Topological Sorting and Strong Connectivity

Randy Truong

rtruong@u.northwestern.edu

February 1, 2024

Outline

1. topological sorting
   - primary intuition
   - motivating problem
   - impl #1: dfs-based algorithm
   - impl #2: kahn's algorithm
   - impl #3: arrival-departure algorithm
2. strong connectivity
   - intuition
   - solution intuition
   - summary

Topological Sort

a motivating problem

a *topological sorting* of a directed graph G is an ordering of all vertices $v \in G$ such that it preserves the dependency relations between all vertices

- ie, if $a \to b$ in G, then we cannot have an ordering where $b \to a$

a motivating problem

based on LC #207. . .

You are a software developer that works for a school. Given a list of courses and their prerequisites, devise an algorithm for developing viable course schedules.

a motivating problem

based on LC #207...

You are a software developer that works for a school. Given a list of courses and their prerequisites, design an algorithm for developing viable course schedules.

problem intuition

construct a directed graph G

- let each vertex v represent a course
- let each edge e represent a "is a prerequisite of" relationship
    - $a \rightarrow b$ means "course a is a prerequisite of course b"

a motivating problem

solution intuition

1. for each course
   1. if the current course that we're looking at has already been checked, do nothing
   2. otherwise, check the current course's descendant courses recursively
   3. add the current course to the schedule only after visiting all of its descendants
2. reverse the ordering of the list

impl #1: dfs-based

algorithm pseudocode

1. initialize a results array `results` $\leftarrow \{\}$
2. iterate through all vertices $v \in G$
    1. if $v \in$ `was`, then do nothing
    2. otherwise, recursively examine its descendant(s)
    3. append current vertex v to `results`
3. after adding all vertices to the results array, reverse the ordering of results

impl #1: dfs-based

algorithm code

```
main() {
    vector<int> graph[100];
    vector<int> top_sort;
    vector<bool> was[N];
    forn(0, N, 1) {
        if (!was[i]) {
            was[i] = true;
            dfs(i);
        }
    }
    reverse(all(top_sort));
}
```

```
dfs(int node) {
    for (int i: graph[node]) {
        if (!was[i]) {
            was[i] = true;
            dfs(i);
        }
    }
    final.pb(node);
}
```

impl #1: dfs-based (with cycle detection)

algorithm intuition

1. for each course
    1. if the current course that we're looking at has already been checked, do nothing
    2. if the current course that we're looking at has already been checked *in the current path*, then we found a cycle
    3. otherwise, check the current course's descendants, *adding each descendant to the path*
    4. add the current course to the schedule only after examining all of its descendants
    5. remove the current course from the path
2. reverse `results`

impl #1: dfs-based (with cycle detection)

algorithm intuition

1. initialize a results array `results` $\leftarrow \{\}$
2. initialize a path array `path` $\leftarrow \{\}$
3. iterate through all vertices $v \in G$
   1. if $v \in$ `was`, then do nothing
   2. if $v \in$ `path`, then cycle $\rightarrow$ cycle found
   3. otherwise, add v to `path`, then recursively examine its descendant(s)
   4. append current vertex v to `results`
   5. remove v from `path`
4. reverse `results`

impl #1: dfs-based

algorithm code

```
main() {
    vector<int> graph[100];
    vector<int> top_sort;
    bool was[N];
    forn(0, N, 1) {
        if (!was[i]) {
            was[i] = true;
            dfs(i);
        }
    }
    reverse(all(top_sort));
}
```

```
dfs(int node) {
    for (int i: graph[node]) {
        if (!was[i]) {
            was[i] = true;
            dfs(i);
        }
    }
    final.pb(node); // post
        order add
}
```

impl #2: kahn's algorithm

algorithm intuition

how to reshape our algorithm?

impl #2: kahn's algorithm

algorithm intuition

replace dfs in the original algorithm with bfs

- start from a course that has zero prerequisites
- iterate through the graph only in *rings*
  - bfs ensures that we only visit an immediate descendant of the current node

impl #2: kahn's algorithm

algorithm intuition

replace dfs in the original algorithm with bfs

- utilize iterative bfs $\rightarrow$ store neighbors in a FIFO queue
- queue neighbors via *indegree* (# of incoming nodes)
  - **intuition.** if a neighbor n has incoming nodes $\rightarrow$ n must have prerequisites $\rightarrow$ we must iterate through them before visiting n

impl #2: kahn's algorithm

algorithm intuition

1. initialize a results vector `results` $\leftarrow \{\}$
2. for each vertex $v \in G$:
   - add v to the results vector `results`
   - iterate through all v's children
     - decrement the number of incoming nodes for the current child
     - if the indegree of a child is 0 $\rightarrow$ it has no other prerequisites $\rightarrow$ push it to the queue

impl #2: kahn's algorithm

```
main() {
  // 1.  create indegree vector
  vector<int> indegree;
  vector<int> results;
  queue<int> q;
  forn (0, n, 1) {
    if (indegree[i] == 0) // 2.  identify a starting point
      q.push(i);
  }
  while (!q.empty()) { // 3.  utilize iterative bfs
    int curr = q.front();
    q.pop();
    results.pb(curr);
    for (int next : graph[curr]) {
      indegree[next]--;
      if (indegree[next] == 0)
      q.push(next);
    }
  }
}
```

impl #3: arrival-departure algorithm

algorithm motivation

can we utilize the notion of indegree and dfs in order to produce a topological sorting of G?

impl #3: arrival-departure algorithm

algorithm motivation

let utilize the notion of *departure time* of all vertices to topologically sort G

impl #3: arrival-departure algorithm

arrival and departure times

we can classify nodes within a graph using the following notions:

1. **Arrival Time.** the time at which a vertex $v \in G$ is first discovered/visited during dfs
2. **Departure Time.** the time at which dfs finishes processing v (ie, comes back to it)

time t is simply represented using a counter, which will increment with each iteration of dfs

impl #3: arrival-departure algorithm

algorithm intuition

using the notion of departure time, it follows that if vertex v_i has a greater departure time than vertex v_j, v_i must be an ancestor of v_j

- thus, it follows that an ordering of the vertices by descending departure time must be a valid topological sorting of G

impl #3: arrival-departure algorithm

algorithm intuition

1. initialize counter $t \leftarrow 0$
2. initialize departure time vector $d \leftarrow \{\}$ where $d[i]$ represents a node whose depart time is i
3. initialize visited array `was` $\leftarrow \{\}$
4. initialize results array `results`
5. for each vertex $v \in G$
    - if $v \notin$ `was`
        - visit v
        - recursively visit neighbors
    - after visiting all descendants, increment t
    - set the departure time of v to be t
6. reverse order of `results`

impl #3: arrival-departure algorithm

algorithm pseudocode

```
main() {
  // let V be num of vertices
  vec<int> dep(V, -1);
  vec<bool> was(V, false);
  int t = 0;
  forn (0, V, 1) {
    if (!vis[i]) {
      vis[i] = true;
      dfs(i);
    }
  }
  reverse(all(dep));
}
```

```
dfs() {
  for (int i : graph[v]) {
    if (!vis[i])
      dfs(i);
  }
  t++;
  dep[t] = v;
}
```

summary of topological sorting

summary

- 3 ways to do topological sorting
  - pure dfs
  - pure bfs (using rings/indegrees)
  - dfs + departure time
- runtime is $O(V + E)$
- space complexity is $O(V)$

Strong Connectivity

intuition

strongly connected components (aka SCCs)

- subsets of vertices in a digraph where each vertex can be reached from every other vertex within the same subset
  - how do we find these SCCs?

a motivating problem

solution intuition

1. perform a dfs on every single vertex in the graph
2. after visiting all neighbors, append the current node to a stack
3. after a full dfs traversal of G, reverse the edges of the graph (ie, find the transpose of G)
    - dfs on a transposed graph ensures that we can only traverse within a strongly connected component
4. pop the stack one by one and perform dfs on the reversed graph
5. every complete dfs iteration from a single node will yield a strongly connected component

a motivating problem

algorithm intuition

1. init `component` and `components` vectors and stack S
2. **1st dfs traversal.** for $v \in G$
   - visit v
   - recursively visit all neighbors of v
   - post-order push v to S
3. reset `visited`
4. **2nd dfs traversal.** while $|S| > 0$
   - pop vertex s from S
   - perform dfs on s on transpose of G
     - visit s
     - let $component[s]$ be the current component number `num`
     - append s to the `num`th component in `components`
     - recursively visit s's neighbors
   - increment `num`

impl # 1: kosaraju's algorithm

algorithm pseudocode

```
main() {
  stack<int> S; bool was[n];
      int num;
  int component[n];
  vec<int> components[n];
  // 1st dfs
  forn (0, n, i++)
    if (!was[i]) dfs_1(i);
  // reset graph
  forn (0, n, i++)
    was[i] = false;
  // 2nd dfs
  while (!S.empty()) {
    int v = S.top();
    S.pop();
    if (!was[v]) {
      dfs_2(v);
      num++;
    }
  }
}
```

```
dfs_1(int v) {
  was[v] = true;
  for (int i : graph[v]) {
    if (!was[i]) dfs(i);
  }
  S.push(v);
}

dfs_2(int v) {
  component[v] = num;
  components[num].pb(v);
  was[v] = true;
  for (int i : graph_t[v]) {
    if (!was[i]) dfs(i);
  }
  S.push(v);
}
```

summary

kosaraju's algorithm

1. algorithm for finding strongly connected components
2. utilizes two dfs traversals on a graph and its transpose
3. additionally utilizes a stack
4. runtime: $O(V + E)$
5. space: $O(V)$

summary

kosaraju's algorithm is just one of the two most common SCC algorithms. from here, examine **tarjan's algorithm**, which improves upon kosaraju's algorithm by *removing the need to reverse the results*

1. focuses on low-link values (the id of the smallest node that is reachable from the current node when doing dfs, aka the root of the SCC) that makes it similar to topological sorting