JOI2014-2015 春合宿 Day2
道路整備 (Road Development)
解説

三谷 庸 (wo)

問題概要

- N 頂点のグラフがある
- 最初は辺はない
- グラフの更新クエリと出力クエリを処理せよ
- N <= 100,000
- Q <= 300,000

問題概要

- 更新クエリ (タイプ 1 のクエリ) (u, v)
  - u と v が非連結ならば、u と v をコスト 1 の辺で結ぶ
  - u と v が連結ならば、それらの最短路上の辺のコストをすべて 0 にする
- 出力クエリ (タイプ 2 のクエリ) (u, v)
  - u と v の最短路のコストを出力する

小課題 1 (10 点)

- N <= 1,000
- Q <= 3,000

小課題 1 (10 点)

- クエリごとに O(N) や O(N logN) かけてよさそう
- クエリごとにその時点のグラフ上で最短路問題を解くことで処理できる

小課題 1 (10 点)

- 更新クエリ (u, v)
- 頂点 u と頂点 v が非連結な場合は辺を張るだけ
- 頂点 u から頂点 v への最短路に辺 (x, y) が含まれる条件は以下のいずれかが成り立つこと
  - dis(u, x) + cost(x, y) + dis(y, v) = dis(u, v)
  - dis(u, y) + cost(y, x) + dis(x, v) = dis(u, v)

小課題 1 (10 点)

- 更新クエリ (u, v)
- u からの最短路と v からの最短路を求めた後、すべての辺について u から v への最短路に含まれるか判定し、含まれるならばコストを 0 にする

小課題 1 (10 点)

- 出力クエリ (u, v)
- 最短路問題を解いて結果を出力すればよい

グラフの形は？

- 小課題 2 以降は N, Q がとても多い
- 特殊な構造を見つける必要がある

- できるグラフの形について考察する

グラフの形は？

- グラフの更新クエリを再検討
- u と v が非連結ならば、辺を張る
- u と v が連結ならば、・・・のコストを 0 にする

グラフの形は？

- グラフの更新クエリを再検討
- u と v が非連結ならば、u と v の間に辺を張る
- u と v が連結ならば、・・・のコストを 0 にする
- u と v が連結ならば、u と v の間には**辺を張らない**

グラフの形は？

- グラフの更新クエリを再検討

- よって、グラフに閉路はできない
- 最終的なグラフの形は森になる (非連結な場合もあります)

グラフの形は？

- 更新クエリで「すべての最短路に対して」などと言っているが、実際には最短路は一つしかない

小課題 2 (25 点)

- 全ての更新クエリが来てから出力クエリがくる
- 全ての更新クエリが来た後のグラフの状態が分かればよい

辺のコストは？

- 最終状態において頂点 u と頂点 v の間にコスト 0 の辺が張られている条件を考える
  - u と v が非連結な状態で更新クエリ (u, v) が呼ばれる
  - その後、あるコストを 0 にするクエリ (w, x) に対して、w から x への最短路で辺 (u, v) を用いる
- もう少しわかりやすく言い換えられないか？

小課題 2 (25 点)

- 全ての更新クエリ (u, v) に対して、頂点 u と頂点 v の間に辺を張る
- 頂点 u と頂点 v の間をコスト 0 で行き来できる条件は、このグラフ上で頂点 u から頂点 v に 2 通り以上で行けること

小課題 2 (25 点)

- 逆に言うと、このグラフ上で辺 (u, v) のコストが 1 である条件は、頂点 u から頂点 v へ 1 通りの方法でしか行けないこと

小課題 2 (25 点)

- 逆に言うと、このグラフ上で辺 (u, v) のコストが 1 である条件は、頂点 u から頂点 v へ 1 通りの方法でしか行けないこと
- グラフ理論の用語でいうと、辺 (u, v) が橋になっていること

橋・二重辺連結成分

- グラフの辺 (u, v) が橋であるとは、辺 (u, v) を取り除いた時にグラフの連結成分が増えることをいう
- 橋でない辺でつながった成分を二重辺連結成分という
- これは DFS を用いて O(N) で検出できる
- 詳しくは調べてください

小課題 2 (25 点)

- 全ての更新クエリ (u, v) に対して (u, v) に辺を張ったグラフを考える
- 二重辺連結成分分解する (多重辺に注意)
- その結果は森になる
- 森の上のパスの長さを求めたい

最近共通祖先 (LCA)

- 木の根を (適当に一つ) 決める
- 頂点 u と頂点 v の共通の先祖であって、最も深い (u や v に最も近い) ものを最近共通祖先 (LCA) という
- O(log N) 時間で求まる
- 詳しくはプログラミングコンテストチャレンジブック (蟻本) などを参照

小課題 2 (25 点)

- 頂点 u と頂点 v (が属する二重辺連結成分) のLCA を求めると、u と v の距離は dep(u) + dep(v) – dep(lca) * 2 と高速に求まる

- これで出力クエリが処理できた

グラフの形は？ (続き)

- 辺のコストに関する考察は後回しにして最終的にできる森を構成する
- T[i] = 1 となるクエリをすべて見て行って、A[i] と B[i] が非連結ならばそれらの間に辺を張ることを i = 1, 2, …, Q に対して順に行う

グラフの形は？ (続き)

- クエリはオフライン (最初にすべて見れる) なのでこのようなことができる
- 今後、この森の上で考える
- つまり、最初にクエリをすべて見て森を構成した後、もう一度最初からクエリを見て出力クエリに答えていく

グラフの形は？ (続き)

- 更新クエリ (u, v)
  - (u, v) が森の辺になっている場合は無視
  - そうでない場合、u から v へのパス上の辺のコストをすべて 0 にする
  - 森の作り方より、このクエリよりも後に追加される辺が u, v 間の最短路に影響することはない
- 出力クエリ (u, v)
  - u から v へのパス上の辺のコストの和を求める

グラフの形は？ (続き)

- 以下の解説では次のことを仮定します
  - できる森は木になる
  - 出力クエリ (u, v) が来るとき u と v は連結である

- 森が木でない (非連結である) 時はそれぞれに対して処理すればよい
- 出力クエリの二頂点の連結性は Union Find 木で判定できる

小課題 3 (25 点)

- 全ての更新クエリ (u, v) に対して、次のいずれかが成り立つ
  - u と v は非連結
  - u と v は距離 200 以下

小課題 3 (25 点)

- 言い換えると、
  - 全ての「辺のコストを 0 にするクエリ (u, v)」に対して、u と v の距離は 200 以下
- 辺のコストを 0 にするクエリは一つずつ辺を見ていけばよい
- 出力クエリを高速に処理する必要がある
- 木のパス上の辺のコストの和を高速に求めたい

小課題 3 (25 点)

- まとめると、やりたいことは
  - 木の辺のコストを 1 から 0 にする
  - 木の上のパス上の辺のコストの和を求める

- ほぼ同じ問題が蟻本にも載っています

小課題 3 (25 点)

- 出力クエリ
  - dis(u, v) = dis(r, u) + dis(r, v) – dis(r, lca) * 2
  - 各頂点 v に対して根と頂点 v の間の辺のコストの和が分かれば、任意のパスのコストが分かる

小課題 3 (25 点)

- 更新クエリ
  - 辺を列挙する
  - u と v から 1 つずつ親にさかのぼっていけばよい
  - LCA を求めてもよい

小課題 3 (25 点)

- 辺 (c, p) (p は c の親)のコストを 1 から 0 にすると dis(r, v) はどう変わるか
- v が c の子孫でないとき
  - 何も変わらない
- v が c の子孫であるとき
  - dis(r, v) は 1 減る

- 「c を根とする部分木」をうまく扱いたい

DFS 順序

- 木を根から DFS する
- 頂点を、初めて訪れた時間が早い順に並べる
- この時、「頂点 v を根とする部分木」は、「頂点 v から始まるある区間」に対応する

DFS 順序

```
c = 0
dfs(v, p)
  pre[v] = c
  c = c + 1
  for each u in (v に隣接している p 以外の頂点)
    dfs(u, v)
  pos[v] = c

dfs(r, -1)
```

DFS 順序

- この時、頂点 v は pre[v] 番目に訪れたことになる
  - このような順序のつけ方を行きがけ順と言います
- v を根とする部分木の頂点は[pre[v], pos[v]) に含まれる

実現したいことを再確認

- 木がある → 数列がある
  - pre[v] 番目の要素は根から頂点 v までの距離 dis(r, v)
  - 初期値は v の深さ
- v を根とする部分木の各頂点と根との距離が 1 減る → 数列のある(v に対応する) 区間の各要素が 1 減る
- v と根の距離を求める → 数列のある (v に対応する) 要素を求める

Segment Tree

- これは、Segment Tree を用いて実現できる
- Segment Tree の各ノードには、「そのノードに対応する区間に含まれる要素すべてに一様に足された数値」を持っておく

Segment Tree

[1,6) の要素を -1 したいときは、以下のノードに -1 と書きこむ

<table>
<tr><td colspan="8">[0, 8)</td></tr>
<tr><td colspan="4">[0, 4)</td><td colspan="4">[4, 8)</td></tr>
<tr><td colspan="2">[0, 2)</td><td colspan="2">[2, 4)</td><td colspan="2">[4, 6)</td><td colspan="2">[6, 8)</td></tr>
<tr><td>[0, 1)</td><td>[1, 2)</td><td>[2, 3)</td><td>[3, 4)</td><td>[4, 5)</td><td>[5, 6)</td><td>[6, 7)</td><td>[7, 8)</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
</table>

Segment Tree

要素 2 の値を知りたいときは、以下のノードの値の和を求める

<table>
<tr><td colspan="8">[0, 8)</td></tr>
<tr><td colspan="4">[0, 4)</td><td colspan="4">[4, 8)</td></tr>
<tr><td colspan="2">[0, 2)</td><td colspan="2">[2, 4)</td><td colspan="2">[4, 6)</td><td colspan="2">[6, 8)</td></tr>
<tr><td>[0, 1)</td><td>[1, 2)</td><td>[2, 3)</td><td>[3, 4)</td><td>[4, 5)</td><td>[5, 6)</td><td>[6, 7)</td><td>[7, 8)</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
</table>

Segment Tree

- このようにすると、区間の更新クエリもある要素の値を求めるクエリも O(log N) で処理できる

小課題 3 (25 点)

- これらのことを使うと小課題 3 が解ける

- 初期化
  - 木の頂点に DFS 順序をつける
  - Segment Tree を用意し、pre[v] 番目の要素を v の深さで初期化する

小課題 3 (25 点)

- 更新クエリ (u, v)
  - u から v へのパスに含まれる辺(c, p)を一つずつ見ていく
    - u および v から LCA までたどっていけばよい
  - cost(c, p) = 1 であった場合
    - cost(c, p) を 0 に更新
    - Segment Tree を用いて c を根とする部分木に対応する区間 [pre[c], pos[c]) の要素を -1 する

小課題 3 (25 点)

- 更新クエリ
  - 一つの辺の処理に O(log N) 時間かかるので、一つのクエリ (u, v) の処理に O((u, v) の距離 * log N) 時間かかる
  - 距離が 200 以下であることから十分高速

小課題 3 (25 点)

- 出力クエリ (u, v)
  - u と v の LCA を求める
  - dis(r, u) + dis(r, v) – dis(r, lca) * 2 を出力
    - dis(r, v) はSegment Tree の pre[v] 番目の要素として求められる
  - 一回あたり O(log N) 時間
  - 十分高速

小課題 4 (25 点)

- 出力クエリは 200 個以下である

- 出力クエリ (u, v) に対しては、u から v へ一本ずつ辺をたどっていけばよい
  - どの辺のコストが 1 でどの辺のコストが 0 かを持っておく必要がある

小課題 4 (25 点)

- この小課題は小課題 3 とは独立に解けます
  - 小課題 3 では更新クエリが良心的で、出力クエリを高速化
  - 小課題 4 では出力クエリが良心的で、更新クエリを高速化

- このようなこともあるので、小課題の制約も全部読みましょう

小課題 4 (25 点)

- 辺のコストは、
  - 最初 1
  - 更新クエリが来ると 0 になる
  - その後ずっと 0

小課題 4 (25 点)

- 辺のコストが変わるのは高々 N – 1 回
- それぞれの更新クエリ (u, v) に対して、毎回 u から v へのパス上の辺をすべて見るのは無駄
- コスト 0 の辺をうまく飛ばしたい

小課題 4 (25 点)

- やりたいことは以下の通り
  - 頂点 v からその祖先の頂点 a に向かうパスの上のコスト 1 の辺を列挙する
  - 辺のコストを 0 にする

Union Find 木

- Union Find 木は集合の併合および二つの要素が同じ集合に属するかの判定ができるデータ構造です
- うまく工夫することでこの問題に応用できます

Union Find 木

- コスト 0 の辺でつながっている頂点を Union Find でくっつける
- Union Find の各成分の根はその成分に属する頂点のうち最も浅いものにする
  - 普通の Union Find では二点 x, y をくっつける時にどちらを親にするかは自由だが、今回は元の木で浅い方を根にする

Union Find 木

- これでもクエリあたりのならし計算量が O(log N) になることが知られています
  - もちろんパス圧縮は行う
  - 普通の Union Find 木で、どちらを親にするかを rank などで管理すると O(α(N)) になります

小課題 4 (25 点)

- Union Find の v を含む成分の根を t とすると、(t, par[t]) はv から親に向かってたどっていったときに最初に出てくるコスト 1 の辺になっている
  - t が木全体の根でない場合

小課題 4 (25 点)

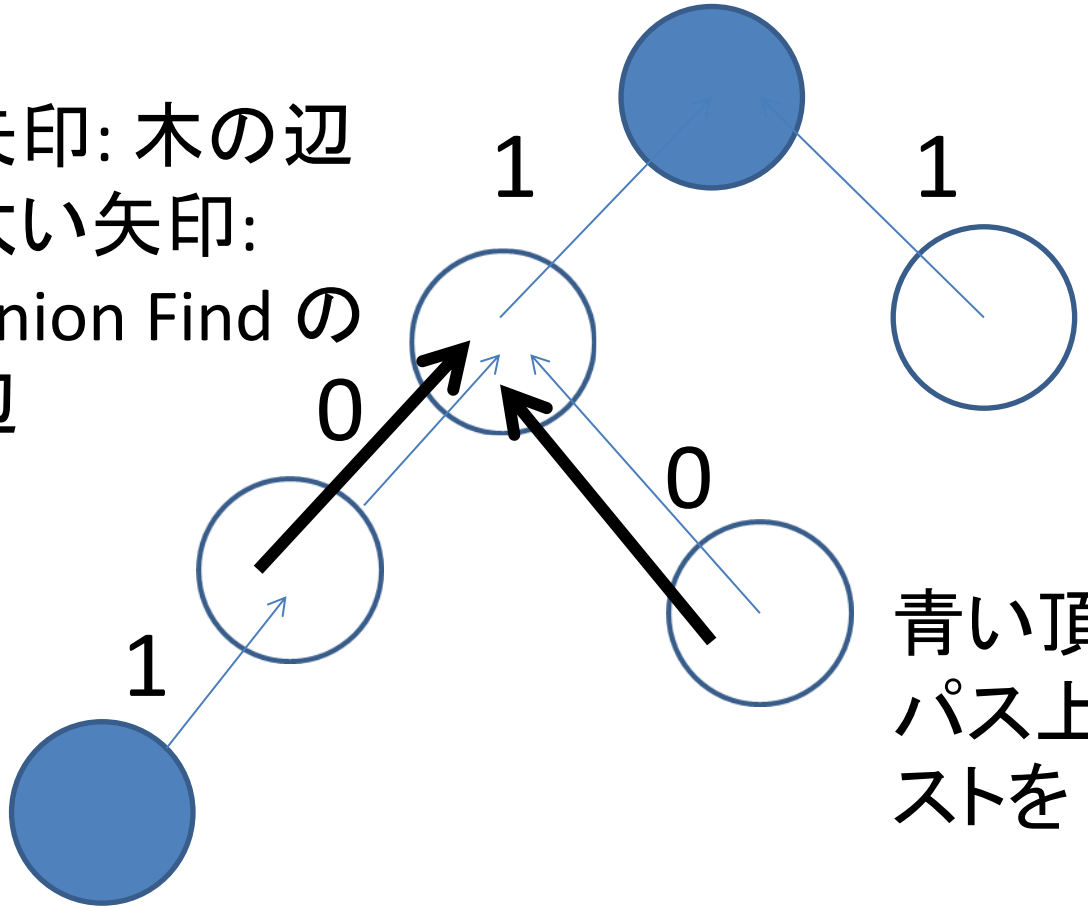
- Union Find を以上のようにして用いると、小課題 4 は次のように解ける

小課題 4 (25 点)

- 更新クエリ
  - v からその先祖 a へのパス上の辺のコストを 0 にする処理
  - v からはじめて以下のことを繰り返す
    - その頂点を含むコスト 0 の辺でつながった成分の根に行く
    - a より浅くなったら終了
    - 今の頂点と親を結ぶ辺のコストを0 にする
    - 親に移動
    - Union Find を更新

小課題 4 (25 点)

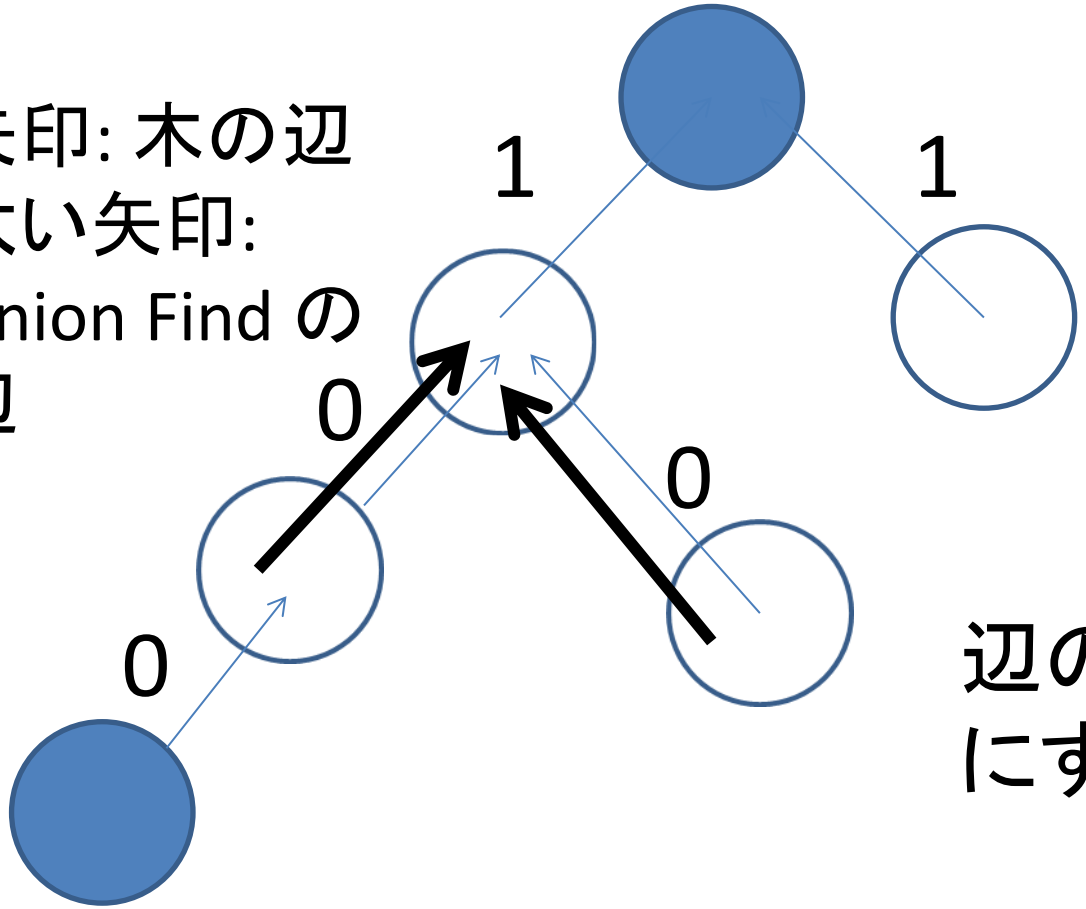
数字: 辺のコスト
矢印: 木の辺
太い矢印: Union Find の辺



青い頂点を結ぶパス上の辺のコストを 0 にする

小課題 4 (25 点)

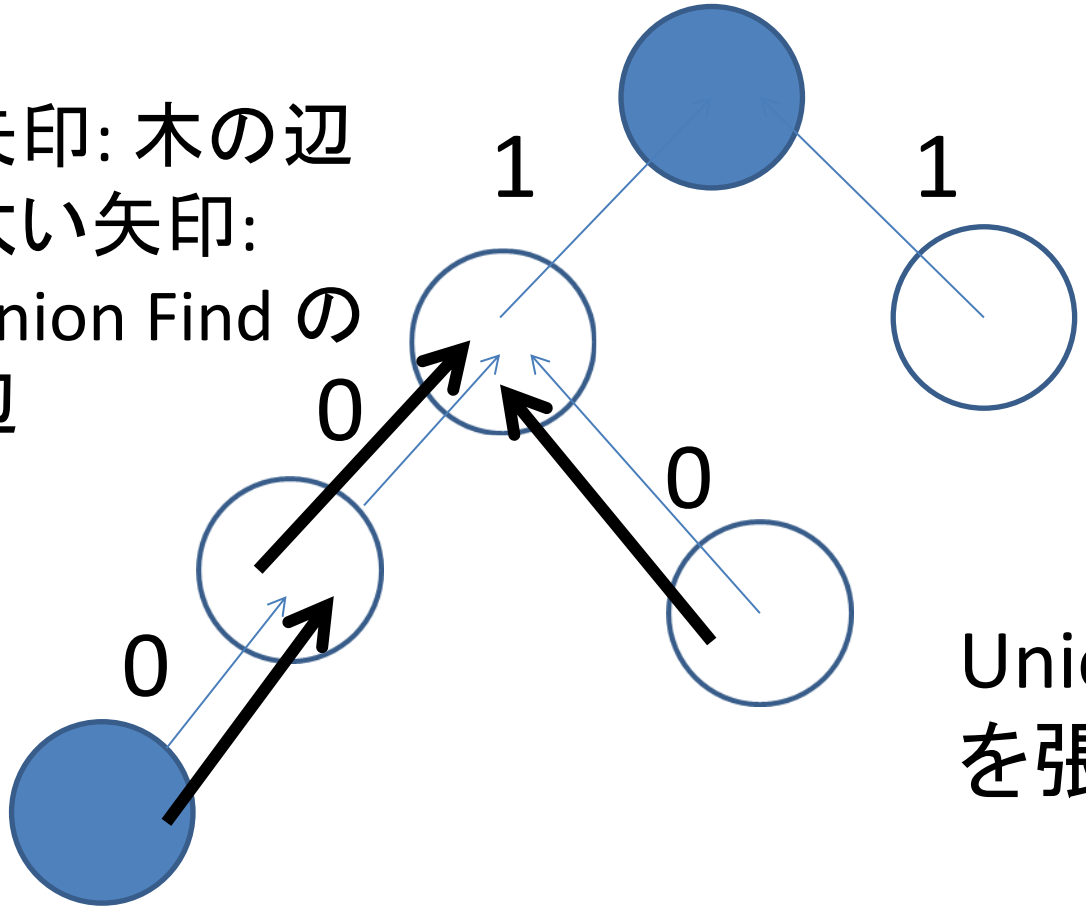
数字: 辺のコスト
矢印: 木の辺
太い矢印: Union Find の辺



辺のコストを 0 にする

小課題 4 (25 点)

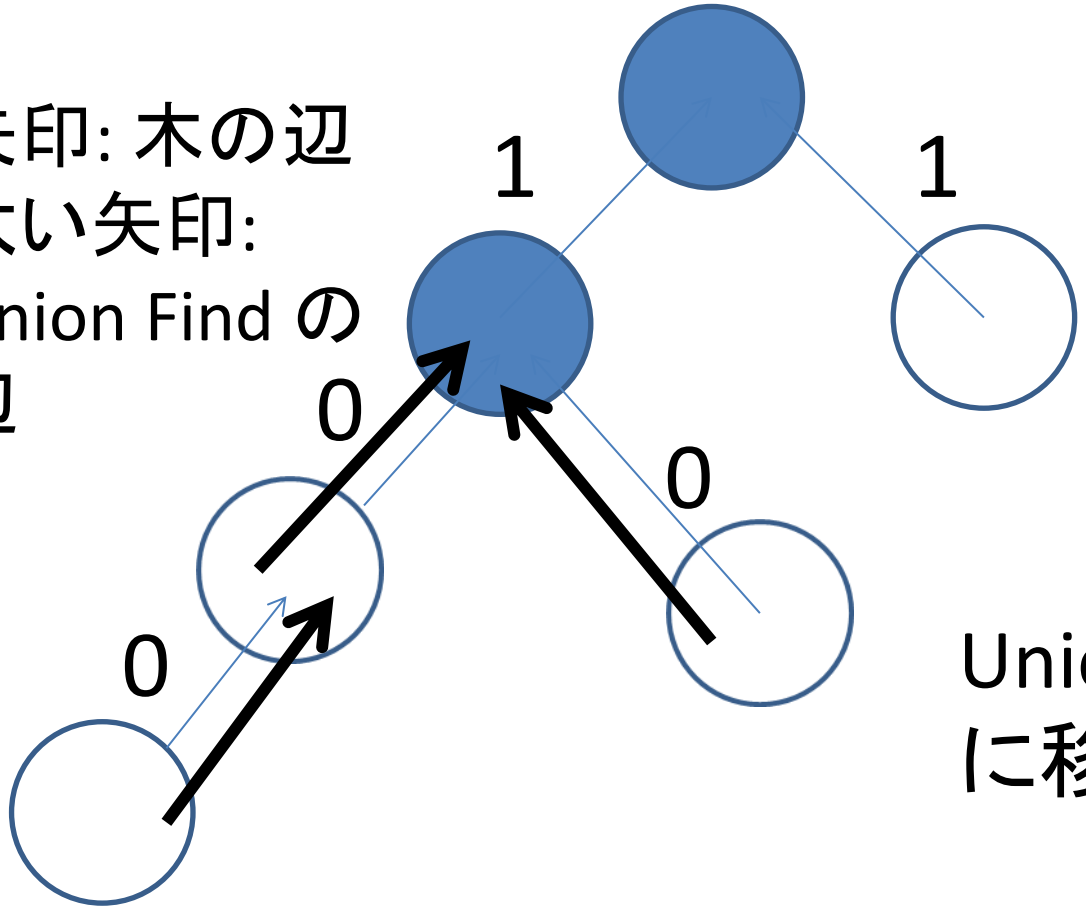
数字: 辺のコスト
矢印: 木の辺
太い矢印: Union Find の辺



Union Find で辺を張る

小課題 4 (25 点)

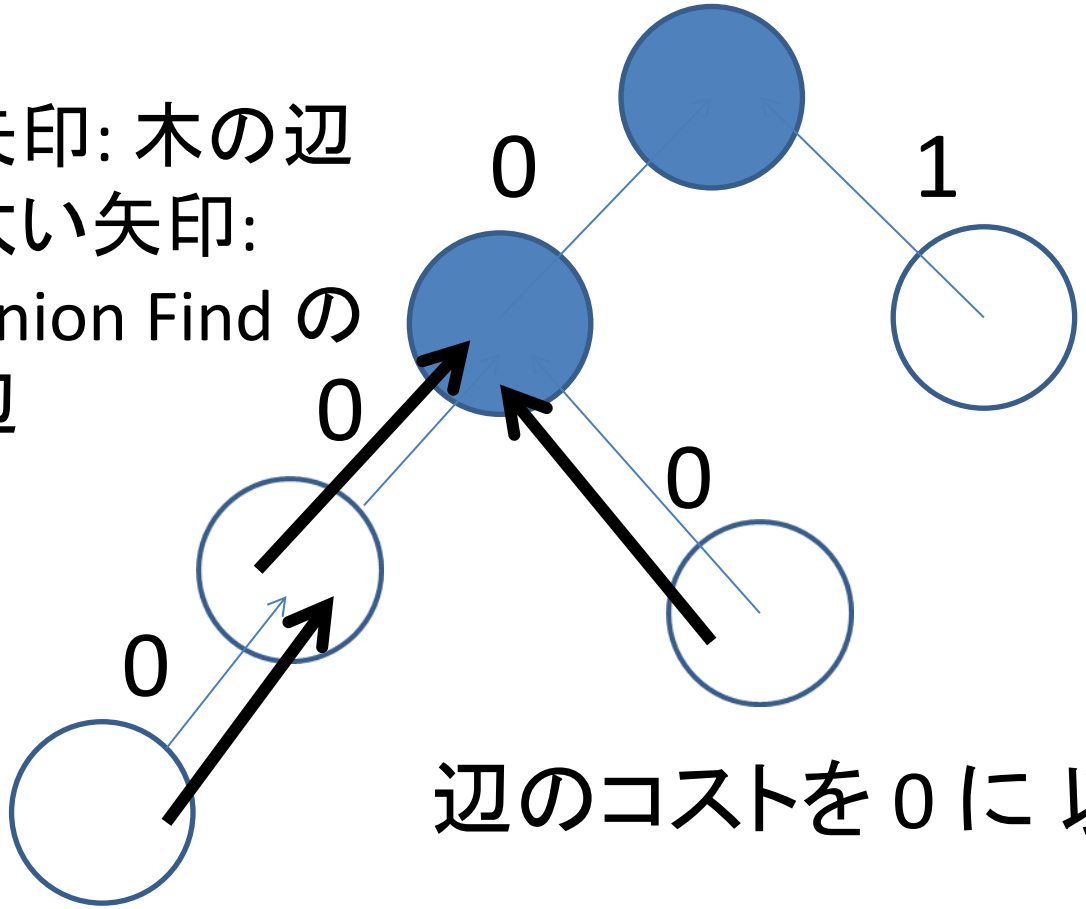
数字: 辺のコスト
矢印: 木の辺
太い矢印: Union Find の辺



Union Find の根に移動

小課題 4 (25 点)

数字: 辺のコスト
矢印: 木の辺
太い矢印: Union Find の辺



辺のコストを 0 に 以下繰り返し

小課題 4 (25 点)

- 辺のコストの更新は O(N) 回しかないことから、更新クエリは全体で O(N log N) 以下の時間で処理できる

小課題 4 (25 点)

- 出力クエリ
  - 毎回 O(N) かけて 木のパスを求める
  - 最短路上の辺のコストを一つずつ足していく
  - 全部で 200 回以下しか来ないのでこれで間に合う

小課題 5 (15 点)

- 追加の制限はない
- と言っても、ここまでくれば解けているようなものです

小課題 5 (15 点)

- 小課題 3 で、出力クエリが高速に処理できるようになった
- 小課題 4 で、更新クエリが高速に処理できるようになった
- これらの高速化を片方しか行ってはいけない、なんていう理由はない
- 両方やれば小課題 5 が解ける

別解

- Heavy Light Decomposition
  - 木の各辺を Heavy Edge と Light Edge に分ける
  - 木を Heavy Edge でつながったパスをくっつけたものとみなす
  - このとき、木の任意のパスがO(log N) 個の Heavy Path (Heavy Edge からなるパス) に分けられる
  - Segment Tree と組み合わせることで、木のパスのいろいろな操作が O(log^2 N) で実現できる
- 詳しくは調べてください

別解

- 実装はそれなりに重いです
- 書ける自信がない人は手を出すかよく考えましょう

まとめ

- この問題では以下のテクニックを使いました
  - 橋・二重辺連結成分 (部分点解法)
  - 最近共通祖先 (LCA)
  - 木の頂点の DFS 順序
  - Segment Tree
  - Union Find (向きをつける) ですでに処理したところを飛ばす

まとめ

- よく知らなかった人は、蟻本やその他の資料で勉強してください
- どれも重要なテクニックです

得点分布