CuWISP

A Massively Parallel Implementation of WISP for Nvidia GPUs

Andy Stokely

September 2021

1 Theory

- CuWISP uses a novel variation of Hedetniemi's algorithm to find suboptimal paths.
- A brief overview of Hedetniemi's algorithm is provided below.

1.1 Hedetniemi's Algorithm

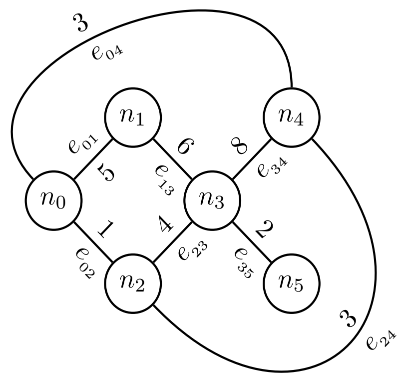
1. Let A be the adjacency matrix representation of a non-negative weighted undirected graph G (Figure 1).

$$a_{ij} = \begin{cases} 0 & i = j \\ w_{ij} & i \neq j \text{ if there is an edge connecting node i } (n_i) \text{ and node j } (n_j). \\ \infty & i \neq j \text{ if there is not an edge connecting } n_i \text{ and } n_j. \end{cases} \tag{1}$$

2. C^n is an all pairs shortest paths matrix for G.
   - Denoted as $C^n = A \dashv\vdash A$, where A = C^{n-1}.
   - Each path has at most n edges.

$$c_{ij}^{(n)} = \min(a_{i0} + a_{0j},\ a_{i1} + a_{1j},\ a_{i2} + a_{2j},\ \cdots,\ a_{im} + a_{mj}),$$
$$\text{where } C \text{ is a m} \times \text{m matrix and } A = C^{n-1} \text{ (Figure 2).} \tag{2}$$

3. If the longest (in terms of edges) shortest path has n edges, C^n is a "complete" all pairs shortest paths matrix for G.



$$A = \begin{pmatrix} 0 & 5 & 1 & \infty & 3 & \infty \\ 5 & 0 & \infty & 6 & \infty & \infty \\ 1 & \infty & 0 & 4 & 3 & \infty \\ \infty & 6 & 4 & 0 & 8 & 2 \\ 3 & \infty & 3 & 8 & 0 & \infty \\ \infty & \infty & \infty & 2 & \infty & 0 \end{pmatrix}$$

Figure 1: Graph G and it's adjacency matrix representation A. The weight of the edge composed of node i and node j is the matrix element $a_{ij} \in A$.

$$A = \begin{pmatrix} \mathbf{0} & \mathbf{5} & \mathbf{1} & \boldsymbol{\infty} & \mathbf{3} & \boldsymbol{\infty} \\ 5 & 0 & \infty & \mathbf{6} & \infty & \infty \\ 1 & \infty & 0 & \mathbf{4} & 3 & \infty \\ \infty & 6 & 4 & \mathbf{0} & 8 & 2 \\ 3 & \infty & 3 & \mathbf{8} & 0 & \infty \\ \infty & \infty & \infty & \mathbf{2} & \infty & 0 \end{pmatrix}$$

$$c_{03}^{(2)} = \min\left(\begin{bmatrix} 0 \\ 5 \\ 1 \\ \infty \\ 3 \\ \infty \end{bmatrix} + \begin{bmatrix} \infty \\ 6 \\ 4 \\ 0 \\ 8 \\ 2 \end{bmatrix} \right) = \min\left(\begin{bmatrix} \infty \\ 11 \\ 5 \\ \infty \\ 11 \\ \infty \end{bmatrix} \right) = 5$$

Figure 2: The shortest path from node 0 to node 3 with two or fewer edges has a length of 5.

4. If $C^n = C^{n-1}$, the longest shortest path for G has n-1 edges.
5. Equation (2) is embarrassingly parallel.

- Each of the sums can be computed in parallel.

6. In the worst case, C^n has to be computed m - 1 times given G has m edges.
7. However, the calculation can be terminated once $C^n = C^{n-1}$.
8. The CUDA version of Hedetniemi's algorithm is roughly 100× faster than NetworkX's python implementation of Bellman Ford's algorithm.
9. The single shortest path between n_i and n_j (SSP_{ij}) can be recovered from C^n as follows.
   (a) Define the source node as n_i.
   (b) Define the sink node as n_j.
   (c) The length of SSP_{ij} is equal to matrix element $c_{ij}^{(n)} \in C^n$.
   (d) n_{k1} is the node in SSP_{ij} that shares an edge with n_j, where k_1 is the column and row index of A and C^n respectively, that satisfies the condition
   $$c_{ij}^{(n)} = c_{k_1 j}^{(n)} + a_{ik_1}. \tag{3}$$
   (e) k_1 is appended to a list (L_n) that stores SSP_{ij}'s node indices.
   (f) a_{ik_1} is appended to a list (L_w) that stores SSP_{ij}'s edge weights.
   (g) The index (k_2) of the node in SSP_{ij} that shares an edge with n_{k_1} is now computed.
   $$c_{k_1 j}^{(n)} = c_{k_2 j}^{(n)} + a_{ik_2}. \tag{4}$$
   (h) k_2 is appended to L_n.
   (i) a_{ik_2} is appended to L_w.
   (j) Once $\|L_w\|_1 = c_{ij}^{(n)}$[1], the procedure is terminated.
   (k) L_n is an ordered list of all node indices in SSP_{ij}.
   (l) This step can be partially parallelized using a parallel reduction.

1.2 Hedetniemi Suboptimal Paths Finding (HSPF) Algorithm

1. Hedetniemi's algorithm can be used to find "important" suboptimal paths between a source and sink node in a graph.
2. The user defines the "importance" criteria by setting the maximum number of suboptimal paths (sp_{max}) they want the algorithm to find.
3. The HSPF algorithm proceeds as follows.

[1] $\|L_w\|_1$ is the euclidean 1-norm, or sum of L_w.

- The single shortest path (ssp) between the source node (n_{src}) and sink node (n_{sink}) is calculated.
- The edges in the ssp are added to a deque (D).
  - The way the edges are added to D depends on which path finding round is being run.
    * **Round 0**: Edges are appended in order to the end of D and popped from the end.

      $D = \textbf{deque}([E_a,\ E_b,\ E_c])$
      $SSP = \big(E_d,\ E_e,\ E_f,\ E_g,\ E_h\big)$
      append: $D = \textbf{deque}([E_a,\ E_b,\ E_c,\ E_d,\ E_e,\ E_f,\ E_g,\ E_h])$
      pop: $D = \textbf{deque}([E_a,\ E_b,\ E_c,\ E_d,\ E_e,\ E_f,\ E_g,\ \cancel{E_h}])$
      $\longrightarrow E_h = (n_x, n_y) \longrightarrow c_{xy}^{(n)} = \infty, \quad c_{xy}^{(n)} \in C^n$

    * **Round 1**: Edges are appended in order to the middle of D and popped from the middle.

      $D = \textbf{deque}([E_a,\ E_b,\ E_c])$
      $SSP = \big(E_d,\ E_e,\ E_f,\ E_g,\ E_h\big)$
      append: $D = \textbf{deque}([E_a,\ E_d,\ E_e,\ E_f,\ E_g,\ E_h,\ E_b,\ E_c])$
      pop: $D = \textbf{deque}([E_a,\ E_d,\ E_e,\ \cancel{E_f},\ E_g,\ E_h,\ E_b,\ E_c])$
      $\longrightarrow E_f = (n_x, n_y) \longrightarrow c_{xy}^{(n)} = \infty, \quad c_{xy}^{(n)} \in C^n$

    * **Round 2**: Edges are appended in order to the end of D and popped from the beginning.

      $D = \textbf{deque}([E_a,\ E_b,\ E_c])$
      $SSP = \big(E_d,\ E_e,\ E_f,\ E_g,\ E_h\big)$
      append: $D = \textbf{deque}([E_a,\ E_b,\ E_c,\ E_d,\ E_e,\ E_f,\ E_g,\ E_h])$
      pop: $D = \textbf{deque}(\cancel{[E_a},\ E_b,\ E_c,\ E_d,\ E_e,\ E_f,\ E_g,\ E_h])$
      $\longrightarrow E_a = (n_x, n_y) \longrightarrow c_{xy}^{(n)} = \infty, \quad c_{xy}^{(n)} \in C^n$

    * **Round 3**: Edges are appended in order to the middle of D and popped from the beginning.

      $D = \textbf{deque}([E_a,\ E_b,\ E_c])$
      $SSP = \big(E_d,\ E_e,\ E_f,\ E_g,\ E_h\big)$
      append: $D = \textbf{deque}([E_a,\ E_d,\ E_e,\ E_f,\ E_g,\ E_h,\ E_b,\ E_c])$
      pop: $D = \textbf{deque}(\cancel{[E_a},\ E_d,\ E_e,\ E_f,\ E_g,\ E_h,\ E_b,\ E_c])$
      $\longrightarrow E_a = (n_x, n_y) \longrightarrow c_{xy}^{(n)} = \infty, \quad c_{xy}^{(n)} \in C^n$

- * **Round 4**: Edges are appended in order to the middle of D and popped from the end.

$$D = \textbf{deque}([E_a,\ E_b,\ E_c])$$
$$SSP = \big(E_d,\ E_e,\ E_f,\ E_g,\ E_h\big)$$
$$\textbf{append: } D = \textbf{deque}([E_a,\ E_d,\ E_e,\ E_f,\ E_g,\ E_h,\ E_b,\ E_c])$$
$$\textbf{pop: } D = \textbf{deque}([E_a,\ E_d,\ E_e,\ E_f,\ E_g,\ E_h,\ E_b,\ \cancel{E_c}])$$
$$\longrightarrow E_c = (n_x, n_y) \longrightarrow c_{xy}^{(n)} = \infty,\quad c_{xy}^{(n)} \in C^n$$

- The ssp is added to a list that stores suboptimal paths (L_{sp}).
- Each edge is a tuple of it's node indices.
- An edge is either popped from the beginning, middle, or end of D based on the path finding round.
- The matrix element in C^n that represents the popped edge is set equal to infinity.
- This edge is now inaccessible.
- The ssp of the new C^n matrix is computed.
- The edges of the new ssp are either appended to middle or end of D based on the path finding round.
  - If an edge is already in D, it is not added.
- This procedure is repeated until D is either empty or the number of paths found is greater than sp_{max}.

4. The HSPF algorithm is embarrassingly parallel, as the path finding rounds can be run concurrently with each other.

5. Multi-GPU compute platforms can increase calculation performance by a factor of n, where n is the number of GPUs.