Homework 1 - Design Writeup

Patrick Collins

January 14, 2014

1 graph.c

File-level constants:

- Error codes:
  - `INVALID_INPUT_ERROR`
    Used for graphs with invalid distances. Violations of Graph invariant #1.
  - `MALFORMED_INPUT_ERROR`
    Used for files with the wrong number of lines, etc. Violations of Graph invariant #2.
- Values:
  - `INF_DIST 1000`
  - `MAX_DIST 10000000`

1.1 Graph

1.1.1 Description

The implementation here closely follows the Wikipedia description of the algorithm linked in the assignment. The description is omitted.

1.1.2 Implementation Overview

Properties:

- `int verticies`
- `int** matrix`

Methods: (Following the Python sytle of a leading underscore for private methods, providing some degree of information hiding without unnecessarily complicating testing. This convention is followed for the remainder of the document.)

- `int _get_element(Graph* self, int row, int column)`
- `void _set_element(Graph* self, int row, int column, int val)`
- `void _update_dist(Graph* self, int i, int j, int k)`
- `void _update_dists(Graph* self, int k)`
- `bool graph_eq(Graph* g1, Graph* g2)`
- `int get_verticies(Graph* self)`
- `void solve_graph(Graph* self)`
- `void print_graph(Graph* self)`

Constructors:

- `Graph* graph_from_file(file *f)`

Destructors:

- `void free_graph(Graph* g)`

Invariants:

1. For each element `e` $\in$ `matrix`, `e < MAX_DIST || e == INF_DIST`
2. `rows == columns == verticies`

2 tsgraph.c

2.1 _DistArgs

Supporting module for `TSGraph`.
Properties:

- `TSGraph* tsgraph`
- `int i_start`
- `int i_end`
- `int k`
- `int tid`

Constructor:

- `_DistArgs* _make_dist_args(TSGraph* tsgraph, int i_start, int i_end, int k, int tid)`

Destructor:

- `_free_dist_args(_DistArgs* args)`

2.2 TSGraph

2.2.1 Implementation Overview

Properties:

- `Graph* graph`
- `int num_threads`
- `int k`
- `volatile bool[] working_threads`

Methods:

- `Graph* get_graph(TSGraph* self)`
- `TSGraph* _update_dist_p(TSGraph* self, int i_start, int i_end, int k, int tid)`
- `void* _update_dist_p_wrapper(_DistArgs* args)`
- `TSGraph* _update_dists_p(TSGraph* self)`
- `bool still_working(TSGraph* self)`
- `int get_k(TSGraph* self)`

Constructor:

- `TSGraph* tsgraph_from_graph(Graph* graph, int num_threads)`

Destructor:

- `void free_tsgraph(TSGraph* self)`

Invariants:

1. No two threads will call `_update_dist_p` at the same time unless the argument `k` is equal in each.

2.2.2 Description

Noting the description of the algorithm in terms of its recursive formula (as listed on Wikipedia):

$$\begin{aligned}\texttt{shortestPath}(i, j, 0) &= w(i, j)\\ \texttt{shortestPath}(i, j, k+1) &= \texttt{min}(\texttt{shortestPath}(i, j, k), \texttt{shortestPath}(i, k+1, k),\\ &\qquad \texttt{shortestPath}(k+1, j, k))\end{aligned}$$

it is evident that successive evaluations of `shortestPath` are independent for values of i and j but not k. Therefore, the `ThreadsafeGraph` will aim to guarantee that every thread is evaluating `shortestPath` for different values of i and j but the same value of k.

This will be implemented by "chunking" out the work on i, the outer loop index, and using a `bool` array, `working_threads`, to signal when all threads have completed work. Specifically, for n threads, `working_threads` will be initialized as an n-element array containing only `true`. When the ith thread completes its work, it will set the ith element of `working_threads` to `false`. When every element of `working_threads` becomes `false`, `self->k` will be incremented, every element of `working_threads` will be reset to `true`, the next loop iteration will continue, until `self->k == self->graph->vertices`.

Based on an implementation of `_update_dist` which will take constant time, `_update_dist_p` will take time that is linear on the difference between `i_min` and `i_max`. For each thread except for the last,

`i_max - i_min == self->graph->verticies / self->num_threads`.

The last thread will be within `self->graph->vertices % self->num_threads` of this value. Therefore, work will be divided equally between threads, providing for a good load imbalance.

Using an array of `bool` values for synchronization will ensure that, for every i and j, `shortestPath`$(i, j, k+1)$ is evaluated after `shortestPath`(i, j, k), preventing a data race. Additionally, it avoids starting and joining `self->num_threads` threads per loop iteration, lowering overhead.

3 Test Plan

3.1 Graph

Hypothesis: Graphs are instantiated correctly.
Tests:

- Every graph is equal to itself.
- The value of each vertex of a newly-initialized graph is equal to the corresponding value in the file it is contained in.
- `INVALID_INPUT_ERROR` is raised for graphs with impossible distances.
- `MALFORMED_INPUT_ERROR` is raised for graphs that do not meet format specifications.

Hypothesis: Graphs are solved correctly.
Tests:

- Nodes with no outgoing edges remain `INF_DIST` from all other nodes after a graph is solved.
- 5 nodes in a horizontal line a unit distance away from each other, bidirectionally, are correctly reported as being 0, 1, 2, 3 and 4 units away from the rightmost node, and vice-versa for the leftmost.
- 5 nodes in a vertical line a unit distance away from each other, bidirectionally, are correctly reported as being 0, 1, 2, 3 and 4 units away from the bottommost node, and vice-versa for the topmost.
- 5 nodes in a horizontal line a unit distance away from each other, bidirectionally, with a 2-unit long path from the leftmost to the rightmost node, are correctly reported as being 0, 1, 2, 3, 3 and 2 units away from each other.
- An undirected graph with two nodes `MAX_DIST - 1` away from a third undergoes no change when solved.

3.2 TSGraph

Hypothesis: No two threads will evaluate `shortestPath`(i, j, k) for two different values of k at the same time. Furthermore, k is evaluated for each i, j exactly once.
Tests:

- Given a `TSGraph*` object `tsg` with `tsg->num_threads = n` and `tsg->k = 0`, for $i = 0$ to $n - 2$, evaluate `_update_dist_p(tsg, 0, 0, i)`. Then:
  - `still_working(tsg) == true`.

  - `get_k(tsg) == 0`.

- After, evaluate `_update_dist_p(tsg, 0, 0, n - 1)`. Then:
  - `still_working(tsg) == true`.
  - `get_k(tsg) == 1`.
- Repeat until `k == tsg->graph->verticies`, which will occur in `k - 1` steps.

Hypothesis: `TSGraph` objects are solved correctly.
Tests:

- Solving the same testcases as listed under the Graph tests yields the same graph as in that case, for 1, 2, 3, 4 and 5 threads.