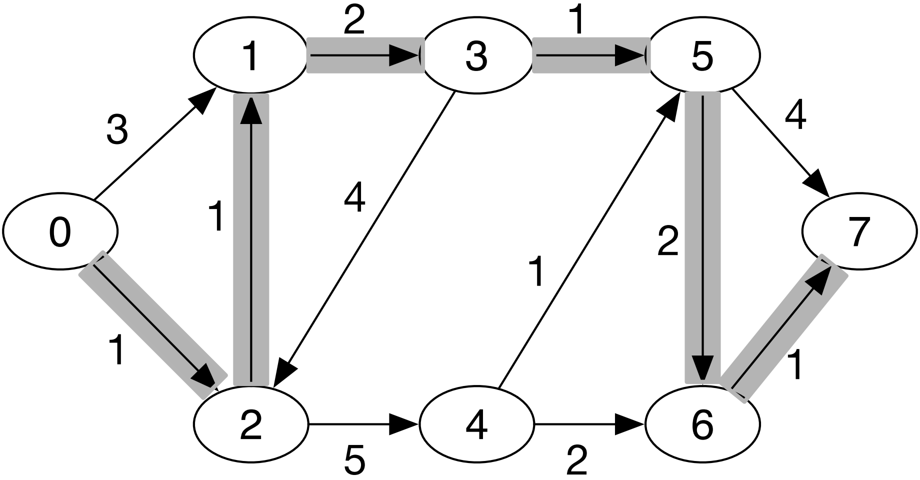
Chapter 6: Single Source Shortest Path

Elements of Parallel Computing

Eric Aubanel

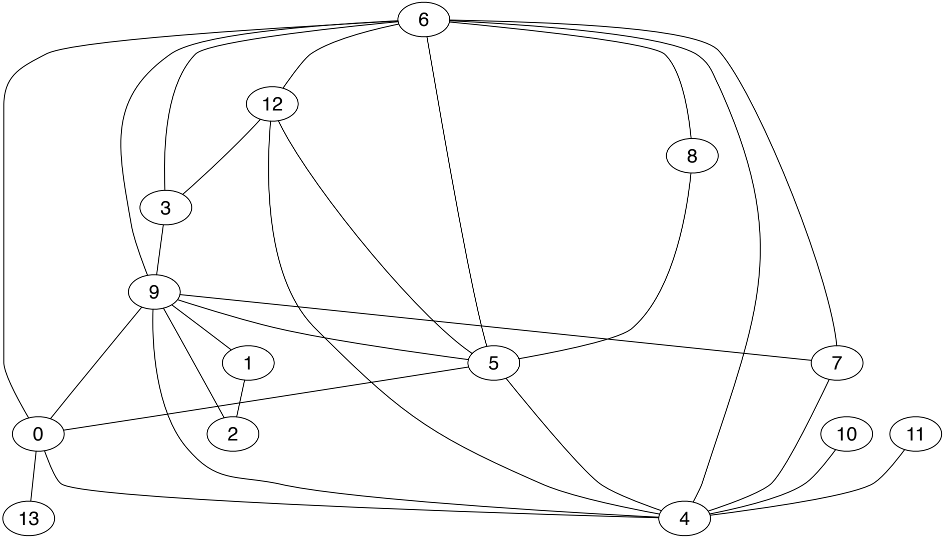
Example



Definitions

- **Shortest path** between two vertices of a graph: path that minimizes sum of edge weights
- **SSSP**: find shortest paths between source vertex all other vertices
- **Diameter** of graph: length of the longest shortest path between all pairs of vertices, ignoring weights
- **Scale-Free Graphs**: degree distribution follows a power law, that is, the number of vertices of degree d is $O(d^{-\lambda})$, where λ is a small constant.

Scale-Free Graph



SSSP Solution with Labelling

Input: Graph with vertices V ($|V| = n$) and edges E with weights C, source vertex s.

Output: D: distance between s and all other vertices. P: pointer to predecessor to each vertex in shortest path.

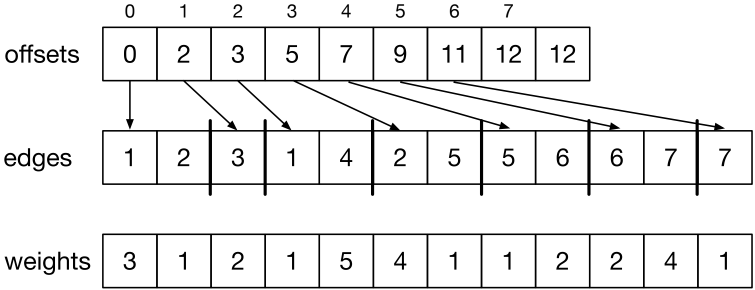
for $i \leftarrow 0$ *to* $n-1$ **do** $D[i] \leftarrow \infty$
$D[s] \leftarrow 0$, list $\leftarrow \{s\}$
while list $\neq \emptyset$ **do**
    remove vertex i from list
    `// Relax each out-edge of vertex` i
    foreach *edge* $e_{ij} \in E$ **do**
        if $D[j] > D[i] + c_{ij}$ **then**
            $D[j] \leftarrow D[i] + c_{ij}$, $P[j] \leftarrow i$
            if $j \notin$ list **then** list $\leftarrow$ list $\cup \{j\}$
        end
    end
end

SSPP Strategies:

- *Label-setting*: in each iteration the distance value of a single vertex is permanently set
- *Label-correcting*: distance value of vertices may be updated multiple times

Data Structures

- Adjacency lists
- Adjacenty matrix
- *Compressed sparse row*: adjacency lists stored contiguously



Bellman-Ford Algorithm

Standard algorithm:

Initialize D
for $k \leftarrow 1$ *to* $n-1$ **do**
    foreach *edge* $e_{ij} \in E$ **do**
        if $D[j] > D[i] + c_{ij}$ **then**
            $D[j] \leftarrow D[i] + c_{ij}$
        end
    end
end

Better to use label correcting algorithm above, with FIFO queue

Bellman-Ford Example

Queue	vertices updated
0(0)	
1(3), 2(1)	1, 2
2(1), 3(5)	3
3(5), 1(2), 4(6)	1, 4
1(2), 4(6), 5(6)	5
4(6), 5(6), 3(4)	3
5(6), 3(4), 6(8)	6
3(4), 6(8), 7(10)	7
6(8), 7(10), 5(5)	5
7(9), 5(5)	7
5(5)	
6(7)	6
7(8)	7

Dijkstra's Algorithm

List algorithm, where list is a min-priority queue.

Queue	**vertices updated**
0(0)	
2(1), 1(3)	1, 2
1(2), 4(6)	1, 4
3(4), 4(6)	3
5(5), 4(6)	5
4(6), 6(7), 7(9)	6, 7
6(7), 7(9)	
7(8)	7

Complexity

- Bellman-Ford: $O(|V||E|)$
- Dijkstra: Min-priority queue takes $O(\log|V|)$ to update vertices and remove the minimum vertex, so $O((|V| + |E|)\log|V|)$ overall.
- In practice Bellman-Ford has lower runtime than complexity suggests, but no known average case
- Delta-Stepping algorithm is a compromise between Bellman-Ford and Dijkstra, and provides average-case linear complexity

Delta-Stepping

- Vertices placed into array B of buckets.
- Bucket i holds vertices with distance in the range $[i\Delta, (i+1)\Delta)$
- Each relaxation will place a vertex in one of the buckets, and may also move it from another bucket

Delta=5

B[0]	B[1]	vertices updated
0(0)		
1(3), 2(1)		1, 2
2(1)	3(5)	3
1(2)	3(5), 4(6)	1, 4
3(4)	4(6)	3
	4(6), 5(5)	5
	5(5), 6(8)	6
	6(7), 7(9)	6, 7
	7(8)	7

Delta Stepping Algorithm

Input: Graph with vertices V ($|V| = n$) and edges E with weights C, source vertex s.
Output: D: distance between s and all other vertices.

foreach *vertex* $v \in V$ **do** `//classify edges as light or heavy`

$H[v] \leftarrow \{e_{vw} \in E \mid c_{vw} > \Delta\}$
$L[v] \leftarrow \{e_{vw} \in E \mid c_{vw} \leq \Delta\}$
$D[v] \leftarrow \infty$

end

`relax(`s, 0`)// places` s `in first bucket (`$B[0]$`)`
$i \leftarrow 0$

while *B is not empty* **do**

    $R \leftarrow \emptyset$

    while $B[i] \neq \emptyset$ **do**

        $Req \leftarrow \{(w, D[v] + c_{vw}) \mid v \in B[i], e_{vw} \in L[v]\}$

        $R \leftarrow R \cup B[i]$`// remember vertices deleted from bucket`

        $B[i] \leftarrow \emptyset$

        foreach $(w, x) \in Req$ **do**

            `relax` (w, x)

        end

    end

    $Req \leftarrow \{(w, D[v] + c_{vw}) \mid v \in R, e_{vw} \in H[v]\}$

    foreach $(w, x) \in Req$ **do**

        `relax`(w, x)

    end

    $i \leftarrow i + 1$

end

```
// Relax edge to vertex w with candidate weight x
// If accepted assign to appropriate bucket
```

Procedure `relax`(w, x)

  if $x < D[w]$ **then**

    $B[\lfloor D[w]/\Delta \rfloor] \leftarrow B[\lfloor D[w]/\Delta \rfloor] \setminus \{w\}$

    $B[\lfloor x/\Delta \rfloor] \leftarrow B[\lfloor x/\Delta \rfloor] \cup \{w\}$

    $D[w] \leftarrow x$

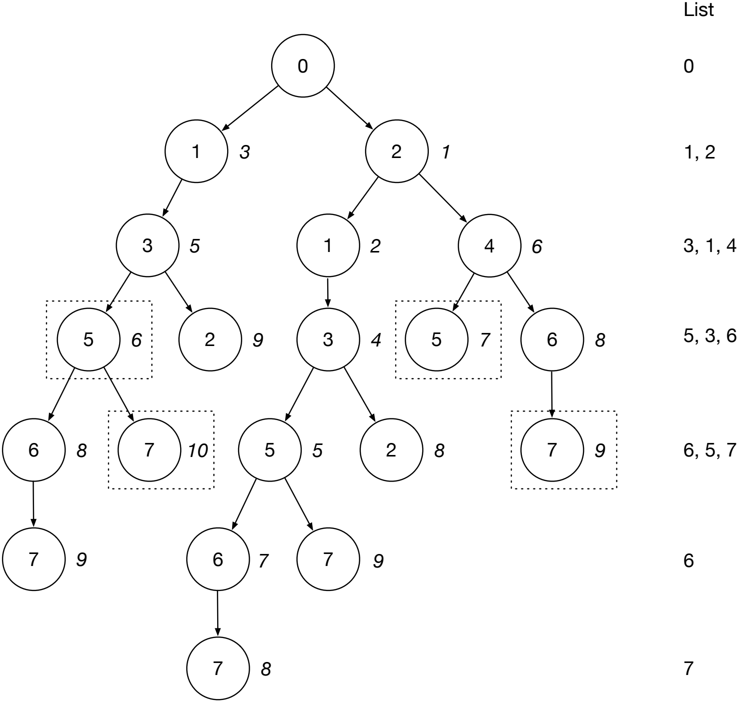
  end

end

Task Decomposition

Bellman Ford, decompose:

- iterations of `foreach` loop (edges adjacent to a vertex)
- process contents of list in parallel


List
0
1, 2
3, 1, 4
5, 3, 6
6, 5, 7
6
7

Dijkstra vs. Bellman Ford

- *Dijkstra*: only have independent tasks for relaxations of edges from one vertex
- *Dijkstra*: $|V| \log |V|$ depth and $O((|V| + |E|) \log |V|)$ work, so $O(|E|/|V|)$ parallelism
- *Bellman-Ford*: in worst case, $O(|V|)$ depth, $O(|E||V|)$ work, and $O(|E|)$ parallelism
- *Delta-Stepping*: tune Δ to balance parallelism and work-efficiency

Data Decomposition: Edge Partitions