CS 61B Data Structures and Programming Methodology

Aug 6, 2008

David Sun
Breadth-first Search

• Starts at an arbitrary source vertex $s$ then visits every vertex that is reachable from it.
  – During this systematic traversal we can compute the distance (in terms of the smallest number of edges) and the shortest path from $s$ to each reachable vertex $v$.

• Called Breadth-first because it:
  – Visits all vertices whose distance from the starting vertex is one, then all vertices whose distance from the starting vertex is two, and so on.
BFS(Graph G) {
    Queue<Vertex> fringe;
    fringe = queue containing {v};
    v.dist() = 0;
    v.parent() = null;

    while (! fringe.isEmpty()) {
        Vertex v = fringe.dequeue();
        For each edge (v,w) {
            if (! marked (w))
                mark(w);
                w.dist() = v.dist() + 1;
                w.parent() = v;
                fringe.enqueue(w);
        }
    }
}
Correctness of BFS

• The starting vertex is enqueued first, then all the vertices at a distance of 1 from the start, then all the vertices at a distance of 2, and so on.

• Why?
  – When the starting vertex is dequeued, all the vertices at a distance of 1 are enqueued, but no other vertex is.
  – When the depth-1 vertices are dequeued and processed, all the vertices at a distance of 2 are enqueued, because every vertex at a distance of 2 must be reachable by a single edge from some vertex at a distance of 1.
  – When the depth-1 vertices are dequeued and processed no vertex at a depth other than 2 will be enqueued, because every vertex at a distance greater than 2 is not reachable by a single edge from some vertex at depth of 1.
Running Time of BFS

• Observations:
  – Each of the $|V|$ vertices is enqueued at most once, and hence dequeued at most once.
  – Enqueuing and dequeuing take $O(1)$ time – total time devoted to queue operations $O(|V|)$.
  – If adjacency list representation is used:
    • each adjacency list is scanned at most once.
    • the sum of the length of all adjacency lists is $\Theta(|E|)$.
    • time spent scanning the adjacency list is $O(|E|)$

• Running time:
  – $O(|V| + |E|)$ if using adjacency list.
  – $O(|V|^2)$ if using adjacency matrix.
Problem:
You want to wire the pins of some circuit component. With $n$ pins, you can interconnect them using $n-1$ wires. Of all possible arrangements, we'd like to find the one that uses the least amount of wire.
Minimum Spanning Tree

• We can model the problem using using a connected, undirected graph $G = (V, E)$ as follows:
  
  – $V$ is the set of pins,
  – $E$ is the set of possible interconnections (between pairs of pins),
  – For each edge $(u,v)$ in $E$, there is a weight($u, v$) specifying the cost (amount of wires) needed to connect $u$ to $v$.
  – Now, find an acyclic, subset $T$ connects all the vertices and whose total weight is minimized.
Minimum Spanning Tree

- $T$ is acyclic and connects all of the vertices,
  - it must form a tree, which we call a spanning tree since it “spans” the vertices of the graph $G$.
  - we are not minimizing the number of edges in $T$, since all spanning trees have exactly $|V|-1$ edges.
  - the problem of determining $T$ given a graph is called the minimum-spanning-tree problem.
Generic Algorithm

• Generic Algorithm for finding the *minimum spanning tree*:
  – A iterative algorithm that uses a *greedy strategy*, which means that at each iteration, we “grow” the current spanning tree by picking an edge with the least weight.

```
Generic-MST(Graph G)
Set<Vertex>A;
while A does not form a spanning tree
  find an edge (u,v) in E of G such that after adding (u,v) to A, A is a subset of a minimum spanning tree.
  Add (u,v) to A
```
Kruskal’s Algorithm

• Based directly on the generic minimum-spanning-tree algorithm:
  – At each iteration we find the edge of the least weight and that does not create a cycle.
  – The set of edges found so far forms a forest.

MST-Kruskal(Graph G)
1. Create a new tree $T$ with the same vertex set as G.
2. Sort the edges of G in nondecreasing order by weight.
3. Iterate through the sorted edges, for each edge $(u,v)$:
   If $u$ and $v$ are not connected by an edge in $T$
   add $(u,v)$ to $T$
Example
Running Time of Kruskal’s Algorithm

- **Step 1** Creating a new graph with the same vertex set:
  - Takes $O(|V|)$ time.
- **Step 2** Sorting $|E|$ edges:
  - Using one of the logarithmic-time sorting algorithms, e.g., mergesort, heapsort or quicksort, we can do this in $O(|E| \log |E|)$ time.
- **Step 3** Determining whether $u$ and $w$ are already connected by a path.
  - A simple way is to do a depth-first traversal on $T$ starting at $u$, and see if we visit $w$, however, potentially taking $\Theta(|V|^2)$ time.
  - We can do better using *Disjoint Sets*, which takes $O(|E| \log |E|)$ time.
- Hence, the running time is in $O(|V| + |E| \log |E|)$.
- If $|E| < |V|^2$, then $\log |E| < 2 \log |V|$. Therefore, Kruskal's algorithm runs in $O(|E| \log |V|)$ time.
Correctness of Kruskal’s Algorithm

• Suppose the algorithm is considering adding an edge \((u, w)\) to \(T\), and there is not yet a path connecting \(u\) to \(w\).

• Let \(U\) be the set of vertices in \(T\) that are connected (so far) to \(u\), and let \(W\) be a set containing all the other vertices, including \(w\).

• Let the **bridge edges** be any edges in \(G\) that have one end vertex in \(U\) and one end vertex in \(W\).

• **Argument:** Any spanning tree must contain at least one of these bridge edges. As long as we choose a bridge edge with the lowest weight, we are safe.
Prime’s Algorithm

• Operates much like *Dijkstra’s* algorithm for finding shortest paths in a graph.
  – The set of edges found so-far always forms a *tree*.
  – Start at some root and grow the tree until it spans all the vertices in $V$.
  – At each iteration, we add to the *tree* the edge of least weight that does not create a cycle.
Prime’s Algorithm

MST-Prime(Graph G)
PriorityQueue fringe;
For each vertex v {
  v.dist() = ∞;
  v.parent() = null;
}
Choose an arbitrary starting vertex, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (! fringe.isEmpty()) {
  Vertex v = fringe.removeMin();
  For each edge (v, w) {
    if (w ∈ fringe && weight(v, w) < w.dist()) {
      w.dist() = weight (v, w); w.parent() = v;
    }
  }
}
Running Time of Prime’s Algorithm

- Initialization and putting all the vertices into the priority queue: $O(|V|)$ time.
- Removing the minimum element from the priority queue in each iteration: $O(\log |V|)$ time. This is executed $|V|$ times: $O(|V| \log |V|)$.
- The body of the for-loop takes $O(\log |V|)$ since by updating \( \text{dist}() \) of a vertex, we are effectively reinserting the vertex into the priority queue. The body of the for-loop is executed $|E|$ times: $O(|E| \log |V|)$.
- Hence the running time of Prime’s algorithm is $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$, which is asymptotically the same as the implementation of Kruskal’s algorithm using Disjoint Sets.