Topological Sorting and Strong Connectivity

Randy Truong

rtruong@u.northwestern.edu

February 1, 2024

Outline

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 that develops feasible **course schedules**.

- A course schedule is an ordering of courses that a student can take
- A feasible course schedule is one that can be completed without any prerequisite issues

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 that develops feasible 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, perform dfs on v's neighbors
    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(i, 0, N) {
        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

- Replace a `was` vector with a `state` vector
  1. unvisited $\rightarrow 0$
  2. visited, but in current path $\rightarrow 1$
  3. completely visited (ie, not in current path) $\rightarrow 2$

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

algorithm intuition

1. for each course
    1. if the current course v we're looking at hasn't been visited, visit it
        1. mark down v as being on the current path
        2. visit all of v's descendants
    2. set v's state as to being completely visited
    3. add the current course to the schedule only after examining all of its descendants
2. reverse `results`

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

algorithm intuition

1. initialize a results array `results` $\leftarrow \{\}$
2. initialize state array `state` $\leftarrow \{\}$
3. iterate through all vertices $v \in G$
    1. if $state[v] = 0$, then perform dfs
        1. set $state[v] \leftarrow 1$
        2. perform dfs on v's descendants
    2. append current vertex v to `results`
    3. set $state[v] \leftarrow 2$
4. reverse `results`

impl #1: dfs-based

algorithm code

```
main() {
    vector<int> graph[100];
    vector<int> top_sort;
    bool was[N];
    forn(i, 0, N) {
        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: departure time algorithm

algorithm motivation

can we utilize properties of graphs + dfs in order to produce a topological sorting of G?

impl #2: departure time algorithm

algorithm motivation

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

impl #2: departure time algorithm

arrival and departure times

we can classify nodes within a graph using the following notion

1. **Arrival Time.** the time at which dfs first processes v
2. **Departure Time.** the time at which dfs finishes processing v (ie, comes back to it)

where time t is a counter, that represents the number of iterations in dfs

impl #3: departure time algorithm

algorithm intuition

using the notion of departure time, it follows that:

1. if vertex v_i and v_j exist in the same path where
2. $v_i \rightarrow v_j$, then $v_i.\texttt{depart} > v_j.\texttt{depart}$

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

impl #2: departure time 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 #2: departure time algorithm

algorithm pseudocode

```
main() {
  // let V be num of vertices
  vec<int> dep(V, -1);
  vec<bool> was(V, false);
  int t = 0;
  forn (i, 0, V) {
    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

- 2 primary ways to do topological sorting via dfs
  - pure dfs
  - dfs + departure time
- runtime is $O(V + E)$
- space complexity is $O(V)$
- other ways?
  - bfs-based topological sorting $\rightarrow$ **kahn's algorithm**
  - **one-sentence summary:** find a reasonable starting point, then just traverse graph in rings

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

from CSES "Planets and Kingdoms"

You are playing a video game that takes place in space. A player can explore various planets, which are connected via teleporters. Two planets belong to a kingdom if and only if there exists a route from one planet to the other and vice versa. Determine which planets belong to which kingdom.

a motivating problem

from CSES "Planets and Kingdoms"

You are playing a video game that takes place in space. A player can explore various planets, which are connected via teleporters. Two planets belong to a kingdom if and only if there exists a route from one planet to the other and vice versa. Determine which planets belong to which kingdom.

problem inituition

construct a directed graph G

- let each vertex v represent a planet
- let each edge e represent a "can teleport to" relationship
  - $a \to b$ means "a player on a can teleport to b"

a motivating problem

solution intuition

1. think of each kingdom (group of planets) as a node
   - topological sort of $G \rightarrow$ topological sort of kingdoms
   - if we can isolate the kingdoms, then we can just iterate through the planets $\rightarrow$ find cycle $\rightarrow$ identify kingdom
2. if two planets exist in the same kingdom, then
   - u must be reachable from v and
   - v must be reachable from u

but how can we isolate kingdoms (or SCCs)?

a motivating problem

solution intuition

- consider two versions of the graph:
  - original graph G and transpose G^T
    - if the graph of kingdoms is acyclic, then the transpose G^T guarantees that we don't traverse between kingdoms (given nature of digraphs)
- **proposed solution.** perform two dfs traversals:
  - on the first dfs traversal, generate a "topological sorting" of G
  - using the topological ordering, perform second dfs on G^T, where each completed dfs iteration yields a kingdom

a motivating problem

solution intuition

1. generate transpose of G for later use
2. perform **departure time algorithm** (denoted as `dfs1`), generating topological sorting via depart. times
3. iterate through topologically sorted vertices, perform regular dfs (denoted as `dfs2`) on transpose
   - every single node visited in a single `dfs` iteration must all belong to the same component

a motivating problem

algorithm intuition

1. init `component` (node's component #), `dep` (depart. time) vectors, and `num` (component #)
2. **1st dfs traversal (depart time algorithm).** for $v \in G$
   - if v is unvisited $\rightarrow$ perform dfs on descendants and record departure time in D
3. **2nd dfs traversal (normal dfs).** for $v \in D$
   - if v is not in a component $\rightarrow$ perform dfs
     - add v to current component
     - perform dfs on v's neighbors $\rightarrow$ add descendants to current component
   - after iterating through a single SCC, increment `num`

impl # 1: kosaraju's algorithm

algorithm pseudocode

```
main() {
  vec<int> = component(-1, n);
  vec<int> results;
  int num = 0;
  // 1st dfs
  forn (i, 0, n)
    if (!was[i]) dfs_1(i);
  reverse(all(results));
  // 2nd dfs
  for (int i: results) {
    if (component[v] == -1) {
        //if v not in a
        component
      dfs_2(v);
      num++;
    }
  }
}
```

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

dfs_2(int v, int num) {
  component[v] = num;
  for (int i : graph_t[v]) {
    if (component[v] == -1)
      dfs_2(i);
  }
}
```

summary

kosaraju's algorithm

1. algorithm for finding strongly connected components
2. utilizes two dfs traversals on a graph
   - first traversal is to topologically sort the SCCs via departure time algorithm
   - second traversal (on transpose) will find all nodes within a single component
3. runtime: $O(V + E)$
4. 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