Introduction to Heuristics Contest 解説

wata

1　取り組み方

最適化問題などを題材にスコアを伸ばしていくタイプのコンテスト (マラソンコンテストとよく呼ばれています) の開催を今後 AtCoder で増やしていくにあたってアンケートを取ったところ、取り組み方が分からないという声が多くありました。そこで、新規の参加者の方々が入門しやすいように、シンプルなスケジューリング問題を題材として、このタイプの問題に対する取り組み方と典型的な手法を紹介します。

まずは問題文を読んでください。普段の ABC 等のコンテストと大きく違う点は、最適解を求める必要はなく、プログラムの出力した解の質に応じてスコアが得られるという点です。このため、「正解」を一つ見つけたら終わりではなく、試行錯誤を重ね改良を繰り返しながらスコアを徐々に上げていくことになります。また、もう一つ特徴的な点として、単に入力の最大サイズが指定されるのではなく[*1]、入力の生成方法が指定されているという点があります[*2]。一般に、最適化アルゴリズムの性能というのは一次元的な指標で評価出来るのもではなく、問題が同じであっても入力の傾向によってその性能は大きく変化します。このため、あらゆる入力における最悪ケースを考えるのではなく、目的の入力に適したアルゴリズムを開発することが重要になります。今回の問題の場合は、入力は単に一様乱数で生成されているため、入力の特徴を活かすことは難しいですが、問題によっては分布を偏らせる[*3]などによって入力の特徴を活用することがより重要となっている場合もあります。ビジュアライザが用意されている場合も多いので、なにか活用できる特徴がないかを考察してみることも重要です。

問題文を読んだら次に、入出力部分の処理と、とりあえず有効な出力を出すプログラム (ランダム出力などでも十分です)、そしてスコア計算プログラムの作成に移りましょう。今回のようにローカル実行用のスコア計算プログラムが提供される場合も多くありますが、ABC などのコンテストと比べて問題の仕様が複雑な場合も多く、スコア計算が一致するかの検証を通して正しく仕様を理解出来ているかを確認するのにも役立ちますし、解答プログラムを作成する際やデバッグ時にもソースコードが流用出来ることが多いため、よほど複雑なスコア計算で無い限りは作っておいて損はないと思います。以下は Rust でのスコア (満足度) 計算の実装例です。

[*1] 今回の問題で、日数 D が最大値ではなく固定の値として指定されたことに違和感を感じたかもしれません。入力サイズがテストデータごとに異なる場合、サイズに応じて手法を切り替えるなどによってスコアを伸ばすことが出来たり、プログラムで用いるパラメータの調整が大変になったり、得られるスコアにバラつきが生じて一部の入力におけるスコアが全体の順位付けにおいて支配的になってしまう場合があったりするなど、難しい点が増えます。このため、特に短期間のコンテストでは固定サイズとすることもよくあります。

[*2] 入力の生成方法が明示的に与えられるかどうかはコンテストの種類によりますが、全くの未知の入力に対してのスコアを競わせるということは滅多にありません。他のよくある形式としては、順位付けに用いる入力を全て公開したり (この場合必然的に手元での実行となります)、もしくはより実践的な問題では現実の入力を用いてその一部のみを事前に公開しておき、未公開の入力で最終順位を決定するなどがあります。

[*3] 現実世界の問題に対する入力は、ソーシャルグラフの次数分布の例など、しばしば非常に偏りがあります。

```rust
struct Input {
    D: usize,
    s: Vec<Vec<i64>>,
    c: Vec<i64>
}
fn compute_score(input: &Input, out: &Vec<usize>) -> i64 {
    let mut score = 0;
    let mut last = vec![0; 26];
    for d in 0..out.len() {
        last[out[d]] = d + 1;
        for i in 0..26 {
            score -= (d + 1 - last[i]) as i64 * input.c[i];
        }
        score += input.s[d][out[d]];
    }
    score
}
```

さて、準備が整ったら解法を考えるフェーズに移りましょう。基本的には大まかな方針を立て (貪欲法を用いる、局所探索を用いるなど)、自分のスコアと順位表とを比較して方針の妥当性を確かめつつ (自分より上に似たようなスコアで大勢が固まっているならおそらく到達しやすい別の良い方針がありそう。上位の参加者が一気に飛び抜けたので重要かつ奇抜なアイデアが必要かもしれない。など)、思いついた改良案の中から重要そうな順に実装して実験結果から考察を深めていく形になります。今回の問題のように典型的な手法が使える問題から、奇抜なアイデアが必要となる問題まで様々です。この解説では典型的な手法の解説と改良の仕方、ならびにどのような問題に有用であるかを解説していきます。章タイトルの右上にある*の数は難易度を表しています。

2　貪欲法

この問題は、1 日目に開催するコンテストを決める、2 日目に開催するコンテストを決める、...という具合に順番に解を構築していくことが出来、更に構築した部分的な解の良さ (満足度) も計算が出来ます。 そこで、d 日目にはその日の終了時点における満足度が一番高くなるコンテストを選択する[*4]、というアルゴリズムを考えることが出来ます。このような、その瞬間でのベストな選択を繰り返す「貪欲法」は、既に ABC などのアルゴリズムコンテストで出会ったことがあるかもしれません。 貪欲法は問題によっては最適解を達成することが保証出来ますが、残念ながらこの問題に対しては最適解を与えるとは限りません。 しかし、最適解は得られずとも、多くの場合にそれなりに良い解を求めることは出来ます。貪欲法は汎用性が高く実装が簡単な上に、他の手法に比べ比較的高速に動作することも多く、巨大な入力を処理する必要がある場合には最有力の手法となることも多々あります。以下は先程実装したスコア計算関数を用いた[*5]Rust での実装例です。提出したところ 62,634,806 のスコアが得られました。

```rust
fn solve(input: &Input) -> Vec<usize> {
    let mut out = vec![];
```

[*4] より単純に $s[d][i]$ の最も大きい i を選ぶという方法も考えられますが、これだと $c[i]$ による満足度の減少を完全に無視することになってしまい、低いスコアしか得ることが出来ません。

[*5] d 日目のコンテストを決める際には一からスコア計算をする必要はなく、d 日目での満足度の増減だけを評価することで計算の高速化出来ますが、速度が問題とならない場合は手を抜いても問題ありません。

```
    for _ in 0..input.D {
        let mut max_score = i64::min_value();
        let mut best_i = 0;
        for i in 0..26 {
            out.push(i);
            let score = compute_score(&input, &out);
            if max_score < score {
                max_score = score;
                best_i = i;
            }
            out.pop();
        }
        out.push(best_i);
    }
    out
}
```

さて、この貪欲法を改良してみましょう。目的はあくまで最終的な満足度の最大化であり、途中段階での満足度を高めることが必ずしもよいとは限りません。例えば、多少その時点では損をしても、満足度の下がりやすい (つまり、$c[i]$ の大きい) コンテストを開催したほうが後々得をするという可能性はあります。そこで、最終的な目的関数値 (今回の場合は D 日目終了時点での満足度) との相関がより高くなるような評価関数を設計し、途中段階での目的関数値の代わりに評価関数値を最大化する行動を選択してみましょう。たとえば今回の問題の場合、以下のような考察をすることが出来ます。

> 毎日一つしかコンテストを開催できないため、各コンテストについて次の開催日は平均 $26/2 = 13$ 日後くらいとなるであろう。従って、d 日目終了時点での満足度よりも、その後 $d + 13$ 日目まで何もコンテストを開催しなかったとした場合における $d + 13$ 日目終了時点での満足度の方が、最終的な満足度との相関が高いのではないだろうか。

実際には前回の開催からの期間が長いコンテストや $c[i]$ の大きいコンテストを先に開催することになるため、平均をとった 13 を用いるのは悲観的で、もう少し小さい値 k を使用したほうが良いスコアが得られます。実行時間に余裕があるので k を 0 から 26 まで全部試した中で一番スコアの良い解を出力したところ、104,195,466 のスコアが得られ、先程の単純な貪欲法による 62,634,806 から大きく改善できました。

```
fn evaluate(input: &Input, out: &Vec<usize>, k: usize) -> i64 {
    let mut score = 0;
    let mut last = vec![0; 26];
    for d in 0..out.len() {
        last[out[d]] = d + 1;
        for i in 0..26 {
            score -= (d + 1 - last[i]) as i64 * input.c[i];
        }
        score += input.s[d][out[d]];
    }
    for d in out.len()..(out.len() + k).min(input.D) {
        for i in 0..26 {
            score -= (d + 1 - last[i]) as i64 * input.c[i];
        }
    }
    score
```

```rust
}

fn solve(input: &Input, k: usize) -> Vec<usize> {
    let mut out = vec![];
    for _ in 0..input.D {
        let mut max_score = i64::min_value();
        let mut best_i = 0;
        for i in 0..26 {
            out.push(i);
            let score = evaluate(&input, &out, k);
            if max_score < score {
                max_score = score;
                best_i = i;
            }
            out.pop();
        }
        out.push(best_i);
    }
    out
}
```

機械学習？

ゲーム AI のような評価関数の作成が難しい問題では、機械学習による評価関数の作成が行われており、研究の進んだ分野では人手で作った評価関数を遥かに凌ぐ性能となっています。今の所、今回の問題のような純粋な最適化問題において、評価関数を機械学習で作成してコンテストで上位を取ったという話はあまり聞きませんが、将来的にはその方向性も発展するのかもしれません。

3　ビーム探索*

貪欲法は高速な反面、制限時間を余らせてしまうことが多々あります。先程の例のように、評価関数を少し変えて何度も実行するというのも一つの手ですが、そのような独立に実行できるタイプの時間の使い方は、実行時間が増えれば増えるほどスコアの伸びが鈍化していき不利になっていきます。貪欲法がうまく適用出来る問題に対しては、より時間を有効活用出来る「ビームサーチ (Beam Search)」という典型手法がよく用いられます。貪欲法では各段階に置いて評価関数値の最も高い解一つに絞りましたが、ビームサーチでは評価関数値の高い解を複数残して進めていきます。

(後で追記)

4　局所探索

出来るだけ良い解を求めるための非常に強力な典型手法の一つが「局所探索法 (Local Search)」です。この手法では、闇雲に一から解を探すのではなく、既に見つけた解を少し変化させる操作によってより良い解を探索していきます。最も単純で基礎的な局所探索法は山登り法です。山登り法では現在の解を少し変化させてみて、解が良くなっていれば更新し、悪くなってしまったら元に戻します。この操作を繰り返し反復することで、時間をかけて徐々に解の質を高めていきます。例えばこの問題の場合、変形操作として「日付 d とコンテ

ストタイプ q をランダムに選び、d 日目に開催するコンテストをタイプ q に変更する」というものを考えることが出来ます。以下は Rust での実装例です。完全にランダムな初期解からスタートし、制限時間が来るまで山登り法の反復を繰り返しています。提出したところ 78,879,391 のスコアが得られました。

```rust
fn solve(input: &Input) -> Vec<usize> {
    const TL: f64 = 1.9;
    let mut rng = rand_pcg::Pcg64Mcg::new(890482);
    let mut out = (0..input.D).map(|_| rng.gen_range(0, 26)).collect::<Vec<_>>(); // random solution
    let mut score = compute_score(&input, &out);
    while get_time() < TL {
        let d = rng.gen_range(0, input.D);
        let q = rng.gen_range(0, 26);
        let old = out[d];
        out[d] = q;
        let new_score = compute_score(&input, &out);
        if score > new_score {
            out[d] = old;
        } else {
            score = new_score;
        }
    }
    out
}
```

局所探索法を用いる上で最も重要なことは、どのように変形を行うかの設計です。

- 変化させる量が小さすぎるとすぐに行き止まり (局所最適解) に陥ってしまい、逆に、変化させる量が大きすぎると闇雲に探す状態に近くなって、改善できる確率が低くなってしまう。
- 反復回数を増やすために、変形後のスコアが高速に計算出来ることが望ましい。

まずは 1 つ目の点を扱います。上記の局所探索法を、ベスト解が更新されたタイミングで経過時刻をデバッグ出力をするように書き換えて実行してみると、動かし始めて数十ミリ秒の時点以降全くベスト解が更新されていないことに気が付きます。これはどんなペア (d, q) を選んでもこれ以上改善が出来ないような局所最適解にはまってしまっているからです。実はこの「日付 d とコンテストタイプ q をランダムに選び、d 日目に開催するコンテストをタイプ q に変更する」という変形操作は、局所最適解にはまりやすい性質を持っています。なぜでしょうか？

d 日目のコンテストをタイプ i から他のタイプに変更することを考えましょう。d 日目の一つ前のタイプ i の開催日を d_0、d 日目の一つ後のタイプ i の開催日を d_1 とすると、元々は $d - d_0$ 日と $d_1 - d$ 日の開催間隔であったのが、変更後には 2 つの和である $d_1 - d_0$ の開催間隔となります。スコアはコンテストの開催間隔に対して二乗のオーダーで減少するため、このような操作でスコアを改善できるのは元々の開催間隔が極めて短かった場合しかありません。局所改善を繰り返すことでそのような短期間に連続する開催は解消されていき、どのコンテストも開催を取りやめると開催間隔が長くなりすぎてしまい、一切変更の出来ない状態になってしまうわけです。

このような局所最適解から脱出するためには、「開催日の近い 2 つのコンテストの開催日を入れ替える」という変形操作が有効であることに気が付きます。この操作ならばコンテストの開催間隔が大きく変化しないため、スコアを改善できる見込みがあります。では、先程の「一点変更」の変形操作を、この「二点スワップ」の変形操作に置き換えれば解決でしょうか？実は単純に置き換えるだけだとまた別の問題が発生します。ス

ワップ操作では各コンテストタイプごとの開催回数が変化しないため、スワップ操作だけでは絶対に到達出来ない解が存在し、探索空間が限られてしまうのです。

実は「一点変更」と「二点スワップ」の両方を採用することで問題は解決します。各イテレーション毎にランダムにどちらの変形を採用するかを決定すればよいのです。以下は Rust での実装例です。スワップには距離が 16 以下のものを選んでいます。提出したところ 112,203,174 点となり、一点変更のみを用いた場合の 78,879,391 から大幅にスコアが伸びました。

```rust
fn solve(input: &Input) -> Vec<usize> {
    const TL: f64 = 1.9;
    let mut rng = rand_pcg::Pcg64Mcg::new(890482);
    let mut out = (0..input.D).map(|_| rng.gen_range(0, 26)).collect::<Vec<_>>(); // random solution
    let mut score = compute_score(&input, &out);
    while get_time() < TL {
        if rng.gen_bool(0.5) {
            let d = rng.gen_range(0, input.D);
            let q = rng.gen_range(0, 26);
            let old = out[d];
            out[d] = q;
            let new_score = compute_score(&input, &out);
            if score > new_score {
                out[d] = old;
            } else {
                score = new_score;
            }
        } else {
            let d1 = rng.gen_range(0, input.D - 1);
            let d2 = rng.gen_range(d1 + 1, (d1 + 16).min(input.D));
            out.swap(d1, d2);
            let new_score = compute_score(&input, &out);
            if score > new_score {
                out.swap(d1, d2);
            } else {
                score = new_score;
            }
        }
    }
    out
}
```

一方で、これでもまだ早期に局所最適解に至っていることが確認できます。局所最適解にはまりにくくするためには、より変化量を大きく (2 点ではなく 3 点スワップにするなど) したり、悪化する変形を確率的に受け入れることで優れた解に到達しやすくした「焼きなまし法」や、変形操作をうまく設計することで巨大な近傍 (現在の解から一回の変形操作で移れる解全体のことを指します) の中から効率的に改善解を探し出す「巨大近傍法」などがよく用いられます。この解説の残りの 2 章では後者の 2 つの手法について解説していきます。

初期解

局所探索を用いるにあたって、初期解をどうするかは難しい問題です。制限時間が厳しい場合には、初期解の与え方が非常に重要になり、この解説での実装例のようにランダムな解からスタートするのではなく、貪欲法などの他の高速な手法で求めた解を初期解としたほうが良いことが多いです。一方で、そのような解は脱出しにくい局所最適にはまっていることもあるため注意が必要です。また、短時間で局所最適解にはまってしまい制限時間に余裕がある場合には、得られた局所最適解をランダムに少しだけ変更することで無理やり局所最適から脱出させ (kick といいます)、それを新たな初期解として再度局所探索を行うという手法もあります。この場合、局所探索で用いた変形操作と同じ変形操作で kick を行うと、すぐに元の局所最適解に戻って来てしまうため、kick には異なる変形操作を用いるようにしましょう。

さて、次に 2 つ目の点、変形後の高速なスコア計算について扱います。変形後のスコアは先に示した実装例のように、一からスコア計算を行うことで計算が可能ですが、変化のあった部分だけに着目して変形前からの差分を計算することで、より高速に行うことができる可能性があります。 逆に、そのような高速化が出来ないということは部分的な変更がスコア計算の大部分に影響を与えているということであり、変形操作の見直しが必要であったり、そもそも局所探索には向いていない問題である可能性が高まります。今回の場合、以下のような高速化が可能です。

■高速化 1　d 日目のコンテストをタイプ i から j に変えることを考えます。満足度は各コンテストタイプごとに独立に計算を行って和を取った式になっており、この変形によって値が変化するのは i と j の部分だけです。先のスコア計算では $D \times 26$ のループを回しましたが、これにより $D \times 2$ のループで更新が可能になります。

■高速化 2　和の公式を用いることで、$c[i]$ による満足度の減少は毎日行う代わりにタイプ i のコンテストを開催した日にまとめて行うことが出来ます。つまり、前回の開催日から次の開催日までの日数が d 日であれば、その間の満足度の減少量は $c[i] \times \sum_{x=1}^{d-1} x = c[i] \times d(d-1)/2$ と計算できます。従って 1 からスコアを計算した場合でも D 回のループでスコア計算が可能です。

■高速化 3　以上 2 つの高速化を合わせると、i と j それぞれについて d 日目の一つ前・一つ後の開催日が分かれば残りは定数時間でスコアの計算が可能であることが分かります。これは各コンテストタイプ毎に開催日を配列に入れて並べておく[*6]ことで、高速に計算が可能です。

以下は高速化 3 の Rust での実装例です。各コンテストタイプごとの開催日を格納した配列 `ds` を用意し、現在のコンテスト日程 `out` とその満足度 `score` と一緒に struct としてまとめてあります。問題 C に提出したところ、高速化前の 442ms から 34ms に改善されました[*7]。

```rust
struct State {
    out: Vec<usize>,
    score: i64,
    ds: Vec<Vec<usize>>,
}

fn cost(a: usize, b: usize) -> i64 {
```

[*6] 二分探索木を用いると最悪計算量は改善しますが、平均的には $D/26 = 14$ の長さしかないため、配列を用いて線形アクセスした場合とあまり変わらないと予想されます。

[*7] 問題 C では入出力にかかる時間が無視できないため、この結果から 13 倍程度の高速化と見積もるのは誤りである点には注意しましょう。

```rust
    let d = b - a;
    (d * (d - 1) / 2) as i64
}

impl State {
    fn new(input: &Input, out: Vec<usize>) -> State {
        let mut ds = vec![vec![]; 26];
        for d in 0..input.D {
            ds[out[d]].push(d + 1);
        }
        let score = compute_score(&input, &out);
        State { out, score, ds }
    }
    fn change(&mut self, input: &Input, d: usize, new_i: usize) {
        let old_i = self.out[d];
        let p = self.ds[old_i].iter().position(|a| *a == d + 1).unwrap();
        let prev = self.ds[old_i].get(p - 1).cloned().unwrap_or(0);
        let next = self.ds[old_i].get(p + 1).cloned().unwrap_or(input.D + 1);
        self.ds[old_i].remove(p);
        self.score += (cost(prev, d + 1) + cost(d + 1, next) - cost(prev, next)) * input.c[old_i];
        let p = self.ds[new_i].iter().position(|a| *a > d + 1).unwrap_or(self.ds[new_i].len());
        let prev = self.ds[new_i].get(p - 1).cloned().unwrap_or(0);
        let next = self.ds[new_i].get(p).cloned().unwrap_or(input.D + 1);
        self.ds[new_i].insert(p, d + 1);
        self.score -= (cost(prev, d + 1) + cost(d + 1, next) - cost(prev, next)) * input.c[new_i];
        self.score += input.s[d][new_i] - input.s[d][old_i];
        self.out[d] = new_i;
    }
}
```

最適解ではなく出来るだけ良い解を求めるタイプのコンテストでは、バグのあるプログラムを提出しても不正解とはならないため、バグに気づくのが遅れる可能性があります。スコアが伸びずに方針が間違っているのかと思っていたら実はバグのせいであったということもしばしば起こります。バグの早期発見のために、複雑な処理を実装した箇所に対しては単体テストをしておくのも一つの手です。例えばスコアの差分計算を実装した際には、一からスコアを計算した場合と一致することをテストしておくと良いでしょう。

局所探索は非常に強力で、変形操作をうまく設計来るような問題の場合には最有力の手法となる場合が多いです。これは貪欲法やビームサーチでは入力を前からしか見ていない (例えば先に実装した貪欲法の場合、問題をオンライン形式に変更し、その日の予定を決めると初めて次の日の情報が与えられるというな設定にしてもそのままで動作する) のに対し、局所探索では全体を知った上で最適化を行っているという点も大きいです。一方で、例えば今回の問題の設定を少し変更し、コンテストの開催期間が 1 日とは限らないようにした場合を考えると、1 点変更も 2 点スワップも適用出来る場面が限られてしまい、貪欲法やビームサーチなどに劣る可能性もあります。

変形してから戻す or スコア計算だけ？

この章で紹介した局所探索法の実装では、まず変形してみてスコアが悪くなっていたら元に戻すという方針を取りました。別の方針として、変形せずに変形後のスコアのみを計算し、スコアが良くなっていたら実際に変形を行うという方針もあります。前者の方針は複雑な変形操作を簡単な変形操作の列で実現出来、実装が簡単になるという利点があります。例えば今回の問題の場合、一点変更の操作のみを実装すれば二点スワップの操作は一点変更を二回行うことで実現が可能です。一方後者の方針は変形を伴わない分だけ高速化が可能になる利点があります。問題に応じて使い分けたり、必要に応じてコンテスト途中で切り替えることも重要です。

5　焼きなまし法*

貪欲法の解説で、目的はあくまで最終的なスコアの最大化であり、途中段階でのスコアを高めることが必ずしもよいとは限らないと述べたのと同じように、局所探索の目的も最終的な到達点のスコアの最大化であって、途中段階でのスコアの最大化ではありません。探索の序盤で見つけた小高い丘に固執するあまり、少し下った先にある高い山に到達できなかったというのは避けたい事態です。焼きなまし法 (Simulated Annealing, SA) では、局所探索法の反復における序盤ではスコアの悪化する変形を高い確率で受け入れ、反復を繰り返すごとに徐々にそのような変形の採用確率を下げていくことで、小高い丘を避けて優れた解に到達しやすくします。

悪化する変形の受け入れやすさは金属工学における焼きなましのアナロジーで「温度」と呼ばれ、温度が T の時にスコアが Δ 増加する変形は以下の確率 $p(\Delta, T)$ で採用されます。

$$p(\Delta, T) = \begin{cases} 1 & (\Delta \geq 0) \\ e^{\Delta/T} & (\Delta < 0) \end{cases}$$

e は自然対数の底ですが、あまり本質ではありません。ここでは最大化問題を扱っているため、Δ が負の場合が悪化する変形に対応します。改善する変形は常に採用し、悪化する変形は悪化度合いに応じた 1 未満の確率で採用されます。

焼きなまし法の開始時点では高い温度 T_0 から開始し、徐々に温度を低い値 T_1 まで下げていきます。温度調整には経過時間 t を開始時刻が 0、終時刻が 1 となるように $[0, 1]$ に正規化して、時刻 t における温度を $T_t := T_0^{1-t} \times T_1^t$ とする指数スケジューリングがよく用いられます。温度 T が高いほど $p(\Delta, T)$ は大きくなるため、悪化する変形も採用されやすくなり、T が 0 に近づくにつれて指数的に採用確率が低下して山登り法の挙動に近づいていきます。以下は Rust による実装例です。先に解説した高速スコア計算の実装を用いています。また、スコア計算が高速になってくると経過時間の取得にかかるオーバーヘッドが無視できなくなってくるため、経過時間の取得と温度の更新は 100 反復に 1 回だけ行うことようにしています。提出したところ 125,726,288 のスコアが得られました。

```rust
fn solve(input: &Input) -> Vec<usize> {
    const T0: f64 = 2e3;
    const T1: f64 = 6e2;
    const TL: f64 = 1.9;
    let mut rng = rand_pcg::Pcg64Mcg::new(890482);
    let mut state = State::new(input, (0..input.D).map(|_| rng.gen_range(0, 26)).collect());
    let mut cnt = 0;
    let mut T = T0;
    let mut best = state.score;
```

```
    let mut best_out = state.out.clone();
    loop {
        cnt += 1;
        if cnt % 100 == 0 {
            let t = get_time() / TL;
            if t >= 1.0 {
                break;
            }
            T = T0.powf(1.0 - t) * T1.powf(t);
        }
        let old_score = state.score;
        if rng.gen_bool(0.5) {
            let d = rng.gen_range(0, input.D);
            let old = state.out[d];
            state.change(input, d, rng.gen_range(0, 26));
            if old_score > state.score && !rng.gen_bool(f64::exp((state.score - old_score) as f64 / T)) {
                state.change(input, d, old);
            }
        } else {
            let d1 = rng.gen_range(0, input.D - 1);
            let d2 = rng.gen_range(d1 + 1, (d1 + 16).min(input.D));
            let (a, b) = (state.out[d1], state.out[d2]);
            state.change(input, d1, b);
            state.change(input, d2, a);
            if old_score > state.score && !rng.gen_bool(f64::exp((state.score - old_score) as f64 / T)) {
                state.change(input, d1, a);
                state.change(input, d2, b);
            }
        }
        if best.setmax(state.score) {
            best_out = state.out.clone();
        }
    }
    best_out
}
```

パラメータの設定

上記の焼きなまし法の実装では開始温度 T_0 と最終温度 T_1 を適切に決める必要があります。公式に用意された入力ジェネレータを用いるなどして入力を手元に用意し、実験的に調整すると良いでしょう。今回の問題のように、全てのテストケースが同じスケールの場合は簡単で、全体で一つの値に固定すれば良い場合が多いです。一方で、テストケース毎に問題のサイズや値の幅が違う場合には、正規化などを行って調節しないと性能が出にくい場合もあります。

$p(\Delta, T)$ の設計思想

$e^{\Delta/T}$ という採用確率には以下のような意図があります。今、解 A から解 B に至る変形と、解 B から解 C に至る変形、解 A から解 C に直接至る変形の 3 つがあったとしましょう。各変形におけるスコアの増加量は全て負で、それぞれ Δ_{AB}、Δ_{BC}、Δ_{AC} であったとします。スコアの変化量の合計は途中に経由する解によらないため、$\Delta_{AC} = \Delta_{AB} + \Delta_{BC}$ が成り立ちます。この時、$A \to B$ と $B \to C$ の変形がともに採用される確率は

$$p(\Delta_{AB}, T)p(\Delta_{BC}, T) = e^{\Delta_{AB}/T}e^{\Delta_{BC}/T} = e^{(\Delta_{AB}+\Delta_{BC})/T} = e^{\Delta_{AC}/T}$$

となり、A から C への直接の変形の採用確率と等しくなります。つまり、変形列の長さによらず、始地点と終地点のスコアの差に応じて採用確率が決まります。このため、広いが低い小高い丘のような局所解から抜け出し長くゆるやかな坂を下った先にある高い山へ到達することが容易となります。一方で、尖っている針のような局所解から数歩先にあるより高い針の先へ到達することは困難です。局所解から抜け出すための他の方針としては変形操作を大きくする (例えば 2 点スワップでなく 3 点スワップにするなど) 方針があります。この方向性の場合は逆に、広いが低い小高い丘のような局所解から抜け出すことが困難になる代わりに、尖った針のような局所解から近くのより高い針の先へと直接移動することが出来るようになります。

6 巨大近傍法**

局所最適解から脱出するためのもう一つの方法は、変形操作を大きくすることです。近傍 (現在の解から一回の変形操作で移れる解全体) が大きければ大きいほど、現在の解より優れた解が近傍に含まれる可能性が高まり、局所最適でなくなる可能性が高まります。しかし一方で、変化させる量が大きすぎると闇雲に探す状態に近くなって、近傍の中から改善解を発見できる確率が低くなってしまいます。極論を言うと今回の問題の場合、2 点スワップではなく任意の k に対する k 点スワップ (多点スワップと呼ぶことにします) にしてしまえば近傍は非常に大きくなりますが、これは解を一からランダム生成するのとあまり変わりません。

巨大近傍法 (Large Neighborhood Search) では、変形操作をうまく設計することで、近傍自体は大きく保ったまま、その中から効率的に改善解を探索します。巨大近傍法の一つとしては、現在の解の一部分を破壊して作り直すという一連の操作を一つの変形操作とする手法があります。この際、作り直す操作を貪欲法や局所探索法など他の強力な手法を用いて行うことにより、単にランダムに作り直すよりも改善解が見つかる可能性を高めます。また、問題によっては破壊する部分をうまく選択することで、作り直す問題の最適解や改善解が多項式時間で求まるようになる場合もあります。

今回の問題の場合、破壊する部分をうまく選択することで、作り直しが多項式時間で解ける最適化問題となります。以下ではまず、現在の解を改善する多点スワップを多項式時間で求める手法を述べます。実はこの手法は誤りを含んでいるのですが、誤りを修正することで巨大近傍法に適用することが出来るようになります。

現在の日程から、u 日目の予定を v 日目に移すという操作をした際の満足度の増加量を $\Delta_{u,v}$ とします。ここで、$\Delta_{u,v}$ を定義する際には毎日ちょうど一つのコンテストを開催するという制約を一旦忘れ、予定を移したことによってコンテストを一つも開催しない日やコンテストを 2 つ以上開催する日が存在しても良いとしています。変形の途中で予定のない日などが出来ても最終的に解消されれば問題はないからです。すると、2 点スワップ $u \leftrightarrow v$ は、まず u 日目の予定を v 日目に移し、そして次に v 日目の予定を u 日

目に移すことにより達成できるので、満足度の増加量は $\Delta_{u,v} + \Delta_{v,u}$ となります。同様に、k 点スワップ $d_1 \to d_2 \to d_3 \to \cdots \to d_k \to d_1$ による満足度の増加量は $\Delta_{d_1,d_2} + \Delta_{d_2,d_3} + \cdots + \Delta_{d_{k-1},d_k} + \Delta_{d_k,d_1}$ となります。従って、各点 u から v へ重み $\Delta_{u,v}$ の辺を張った D 頂点の有向グラフを考えると、グラフに重み正の閉路が存在すれば現在の解を改善できる多点スワップが得られ、逆にそのような多点スワップが存在するならばグラフに重み正の閉路が存在することが成り立ちます。グラフに重み正の閉路が存在するかは、重みを反転して負閉路検出のアルゴリズムを用いることで多項式時間で判定できるため、現在の解を改善する多点スワップが存在するならば多項式時間で求めることが出来ます。

上の議論は赤文字の部分が実は誤っています。2 点スワップに対する議論は正しいのになぜ多点スワップに拡張出来ないのでしょうか？それは $\Delta_{u,v}$ が現在の解を変形した際の満足度の増加量であるからです。もし d_i 日目と d_j 日目のコンテストのタイプが同じであったら、d_i を動かしたことで $\Delta_{d_j,d_{j+1}}$ の値が変化している可能性があります。逆に、各コンテストタイプについて高々一つのコンテストしか動かさない場合に限定すれば、上記の議論は正しいです。

そこで次のような巨大近傍法を考えることが出来ます。まず、各タイプから一つずつコンテスト開催日をランダムに選び、それを集合 D' とします。すると、D' 内で日程を入れ替えることで解を改善する問題は、上の議論を適用することで負閉路検出問題に帰着されます[*8]。つまり、各点 u から v へ重み $\Delta_{u,v}$ の辺を張った $|D'|$ 頂点の有向グラフから重み正の閉路を探せばよいのです。そのような閉路が見つかれば対応する多点スワップにより解を改善し、見つからなければ別の集合 D' を選ぶことを繰り返します。あまり離れた日程を入れ替えると効果が薄いことが予想されるため、出来るだけ近くにまとまった D' を選ぶようにしましょう。これは例えばランダムに開始日 s を選び、各タイプについて s 日以降に初めてタイプ i のコンテストを開催する日を選ぶことで達成できます。

この解法は以下のようにして更に改良できます。

- d 日目に予定されたタイプ i のコンテストを、同じタイプ i のコンテストが開催される別の日 $d'(> d)$ をまたいだ先 $d''(> d')$ に移動させることは無駄が多いように思えます。そのような移動を禁止した場合、d 日目のコンテストを移動させることで移動コストが変化するのは、d のちょうど一つ前・一つ後の同じタイプのコンテストだけです。従って、D' の条件を「各タイプから高々 1 つまで」から「同じタイプの連続する 2 つのコンテストを同時に選ばない」に緩和することが出来ます。
- d 日目より前の最後のタイプ i のコンテストと d 日目より後の最初のタイプ i のコンテストがともに D' に含まれない場合、d 日目のコンテストをタイプ i に変更するという操作は D' 内の他のコンテストの移動コストに影響を与えません。新たに頂点 i' を用意し、$i' \to d_1 \to \cdots \to d_k \to i'$ という閉路が d_1 日目に開催するコンテストのタイプを i に変更し、元々 d_1 日目に開催する予定だったコンテストを d_2 日目に移動し、…、元々 d_k 日目に開催する予定だったコンテストの開催は取りやめる、という変形に対応するとみなすと、このような操作を含めた変形による改善解も負閉路検出に帰着されます。

焼きなまし法で得られた解を上記の巨大近傍法で更に改善するという解法を Rust で実装したところ、126,904,629 のスコアが得られ、焼きなまし法のみの 125,726,288 から更に改善できました。

[*8] 実は D' に対する最適な多点スワップを求める問題は最大重み二部マッチングと呼ばれる有名な組合せ最適化問題となり、最小費用流などのアルゴリズムで多項式時間で解くことが出来ます。負閉路検出をすることは現在のフローが最適であるかを確認することに対応します (詳しくはプログラミングコンテストチャレンジブックの 3.5 章例題 Evacuation Plan の解説などを参照してください)