31251 – Data Structures and Algorithms

Week 9, Autumn 2020 - Graphs Part II

Xianzhi Wang

Does anyone read these titles?

- Dynamic Programming
    - Some Examples
    - Dijkstra’s Algorithm
- Connectivity in Graphs
    - Connectivity and Connected Components
    - Strongly Connected Components
    - Articulation Points
- Dependencies in Directed Graphs

Dynamic Programming

Dynamic Programming

- Dynamic programming looks a lot like Divide-and-Conquer, but subinstances overlap.
- Usually implemented with some sort of table that keeps track of subinstance solutions.
- Normally they work bottom-up (Can be done top-down recursively):
    - Solve the smallest subinstances, combine them to make bigger subinstances, keep going until you have the whole thing.

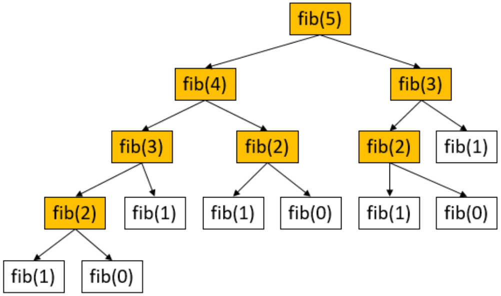
What makes it work?

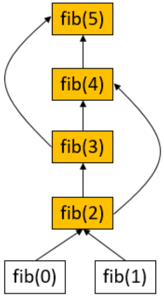
- Optimal Substructure: same as greedy algorithms and D&C.
  - $\mathit{Solution}(P) \leftarrow f(\mathit{Solution}(SP_1), \cdots, \mathit{Solution}(SP_n))$
- Overlapping Subproblems:
  - e.g., the recursive algorithm (practice 4a, week 4) repetitively calculates the subinstances of the problem.

```
fib(5)
=fib(4)+fib(3)
=(fib(3)+fib(2))+(fib(2)+fib(1))
=((fib(2)+fib(1))+(fib(1)+fib(0)))+((fib(1)+fib(0))+fib(1))
=(((fib(1)+fib(0))+fib(1))+(fib(1)+fib(0)))+((fib(1)+fib(0))+fib(1))
```

A Comparison

Recursive (on the left) vs. Dynamic Programming (on the right)





0/1 Backpack Problem

"Given n items $\{o_1, o_2, \cdots, o_n\}$ and their weights $\{w_1, w_2, \cdots, w_n\}$ and values $\{v_1, v_2, \cdots, v_n\}$, put these items in a knapsack of capacity W to get the maximum total value in the knapsack."[1]

- Overlapping subproblems:
  - Consider all possible solutions, especially those near-to-optimal ones, they are likely to contain similar subsets of items.
- Optimal substructures:
  - If the optimal solution contains only one item:

$$optsol_1 = \underset{i \in \{1,2,\cdots,n\}}{\arg\max} \{v_i | w_i < W\}$$

  - If the optimal solution contains m items:

$$optsol_m = \underset{i \in \{1,2,\cdots,n\}/sol_{m-1}}{\arg\max} \{v_i + v(sol_{m-1}) | w_i + w(sol_{m-1}) < W\}$$

[1] https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Dijkstra's Algorithm

Given G with non-negative edge weights and a starting vertex u:

1. Set distances: 0 for u, $+\infty$ for others vertices.
2. Mark all vertices as unvisited.
3. Set the current vertex to u.
4. While there are unvisited vertices:
    1. For each unvisited neighbour of the current vertex:
        1. Compare the distance of each neighbour to the distance to current plus the edge weight of the edge joining them.
        2. Keep whichever is smaller.
    2. Mark the current vertex as visited.
    3. Select the unvisited vertex with smallest tentative distance and set it as the current vertex.

Dijkstra's Algorithm

- Where was the dynamic programming?

Suppose *dist*[*v*] is the distance from *u* to *v*, then the algorithm is repeatedly performing

$$\underbrace{dist[v]}_{updated\ \ distance} = \min\{ \underbrace{\overbrace{dist[v]}^{past\ \ result}}_{known\ \ shortest\ \ distance}, \underbrace{\overbrace{dist[current]}^{past\ \ result} + w(current, v)}_{newly\ \ calculated\ \ distance} \}.$$

- As such, we always keep the shortest distance of a vertex from the starting vertex, representing an optimal substructure.

Dijkstra Complexity

- Suppose $|V| = n$ and $|E| = m$ in G. We have $n - 1 < m < n(n-1)$.
- There are $O(n)$ inserts, $O(m)$ updates, $O(n)$ Find/Delete Minimums
- Data structure-dependent:
  - Unsorted array: $O(n^2)$.
  - Binary min heap: $O(m \log n)$.
  - Fibonacci heap[2]: $O(m + n \log n)$

[2] https://en.wikipedia.org/wiki/Fibonacci_heap

Dealing with Negative Edges

- Dijkstra's algorithm ran into trouble with negative edges because we never look at marked vertices again.
- Bellman-Ford algorithm can handle negative edges – it just looks at everything again!
- Doesn't work with negative *cycles* though – but it can detect them.

Bellman-Ford Algorithm

Bellman-Ford Algorithm:

1. Set things up as with Dijkstra's.
2. For 1 to the number of vertices
    - For each edge *uv*
        - If using *uv* improves the distances, use it and update as needed.

Check for negative cycles:

3. For each edge *uv*
    - If *uv* improves the distances, there's a negative cycle.

Bellman-Ford – Complexity

Suppose $|V| = n$ and $|E| = m$ in G.

Bellman-Ford gets:

- $O(nm)$ time – can't really get around this.
- $O(n)$ space.

Connectivity

Connectivity of Graphs

- When using graphs to model things, we usually want to know if they're all in one piece.
- A `connected` graph allows any vertex to get to any other vertex via edges.
- It is useful to check the reachability between nodes in computer networks or transport networks, etc.

Testing Connectivity

- Testing connectivity is easy — Just do a traversal!
    - If we visit all the vertices, the graph must be connected.
- What if it's not connected?
    - We might be interested in finding out what the subgraphs are.
    - We call them connected components.
- Finding them is almost as easy as testing connectivity:

1. While not all vertices have been visited:
    1. Pick an unvisited vertex to start a new traversal.
    2. Mark all the vertices you can reach as visisted and record them as a component.

Strongly Connected Components

For directed graphs,

- Two vertices are *strongly connected* if there is a path from one to the other, as well as a path back.
- *Strongly connected component*: a subgraph where every pair of vertices is strongly connected.

Note that,

- In a directed graph, a may not be adjacent to b when b is adjacent to a.

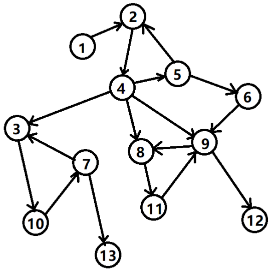
Strongly Connected Components

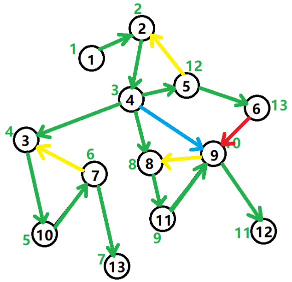
Detecting Strongly connected components in a graph:

- We can just run a bunch of reachability queries.
    - $O(n^2(n+m))$ without clever optimisation.
    - $O((n+m)\log n)$ if we make the approach complicated enough (uses divide and conquer).
- Parallelises well.

We can do it faster using *Tarjan's Algorithm*.

Tarjan's Algorithm





DFS spanning trees have four kinds of edges[3]:

- `Tree edge (green)`: *points to* an unvisited node.
- `Back edge (yellow)`: *points to* an ancestor.
- `Cross edge (red)`: *points to* a visited non-ancestor.
- `Forward edge (blue)`: *points to* an visited offspring.

[3]`https://programmer.ink/think/5d2cd3c392a88.html`

Tarjan's Algorithm

- Each vertex has an id $\in \{1, 2, \cdots, n\}$.
- Each vertex u maintains two variables:
  - `DFN[u]`: The order in which nodes are searched during depth-first search traversal.
  - `LOW[u]`: Let the subtree rooted in u be Subtree(u). LOW[u] is defined as the minimum value of the following nodes: the node in Subtree(u); the node from Subtree(u) through an edge not on the search tree.
- An ancestor has smaller `DFN` than offspring in the tree.
- `DFN` on a path starting from the root increases strictly while `LOW` does not.

Tarjan's Algorithm

- Nodes in the graph are searched and SCCs are detected during depth-first-search (DFS) traversal.
- Given the current vertex u, consider three cases its adjacent node v (not the parent of u):
    - Case 1: v is not accessed: continue to search for v in depth. In the retrospective process, update `LOW[u]` with `LOW[v]` — whatever v can trace back to, u can as well.
    - Case 2: v has been visited and is in the stack: v could be the earliest vertex that can be traced back from u. Update `LOW[u]` with `DFN[v]`.
    - Case 3: v has been visited but is not in the stack: v has been added to a connected component. There is nothing left to do.

Tarjan's Algorithm

```
int dfn[n] = {0}, low[n], dfn_cnt = 0; // variables for each vertex
stack<int> s; // stack
set<int> in_s; // a set for checking if a node is in stack
set<set<int>> all_sccs; // algorithm's result: all SCCs

void tarjan(int u) {
    low[u] = dfn[u] = ++dfncnt; // initialise dfn and low
    s.push(u); in_s.insert(u); // add u to stack & mark it as visited
    for (v: every adjacent vertex of u) {
        if (!dfn[v]) // case1: v unvisited
            tarjan(v); // recur for v
            low[u] = min(low[u], low[v]); // retrospective
        } else if (in_s.find(v) != in_s.end()) // case2: v visited & in stack
            low[u] = min(low[u], dfn[v]); // update low[u]
    }
    if (dfn[u] == low[u]) { // this only happens when u is the root of a SCC
        set<int> scc; // create a new scc
        while (s.top() != u) // add all vertices above u in stack to scc
            scc.insert(s.top()); in_s.erase (s.top()); s.pop();
        scc.insert(s.top()); in_s.erase (s.top()); s.pop(); // add u
        all_sccs.insert[scc]; // put scc to the result
    }
}
```

Tarjan's Algorithm

```
int dfn[n] = {0}, low[n], dfn_cnt = 0; // variables for each vertex
stack<int> s; // stack
set<int> in_s; // a set for checking if a node is in stack
set<set<int>> all_sccs; // algorithm's result: all SCCs

void find_scc(directed_graph d) {
    for (each vertex u in d) {
        if (!dfn[u]) // u unvisited
            tarjan(u);
    }
}
```

Complexity:

- Time: $O(n + m)$.
- Space: $O(n)$

Finding Articulation Points

A similar algorithm can be used to find the equivalence of strongly connected components in an undirected graph.

- "A vertex in an undirected connected graph is an articulation point (or cut vertex) *iff* removing it (and edges through it) disconnects the graph."[4]
- "Articulation points represent vulnerabilities in a connected network — single points whose failure would split the network into two or more disconnected components."

[4] `https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/`

Finding Articulation Points[5]

- Pick an arbitrary vertex of the graph root and run depth first search from it.
    - Undirected graphs have only tree and back edges (why?)
    - All vertices visited before u in DFS are ancestors of u.
- Given the current vertex in DFS, v ($\neq$ *root*),
    - Case 1: if none of its descendants has a back-edge to its ancestors, v is an articulation point;
    - otherwise, v is not an articulation point.
- Case 2 (special case): *root* is an articulation point *iff* it has more than one child in the DFS tree.
    - Why? That is at least one subtree that can only be reached (by other vertices) through *root*.

[5] `https://cp-algorithms.com/graph/cutpoints.html`

The Algorithm

- Each vertex has an id $\in \{1, 2, \cdots, n\}$.
- Each node u maintains two variables:
    - `tin[u]`: the entry time (*or* age) for u.
    - `low[u]`: the lowest entry time reachable by u[6].

$$low[u] = \min \begin{cases} tin[u] & \text{the entry time of u} \\ tin[v] & \text{for all v for which (u,v) is a back edge} \\ low[to] & \text{for all to for which (u,to) is a tree edge} \end{cases}$$

- u in the DFS tree is an articulation point *iff* its depth is smaller than or equal to the lowpoint of any of its children
    - $tin[u] \leq low[to]$

[6] *low*[] is just a way of tracking whether there are at least two paths to get to a vertex — Vertices in biconnected components all have the same *low* value.

The Algorithm

```
int tin[n] = {0}, low[n], timer = 0;
bool visisted[n] = {false};

void find_cutpoints(undirected graph d) {
    for (u: every vertex in d) {
        if (!visited[u])
            dfs(u);
    }
}
```

The Algorithm

```
int tin[n] = {0}, low[n], timer = 0;
bool visited[n] = {false};
set<int> cutpoints;

void dfs(int u, int v = -1) { // no parent of u is known by default
    visited[u] = true;
    low[u] = tin[u] = ++timer;
    int childCount = 0;
    for (int to : all children of u) {
        if (to == v) // (u,to) leads back to u’s parent in DFS tree
            continue;
        if (visited[to]) { // (u,to) is a back edge to an ancestor
            low[u] = min(low[u], tin[to]);
        } else { // u unvisited, (u,to) is a tree edge
            dfs(to, u);
            low[u] = min(low[u], low[to]);
            if (low[to] >= tin[u] && p!=-1) // case 1
                cutpoints.insert(u);
            ++childCount;
        }
    }
    if(v == -1 && childCount > 1) // case 2
        cutpoints.insert(u);
}
```

The Algorithm

```
FindArticulationPoints(i, d)
  visited[i] = true, depth[i] = d
  low[i] = d, child_count = 0
  is_articulation = false
  for (each neighbour n of i)
      if (not visited[n])
        parent[n] = i
        FindArticulationPoints(n, d+1)
        child_count = child_count + 1
        if (low[n] >= depth[i])
            is_articulation = true
        low[i] = min{low[i], low[n]}
      else if (n != parent[i])
        low[i] = min{low[i], depth[n]}
  if ((i has a parent and is_articulation) or
     (i has no parent and child_count > 1))
    i is an articulation point
```

Complexity

- Time: $O(n + m)$ — it's just a depth-first traversal.
- Space: $O(n)$

- Can be modified to retrieve the biconnected components with the same complexity.

- What if we just remove each edge and test if the remaining graph is still connected?
    - $O(n * (n + m))$

- Can be done in parallel in $O(\log n)$ time! (But with $O(n + m)$ processors...)

Dependencies in Directed Graphs

DAGs

If a directed graph has no directed cycles, we called it a ***Directed Acyclic Graph***, or ***DAG***.

How many strongly connected components does this have?

DAGs

DAGs are like the trees of the directed graph world.

As such, they have many applications:

- Anything where dependencies are important (anyone running a *NIX distro will understand this one).
- Task scheduling.
- Compilation.
- Computing information in a spreadsheet.
- Modelling multi-stage processes.

Topological Sorting

DAG describes the dependencies among things (vertices).

We are interested in how to do things in an order that is complaint with the dependencies — *topological ordering*.

It may not be unique. (why?)

We do *topological sorting* to find a topological ordering.

Kahn's Algorithm

```
list<vertex> topo_order;

void Kahn_topo_sort(directed graph g) {
    set<vertex> s = {all vertices with no incoming edges};
    while (!s.empty()) {
        v = s.remove_something_from_a_set();
        topo_order.append(v);
        for (each neighbour u of v) {
            remove (v,u) from g;
            if (u has no incoming edges)
                add u to s;
        }
    }
}
```

Depth-First Topological Sort

```
stack<vertex> topo_order;

void topo_sort (directed graph g) {
    while (not all vertices marked) {
        v = next unmarked vertex;
        visit(v)
    }
}

visit (vertex v) {
    if (v is marked) // terminate the iteration
        return;
    for (each (v,u) in edges) // do recursion
        visit(u);
    mark v;
    topo_order.push(v); // update the result
}
```

Complexity

- Both linear time algorithms – $O(n + m)$.
- Both use $O(n)$ extra space.
    - Can get $O(1)$ for Kahn's algorithm at the cost of increasing time to $O((n + m) \log n)$.
- Can find a topological (a different way) in $O(\log^2 n)$ time ... if we have a polynomial number of processors.

Appendices

Dijkstra's Algorithm – Correctness

Let S be the set of marked vertices, u the starting vertex, and $d(v)$ the calculated distance from u to v, then the correctness of Dijkstra's algorithm can be stated as:

Lemma

For every vertex $v \in S$ $d(v)$ is the shortest path from u to v.

Dijkstra's Algorithm – Correctness

Proof. By induction on $|S|$.

- **Base Case.** Easy, $|S| = 1$ implies $S = \{u\}$, and the distance is 0. Alternatively $|S| = 0$ is trivial.
- **Induction Hypothesis.** Assume the theorem is true for $|S| = k \geq 1$.
- **Inductive Step.**
    - Let v be the next vertex added to S, and w be the neighbouring path vertex according to the algorithm.
    - By the assumption, $d(w)$ is minimum.
    - Let P, for contradiction, be the shortest path from u to v.
    - Let xy be the first edge in P that leaves S.
    - Then the length of P is at least $d(x) + w(x, y)$, which is the tenative distance to y.
    - This distance can't be smaller than the tentative distance to v, otherwise we would've chosen it instead of v to add next.
    - Therefore the length of P is at least $d(v)$. □

Bellman-Ford Algorithm - Correctness

Formally, we need a slightly more precise statement:

Lemma

After i iterations of the outer for loop:

- *If $d(v) \neq \infty$, there is a (known) path from the start vertex u to vertex v with distance $d(v)$.*
- *If there is a path from u to v with at most i edges, $d(v)$ is the length of the shortest path with at most i edges.*

Bellman-Ford Algorithm - Correctness

Proof. By induction on i:

- **Base Case:** $i = 0$, nothing has happened yet, trivial!
- **Inductive Assumption:** Assume that at step i we have the shortest paths with at most i edges.
- **Inductive Step:**
    - (First part) When $d(v)$ is updated to $d(r) + w(r, v)$, there is a path from u to r to v with weight $d(v)$.
    - (Second part) Before updating, we have by assumption the distances of all the shortests paths with at most i edges computed (if they exist).
    - Then at each possible update, we compare the shortest path to a neighbour plus an edge, with the shortest path on at most i vertices, and keep the minimum.
    - So we have a path on $i + 1$ edges that is shorter than the i edge path (and we pick the smallest of all these), or we keep the i edge path – thus the path must be the shortest path on $i + 1$ edges – there are no other possible ways to get it. $\square$