bellman_ford_v4_1(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v4(graph,startNode,endNode,1);
}

uint32_t sssp_parallel_bellman_ford_v4_2(Graph &graph, uint32_t startNode, uint32_t endNode)
{
    return _bellman_ford_v4(graph,startNode,endNode,2);
}
```

```
uint32_t sssp_parallel_bellman_ford_v4_3(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   return _bellman_ford_v4(graph,startNode,endNode,3);
}

uint32_t sssp_parallel_bellman_ford_v4_4(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   return _bellman_ford_v4(graph,startNode,endNode,4);
}
```

12.2.8 sequential_bellman_ford.cpp

```
#include "sssp.hpp"
#include "graph.hpp"
#include <iostream>
using std::cout;
using std::endl;

typedef struct {
   uint32_t distance;
} user_data_t;

#define UD(node) ((user_data_t*)(node).userData)

/**
 * Run bellman-ford sequentially
 */
uint32_t sssp_sequential_bellman_ford(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   int n, e;
   uint32_t shortestDistance = c_infinite_distance;
   uint64_t newDistance, oldDistance; //64 bits to avoid overflow in new distance calc

   //initialize node userData
   for (n = 0; n < graph.nodes.size(); n++)
   {
      GraphNode_t &node = graph.nodes.at(n);
      node.userData = new user_data_t();
      UD(node)->distance = c_infinite_distance;
   }
   UD(graph.nodes.at(startNode))->distance = 0;

   //1 to |V|-1
   for (n = 1; n < graph.nodes.size(); n++)
   {
```

```
        //all edges
        for (e = 0 ; e < graph.edges.size(); e++)
        {
            GraphEdge_t &edge = graph.edges.at(e);
            GraphNode_t &source = graph.nodes.at(edge.source);
            GraphNode_t &destination = graph.nodes.at(edge.destination);
            //relax edge
            newDistance = UD(source)->distance;
            newDistance += edge.weight;
            oldDistance = UD(destination)->distance;
            if (newDistance < oldDistance)
            {
                UD(destination)->distance = newDistance;
            }
        }
    }

    //return the distance we found
    shortestDistance = UD(graph.nodes.at(endNode))->distance;

    //delete userData
    for (n = 0; n < graph.nodes.size(); n++)
    {
        GraphNode_t &node = graph.nodes.at(n);
        delete (user_data_t*)node.userData;
        node.userData = 0;
    }
    return shortestDistance;
}
```

12.2.9 sequential_binary_dijkstra.cpp

```
#include <stdint.h>
#include "sssp.hpp"
#include <iostream>
#include <cstdlib>
#include <set>
#include <utility>

using std::cout;
using std::endl;
using std::cerr;

class NodeWrapper;
```

```
typedef struct {
   uint32_t distance;
} user_data_t;

#define UD(node) ((user_data_t*)(node).userData)


uint32_t sssp_sequential_binary_dijkstra(Graph &graph, uint32_t startNode, uint32_t endNode)
{
   uint32_t n,e,alt;
   uint32_t shortestDistance = c_infinite_distance;
   std::set<std::pair<uint32_t, GraphNode_t*>> PQ;

   //initialize node userData and fill priority queue
   for (n = 0; n < graph.nodes.size(); n++)
   {
      GraphNode_t &node = graph.nodes.at(n);
      node.userData = new user_data_t();
      UD(node)->distance = c_infinite_distance;
   }
   GraphNode_t &start = graph.nodes.at(startNode);
   UD(start)->distance = 0;

   PQ.insert(std::make_pair(UD(start)->distance, &start));

   while (!PQ.empty())
   {
      //Get next closest node to source
      GraphNode_t &node = *(PQ.begin()->second);
      PQ.erase(PQ.begin()); //remove it from the queue

      //found the shortest path
      if (node.id == endNode)
      {
         shortestDistance = UD(node)->distance;
         goto sssp_seq_bin_dijk_cleanup;
      }

      for (e = 0; e < node.edges.size(); e++)
      {
         GraphEdge_t &edge = node.edges.at(e);
         GraphNode_t &neighbor = graph.nodes.at(edge.destination);

         //Relax neighbor weights if the path is
```

```
            //shorter through this node
            alt = UD(node)->distance + edge.weight;
            if (alt < UD(neighbor)->distance)
            {
               PQ.erase(std::make_pair(UD(neighbor)->distance,&neighbor));
               UD(neighbor)->distance = alt;
               PQ.insert(std::make_pair(UD(neighbor)->distance,&neighbor));
            }
         }
      }
sssp_seq_bin_dijk_cleanup:
      //delete userData
      for (n = 0; n < graph.nodes.size(); n++)
      {
         GraphNode_t &node = graph.nodes.at(n);
         delete (user_data_t*)node.userData;
         node.userData = 0;
      }
      return shortestDistance;
}
```

12.2.10 graph.hpp

```
#ifndef graph_hpp
#define graph_hpp

/*
 * Graph class
 * Shane Loretz 2014
 */

#include <stdint.h>
#include <vector>

using std::vector;

typedef struct{
   uint32_t source;
   uint32_t destination; //node being pointed at
   uint32_t weight;
} GraphEdge_t;

typedef struct{
   uint32_t id;
   vector<GraphEdge_t> edges;
```

```
    void *userData;
} GraphNode_t;


class Graph {
    public:
        //Constructor generates a new graph
        //Create a dag
        Graph(uint32_t numNodes, uint32_t minEdges, uint32_t maxEdges, uint32_t minWeight, uint32_t maxWeight);
        //create a weakly connected directed graph
        Graph(uint32_t numNodes, uint32_t avgEdges, uint32_t minWeight, uint32_t maxWeight);
        ~Graph();

        void print_dot();

        //No secrets here, make everything public
        vector<GraphNode_t> nodes;
        vector<GraphEdge_t> edges;
};

#endif
```

12.2.11 graph.cpp

```
#include "graph.hpp"

#include <cstdlib>
#include <iostream>
using std::cout;
using std::endl;

Graph::Graph(uint32_t numNodes, uint32_t avgEdges, uint32_t minWeight, uint32_t maxWeight)
{
    GraphNode_t firstNode;
    uint32_t numEdges = avgEdges * numNodes / 2;
    uint64_t n,e,e2;
    uint64_t src,dest;
    GraphNode_t *pSource;

    //Generate a weakly connected directed graph

    nodes.reserve(numNodes);
    edges.reserve(numEdges);

    firstNode.id = 0;
```

```
nodes.push_back(firstNode);
for (n = 1; n < numNodes; n++)
{
   //Generate a node with an edge with previous node
   //Direction of this edge is random
   GraphNode_t node;
   GraphEdge_t edge;
   edge.weight = rand() % (maxWeight-minWeight) + minWeight;

   if (rand() % 2)
   {
      //From this node to last
      edge.source = n;
      edge.destination = n-1;
      node.edges.push_back(edge);
      edges.push_back(edge);
   }
   else
   {
      //from last node to this
      edge.source = n-1;
      edge.destination = n;
      nodes.back().edges.push_back(edge);
      edges.push_back(edge);
   }
   node.id = n;
   nodes.push_back(node);
}

//Variable holds number of edges left to create
numEdges -= numNodes - 1;

//graph is already weakly connected
//now add random edges
for (e = 0; e < numEdges; e++)
{
   GraphEdge_t edge;

   do
   {
      src = rand() % numNodes;
      dest = rand() % numNodes;
      pSource = &nodes.at(src);
      //make sure edge does not exist already
```

```
            for (e2 = 0; e2 < pSource->edges.size(); e2++)
            {
               if (pSource->edges.at(e2).destination == dest)
               {
                  //Edge exists already, skip
                  src = dest;
                  break;
               }
            }
         } while (src == dest);

         edge.source = src;
         edge.destination = dest;
         edge.weight = rand() % (maxWeight-minWeight) + minWeight;
         pSource->edges.push_back(edge);
         edges.push_back(edge);
      }
   }

   void Graph::print_dot()
   {
      cout << "digraph gen_graph {" << endl;
      for (int e = 0; e < edges.size(); e++)
      {
         GraphEdge_t &edge = edges.at(e);
         cout << edge.source << " -> " << edge.destination << "[label=" << edge.weight << "]; ";
      }
      cout << "}" << endl;
   }


   Graph::~Graph()
   {
      //nothing to do...
   }
```

12.2.12 timer.hpp

```
#include <ctime>

class Timer
{
   public:
      Timer()
      {
```

```
        _start = 0;
        _stop = 0;
        }
        ~Timer()
        {
        }

        void begin()
        {
            _start = clock();
        }

        double stop()
        {
            _stop = clock();
            if (_stop < _start)
                return -1;
            double ret = _stop-_start;
            ret = ret / CLOCKS_PER_SEC;
            return ret;
        }

    private:
        clock_t _start;
        clock_t _stop;
};
```

12.2.13 run_tests.py

```
# Python script to run a bunch of tests and collect results

import random
import subprocess
import re
import code
import sys
import csv

def run_test(seed, numVerts, avgEdges):
    results = dict()
    results["seed"] = seed
    results["numVerts"] = numVerts
    results["avgEdges"] = avgEdges
    distances = set()
    output = ""
```

```
    #Run the program
    try:
        output = subprocess.check_output(["./test_prog", str(seed), str(numVerts), str(avgEdges)])
    except subprocess.CalledProcessError:
        pass

    #Look for results in output
    matches = re.findall(r"result:([a-zA-Z_0-9]+) ([0-9]+[.]?[0-9]*) distance ([0-9]+)",output)
    if not matches:
        print "Did not find results"
        print output
        return None

    for func,time,distance in matches:
        results[func] = time;
        distances.add(distance);

    if len(distances) != 1:
        print "Distance disagreement"
        print output
        sys.exit()
        return None

    results["distance"] = distances.pop()
    return results


if __name__ == '__main__':

    output_file = open(sys.argv[1],"w");
    dictWriter = None

    numNodes = [1000, 2500,5000,7500]
    avgEdges = [10, 20, 30, 40]


    for verts in numNodes:
        for edges in avgEdges:
            for runNum in range (0,3):
                result = None
                while not result:
                    seed = random.randint(0,2000000000)
                    print "Batch", seed, verts, edges
```

```
    result = run_test(seed,verts,edges)

result["runNum"] = runNum
#Write results to CSV
if not dictWriter:
    dictWriter = csv.DictWriter(output_file, result.keys())
    dictWriter.writeheader()
dictWriter.writerow(result)
print result
```