CS 598: Communication Cost Analysis of Algorithms

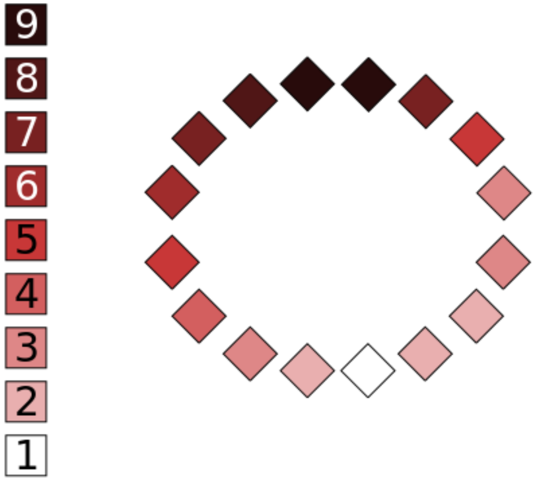
Lecture 12: Bitonic sort revisited and single-source shortest path graph algorithms

Edgar Solomonik

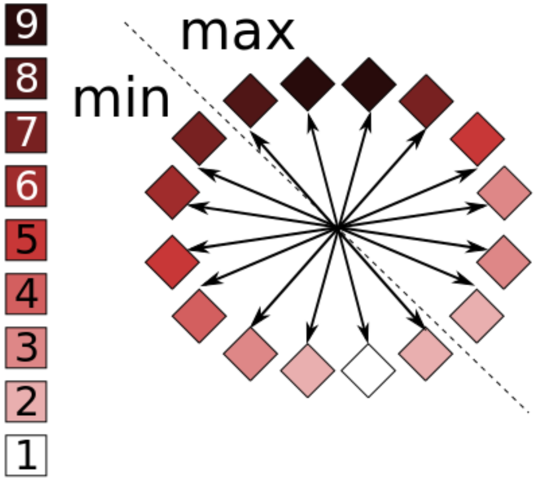
University of Illinois at Urbana-Champaign

October 3, 2016

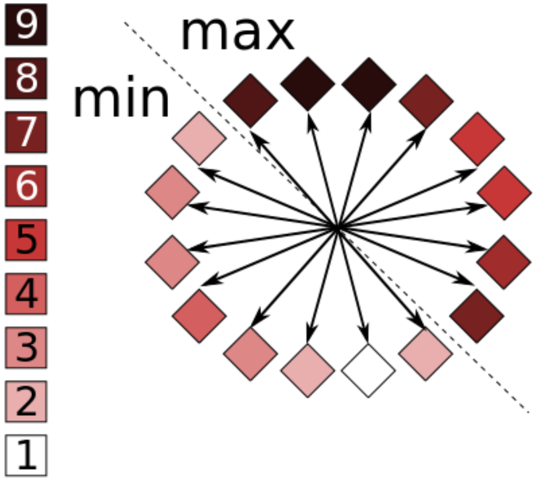
Bitonic sequence as a circle



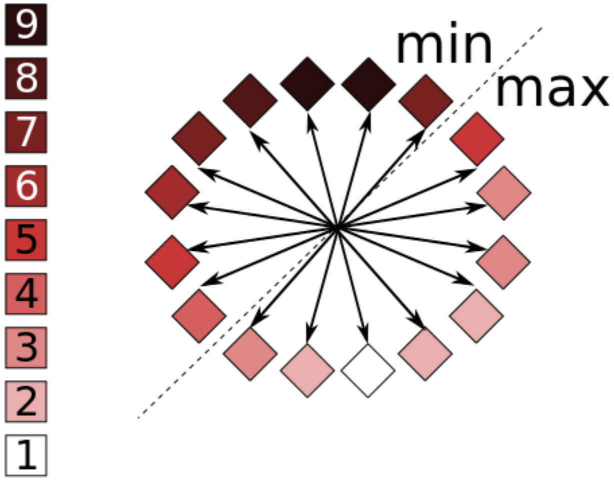
Matching opposite pairs in the circle



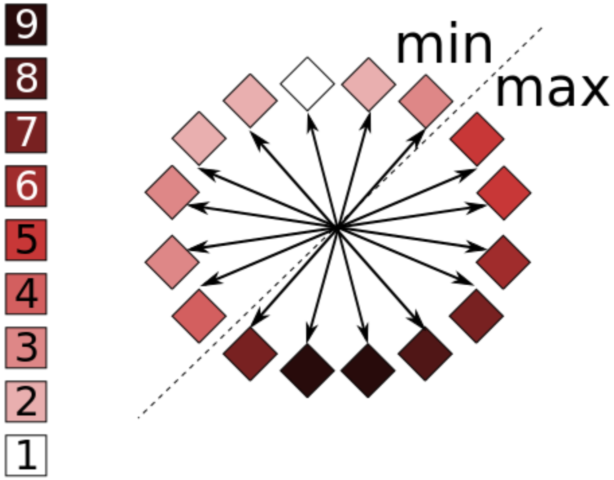
Swapping opposite pairs in the circle



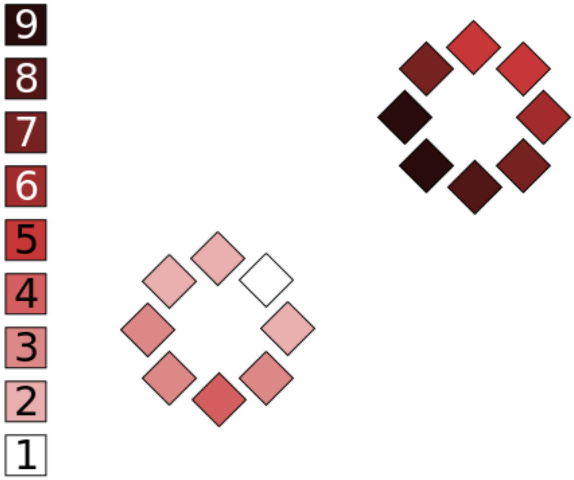
Collecting the min/max into different subsequences



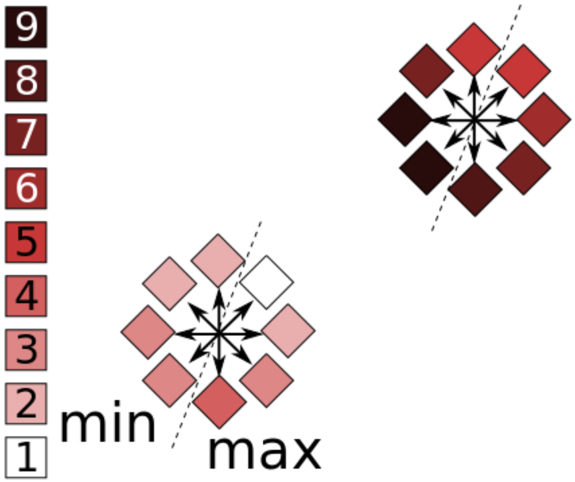
Any partition subdivides smaller/greater halves



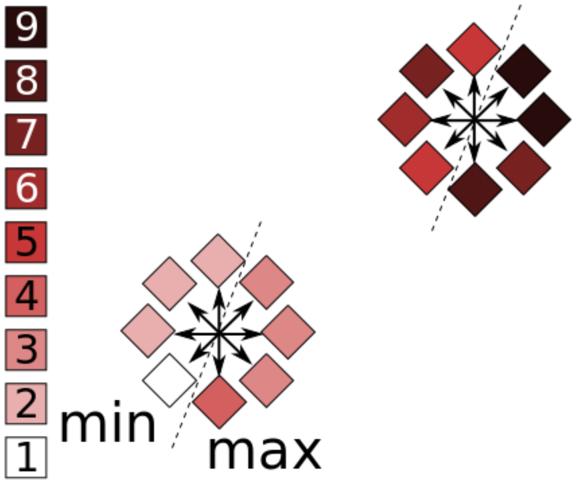
Arranging the two halves into new circles



Swapping opposites again



Continuing with bitonic merge recursively



Bitonic merge

A *bitonic sequence* is any cyclic shift of the sequence $\{i_0 \leq \cdots \leq i_k \geq \cdots i_{n-1}\}$

- each step of bitonic merge partitions the sequence into smaller and greater sets of size $n/2$, both of which are bitonic sequences
- each compare-and-swap acts on elements a distance of $n/2$ away
- these pairings are unaffected by a cyclic shift
- therefore, it suffices to consider swaps on the sequence $S = \{i_0 \leq \cdots \leq i_k \geq \cdots i_{n-1}\}$
- there exists $l \leq k$, such that the largest $n/2$ elements of S are the subsequence $\{i_l, \ldots, i_{l+n/2-1}\}$
- since every element is compared with one $n/2$ away, all of these will be paired with an element outside of the subsequence
- hence the elements of this subsequence are the larger elements in the $n/2$ comparisons
- any subset of a bitonic sequence is a bitonic sequence

Shortest paths in graphs

Given a connected graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$

- find paths $P = (v_1, \ldots, v_s)$, $v_i \in V$, $(v_i, v_{i+1}) \in E$, with min weight

$$W(P) = \sum_{i=1}^{s-1} w((v_i, v_{i+1}))$$

- we define the distance between $u, v \in V$, $d(u, v)$ as the minimal weight $W(P)$ of any path $P = (u, \ldots, v)$ in G
- single-source shortest-paths (**SSSP**) computes $d(s, v)$ from **a source** $s \in V$ to all destinations $v \in V$
- all-pairs shortest-paths (**APSP**) computes $d(u, v)$ from **all sources** $u \in V$ to all destinations $v \in V$
- shortest paths from u can be constructed from distances, by computing the predecessor(s) of each node v: $\{x : d(u, x) + w(x, v) = d(u, v)\}$

Breadth first search (BFS)

Given an unweighted graph ($w(e) = 1$ for all $e \in E$), BFS computes SSSP

- BFS is also a primitive in many other graph algorithms
- a good way to think of BFS is as iterative computation of frontiers
- the root vertex r is the first frontier, and each subsequent frontier is connected to the previous
- the frontiers are a disjoint partition of vertices

$$\{F_1, \ldots, F_d\}, \quad F_1 = \{r\}, \quad V = \bigcup_{i=1}^{d} F_i,$$
$$F_i = \{v : v \in V \setminus (F_{i-1} \cup F_{i-2}), \exists u \in F_{i-1}, (u, v) \in E\}$$

- for each vertex $u \in F_i$, there is a path of $i - 1$ edges from r to u
- therefore the unweighted distance $d(r, u) = i - 1$ if $u \in F_i$

Expressing BFS algebraically

BFS is repeated multiplication of a sparse matrix and a sparse vector

- let the $|V| = n$ vertex labels be unique numbers, so $V = \{1, \dots, n\}$
- consider the adjacency matrix A, where $A(i,j) = 1$ if $(i,j) \in E$
- Q: if G is undirected what property would A satisfy?
- A: A would be symmetric
- we can think of a non-existent edge as an edge with infinite weight, so $A(i,j) = \infty$ if $(i,j) \notin E$
- we represent each frontier F_i as a vector f_i, where $f_i(j) = i - 1$ if $j \in F_i$ and $f_i(j) = \infty$ otherwise
- so, if the root vertex is $r = 1$, $f_1 = \begin{bmatrix} 0 & \infty & \cdots & \infty \end{bmatrix}^T$
- now, we can compute each frontier and tenstative distances D_i, with $D_1 = f_1$ from the subsequent via

$$f_{i+1}(j) = \begin{cases} \infty & : D_i(j) \neq \infty \\ \min_k(f_i(k) + A(k,j)) & : \text{otherwise} \end{cases}$$

and set $D_{i+1}(j) = \min(D_i(j), f_{i+1}(j))$

Semirings

To express graph operations as matrix operations, we need to redefine the elementwise operators

- a **semiring** $(S, \oplus, \otimes)$ is an algebraic structure
  - it defines an additive operator $\oplus$ and a multiplicative operator $\otimes$ on elements in set S
  - both operators should have an identity
  - the additive operator should be commutative and the multiplicative operator should be distributive
  - the additive operator *need not* have an inverse, which differentiates semirings from rings
- other algebraic structures, in particular monoids can make sense for graph algorithms when combined with appropriate functions
- a semiring induces corresponding matrix/vector operations

$$C = A \oplus B \rightarrow C(i,j) = A(i,j) \oplus B(i,j)$$

$$Z = X \otimes Y \rightarrow Z(i,j) = \bigoplus_{k=1}^{n} X(i,k) \otimes Y(k,j)$$

The tropical semiring

The tropical semiring $(\mathbb{R} \cup \{\infty\}, \min, +)$ enables shortest path computation

- note that $+$ is the *multiplicative* operator in the tropical semiring
- Q: what are the additive and multiplicative identities of the tropical semiring?
- A: ∞ and 0
- the tropical semiring allows us to compute frontier in BFS, recall

$$f_{i+1}(j) = \begin{cases} \infty & : D_i(j) \neq \infty \\ \min_k(f_i(k) + A(k,j)) & : \text{otherwise} \end{cases}$$

- perform $x_{i+1} = f_i \otimes A$ to get $x_{i+1}(j) = \min_k(f_i(k) + A(k,j))$ then set

$$f_{i+1}(j) = \begin{cases} \infty & : D_i(j) \neq \infty \\ x_{i+1}(j) & : \text{otherwise} \end{cases}$$

- with unweighted graphs we could also choose do BFS with other semirings

BFS cost

Lets now analyze the cost of BFS

- the number of operations needed to compute BFS is $O(|E|)$, since each edge is traversed once
- the bandwidth cost is at least $O(|E| \cdot \nu)$, since we need to read each edge from memory to cache
- parallelizing BFS in shared or distributed memory can be challenging
    - partitioning the graph could reduce communication costs, but is generally more expensive than BFS
    - in shared memory, threads can branch and perform atomic updates or do redundant work
    - in distributed memory, it makes sense to use a 2D processor grid distribution for A (the edges)
    - the dominant cost is multiplication of sparse matrices with sparse vectors
    - if we are able to balance the work of the d (depth of G) SpMSpVs, we obtain

$$T_{\text{BFS}} = O(d\log(P) \cdot \alpha + n/\sqrt{P} \cdot \beta + |E|/P \cdot (\nu + \gamma))$$

Load balancing by randomization

So how can we balance the work for arbitrary graphs?

- randomly ordering the vertices should achieve load balance with high probability
- **balls-into-bins problem:**
  - place m balls randomly into k bins, what maximum load l is obtained with high probability?
  - $l = m/k$ would be ideal, answer depends on ratio of m to k
  - if $m > k \log k$, we get $l = O(m/k)$, in particular $l = m/k + O(\sqrt{m \log k/k})$
  - in other scenarios there can be (poly)logarithmic factors of imbalance (less than $O(\min(\log(m), \log(k)))$)
  - we will return to this problem in more depth in a subsequent lecture

Load imbalance in BFS

What does the load balance of BFS depend on, given a 2D distribution with randomized vertex ordering?

- if $|F_i| > \sqrt{P}\log(P)$, we can expect the vertices in the frontier to be balanced across columns of the processor grid
- so when $|F_i|$ is small, we expect to have more load imbalance
- we can argue that the distribution of edges (sparse matrix A) among processors is load balanced by a similar argument
    - given a uniform degree graph, we can assign each ball a constant weight and think of bins as processor grid rows/columns
    - variance of vertex degree would increase load imbalance, having some fully connected vertices (few dense columns/rows in A) is a worst case

Load imbalance in BFS for quickly growing frontiers

What might the load imbalance in BFS be for some typical graphs?

- many "real-world" graphs have high *vertex expansion*, typically defined as

$$h(G) = \min_{|S| \leq n/2} |\delta(G,S)|/|S|$$

  where $\delta(G,S)$ is the outer boundary of S in G (its also $F_{i-1} \cup F_{i+1}$ if $S = F_i$ and G is undirected)

$$\delta(G,S) = \{v : v \in V \setminus S, \exists u \in S, (u,v) \in E\}$$

- one can also measure expansion with respect to a subset of size s, namely $h(G,s) = \min_{|S|=s} |\delta(G,S)|$
- if $h(G) > 2$ or $h(G,s) \geq 2s$ then $|F_i|$ will grow geometrically with i
- even if these conditions don't hold, $|F_i|$ may grow very quickly, for instance in power-law graphs, which contain high-degree vertices
- in such cases, parallel BFS would be load imbalanced when $|F_i|$ is small, but the bandwidth costs will be dominated by processing the larger frontiers

Short pause

Projects

Project is in total 60% of the course grade

- first proposal grade deferred, 10% of total grade or 1/6 of project grade is proposal
- 30-min presentation and project report need to be done by end of semester
- guidelines for stage 2 proposal (due Oct 19)
  - project should be set into context with respect to at least 2 previous work citations
  - novelty of the proposed work should be discussed
  - an ideal proposal should be the first ~2 pages of your project report: problem statement, previous work, methodology
- project report should additionally detail the completed work and results (~5 pages)

Dijkstra's algorithm

Lets now return to SSSP for weighted graphs

- BFS is not generally correct, since it only considers paths with a minimal number of edges
- the classical solution for graphs with nonnegative edge weights is Dijkstra's algorithm
  - visit the closest unvisited node and update distances by relaxing edges connected to that node
  - priority queue typically used to find closest node
  - each edge relaxed once and queue modified once for each node, for a cost of $O(|E| + n \log n)$
- Dijkstra's algorithm has very little parallelism
- expressed algebraically, it performs $n - 1$ SpMSpVs with a vector containing a single nonzero
- Δ-stepping [Meyer, Sanders 2003] modifies Dijkstra to exploit more parallelism, by relaxing edges of all nodes within a distance of Δ from the visited nodes

Bellman-Ford algorithm

The Bellman-Ford algorithm computes shortest shortest paths in an arbitrary graph

- if there are negative cycles the problem of computing distances is not well-defined
- Dijkstra's algorithm is not correct in the presence of negative edges
    - we cannot just visit each vertex once ("set labels"), we may always detect a shorter path later
- the Bellman-Ford algorithm relaxes all edges ("updates labels") in the graph at every iteration
    - for sequential execution, the edges are relaxed in some order
    - for parallel execution we can think of an iteration as relaxing all vertices simultaneously
    - of course, we should avoid relaxing outgoing edges from nodes with tentative distance (label) ∞
    - furthermore, we can avoid relaxing edges from nodes whose distance was unchanged since the last set of relaxations

Bellman-Ford algorithm algebraically

At each iteration, we relax a subset of vertices (a frontier), and take the next frontier to be the set of vertices with modified distance labels

$$x_{i+1} = f_i \otimes A \qquad f_{i+1}(j) = \begin{cases} \infty & : x_{i+1}(j) = D_i(j) \\ x_{i+1}(j) & : \text{otherwise} \end{cases}$$

and as in BFS, set $D_{i+1}(j) = \min(D_i(j), f_{i+1}(j))$

For a worst case graph, every node appears in every frontier, for a cost of

$$T_{\text{BF}} = O(h\log(P)\cdot\alpha + hn/\sqrt{P}\cdot\beta + h|E|/P\cdot(\nu+\gamma))$$

where h is the max number of edges in any shortest path, and assuming a load balanced 2D layout of A

This is the cost of h SpMVs (sparse-matrix times dense vector) with $|E|$ nonzeros in the matrix, rather than d SpMSpVs, which gave the BFS cost

$$T_{\text{BFS}} = O(d\log(P)\cdot\alpha + n/\sqrt{P}\cdot\beta + |E|/P\cdot(\nu+\gamma))$$