@drken 2019年05月22日に更新

重み付き Union-Find 木とそれが使える問題のまとめ、および、牛ゲーについて

アルゴリズム 最適化 競技プログラミング データ構造 組合せ最適化

はじめに

ABC 087 D - People on a Line にて、重みつきUnion-Find 木を貼るだけで解ける問題が出題されたので簡単にまとめてみます。なお、通常の Union-Find 木については以下にわかりやすくまとまっています。

- Union find (素集合データ構造) - SlideShare

重みつき Union-Find 木が使える問題として以下のものがあります。これらの問題の簡単な解説を最後につけています。

- ABC 087 D People on a Line
- AOJ 1330 Never Wait for Weights
- AOJ 2207 Consistet Unit System
- AOJ 2427 ほそながいところ

重みつき UnionFind 木の処理

普通の UnionFind 木のサポートする処理は

クエリ	処理内容
merge(x, y)	x を含むグループと y を含むグループをマージする
issame(x, y)	x と y が同じグループにいるかどうかを判定する

ですが、重みつきUnionFind木は少し発展させて、各ノード v に重み weight(v) を持たせ、**ノード間の距離**も管理するようなものになっています。

クエリ	処理内容
merge(x, y, w)	weight(y) = weight(x) + w となるように x と y をマージする
issame(x, y)	x と y が同じグループにいるかどうかを判定する
diff(x, y)	x と y とが同じグループにいるとき、weight(y) - weight(x) をリターンする

をサポートするものになります。

- x と y がすでに同じグループにいるときに merge(x, y, w) を呼ぶ
- x と y が同じグループにいないときに diff(x, y) を呼ぶ

については未定義動作で、この場合の処理をどうするかはライブラリ作者の裁量に委ねられそうです。なお、通常の Union-Find 木の実装は以下を想定しています (このスライドを参照)。これに「重み」を付け加えることを考えます。

```
struct UnionFind {
    vector<int> par; // 親ノード
    vector<int> rank; // ランク

    UnionFind(int n = 1) {
        init(n);
    }

    void init(int n = 1) {
        par.resize(n); rank.resize(n);
        for (int i = 0; i < n; ++i) par[i] = i, rank[i] = 0;
```

```
    }

    int root(int x) {
        if (par[x] == x) {
            return x;
        }
        else {
            int r = root(par[x]);
            return par[x] = r;
        }
    }

    bool issame(int x, int y) {
        return root(x) == root(y);
    }

    bool merge(int x, int y) {
        x = root(x); y = root(y);
        if (x == y) return false;
        if (rank[x] < rank[y]) swap(x, y);
        if (rank[x] == rank[y]) ++rank[x];
        par[y] = x;
        return true;
    }
};
```

ポテンシャルつき Union-Find 木と呼ぶべき？

この Union-Find を重みつき Union-Find を呼ぶことに違和感を感じている人も多いようです。kirika_comp さんはポテンシャルつき Union-Find と呼ぶことを提唱しています。

https://twitter.com/kirika_comp/status/958410332120137728

理由は**差分制約系**に登場する**ポテンシャル**に相当するものを Union-Find の各頂点がもつからですね。差分制約系は、**牛ゲー**、**重みつきUnion-Find**、**最小カット**、**最小費用流の双対問題**といった具合に話題が豊富なので、いずれ記事にまとめてみたいです。

重みつき Union-Find 木の実装

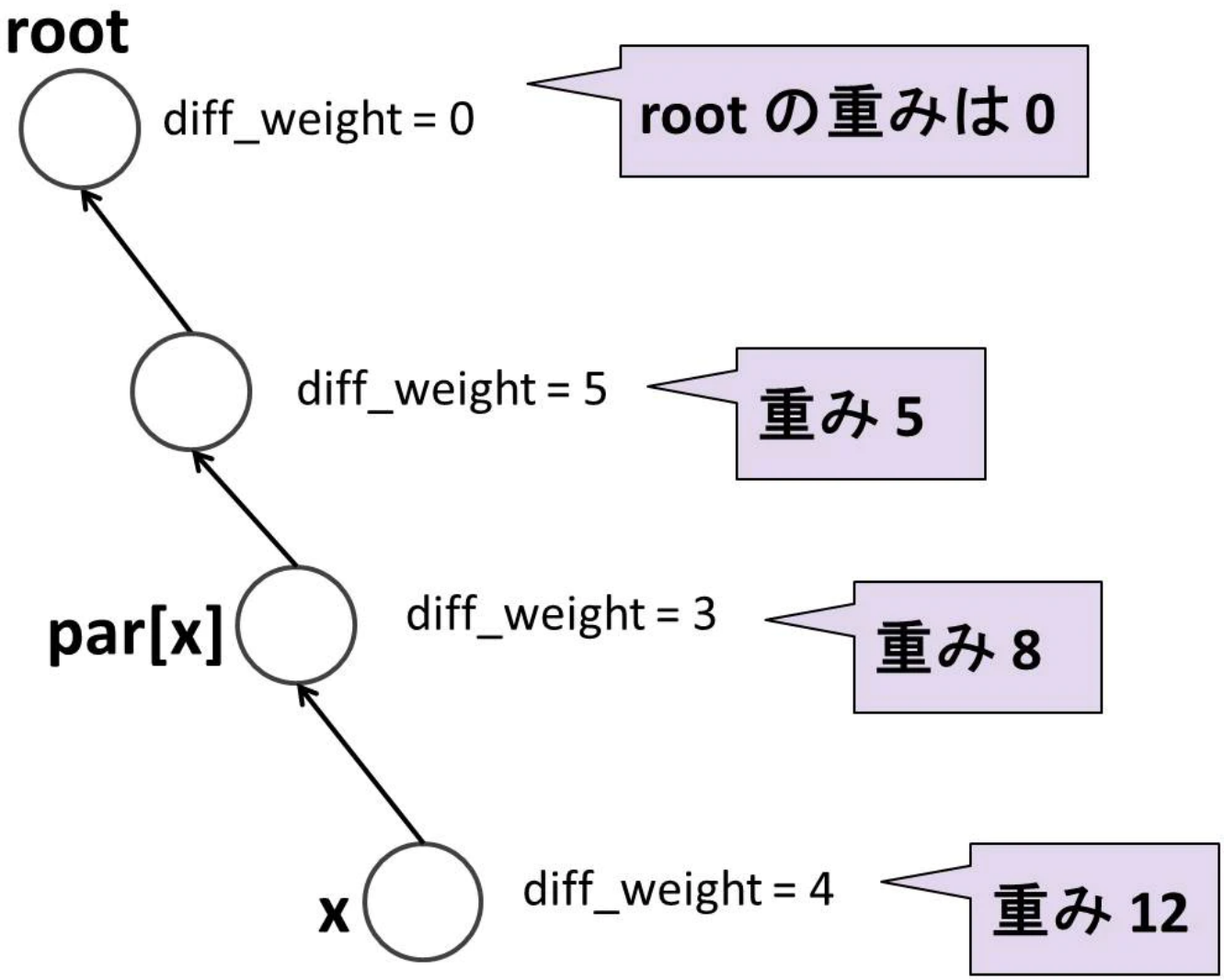
概ね普通の Union-Find 木と一緒です。普通の Union-Find 木は各ノード v に

- 親ノード par[v]
- ランク rank[v] （rank を用いない簡易版の Union-Find もあります)

を持たせていますが、これに追加で

- 親ノードとの値の差分 diff_weight[v]

を持たせます。



経路圧縮

Union-Find 木では元々 issame(x, y) を実施するときに**経路圧縮**という操作を行っていました。issame(x, y) の原理は「共通の root をもつかどうか」を判定するというものですが、親をたどって root に辿り着くまでの間に経路圧縮と呼ばれる操作を行っています。

重みつき Union-Find では、経路圧縮のついでに差分重み diff_weight も更新します。root(x) 関数を呼ばれると、x は次々と親を親をと遡りながら、最終的には root の下に接続されます。その都度 diff_weight の累積和をとっていきます。root の重みは 0 と決めているので、root の下に接続された時点で、x の重みは diff_weight[x] そのものとなっています。

```
int root(int x) {
  if (par[x] == x) {
    return x;
  }
  else {
    int r = root(par[x]);
    diff_weight[x] += diff_weight[par[x]]; // 累積和をとる
    return par[x] = r;
  }
}
```

各ノードの重みの取得, diff の計算

単純に経路圧縮をした上で diff_weight[x] を返せばよいです。また diff の計算は weight 関数の差をとればよいです。

```
int weight(int x) {
  root(x); // 経路圧縮
  return diff_weight[x];
}

int diff(int x, int y) {
  return weight(y) - weight(x);
}
```

併合

merge(x, y) 操作です。weightに関する行がなければごく普通の Union-Find 木です。
最初の

w += weight(x); w -= weight(y);

と補正するところが少しわかりにくいかもしれません。merge(x, y) 操作では、元々の x と y との間に辺を繋ぐのではなく、root(x) と root(y) の間をつなぐので、つなぐべき辺の重みは

w ではなく修正が必要になります。

```
// weight(y) - weight(x) = w となるように merge する
bool merge(int x, int y, int w) {
  // x と y それぞれについて、 root との重み差分を補正
  w += weight(x); w -= weight(y);

  // x と y の root へ (x と y が既につながっていたら false を返すようにした)
  x = root(x); y = root(y);
  if (x == y) return false;

  // rank[x] >= rank[y] となるように x と y を swap (それに合わせて w も符号反転します)
  if (rank[x] < rank[y]) swap(x, y), w = -w;

  // y (のroot) を x (のroot) の下にくっつける
  if (rank[x] == rank[y]) ++rank[x];
  par[y] = x;

   // x が y の親になるので、x と y の差分を diff_weight[y] に記録
  diff_weight[y] = w;

  return true;
}
```

普通の Union-Find 木 (rank 付き、このスライド の P.17 参照) ではこんな感じでした。

```
void merge(int x, int y) {
  // x と y の root へ
  x = root(x); y = root(y);
  if (x == y) return;

  // rank[x] >= rank[y] となるように x と y を swap
  if (rank[x] < rank[y]) swap(x, y);

  // y (のroot) を x (のroot) の下にくっつける
  if (rank[x] == rank[y]) ++rank[x];
  par[y] = x;
}
```

重み付き Union-Find 木ライブラリ

以上の処理をまとめて、以下のようなライブラリができました。
ここまでの説明では重みは int 型と考えて来ましたが、template を用いて少し抽象化します。
template 引数には、重みのとるべき型を指定します。コンストラクタ引数は以下の通りです:

- n: 要素数
- SUM_UNITY: 基本的に 0 を入れます

SUM_UNITY について補足ですが、重み付き UnionFind の重みは一般にアーベル群 (足し算と引き算ができる代数系) を乗せられます。その場合は、SUM_UNITY には「アーベル群の単位元」を入れます。

```
#include <iostream>
#include <vector>
using namespace std;

template<class Abel> struct UnionFind {
    vector<int> par;
    vector<int> rank;
    vector<Abel> diff_weight;

    UnionFind(int n = 1, Abel SUM_UNITY = 0) {
        init(n, SUM_UNITY);
    }

    void init(int n = 1, Abel SUM_UNITY = 0) {
        par.resize(n); rank.resize(n); diff_weight.resize(n);
        for (int i = 0; i < n; ++i) par[i] = i, rank[i] = 0, diff_weight[i] = SUM_UNITY;
    }

    int root(int x) {
        if (par[x] == x) {
            return x;
        }
        else {
            int r = root(par[x]);
            diff_weight[x] += diff_weight[par[x]];
            return par[x] = r;
        }
    }

    Abel weight(int x) {
        root(x);
        return diff_weight[x];
    }
```

```
    bool issame(int x, int y) {
        return root(x) == root(y);
    }

    bool merge(int x, int y, Abel w) {
        w += weight(x); w -= weight(y);
        x = root(x); y = root(y);
        if (x == y) return false;
        if (rank[x] < rank[y]) swap(x, y), w = -w;
        if (rank[x] == rank[y]) ++rank[x];
        par[y] = x;
        diff_weight[y] = w;
        return true;
    }

    Abel diff(int x, int y) {
        return weight(y) - weight(x);
    }
};
```

差分制約系, 牛ゲーについて

重みつき Union-Find 木は

Maximize:

特になし

Subject to:

$$x[r[i]] - x[l[i]] = d[i]$$

の形の最適化問題 (差分制約型の一次方程式系) を解くことのできる方法論と言えますが、これを不等式に拡張して

Maximize:

$$x[T] - x[S]$$

Subject to:

$$x[r[i]] - x[l[i]] \leq d[i]$$

という形の最適化問題が考えられます。これはいわゆる**牛ゲー**と呼ばれるものです。牛ゲーは

- 頂点 $l[i]$ から 頂点 $r[i]$ へ重み $d[i]$ の枝を張る
- S から T への最短路が答え

というもので、例外処理としては

- 負閉路あり: **実行可能解なし**
- S から T へたどり着けない: **非有界** ($x[S]$ と $x[T]$ をいくらでも引き離せる)

となります。なお牛ゲーは **2 頂点間の最短経路問題の双対問題**になっています。

また、牛ゲーの枠組みは前者を含みです。なぜなら、

$$x[r[i]] - x[l[i]] = d[i]$$
$$\Leftrightarrow$$
$$x[r[i]] - x[l[i]] \leq d[i]$$
$$x[l[i]] - x[r[i]] \leq -d[i]$$

が成立するからです。ABC 087 D - People on a Line を最短経路問題を解く解法も考えられますが、牛ゲーからの輸入と思うと自然に思えて来ます。

問題例

ABC 087 D People on a Line

【問題概要】

整数変数 $x[0], x[1], \ldots, x[N-1]$ のうち、すべての $i = 0, 1, \ldots, M-1$ に対して

$x[R[i]] - x[L[i]] = D[i]$

を満たすような x が存在するかどうかを判定せよ

【制約】

- $1 \leq N \leq 10^5$
- $1 \leq M \leq 2 * 10^5$

【解法】

重みつき Union-Find 木のライブラリがあれば、貼るだけです。

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

template<class Abel> struct UnionFind {
    vector<int> par;
    vector<int> rank;
    vector<Abel> diff_weight;

    UnionFind(int n = 1, Abel SUM_UNITY = 0) {
        init(n, SUM_UNITY);
    }

    void init(int n = 1, Abel SUM_UNITY = 0) {
        par.resize(n); rank.resize(n); diff_weight.resize(n);
        for (int i = 0; i < n; ++i) par[i] = i, rank[i] = 0, diff_weight[i] = SUM_UNITY;
    }

    int root(int x) {
        if (par[x] == x) {
            return x;
        }
        else {
            int r = root(par[x]);
            diff_weight[x] += diff_weight[par[x]];
            return par[x] = r;
        }
    }
```

```
    Abel weight(int x) {
        root(x);
        return diff_weight[x];
    }

    bool issame(int x, int y) {
        return root(x) == root(y);
    }

    bool merge(int x, int y, Abel w) {
        w += weight(x); w -= weight(y);
        x = root(x); y = root(y);
        if (x == y) return false;
        if (rank[x] < rank[y]) swap(x, y), w = -w;
        if (rank[x] == rank[y]) ++rank[x];
        par[y] = x;
        diff_weight[y] = w;
        return true;
    }

    Abel diff(int x, int y) {
        return weight(y) - weight(x);
    }
};

int main() {
    int N, M;
    cin >> N >> M;
    UnionFind<int> uf(N);
    for (int i = 0; i < M; ++i) {
        int l, r, d;
        cin >> l >> r >> d;
        --l, --r;
        if (uf.issame(l, r)) {
            int diff = uf.diff(l, r);
            if (diff != d) {
                cout << "No" << endl;
                return 0;
            }
        }
        else {
            uf.merge(l, r, d);
        }
    }
    cout << "Yes" << endl;
}
```

AOJ 1330 Never Wait for Weights

【問題概要】

N 個の値 $x[i]$ を特定したい。次の 2 種類のクエリが与えられるので適切に処理せよ。

- a, b, w が与えられて、$x[b] - x[a] = w$ という情報が与えられる
- a, b を与えるので、今までの情報で $x[b] - x[a]$ が特定できるならその値を出力し、特定できないならば "UNKNOWN" と出力せよ

【制約】

- $2 \leq N \leq 10^5$
- $1 \leq M \leq 10^5$

【解法】

ほとんど ABC 087 D People on a Line と同じです。

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdio>
using namespace std;
template<class Abel> struct UnionFind {
	vector<int> par;
	vector<int> rank;
	vector<Abel> diff_weight;
	UnionFind(int n = 1, Abel SUM_UNITY = 0) {
		init(n, SUM_UNITY);
	}
	void init(int n = 1, Abel SUM_UNITY = 0) {
		par.resize(n); rank.resize(n); diff_weight.resize(n);
		for (int i = 0; i < n; ++i) par[i] = i, rank[i] = 0, diff_weight[i] = SUM_U
	}
	int root(int x) {
		if (par[x] == x) {
			return x;
		}
```

```
                else {
                        int r = root(par[x]);
                        diff_weight[x] += diff_weight[par[x]];
                        return par[x] = r;
                }
        }
        Abel weight(int x) {
                root(x);
                return diff_weight[x];
        }
        bool issame(int x, int y) {
                return root(x) == root(y);
        }
        bool merge(int x, int y, Abel w) {
                w += weight(x); w -= weight(y);
                x = root(x); y = root(y);
                if (x == y) return false;
                if (rank[x] < rank[y]) swap(x, y), w = -w;
                if (rank[x] == rank[y]) ++rank[x];
                par[y] = x;
                diff_weight[y] = w;
                return true;
        }
        Abel diff(int x, int y) {
                return weight(y) - weight(x);
        }
};
int main() {
        int N, M;
        while (cin >> N >> M) {
                if (N == 0) break;
                UnionFind<int> uf(N);
                for (int i = 0; i < M; ++i) {
                        char c; int a, b, w;
                        cin >> c;
                        if (c == '!') {
                                cin >> a >> b >> w; --a, --b;
                                uf.merge(a, b, w);
                        }
                        else {
                                cin >> a >> b; --a, --b;
                                if (!uf.issame(a, b)) puts("UNKNOWN");
                                else cout << uf.diff(a, b) << endl;
                        }
                }
        }
}
```

AOJ 2207 無矛盾な単位系

【問題概要】

1 kilometre = 10^3 metre

1 megametre = 10^3 kilometre

1 metre = 10^-6 megametre

1 terametre = 10^3 gigametre

1 petametre = 10^3 terametre

1 gigametre = 10^-6 petametre

1 metre = 10^-15 petametre

のような単位換算関係式があたえられる。これが無矛盾かどうかを判定せよ。

【制約】

- 1 <= 単位換算式の数 <= 100

【解法】

色んな解法がありそうですが、重み付き Union-Find 木を使うと楽です。

```
#include <iostream>
#include <sstream>
#include <vector>
#include <string>
#include <map>
using namespace std;

template<class Abel> struct UnionFind {
    vector<int> par;
    vector<int> rank;
    vector<Abel> diff_weight;

    UnionFind(int n = 1, Abel SUM_UNITY = 0) {
        init(n, SUM_UNITY);
    }

    void init(int n = 1, Abel SUM_UNITY = 0) {
```

```
        par.resize(n); rank.resize(n); diff_weight.resize(n);
        for (int i = 0; i < n; ++i) par[i] = i, rank[i] = 0, diff_weight[i] = SUM_UNITY;
    }

    int root(int x) {
        if (par[x] == x) {
            return x;
        }
        else {
            int r = root(par[x]);
            diff_weight[x] += diff_weight[par[x]];
            return par[x] = r;
        }
    }

    Abel weight(int x) {
        root(x);
        return diff_weight[x];
    }

    bool issame(int x, int y) {
        return root(x) == root(y);
    }

    bool merge(int x, int y, Abel w) {
        w += weight(x); w -= weight(y);
        x = root(x); y = root(y);
        if (x == y) return false;
        if (rank[x] < rank[y]) swap(x, y), w = -w;
        if (rank[x] == rank[y]) ++rank[x];
        par[y] = x;
        diff_weight[y] = w;
        return true;
    }

    Abel diff(int x, int y) {
        return weight(y) - weight(x);
    }
};

int main() {
    int N;
    while (cin >> N) {
        if (N == 0) break;
        bool ok = true;
        UnionFind<int> uf(200); // 単位個数は最悪で 200 個
        map<string,int> str2id; // 単位 -> id の対応
        int final_id = 0;
```

```
        for (int i = 0; i < N; ++i) {
            string ichi, unit1, equal, val, unit2;
            cin >> ichi >> unit1 >> equal >> val >> unit2;

            // 1個目の単位の番号
            if (!str2id.count(unit1)) {
                str2id[unit1] = final_id++;
            }
            int id1 = str2id[unit1];

            // 2個目の単位の番号
            if (!str2id.count(unit2)) {
                str2id[unit2] = final_id++;
            }
            int id2 = str2id[unit2];

            // 単位換算の倍率の取得
            stringstream si(val.substr(3));
            int diff;
            si >> diff; // "1 kilometre = 10^3 metre" の 3 の部分を取得

            // 重み付き Union-Find 木の処理
            if (uf.issame(id1, id2)) {
                int curdiff = uf.diff(id1, id2);
                if (diff != curdiff) ok = false;
            }
            else {
                uf.merge(id1, id2, diff);
            }
        }

        if (ok) puts("Yes");
        else puts("No");
    }
}
```

AOJ 2427 ほそながいところ

【問題概要】

2 つ以上の馬車がすれ違うこともできない細長い道路 ($dist$ km) を N 個の馬車が端から端まで移動します。それぞれの馬車は 1km あたり $S[i]$ 分で等速で止まることなく移動します。

細長い道路には m 箇所だけ「少し広いところ」があり (出発点から $D[0], D[1], \ldots, D[m-1]$ km 地点) 、そこでなら、ある馬車が他の馬車を追い越すことができます。
N 個の馬車はどの 2 つも出発時刻を 1 分以上空ける必要があります。N 個の馬車の出発時刻として考えられるもののうち、馬車の追い越しが生じる場所が「少し広いところ」だけになるようなものを考えて、最初の馬車が出発してから最後の馬車が到着するまでの時間を最小にせよ。

【制約】

- $1 \leq dist \leq 10^8$
- $1 \leq n \leq 5$
- $0 \leq m \leq 5$
- $1 \leq S[i] \leq 100$
- $0 < D[i] < dist$

【解法】
今回挙げた 4 つの問題の中では最も難しいです。しばしば「想定解法が全探索でも易しいとは限らない」問題例として語られています。

馬車のペア (i, j) (最大で nC2 <= 10 通り) について、以下のパターンを全探索します:

- j が i より 1 分遅れでスタート
- j と i が同時にゴール
- j が i を地点 k (0 <= k <= m-1) で追い越す (j の方が速い場合のみ): m 通りある
- それ以外

最悪で 8 通りのパターンがあるので、最悪で 10^8 通りになります。これらそれぞれのパターンについて、n 個の馬車のそれぞれのスタート時刻の間隔としてふさわしいものを重み付き Union-Find 木によって求めます。最後に以下の項目の整合性を check します:

- どの 2 つの馬車も、index が大きい方が後から出発していること
- どの 2 つの馬車も「少し広いところ」以外では追い越していないこと
- どの 3 つの馬車も「少し広いところ」を同時に通過しないこと (結構なコーナーケース！！！)

また注意点として、馬車 i と馬車 j のスタート時刻の間隔が不定となってしまうことがありうるが、そのような状況が発生したら最適解ではあり得ない (条件を満たしながら解を改善可能) ので無視してしまってよいです。

```
#include <iostream>
#include <sstream>
#include <vector>
#include <string>
#include <map>
using namespace std;

template<class Abel> struct UnionFind {
    vector<int> par;
    vector<int> rank;
    vector<Abel> diff_weight;

    UnionFind(int n = 1, Abel SUM_UNITY = 0) {
        init(n, SUM_UNITY);
    }

    void init(int n = 1, Abel SUM_UNITY = 0) {
        par.resize(n); rank.resize(n); diff_weight.resize(n);
        for (int i = 0; i < n; ++i) par[i] = i, rank[i] = 0, diff_weight[i] = SUM_UNITY;
    }

    int root(int x) {
        if (par[x] == x) {
            return x;
        }
        else {
            int r = root(par[x]);
            diff_weight[x] += diff_weight[par[x]];
            return par[x] = r;
        }
    }

    Abel weight(int x) {
        root(x);
        return diff_weight[x];
    }

    bool issame(int x, int y) {
        return root(x) == root(y);
    }

    bool merge(int x, int y, Abel w) {
        w += weight(x); w -= weight(y);
```

```
        x = root(x); y = root(y);
        if (x == y) return false;
        if (rank[x] < rank[y]) swap(x, y), w = -w;
        if (rank[x] == rank[y]) ++rank[x];
        par[y] = x;
        diff_weight[y] = w;
        return true;
    }

    Abel diff(int x, int y) {
        return weight(y) - weight(x);
    }
};

typedef pair<int,int> pint;

long long dist;
int n, m;
long long S[11], D[11];

const long long INF = 1LL<<60;
long long res = INF; // 答えを更新していく

// 各馬車が「どこですれ違うか」
void check(vector<long long> diffs) {
    UnionFind<long long> uf(n);
    int iter = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            long long diff = diffs[iter++];
            if (diff == -1) continue;

            if (uf.issame(i, j)) {
                long long curdiff = uf.diff(i, j);
                if (diff != curdiff) return;
            }
            else {
                uf.merge(i, j, diff);
            }
        }
    }

    // 暫定回を求める
    long long start = 0, goal = 0;
    for (int i = 0; i < n; ++i) {
        long long start_i = start + uf.diff(0, i);
        long long goal_i = start_i + S[i] * dist;
        if (start > start_i) start = start_i;
```

```
        if (goal < goal_i) goal = goal_i;
    }

    // 整合性を確認
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (!uf.issame(i, j)) return; // 繋がっていない箇所があったら明らかに最適でないので r
            if (uf.diff(i, j) < 1) return; // j が i の 1分後よりも早く出ていたらダメ

            // 追い越している場合はどこで追い越してるか
            long long start_i = uf.diff(0, i);
            long long goal_i = start_i + S[i] * dist;
            long long start_j = uf.diff(0, j);
            long long goal_j = start_j + S[j] * dist;
            if (goal_j < goal_i) {
                int pl = -1;
                for (int k = 0; k < m; ++k) {
                    long long it = start_i + S[i] * D[k];
                    long long jt = start_j + S[j] * D[k];
                    if (it == jt) {
                        pl = k;
                    }
                }
                if (pl == -1) return; // 「少し広いところ以外で交わったらダメ
            }
        }
    }

    // 本当に m 箇所それぞれについて、3 個以上の馬車が交わることがないか確認
    for (int i = 0; i < m; ++i) {
        map<long long,int> ma;
        for (int j = 0; j < n; ++j) {
            long long t = uf.diff(0, j) + S[j] * D[i];
            ma[t]++;
        }
        for (map<long long,int>::iterator it = ma.begin(); it != ma.end(); ++it) {
            if (it->second >= 3) return;
        }
    }

    // 確認を終えたら OK
    if (res > goal - start) res = goal - start;
}

void dfs(vector<long long> diffs, int i, int j) {
    if (i == n-1) {
        check(diffs);
        return;
```

```
    }

    // 馬車 i, j が何分でゴールするか
    long long total_i = S[i] * dist;
    long long total_j = S[j] * dist;

    // 次の index
    int ni = i;
    int nj = j+1;
    if (nj == n) {
        ++ni;
        nj = ni+1;
    }

    // weight(j) - weight(i) = 1 (j が i より 1 分遅れスタートで、途中で抜かさない場合
    if (total_j >= total_i - 1) {
        diffs.push_back(1);
        dfs(diffs, ni, nj);
        diffs.pop_back();
    }

    // ゴールでピッタリ
    {
        long long diff = total_i - total_j;
        if (diff >= 1) {
            diffs.push_back(diff);
            dfs(diffs, ni, nj);
            diffs.pop_back();
        }
    }

    // m 箇所それぞれ
    if (S[i] != S[j]) {
        for (int k = 0; k < m; ++k) {
            long long diff = (S[i] - S[j]) * D[k];
            if (diff >= 1) {
                diffs.push_back(diff);
                dfs(diffs, ni, nj);
                diffs.pop_back();
            }
        }
    }

    // それ以外
    diffs.push_back(-1);
    dfs(diffs, ni, nj);
    diffs.pop_back();
}
```

```cpp
int main() {
    while (cin >> dist) {
        cin >> n;
        for (int i = 0; i < n; ++i) cin >> S[i];
        cin >> m;
        for (int i = 0; i < m; ++i) cin >> D[i];

        vector<long long> diffs;
        dfs(diffs, 0, 1);

        cout << res << endl;
    }
}
```


けんちょん (Otsuki) @drken

NTTデータ数理システムでリサーチャーをしている大槻です。 機械学習やアルゴリズムに関して面白いと思ったことを記事にしていきたいと思います。記事へのリンク等についてはお気軽にしていただいて大丈夫です。よろしくお願いします。

http://www.msi.co.jp/

フォロー

株式会社NTTデータ数理システム

数理科学とコンピュータサイエンスの融合！！

http://www.msi.co.jp/

コメント

この記事にコメントはありません。

あなたもコメントしてみませんか :)

ユーザ登録

すでにアカウントを持っている方はログイン