電脳

2020

制作・著作 Paken

目次

はじめに

72nd define

「どうもー、筑駒パ研中学部長の平田です。この度は電脳 2020 をお読み頂き...」

『長いわ！毎年毎年同じ事言って飽きないんか？』

「まぁ、そういうもんやろ？」

『んーまぁええわ。そうそう、うちのオカンがね、読みたい部誌があるらしいんやけど、その名前をちょっと忘れたらしくてね』

「いやこの話の流れで行ったらどう考えても電脳やん」

『それが違うらしいねん』

「へぇ、違うの？ほなちょっと一緒に考えたるから、どんな部誌言うとったか教えてみてよ」

『まず、毎年秋に筑駒で配ってて、集めると技術の変遷とか分かって面白いらしいねん」

「おー、電脳やないかい。てか、オカン電脳好きすぎやろ！どんだけ集めてんのや」

『いやー俺も電脳思ったんやけどな、でもちょっと分からへんのよな』

「何が分からへんのよー」

『オカンが言うにはね、内容が難しくて JOI 本選 er でも全然分からへんらしいんよ』

「あー、ほな電脳と違うかぁ。電脳は簡単なものから難しいものまであって毎年充実してるねんな」

『そやねん。今年も競プロのテクニックから暗号まで色々あって面白いで』

「あれほなもう一度詳しく教えてくれる？」

『そんでね、記事は計画的に書かれてて、締切前日にはもう大体書き終わってるらしいねん』

「ほな電脳ちゃうやないかい！電脳は部員が締切当日に書き始めるから編集大変って有名や。大体これも文実への提出期限ギリギリに書いてるし」

『せやろ？てかさっさと終わらせんか、HR の動画編集もあるやろ』

「せやな、でもこれどうなってんねんもう」

『んでオトンが言うにはな、今年部活に昇格したパズ同の部誌、こまぱちゃうか？って言うねん」

「いや絶対ちゃうやろ！もうええわ」

「『ありがとうございましたー。それでは、電脳 2020 をお楽しみ下さい！』」

ソートアルゴリズムとは？役に立つの？調べてみました！！

73rd aspi

はじめに

こんにちは 73 期中 2 の aspi です。
今年 Paken 幽霊部員から部員になりました。
今回ソートアルゴリズムを選んだ理由は、簡単で分かりやすいものの奥が深く、部誌で書く題材に良いと思ったからです。
タイトルはネタです。

そもそもソートアルゴリズムとは？

そもそもソートアルゴリズムとはいったい何なのでしょう？みんな大好き wikipedia によると「データの集合を一定の規則に従って並べること」とありました。
アルゴリズムとしては、数や文字列の配列を昇順（小さい順）に並べる操作を指すことが多いです。
ソートは愚直に実装すると 100 要素の数列ならば 10000 回程度の操作が必要なのですが、これをどうにかして高速化することを考えていきたいと思います。
「愚直な実装 1-バブルソート」からは要素数 N の数列 A$\{A_1, A_2, ..., A_N\}$ があって、これをソートするための様々なアルゴリズムについて見ていきます。

前提知識：オーダー記法について

本格的に説明を始める前に、スムーズに理解するための前提知識としてオーダー記法について説明しておこうと思います。計算機科学や競技プログラミングなどで、厳密な計算

ステップ数や必要なメモリの量は実装に用いるプログラミング言語や実装方法などによって変わるので、計算量を厳密に見積もるのは大変です。
そこでオーダー記法を用いて計算量を大まかに見積もります。例えば入力 N に対して $3N^2+7N+4$ 回の計算ステップで答えが求まるプログラムがあったとします。
この時、N が十分に大きいとき $3N^2$ に対して $7N+4$ は無視できるほど小さくなります。
よって、オーダー記法では、係数を省略した上で N を大きくしたときに一番影響が大きい項を取り出し O(項) と表します。
例えば先ほどの例 $3N^2+7N+4$ では、$O(N^2)$ となります。また、計算量が $\log_2 N$ であるとき、オーダー記法では $O(\log N)$ と底が省略されます。
また、AtCoder 社のプログラミングガイド APG4b によると、「$\log N$ は N に対して非常に小さくなるので計算量の中に $O(\log N)$ が出てきた場合でも実行時間にそこまで影響しないことが多い」とあります。（AtCoder 社. ”W - 2.06. 計算量”. AtCoder. 2020-10-24. https://atcoder.jp/contests/apg4b/tasks/APG4b_w, 2020-10-24）

愚直な実装 1-バブルソート

最初に思いつく愚直な実装として、**バブルソート**が考えられます。

アルゴリズム

1. 連続した 2 つの要素 (A_k, A_{k+1}) を選んび、$A_k > A_{k+1}$ だったら二つの要素を入れ替える」という操作を左端まで行っていく。
2. 1 の操作を何回か繰り返す。
3. 1 の操作で一度も要素の入れ替えが行われなかった場合終了する。

計算量

このアルゴリズムでは、右端に最大値があった場合、$N-1$ 回左端から右端まで回る必要があるため、計算量は $O(N^2)$ となります。

愚直な実装 2-選択ソート

もう一つ愚直な実装として、**選択ソート**が挙げられます。

アルゴリズム

1. 配列で一番小さい要素を探し、その要素を配列の 1 番目と交換する。
2. 2 番目から N 番目までの中から一番小さい要素を探し、2 番目に持っていくというような操作を最後まで ($N-1$ 回) 行う。
3. 1,2 のような操作を最後まで $N-1$ 回繰り返していく。

計算量

このアルゴリズムでは、N 回最小値を探すために $(N-1)+(N-2)+...+1 = N \times (N-1)/2$ 回比較する必要があるため $O(N^2)$ となります。
ちなみに選択ソートとバブルソートではやや選択ソートの方が高速だと言われています。

高速なアルゴリズム 1-マージソート

上の二つより高速なアルゴリズムの一つがマージソートです。

アルゴリズム

まず、ソートされた 2 つの配列 A, B をマージ（併合）し、新しくソートされた配列 C を作る方法について考えます。
A, B の要素の中で、最小のものは A の先頭（最小値）か B の先頭なので、A の先頭と B

の先頭の小さい方を削除し、C の末尾に追加していくことで、A, B の要素数の和を N とすると $O(N)$ でマージすることができます。
1. 配列を 1 要素の配列に分割する。
2. 配列を二つごとにマージする。(1 要素の配列が 9 個あった場合要素数 2 の配列 4 個と要素数 1 の配列 1 個ができる)
3. 2 を繰り替えし、最終的に一つの配列になったら操作を終了する。

計算量

配列の大きさは一回マージするごとに $2, 4, 8, 16, 32, ...$ と倍になっていきます。よって要素数 N の配列の場合、$O(\log N)$ 回のマージが必要です。
また、一連のマージに $O(N)$ かかるため計算量は $O(N \log N)$ です。

高速なアルゴリズム 2-クイックソート

クイックソートは他のソートアルゴリズムに比べて平均的に最も高速だと言われています。

アルゴリズム

1. 配列の中からピボットと呼ばれる適当な数を選択する。(末尾の数をピボットとするなど)
2. ピボットより小さい数を前方に、大きい数を後方に移動させ、ピボットがその真ん中に来るようにする。
3. 配列の要素数が 2 以上の時 2 の操作で分けたピボットより小さい数の配列とピボットより大きい数の配列にも 2 の操作を行う。

計算量

ピボットに中央値が選ばれた場合、$O(\log N)$ 回 2 の操作が行われるため、最良計算量・平均計算量は $O(N \log N)$ です。
しかし、ピボットに最大値や最小値が選ばれた場合の $(N-1)$ 回 2 の操作が行われるため $O(N^2)$ となってしまいます。
これは、ピボットを選ぶのに乱数を使ったり、いくつか適当に値を取ってその平均値をピボットとしたりすることである程度改善できます。

まとめ

いかがでしたでしょうか？
ソートアルゴリズムの面白さについて知っていただけたでしょうか？
これを機会にアルゴリズムや競技プログラミングに興味を持って頂ける人がいれば何よりです。ほかにも平衡二分探索木のことも書きたかったのですが、難しいアルゴリズムなので~~筆者が理解できておらず~~省きました。
好きなデータ構造なのでまた強くなったら平衡二分探索木に絞って記事など書いてみたいです。
最後まで読んで頂きありがとうございました。
部誌

動的計画法を広めたい

71st penguinman

はじめに

高校一年生の penguinman と申します。実は当方絶賛休学留学中なんですが、こっそり文化祭に紛れ込んじゃいました！（害）
さて、今回は僕が好きな動的計画法を、初心者の方にも分かるように（というかそれをメインに）解説していこうと思います。もちろんこの初心者には、プログラミング初心者も含んでいます。
ただ、それだけだと動的計画法を既に知っている人には退屈だと思うので、最後の方にオリジナルの問題を載せておきました。競技プログラミングをやっている人向けに言うと緑 diff くらいでしょうか。頭の体操だと思って解いてみてください。

動的計画法って何？

ざっくりと言うと、既に計算した値を利用して次の値を求めることで、計算を効率化するアルゴリズムのことです。
例えば、フィボナッチ数列の n 番目の値を求めることを考えます。
フィボナッチ数列とは $F_1 = F_2 = 1,\ F_n = F_{n-1} + F_{n-2} (n > 2)$ で表される数列のことです。
さて、これを求める時、多くの人は添字が小さい順に、前の結果を利用して

$$1,\ 1,\ 2(1+1),\ 3(1+2),\ 5(2+3),\ 8(3+5),\ 13(5+8)...$$

といった風に値を求めていくと思います。
実は、これも立派な動的計画法なのです。
この章の始めに書いてある、動的計画法の定義を思い出してください。
今挙げたフィボナッチ数列の例では、**前の結果を利用して**次の値を求めていますね？
よって、これも動的計画法に含まれるのです。
もちろんこれはかなり初歩的な動的計画法の例ですが、応用的な動的計画法も考え方は同じです。

表記

実は前の章で説明できておらず、かつ説明しておく必要があることが 1 つだけあります。それは dp を表す際の式です。
前の章で説明したように、dp は既に計算した値を利用して次の値を求めるアルゴリズムです。よってその「次の項」は、「前の項」を用いて表すことができます。
例えばフィボナッチ数列の例だと、フィボナッチ数列の i 項目を $F_i = F_{i-1} + F_{i-2}$ といった風に表すことができます（自明...）。
これだけだと表記について知っておく必要は少なく感じると思いますが、次の章で紹介するナップサック問題を解く際などには、表記についてしっかりと知っておく方が理解がしやすいです。
また余談ですが、一般的に dp の漸化式を表す際には dp という名前を用いることが多いです。
例えばフィボナッチ数列の例だと $dp[i] = dp[i-1] + dp[i-2]$ という表し方をするのが一般的です。
今後の解説では僕もこの表記で統一しようと思うので、覚えておいてください。

ところで動的計画法って長い

今更ながら、動的計画法って長くないですか？長いです。
そのため巷では dp と呼ばれたりしています。これは動的計画法の英名、Dynamic programming の略です。
僕の説明でも、ここから先は dp 表記で統一しようと思います

本題「ナップサック問題」

世の中には「ナップサック問題」と呼ばれる、以下のような有名問題が存在します。

ナップサック問題

N 個の品物があり、i 個目の品物の価値は v_i、重さは w_i である。
これらの品物から幾つかを、重さの合計がナップサックの容量である W を超えないように選んでナップサックに詰める時、詰められた品物の価値の合計の最大値を求めよ。

まず、この問題を愚直に計算することを考えてみましょう。
この場合、それぞれの品物について選ぶ or 選ばないの 2 通りを選択し、それらの組み合わせを全て列挙した上で、それぞれについて価値の合計は幾つか、重さの合計が W を超えないかを計算する必要があります。組み合わせ全体で 2^N 通りあり、それぞれについて価値の合計、重さの合計を求める必要があるため、おおよそ $2^N * N$ 回の計算をする必要があります。
一般的にコンピューターが計算できる回数は、計算の内容にもよりますが 1 秒あたり 10^8 ～10^9 程度と言われているので、これだと N が 40 くらいになっただけで計算に 1 時間を費やしてしまいます。

これだとやはり遅すぎるので、dp を利用することで、おおよそ $N*W$ 回の計算で最大値を求めてみましょう。因みにこれだと、人力では求められないような、$N = 3000$, $W = 3000$ のケースでもたったの 1 秒足らずで答えを求めることができます。
さて、今回のケースでは、dp の添え字を 2 つ取ります。1 つは「何個目の品物までの情報か」、もう 1 つは「重さがいくつのときの情報か」です。
具体的には、$dp[i][j]$ を、「i 個目までのの品物を上手に取捨選択した場合の、重さの合計が j 以下のときの価値の最大値」と定義します。
例えば $dp[2][3]$ は、「1 個目の品物と 2 個目の品物を品物を上手に取捨選択した場合の、重さの合計が 3 以下のときの価値の最大値」です。
また、今回のケースでも、フィボナッチ数列の時と同様、$0 \leq k < i$ を満たす任意の k において $dp[k][0]$～$dp[k][W]$ が求まっている時、それを利用して $dp[i][j]$ を求めることを考えます。

まず、$j < w_i$ の場合を考えます。
このとき、$j - w_i$ が 0 未満であることから、i 個目の品物を取ったときに重さの合計が j 以下になることはありません。
よって i 個目の品物を取ることはできず、$dp[i-1][j]$ と $dp[i][j]$ は等しくなります。

次に、$j \geq w_i$ の場合を考えます。
この場合、先程とは違い $j - w_i$ が 0 以上であることから、i 個目の品物を取ったときに重さの合計が j 以下になることはありえます。
よって、i 個目の品物を取るかどうか選択することができます。
また、i 個目までの品物を取るかどうかについてこれ以降に議論することはありません。
よって、$dp[i][j]$ は可能な範囲で最大化するのが最適ということになります。
これらを踏まえると、$dp[i][j]$ は i 個目の品物を取った場合と取らなかった場合それぞれについての 最大値のうち、大きい方ということになります。
i 個目の品物を取らなかった場合の最大値は $j \geq w_i$ のときと同様 $dp[i-1][j]$ に等しくなります。
それに対し、取った場合の最大値は $dp[i-1][j-w_i] + v_i$ に等しくなります。
よって、$dp[i][j] = max(dp[i-1][j-1], dp[j-1][j-w_i] + v_i)$ であることが分かりました。

このようにして、$dp[i-1][0]$ から $dp[i-1][W]$ の結果を利用して $dp[i][0]$ から $dp[i][W]$ を求めることができました。
後はこの性質を利用した上で、i の昇順に $dp[i][0]$ から $dp[i][W]$ を求めてあげれば良いです。
計算量は、 $0 \leq i \leq N,\ 0 \leq j \leq W$ を満たす任意の $i,\ j$ について 1 回ずつ計算するため最初に説明したとおり、おおよそ $N * W$ 回です。（オーダー表記というものを使うと、$O(N * W)$ で表されます。）

penguinman からの挑戦状

さて、ここまで dp の基礎を入念に学んできましたね。そろそろ output をしたくなる季節ではないでしょうか。
そんな皆さんのために、ここまでやった内容で解ける（はず）の問題を用意しました。力試しにぜひ解いてみてください！！！

問題

あなたは、N 日間に渡って市場を開くことにしました。
i 日目には、以下の行動のいずれかをすることができます。

・商品を 1 個、 A_i 円で仕入れる。
・商品を 1 個、仕入れ値の 2 倍で売る。ただし仕入れてある商品が 0 個の場合は何もしない。
・何もしない。

適切に仕入れと売却を行うとき、得られる利益の最大値をおおよそ N^2 回の計算で（$O(N^2)$ で）求めてください。
ただし商品を仕入れるための資金は無限にあるものとし、
また 0 日目（つまり市場が開かれる前）の時点では売るための品物は 1 つも持っていないものとします。

この問題はそれなりに難しいと思うので軽めのヒントを出しておくと、競技プログラミングで頻出の、「考察」というステップを踏む必要があります。
解説は次の頁に書いてあるので、見たくなったタイミングで見てください。

解説

想定解法はもちろん dp です。しかし、dp の内容を詰める前に、一番最初に考察ステップを踏む必要があります。
まず、最終日が終わった段階でいくつかの品物が余っていたとします。この場合、それらを買った日に何もしないことで、それらの品物の原価の合計分得をすることができます。
よって最終日が終わった時点で残っている品物の数は 0 個であるべきです。
また、このことから売れ残りについては考慮しなくてもいいことが分かります。**そのため、ある品物を仕入れた時点で、それを売ることで得られる利益（= 原価）を加算してもいいことになります。**
ここまで考察ステップを踏むと、次第に解法が見えてくる人もいるのではないでしょうか。ここからはノンストップで、具体的な解法の説明をしていきます。
まず、今回 dp の添え字に取るのは「何日目までの情報か」「品物の在庫がいくつか」です。つまり、$dp[i][j]$ は「i 日目までで、品物の在庫が j 個ある時の利益の最大値」を表します。
さて、これまた同じように、 $dp[i-1][0]$ から $dp[i-1][N]$ までを利用して、 $dp[i][0]$ から　$dp[i][N]$ までを求めて行きましょう。
今回はなんと、 3 段階に分けて遷移します！（やばそう）
とはいえ、うち 1 つはあってないようなものなので、ご安心ください。

まず、 $i < j$ の場合です。
この場合、1 日に 1 個までしか入荷できないことから条件を満たす仕入れ方は存在しません。よって無視します。

次に、$j = 0$ の場合です。
この場合、i 日目に商品を仕入れたと仮定すると元の在庫が負の数になり、矛盾します。
よって、商品を売ったか、何もしなかったかの 2 通りしか存在しません。
ナップサック問題の時同様、i 日目までの行動についてこれ以降に議論することはないので、$dp[i][j]$ は可能な範囲で最大化するべきです。
故に、i 日目に商品を売った場合と何もしなかった場合のそれぞれについて最大値を求め、それらのうち大きいものを取ればよいです。
よって、$dp[i][0] = max(dp[i-1][0],\ dp[i-1][1])$ と表すことができます。

最後に、$1 \leq j \leq i$ の場合です。
この場合、i 日目に品物を売っても、仕入れても、何もしなくても矛盾は起きません（$i-1<j$ の時等を除く）。
そのため、それぞれについての最大値を求めた上で、その max を取って上げればいいです。因みにですが、「何もしない」という行動は明らかにメリットがないので、遷移時には無視してしまっていいです。
よって、$dp[i][j] = max(dp[i-1][j-1] + A_i, (i-1 \geq j+1 \text{ なら}) dp[i-1][j+1])$ と表すことができます。

さて、このようにして $dp[i-1][0]$ から $dp[i-1][N]$ の結果を利用して $dp[i][0]$ から $dp[i][N]$ を求めることができました。
最初に示したとおり、売れ残りが 0 個の時以外は考慮しなくていいので、答えは $dp[N][0]$ になります。
よってこの問題を解くことができました。

さて、「penguinman からの挑戦状」はいかがだったでしょうか。考察パートの部分だけでも楽しんでもらえたら嬉しいなーなどと思いながら作問していたので、そこだけでも楽しんでいただけたら嬉しいです。

終わりに

長い記事をお読みくださりありがとうございました。お楽しみいただけたでしょうか？これをきっかけに dp や競技プログラミングにのめり込んでいただけたらこちらとしても嬉しい限りです。
一応そういった方向けにちょっとしたメモを残しておくと、dp に興味がある人は bit dp や LCS について、競技プログラミングに興味がある人は AtCoder について調べてみるといいかもしれません。

先生にやってほしい公平な指名方法

71st nuhununе

はじめに

多くの方は初めまして。高校一年生の nuhununе と申します。パ研に入ったのは中 3 の時ですが、~~日頃の仕事ぶりが評価されたのか、~~71 期のアクティブ部員が他にあまり居なかったため、デコ責をやらせていただいています。そんな責任ある役職に就いてはいますが、部誌を書き始めた只今の時刻は部誌の締切の三時間半後。編集者の怖い眼光に怯えながら、書き始めたところでございます。(本当にごめんなさい)
もう少し自己紹介をしますと、AtCoder は水色で、主に Java を使ってコンテストに参加していました。最近は主に開発っぽいことをしており、最も書く言語は JavaScript です。

突然ですが

みなさん、学校の授業に参加したことはありますか？ないわけがないですね。学校の授業に参加したことがあれば、先生が「じゃあ今日は 25 日だから〜、25 番の○○この問題に答えろ」などという光景も見たことがあるのではないでしょうか。少なくともこの筑駒ではごく日常的に行われていることであり、たまに物議を醸すことがある話題でもあります。最近私が参加した授業ではこんなことがありました。
先生「えっと今日は 22 日だから 22 の約数で〜、」
一番の生徒「えっこれ俺絶対当たらね？ (注：1 はあらゆる自然数の約数)」
先生「あっほんとだ。じゃあ (一番の生徒) くん！この問題に〜」
一番の生徒「ひどいよー！」
とても可哀そうですね。こんな~~面白い~~ことがあったのが、私がこのテーマで部誌を書こうと思った理由です。今回犠牲者になってしまった彼を救うために、できるだけ公平に生徒を指名する方法を見つけていこうと思います。

問題の定義

今回はこの問題をプログラムを使って研究したいので、扱いやすいように問題を再定義します。定義したものが以下です。
・何人かの生徒が集まった架空のクラスを考える。
・クラスの生徒は、筑駒のひとクラスと同じ 41 人とする。それぞれに 1 から 41 の出席番号が一つずつ割り当てられている。
・日付をもとにした計算で出席番号を定めることによって生徒を指名していく方法を提示する。
・一年間 (365 日) のそれぞれの日に、その方法を用いて 5 人ずつ生徒を指名することを考える。
・それぞれの生徒について、一年間で指名される回数の期待値 (と言うより単に回数) を求める。
・期待値のばらつきが小さいほうが公平な指名方法である。ばらつきは標準偏差を求めて評価する。

提示していく方法は以下の条件を満たすものとします。
・最初に指名する生徒、二番目に指名する生徒・・・が日ごとに一意に定まる。
日ごとに「この生徒は指名される可能性がある」という集合を作ってランダムに選んだときの期待値も面白そうです。しかし、これができてしまうとなるとその集合をクラス全員にすれば万事解決であり、趣旨から外れてしまうと思いました。また、ランダム性を先生に作らせようとすると私情が入ってしまうこともあるでしょう。というわけで、私情を挟みようがないこの条件で行きたいと思います。

実験の方法

JavaScript を用いて、期待値が入った CSV ファイルを出力するプログラムを書きます。それを Excel で分析します。

実際に方法を考えて実験！

適当に考えた方法で実験を行い、その結果を統計しました。挿入されているグラフは横軸が出席番号、縦軸が指名される回数です。

方法 1

説明

その日が 1 年の中で N 日目だとします。最初に N 番の生徒、次は $N+1$ 番、その次は $N+2$ 番・・・というように指名していきます。この番号は多くの場合 41 を超えるので、1 以上 41 以下の整数の中で、41 を法としてこの番号と合同な番号の生徒を選びます。

例

3 月 20 日 (1 年の 79 日目) の場合、$38, 39, 40, 41, 1$ 番が指名されます。

結果

最大値　45

最小値　41

標準偏差　1.12

コメント

偏りは少なく、シンプルながらも優秀な方法ですね。

方法 2

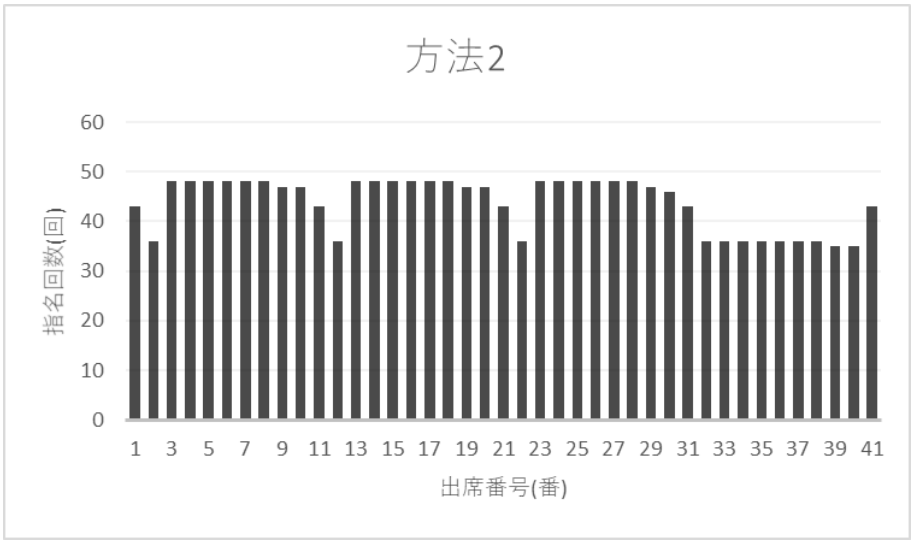
説明

その日が N 月 M 日だとします。最初に M 番の生徒を指名し、次は $M+10$ 番、$M+20$ 番と、M と一の位が同じ番号を順に選んでいきます。番号が 41 を超えたら、1 番から 10 番までの中から 1 の位が合致するものを選びます。既に指名された生徒に戻ってきたら、番号に 1 を加算します。

例

3 月 19 日の場合、$19, 29, 39, 9, 20$ 番が指名されます。

結果



最大値　48

最小値　35

標準偏差　5.34

コメント

回数に最大で $\frac{4}{3}$ 倍の差があり、そこそこ不公平な方法と言えるでしょう。この方法は併校のとある数学教師が愛用しているものですが、彼はこのことに気づいているのでしょうか。

方法 3

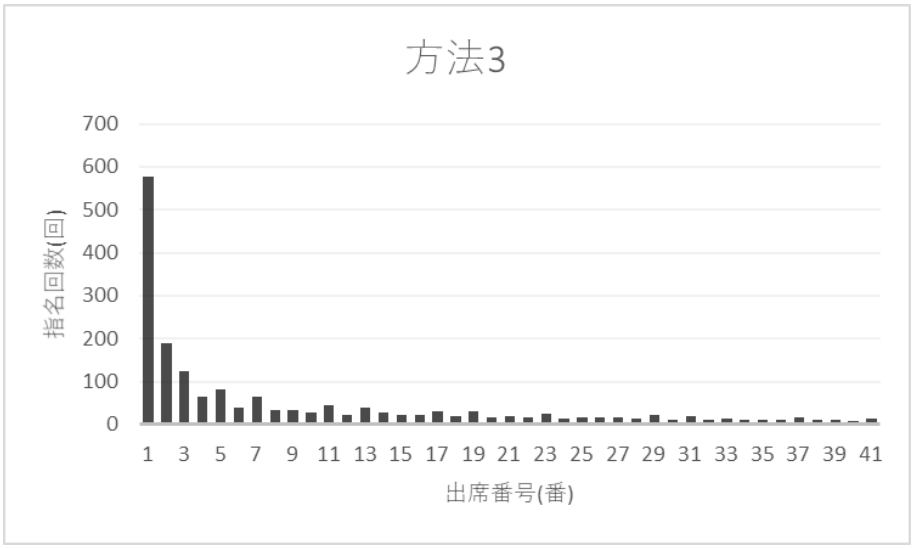
説明

その日が 1 年の中で N 日目だとします。N の約数であって 1 以上 41 以下である番号を、大きい方から順に選んでいきます。1 まで来たら、最大のものに戻ります。

例

1 月 14 日の場合、$14, 7, 2, 1, 14$(二回目) 番が指名されます。

結果



最大値　576

最小値　9

標準偏差　91.55

コメント

やはりこの方法は不公平ですね。偏り方がものすごい。大きい方から取れば少しはマシになるかと思いましたが、そんなことはなかったです。

一旦まとめ

最もシンプルな方法が最も優秀でした。

問題点

今回の実験で考案できたもっとも優秀な方法は、1 年という長い期間で見れば公平なものでした。しかし、もう少し具体的な期間について考えてみると、ある時 5 日連続で指名され、そのあとは 36 日間一度も指名されないことがわかります。これでは生徒側としては不快さを感じ得ます。
というわけで、公平になる上で指名される日も均等に散らばる方法を作り出したいのですが難しそうです。何か解決策は無いのでしょうか。

ところで

今まで考えてきたのは、日付の値を元に、1 から 41 までの数字をできるだけ均等に、あわよくばランダムに発生させる方法でした。これって何かに似てませんかね？
そうです、疑似乱数です。~~(みんなもうわかってる)~~
わからない方のために説明しておきます。ゲームを作るときに確率で分岐したい時など、コンピューターでランダムな数字 (乱数) を生成したくなることがあります。しかし、コンピューターとは命令に沿って決まった動きをするもの。完全にランダムな数字を生成するのは不可能です。そこで出てくるのが擬似乱数。時刻や株価など、時によって異なる値 (このような初期化に使う値をシード値と言います) を元に、数字を生成します。今回の実験の内容と非常に似ていますね。疑似乱数については、先人が優秀なロジックを見つけてくれています。有名なものには、線形合同法、メルセンヌ・ツイスター、XOR Shift などがあります。今回はメルセンヌ・ツイスターについて軽く解説したいと思います。ここから話の難易度が上がります。

メルセンヌ・ツイスターの解説

最初に、0 と 1 しか数が存在しない世界を考え、数どうしの足し算と掛け算を定義します。といっても私達の世界と違う規則は一つだけです。$1+1$ を 2 ではなく 0 と考えて、演算をします。これからこの世界でベクトルの計算をしていくことになるのですが、その前に演算の例を見てみましょう。

$$(0,0,1,1)+(0,1,0,1)=(0,1,1,0)$$

次に、このようなベクトルの漸化式を作ります。ベクトルの成分数は、そのコンピューターが一括に処理できるビット数 (これをワードサイズと言います、32bit マシンであれば 32 など) と同じにします。以下のような漸化式を考えてみましょう。$\overrightarrow{x}$が、考えているベクトル列です。

$$\overrightarrow{x_{n+p}}=\overrightarrow{x_{n+q}}+\overrightarrow{x_n}$$

p や q には適切な値を入れます (p には 607 などを用いるようです、これ以上詳しい情報は見つけられませんでした・・・)。$\overrightarrow{x_1}$ から $\overrightarrow{x_p}$ までの成分は先程言ったシード値です。
こうして作ったベクトルを二進法の数として見るなどして、乱数として用います。この方法を GFSR と言います。
この方法は私が考えていた方法などよりも遥かに優秀なのですが、乱数として穴があります。ベクトルの足し算のみによって列を生成していく以上、異なる位置の成分間では値のやり取りが行われません。このままでは、例えば初期化時のベクトル全てに置いて 0 であるような成分があればその位置は必ず 0 になってしまうなど、問題が生じます。
そこで登場するのが Twister という行列です。漸化式においてベクトルに行列を掛け合わせることで各桁間の情報を混ぜ合わせ、より高い乱数性と長い周期を達成します。A(n 次正方行列、n は x の要素の成分数と同じ) を Twister として、以下のような漸化式を用います。

$$\overrightarrow{x_{n+p}}=\overrightarrow{x_{n+q}}+\overrightarrow{x_n}A$$

この方法を Twisted GFSR と言い、GFSR より高い乱数性と長い周期を持ちます。具体的には、GFSR ではうまくやっても 2^p-1 であった有り得る最長の周期が、w をワードサイズとして、$2^{wp}-1$ まで長くできるようになります。GFSR では「漸化式でこれから使われる数字」の有り得る状態数が、異なる桁間で撹拌がないために 2^p だったのに対し、こちらでは 2^{wp} になったからです。w が 32 などの数字だと考えると、かなり大きい差ですね。

これをさらにもう一段階発展させたのが Mersenne Twister(メルセンヌ・ツイスター) です。B と C を Twister として、以下のような漸化式を用います。

$$\overrightarrow{x_{n+p}} = \overrightarrow{x_{n+q}} + \overrightarrow{x_{n+1}}B + \overrightarrow{x_n}C$$

このようにすることで、先程 2^{wp} であった状態数を 2^{wp-r} の形に制御できるようになり、周期の長さを素数にすることが可能になりました (例えば $w = 32, p = 624, r = 31$ のとき、$2^{wp-r} - 1 = 2^{19937} - 1$ は素数)。ここも正直よく理解できていないのですが、これが素数なことによって周期の長さの判定が容易になるらしいです。
最後がグダグダで申し訳ないですが、これでメルセンヌ・ツイスターの紹介を終わります。

おわりに

授業中の何気ない出来事からこの題材を思いつき、ここまで原稿を書くことになりました。授業中に当てられまくって悩んでいる人は、先生に高度な乱数生成アルゴリズムを使うこと、つまりスマホの乱数アプリを使うことを薦めるといいかもしれませんね。ただし、もともと目をつけられているからあてられる人がそんなおすすめをすると、更に目をつけられる可能性があるので注意です。
先程もちょこっと書いたとおり、乱数はゲームやシミュレーションなど様々な場面で活用されています。皆さんも一度はお世話になったことがあると思います。がチャを回すときなど、是非乱数の細かい仕組みにも思いをはせてみて下さい。レアキャラが当たる確率が UP するかもしれませんよ？
ここまでお読みくださり、ありがとうございました。

参考文献

松本 眞 (2014 年),「あなたの使っている乱数、大丈夫？-危ない標準乱数と、メルセンヌ・ツイスター開発秘話-」,(第 50 回市村学術賞記念　先端技術講演会),http://www.math.sci.hiroshima-u.ac.jp/　m-mat/TEACH/ichimura-sho-koen.pdf,2020 年 10 月 25 日アクセス

Monopoly を AI で解析する

70th capra314cabra

はじめに

もう高校二年生になってしまった capra314cabra です。

気付いたらパ研を引退するまであと 1 年もない状態になっていました。寂しい。

まあ、部誌を LaTeX 化し文実の器材管理システムを実装するという功績 (というほどでもない) を残せたので未練なく引退できそうです。

競技プログラミング以外のことをして筑駒生活を過ごしてきたので、そもそも競技プログラミング関係の部誌を書くことができません。それらのことは他の人に任せて、今年も機械学習の話をしていきたいと思います。

突然ですが、Monopoly を知ってますか?

Monopoly はアメリカ発のボードゲームでとても長い間世界中の人に楽しまれているボードゲームです。
ルールは意外と複雑なのですが、ゲームの大枠は簡単です。

1. サイコロを振って出た数だけ進む
2. 土地のマスに止まったら、土地を所有しているプレイヤーにお金を払う
3. これを繰り返して、お金が払えなくなった (破産した) 人からゲームを脱落する
4. プレイヤーは土地に家を建てて土地の収益を強化できる

これだけだとただサイコロを振って人の土地に止まらないことを祈るだけのゲームで戦略要素が全くありません。実は Monopoly にはこれに加えて戦略要素を高めるあるルールがあります。

プレイヤー同士で交渉して自由に土地を取引してよい

Monopoly の土地は全部同じというわけではなく、少ない投資コストで家を立てられる

土地 (New York 通りなど) から投資コストが莫大な代わりに大きな収入が見込める土地 (Boardwalk など)、そもそもサイコロのマス目の都合で止まりやすい土地 (Illinois 通りなど) など違いがあります。

交渉が自由に出来ることで、自分が欲しい土地を持っている人に「自分の土地とあの土地交換させて」などと持ち掛けたり、手元のお金が不足している時は要らない土地を他の人にお金で買ってもらうよう提案したりでき、Monopoly における戦略性は大きく高まります。

Monopoly の交渉を助けてくれる AI を作りたい

去年の秋くらいに友達と Monopoly をやったのですが、その時に、ゲーム後に友達が「実は自分に利益のない交渉ばかりのってしまったのではないか?」と延々と一人で Monopoly 反省会をしていました。彼はゲームの途中まで順調だったのにも関わらず、終盤に持ち金が $a - e^x$ のように急激に少なくなってしまい、そのまま破産してしまったのです。そこで彼は、中盤に行った幾つかの交渉の中に自分の将来の利益を大きく損ねたものがあったのではないか、と考えたようです。しかし結局、彼は結論を出すことができませんでした。

そんな様子を見て、実はこれ、機械学習で交渉の良し悪しを評価できるのでは?と考えました。

使ったもの

- Python (プログラミング言語)
- Tensorflow (機械学習で使えるライブラリ)
- Monopoly のボード (土地のデータを Excel sheet にするのに使いました)
- 自作パソコン (GPU 積んで機械学習へのやる気を見せています)
- 自分の脳のリソース

目標

Monopoly で交渉後に、将来的に誰が得するのかを機械学習の手法を用いて予測すること。

これが出来れば、あとは考えられる交渉 (金銭的に余りにも差がある交渉以外) を列挙して、その交渉に関係する 2 人が他の 2 人に比べてどれくらい得するかを判定すれば、「最善な交渉」を探索できます。

とりあえず式で表す

今から当たり前のことを式にします。

任意のプレイヤーについて、$k(\in \mathbb{N})$ ターン後の利益を $r_k(\in \mathbb{Z})$、全財産を $m_k(> 0)$ とすれば

$$m_{k+1} = m_k + r_k$$

当たり前ですね。これから予想したいのはゲームがある程度進んだ k ターン目での m_k です。愚直にやろうとすると、この式を変形して

$$m_k = \sum_{t=0}^{k-1} r_k$$

としてから r_k をシミュレートすることになります。しかし考えてみて下さい。最初の式の形から、機械学習の手法の、あれ、使えそうですよね。

Q-learning したくなってきます

はい。Q-learning のお時間です。

k ターン目の盤面を行列 b_k、その盤面での交渉を a_k で表現する時に、$Q(b_k; a_k)$ で現在がその盤面の時に将来的に得られる利益を表すとします。この時、

$$Q(b_k; a_k) = r_k + Q_{max}(b_{k+1})$$

となるはずです。しかし、ゲームの終盤の利益が全て現在の交渉に帰結していると考えるのには無理がありますし、ゲームが一生続くわけではないので、割引率 $\gamma(0 < \gamma < 1)$ を利用して

$$Q(b_k; a_k) \leftarrow r_k + \gamma Q_{max}(b_{k+1})$$

という $Q(b_k)$ の近似式を作ります。この近似式から $Q(b_k)$ の値を考えられればゴールはすぐ目の前です。あとはここに NN(Neural Network) を持ち込みます。

遂に Deep Q-learning します

$Q(b_k)$ を NN でおいて、先程の式を基に学習を行います。

データセットの作り方ですが、私は、Monopoly の盤面を乱数で生成して、それを一世代前の Q 値関数に代入して教師データを生成することを繰り返しました。Monopoly を最初からやらせると特殊な盤面に対応できないと考えたからです。

Monopoly を最初からやっていって学習させるのも別に問題はないと思います。

定義したモデル

$Q(b_k)$ の近似を行うモデルです。各プレイヤーの利益を一括で計算するようにしました。以下の順番に計算を行います。

1. 入力層 (実は盤面を画像のような構造にして与えています)
2. 畳み込み層
3. 平坦化層 (tf.keras.layers.Flatten 呼んでるだけです)
4. 隠れ層 1 (Fully connected しています)
5. 隠れ層 2 (上に同じ)
6. 出力層 (上に同じ)

畳み込み層を追加したので、実は NN ではなく CNN(Convolutional Neural Network) になっています。

盤面上で近い土地同士は賃貸料に影響を及ぼすのでそれは画像のような構造で盤面を入力として与えた方が良いと考えたので CNN にしました。

結果

プレイヤーが 4 人の時に 68% の確率 ($-\sigma$ から σ) で Q 値関数の誤差が\$400 に収まるという結果になりました。お世辞にも良い確率とは言えません。止まった時に請求できる額が大きい土地のデータに引っ張られていそうです (例： Boardwalk)。値の正規化はしているので、あとはゲームの単純化が必要な気もします。

実用的なものにするにはまだまだかかりそうです。課題しかない。

余談

実はこのプロジェクトで一番時間を使ったのは、モデル作りではなく、Monopoly のシミュレーションするスクリプトを書くことでした。Tensorflow を使うと比較的簡単にモデルを構成できるので、学習の方針が立ってしまえば、意外とすぐ書くことができました。一方、Monopoly のシミュレーターは、日本モノポリー協会の公式大会ルールに基づいて結構細かいルールまでプログラムしたのでだいぶ時間がかかりました。

~~簡略化しても変わらんやろ、とか言わない。~~

まとめ

本日の一品： シェフの気まぐれ Deep Q-learning ～*CNN* を添えて～

参考文献

日本モノポリー協会の公式大会ルール (https://monopoly-championship.jp/rule.html)

Tensorflow Docs (https://www.tensorflow.org/api_docs/python/tf)

DFS と UnionFind

74th cpcznksutbeoa

はじめに

こんにちは。中学 1 年の cpcznksutbeoa です。

ちなみに、今は 10 月 21 日の 18 時 37 分です。あと 3 日しかないですが、できるだけ書きたいです。ちなみに、今回はテストケースの言及や、解法の言及がかなりされているので、言及を見たくない方は、読み飛ばしてもらって構いません。

UnionFind をメインで書くつもりですが、DFS も書くので、とても長くなりそうです。最後まで読んでくだされば嬉しいです。ちなみに、僕が初めて UnionFind に出会ったときには、OnionFind と読んで爆笑した記憶があります。

ところで、今 tex ファイルを作っているのが僕も含めて 6 人で、去年の部誌は 10 人執筆しているので、人数足りるのか？と思っています。

自己紹介

ユーザー名　cpcznksutbeoa
AtCoder ユーザー名の中で読みにくいランキング TOP50 に入っている自信があります。
学年　中学 1 年
パ研には 5 月下旬に入り、AtCoder のコンテストには 16 回参加しており、レートは 825 の緑コーダーです。(10/21 現在)AtCoder の AC 数は 983 ですが、虚無埋めをよくしているので RPS は 140500 と、AC 数の割には低いです。

diff の現状

最近のコンテストの問題で、結構苦戦した問題が灰 diff だったり、コンテスト中に解けなかった問題が茶 diff だったり…最近文化祭の準備であまり精進できてないのも原因の一つだと思いますが、明らかに昔より同じような問題でも diff は最近の方が圧倒的に低く感

じます。実際、square1001 さんの記事によると、
「問題の難易度はほぼ同じ でありかつ正解率がほぼ同じなのにもかかわらず、解くのに必要なレーティングだけは 250~300 程度下がっているように見える」
とあります。ただ、これを理由にしてしょうがないと言っていてはレートは下がるばかりなので避けたいところです。

注目した観点

では、どのように対策すればいいのかを考えたところ、アルゴリズムを習得することが重要ではないかと思いました。例えば、bit 全探索では、abc173-C の H and V と、abc128-C の Switches は、ほとんど同じような問題ですが、diff は 207 も違います。特に、UnionFind もしくは DFS で解くことができる問題に関しては、ARC032-B の道路工事と ABC177-D の Friends では、diff は 492 違い、UnionFind でしか解けない ABC049-D と ABC120-D では、diff が 698 違います。今回は、この UnionFind と DFS について書いていきます。

DFS とは

DFS は、深さ優先探索 depth-first search の略です。ちなみに、BFS(Breadth-first search) という、幅優先探索もあります。BFS に関しては Cyanmond 氏が書いてくれているようなので、飛ばします。
この DFS は、再帰関数を用います。

再帰関数とは何ぞや

関数は前提知識として話を進めます。
再帰関数は、簡単に言えばある関数の中で同じ関数を呼び出し、再帰的にループをすることです。例を挙げると、

```
int sum(int N){
  if(N==0)return 0;
```

```
  return sum(N-1)+N;
}
```

のように、再帰関数の中で再帰関数を呼び出しています。N に 2 を代入すると、
sum(2) → sum(1)+2 → sum(0)+1+2 → 0+1+2 → 1+2 → 3
のように変化していきます。この関数では、0〜N までの総和を求めています。はっきり言って、N(N+1)/2 の方が早いです。再帰関数は、引数の値が一定の数、もしくは範囲になった時の挙動を決めておかないと、永遠に再帰するので、無限ループが発生してしまいます。

そろそろ本題に入りたいので、DFS の説明をしていきます。グラフ、グラフに関する基本的な用語は前提知識として先に進みます。

DFS の前処理

DFS には、ある程度の前処理が必要です。問題によって多少変わることがありますが、大体一緒です。入力を受け取ることは、当然のことなので、省略します。まず、計算量をできるだけ減らすために、int 型の二次元配列 (以後、G と書く。) を持ち、ある一点からつながっている点を記録しておきます。次に、ある点に対して、その点が探索済みかどうかを調べるために、bool 型の配列 (以後、b とかく。) を持ちます。前処理は以上です。意外と簡単ですね！ (いいえ)

DFS の本質部分

まず、探索する点を受け取ります。このときに、この点が訪問済みであることを示すために b に記録します。次に、前処理で用意した G を利用して、受け取った点からつながっている点が訪問済みか否かを前処理で用意した b で判定し、まだ訪問していない点だった場合、その点を探索する点とくして DFS に送ります。これを再帰的に繰り返すことですべての場所を探索することができます。

なぜこれで無限ループにならないのか

これを証明しようとすると部誌の提出期限に間に合わないので、簡単に説明します。すでに訪問している点には立ち寄らなく、すべての辺を端辺の 2 つから調べるので、オーダー記法で表すと、O(点の数 + 辺の数) となるのです。

UnionFind

やっとメインディッシュです。もう 4 ページ目ですね！この章では一般的な UnionFind、重み付き UnionFind、部分永続 UnionFind の三本立てです。ちなみに、部分永続 UnionFind、重み付き UnionFind に関しては、今偉そうに書いていますが、名前しか知りません。

一般的な UnionFind

一般的という定義はあいまいですが、普通の UnionFind 木です。(結局あいまいなのは変わりませんでした、すみません)
そもそも UnionFind とは、グループ分けされている複数の要素があり、それをグラフとしてあらわしたときに、与えられた 2 つの点同士が連結かどうかを調べることのできるアルゴリズムです。

UnionFind の前処理

UnionFind も、DFS と同様に前処理が必要です。UnionFind の場合は、空間計算量をできるだけ落とそうとすると、入力を受け取るのは本質部分と同時並行のほうがいいので、後で説明します。ただし、問題によって異なり、一度入力されたものに手を加えてから本質部分を行わなければいけないものもあります。まず、int 型の一次元配列 (以後、p と呼ぶ) を持ち、
p[a]=b
の時、a の親が b であることを示しています。これを、$1 \leqq i \leqq N$ を満たす、すべての i に対して、

p[i]=i
とします。次に、もう一つ int 型の一次元配列 (以後、s と呼ぶ。) を持ちます。前処理は以上です。DFS のときと、前処理の量はそこまで変わりません。

UnionFind の本質部分

まず、UnionFind は、構造体を使います。構造体は、データ構造の一つで、
struct 構造体名
型 1 メンバ変数名 1
型 2 メンバ変数名 2
... (必要なだけ書く) ;
という風に宣言します。ここで先程後回しにしていた入力を受け取る作業です。まず、入力で x と y を受け取ったとします。ここで、p を使って再帰的に x と y の根を求めます。このとき、x の根を rx、y の根を ry とします。この時、rx と ry が一緒だった場合は、何もせず、違った場合、s[rx]<s[ry] のとき、s[ry] に s[rx] を足し、p[rx] を ry に、s[rx]>s[ry] のとき、s[rx] に s[ry] をたし、p[ry] を rx にします。次に、連結かどうかは n と m を受け取り、先程と同じように n と m の根を再帰的に求めます。これが等しかった時、n と m は連結、等しくなかったとき、n と m は連結ではないということになります。

UnionFind の計算量

ここで気になるのが計算量です。こんなことして 2sec に間に合うのか？と思いがちですが、制約が N ≦ 100000 のときでも使われていることもあり、多くの場合、500msec 以内に収められます。これも、証明をし始めると期限に間に合わなくなってしまうので、証明は省きますが、UnionFind の計算量はアッカーマン関数の逆関数と等しくなるようです。

そもそもアッカーマン関数の逆関数とは何ぞや

まず、そもそもアッカーマン関数について。(人生初めて Tex command を使いました)

$$A_k(x) : \mathbb{R} \to \mathbb{R}$$
$$A_0(x) = x + 1$$
$$A_{k+1}(x) = A_k^x(x)$$

この関数のことをアッカーマン関数と言います。~~ちなみに、Paken には頭がアッカーマンなどといったネタを飛ばしている人もいます。~~次に、これの逆関数についてです。逆関数があることの証明も省きますが、逆関数があることの証明は、その関数が全単射であれば逆関数が存在するので、興味がある人はやってみてください。ちなみにこれの逆関数は、

$$\alpha(n) = \min\{k \mid A_k(2) \geq n\}$$

です。……正直、どのくらいか想像がつかないと思います。もう少し計算量解析を進めます。まず、アッカーマン関数で
$A_k(2)$
を考えると、k=0 のとき、3、k=1 のとき、4、k=2 のとき、8、k=3 のとき、1024 となっています。ここまではいいのですが、次の k=4 のときは、約 $2^{6.62\times10^{619}}$ と、膨大な数です。このことから、k の増加は非常に遅いため、$A_k(2)$ から k を求めることを考えると、k はほぼ定数と見ていいとかんがえられます。実際、計算のどこかで 約 $2^{6.62\times10^{619}}$ を超えるような数が出てきていたら TLE 確実なので、k が 5 以上になることは可能性が 0 と言っても過言ではないと思います。

重み付き UnionFind... に、行く前に

今まで、DFS と UnionFind について書いてきて、計算量解析も~~証明はさぼりましたが~~してきました。でも結局、UnionFind と DFS ってどっちの方が早いのか、分かりにくいですよね。数学的に比べようと思えば比べられるのかもしれないですが、正直数学がそこまでできないので数学では諦めます。ということで実際に比べてみましょう。

実験問題 1 Friends

まずは DFS で書いたコードを。https://atcoder.jp/contests/abc177/submissions/17577898
152ms です。
次に UnionFind で書いたコードを。https://atcoder.jp/contests/abc177/submissions/17578153
105ms です。~~UnionFind を手で書いたのが初めてだったので 4CE したことはさておき、~~UnionFind の方が速そうですね。もう一つ例を試そうと思います。

実験問題 2 道路工事

先程と同様に、DFS で書いたコードを。https://atcoder.jp/contests/arc032/submissions/17578303
71ms です。
次に UnionFind で書いたコードを。https://atcoder.jp/contests/arc032/submissions/17578382
60ms です。今回も UnionFind の方が速そうですね。

考察

テストケース全体の結果だけから見ると、UnionFind の方がかなり速いように見えます。もちろんですが、N や M が大きいほど実行時間は長いはずで、先程提示した実行時間は最も長いテストケースの実行時間がでてくるので、N や M の値が大きいときには UnionFind の方が早く実行できると分かります。では、N や M の値が小さいときにはどうなるのかを調べていきます。
問題 1 の方も、問題 2 の方も、多くのテストケースで UnionFind の実行時間のほうが速い、もしくは等しいです。ただ、逆に言えば DFS の方が実行時間が速い場合もあるということです。
DFS と UnionFind では得意分野がありそうですね。ここで問題 1 の方のテストケースをみてみましょう。残念ながら問題 2 は、昔の問題のため、テストケースは見ることができません。
問題 1 で DFS の方が実行時間が短いテストケースを挙げると、random02, random31, sumple01 である。random02 は、N が 190000 近くあり、M は 14000 近くしかありません。また、random31 は、N は 600 近くしかないのに、M は 200000 近くあります。sumple01 に関しては、差が 1ms しかないため、誤差と考えていいでしょう。このことか

ら、N と M の差が大きいと DFS の方が速くなる可能性が高いということが分かります。

結論

N と M の差が大きいと DFS の方が早くなる可能性が高くなる。
ちなみに、一生懸命 DFS と UnionFind の計算量の差を調べてきて、上記のような結論に至りましたが、N と M の差が大きいものがあった場合、N と M がどちらも最大に近いものもあるということです。よって、N と M がどちらも最大に近いテストケースがあることが多い競プロでは、そこまで関係ないとも言えますが、結果として、「DFS と UnionFind があるときでは UnionFind の方が速い。しかし、N と M の差が大きいと DFS よりも遅くなることもある。」ということを得られました。

重み付き UnionFind

三本立ての一本目に 4 ページくらい使ってしまいました。やっと二本目です。でも、正直言って重み付き UnionFind って BFS が dijkstra 法になるようなものなので、そこまで変わりません。重み付き UnionFind よりもポテンシャル付き UnionFind の方が呼び名として適していると言っている方もいますが、今回は重み付き UnionFind と呼ぶことにします。今回は、本質ではなく、項目ごとに書いていきます。前処理は別物なので一つの項目として書きます。

重み付き UnionFind の前処理

一般的な UnionFind の項でもでてきた、p, s に加えて、Abel 型の一次元配列 (以後 d と呼ぶ) を持ちます。p, s は先と同じように処理し、d は自分で決めた値 (0 の場合が多い) で初期化します。

経路圧縮

x を受け取ります。一般的な UnionFind の項で説明した再帰的に求めていくことをこういいます。このときに d[x] に d[par[x]] を足します。

diff の計算

x と y を受け取ります。x と y の差を調べる時には、y のノードの重みから x のノードの重みを引けばいいです。

各ノードの重みの取得

x を受け取ります。経路圧縮をしてから d[x] を調べればよいです。

併合

x, y, w を受け取ります。まず、w から y と x の diff を引きます。次に、rx を x の根、ry を y の根とし、rx==ry ならここで中断します。r[x]<r[y] のとき、x と y を swap 関数を用いて入れ替え、x の符号を入れ替えます。r[x]==r[y] のとき、r[x] に 1 を足します。最後に、p[y]=x, d[y]=w とします。

差分制約系の問題を解く

この重み付き UnionFind では、差分に制約のある問題を解くことができます。通称、牛ゲーと呼ばれています。差分制約系の問題とは、このような問題のことです。
$https://atcoder.jp/contests/abc087/tasks/arc090_b$
差分制約系の問題では、l[i] から r[i] に d[i] の線を引き、ノードの重さを求めることで、矛盾が発生していることを見つけ出すことができます。

部分永続 UnionFind

ここを理解するのに非常に時間がかかりました。まあ、このアルゴリズムを使用する例題として、橙 diff の問題がよく挙げられているので、かなり難しいアルゴリズムなのかなと思います。そもそも部分永続 UnionFind は、ある時刻に与えられた 2 つの点が連結かどうかを調べられるアルゴリズムです。一般的には、入力を受け取って、sort した後に UnionFind をすることで求められる場合がありますが、中には、入力を受け取った順番で操作をしないと解けない問題もあり、このような問題は、sort をすると順番がバラバラになってしまうので、解けなくなってしまいます。部分永続 UnionFind は、このような

sort をしてから UnionFind をすることでは解けない問題を、sort せずに解けるアルゴリズムです。

部分永続 UnionFind の前処理

まず、p と r を用意します。次に、現在の時刻を記録するための整数 (以後、n と呼ぶ)、親がいつ更新されたかを記録するための int 型の一次元配列 (以後、t と呼ぶ)、その頂点を根とする木の頂点数と時刻を記録する pair 型 (中はどちらも int 型) の二次元配列 (以後、N と呼ぶ) を持ちます。p,r は前と同様、n は 0、N には全ての n[i](0 ≦ i ≦ n.size()) に 0,1(時刻 0 には期の頂点数が 1 という意味) を pushback し、t はすべての t[i](1 ≦ i ≦ t.size()) 非常に大きい数 (1e9+7 など) にします。

根を探す

頂点 x と、時間 T を受け取ります。時刻 T においての、x の根を求めるのですが、これも再帰的にしていきます。T<t[x] のとき、まだ x の親が更新されていないということなので、x は木の根になっているので、x を返します。そうでなければ、再帰的に、p[x] と T で探します。

木のサイズ

頂点 x と、時間 T を受け取ります。まず、rx を x の根として、頑張って二分探索をして、知りたい時刻 z を出します。このとき、N[rx][z] を返すことでできます。

併合

まず、時間が 1 進んだので、n に 1 を足し、rx を x の根、ry を y の根とします。rx==ry なら、そこで操作をやめ、r[rx]>r[ry] ならば、p[ry]=rx、t[ry]=n にして、N[rx] に n, rx の現在の木のサイズ +ry の現在の木のサイズを pushback します。そうでなければ、p[rx]=ry、t[rx]=n にして、N[ry] に n, rx の現在の木のサイズ +ry の現在の木のサイズを pushback します。また、r[rx]==r[ry] なら、r[rx] に 1 を足します。

部分永続 UnionFind の計算量

部分永続 UnionFind の計算量は、まず、木のサイズを求めるのには、二分探索をするため、O(logN) になります。根を探すのも、rank をベースにしているため、O(logN) で済みます。併合は、根を探すことと、木のサイズを求めることをどちらもしており、これらがボトルネックなので、O(logN) でできます。

おわりに

10 ページを超える文章を読んでいただき、ありがとうございました。最初の 3 つのデータ構造は水 diff 以下、部分永続 UnionFind は橙 diff 程度 (例が少ないので判断しにくいですが) の問題でもでてきます。難しいと思っても、早めにアルゴリズムを覚えておくことで、パフォーマンスがかなり高くなることがあるので、頑張って覚えましょう！ほかにもアルゴリズムはたくさんあるので、基本的なアルゴリズムを知りたい場合は、最近出版されたけんちょん本と呼ばれるもの、多くのアルゴリズムを知りたい場合は、蟻本と呼ばれているものを買うことをお勧めします。それでは、また来年部誌を書くときにお会いしましょう！

参考文献

square1001 さん「AtCoder の色到達難易度は本当に上がったか?」
https://drive.google.com/file/d/1OQqJsd33UVq7FIznkGrDOEBUGa3-hUZs/view
camypaper さん「やぶについて書きます」
https://camypaper.bitbucket.io/2016/12/18/adc2016/ kopricky さん「UnionFind の計算量の話」
https://qiita.com/kopricky/items/3e5847ab1451fe990367 Misteer さん「部分永続 UnionFind の実装」
https://misteer.hatenablog.com/entry/persistentUF
drken さん「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」
https://qiita.com/drken/items/cce6fc5c579051e64fab

暗号理論入門

71st kenkenken2004

あいさつ

今年筑駒に入学しました高校一年の kenkenken2004 です。

AtCoder 歴 5 カ月の緑コーダーですが、夏休みが終わってからずっと不振状態で、半ば現実逃避で整数論や CTF、CG に手を出し始めてます。今日は、CTF（セキュリティコンテスト）の一分野でもある暗号についてやっていきたいと思います。

暗号とは何ぞや

暗号については皆さんはご存じでしょうか。

暗号とは、

「他の人に通信の内容を知られないようにデータを変換する手段」のことです。

簡単なものだと、「たぬき」があります。これは、要は伝えたい文の文字の間に「た」をたくさん入れ、一見言語として成り立たないようにするものですね。知っている方は多いでしょう。ですが、これは「たぬき」の暗号であることを知っていれば誰でも解読できるので全くもって安全ではありません。

このような暗号で重要な通信を行うわけにはいきません。古今東西、特に戦争では「情報」は非常に重要なものです。だから、ブルータスの凶刃に斃れたカエサルより前の時代から現代に至るまで人々は暗号を発展させ続けているのです。今回は、いろいろな暗号の中から、代表的なものを紹介します。

暗号の分類

一口に暗号といっても、その手段によって大まかに 2 つに分類することができます。

古典暗号　：1950 年代にコンピュータが実用化されるより前の暗号。

現代暗号　：コンピュータの時代が到来したことで使用されるようになった数学的な暗号。

古典暗号は基本的に鍵すなわちパスワードが無く暗号化方式自体を秘密にして使うことがほとんどです。であるので、方法がバレた時点でジ・エンドです。それに対し現代暗号はほぼ全てが鍵を用いて暗号化します。そしてアルゴリズムも古典暗号と比較にならないほど複雑なので極めて安全です。

また、古典暗号は

置換式暗号　：1 文字ずつある規則に従い他のものに置き換える暗号。主に筆記で行う。
転置式暗号　：ある規則に従い元の文を並び替える暗号。主に筆記で行う。
機械式暗号　：上の二つを複雑に組み合わせた暗号。基本的に専用の機械で行う。

に、現代暗号は

共通鍵暗号　：暗号と復号（元に戻すこと）に使う鍵が同じ暗号。
公開鍵暗号　：暗号と復号に使う鍵が異なる暗号。

に分けることができます。では、それぞれの代表的な暗号を説明していきます。

シーザー暗号

名前の通り、古代ローマの偉大な軍人カエサルが使っていたとされる暗号です。置換式暗号で、アルファベットを全て一定だけシフトさせるという非常に単純な暗号ですね。

一番有名なのは A を D、B を E、というように 3 文字右にずらすというものでしょう。

例）TSUKUKOMA　⇒　GFHXHXBZN　（１３文字右にシフト）

スキュタレー暗号

古代ギリシア、「スパルタ教育」の語源として有名なスパルタが用いていた暗号です。

転置式暗号で、決められた直径の円筒に細長い羊皮紙を巻き付け、書き込んでいました。

これによって文の各文字は飛び飛びで羊皮紙に記されます。

とはいえ意味不明の文字の羅列が書かれた細長い羊皮紙では簡単に方法が推測されたようです。

例）TSUKUKOMA　⇒　TKSOUMKAU　（1 文字飛ばし）

エニグマ

エニグマとは、第二次世界大戦中ナチス・ドイツが使用していた機械式暗号のことです。
エニグマはコンピュータの登場以前の古典暗号の集大成と言うことができます。
連合軍はこれにひどく悩まされましたがイギリスのチューリングが解読しました。
連合軍は解読したことを徹底的に秘匿したため、ナチスは使い続け、情報がダダ洩れだったようです。
エニグマは暗号表が刻まれた複数のローターとプラグボードの配線によって暗号化します。シーザー暗号とは違い、変換のための表が複数あるので、安全性も桁違いに高いです。

Vernam 暗号

一応時期的には古典暗号に入りますがこれはかなり異質な暗号で、他の古典暗号はおろか現代暗号を含めたほぼ全ての暗号と一線を画してます。
他の現代暗号は「計算量的安全性」という性質を持っています。要は、原理上解読は可能であるが現在の技術では数十億年のような非現実的な時間がかかる、ということです。ですが、結局はコンピュータが進歩すれば解かれてしまうのです。
実際に、40 年ほど前の暗号はすでにレッドゾーンです。
それに対しこの暗号は「情報理論的安全性」というものを持っています。これは、
「理論的に絶対的に安全であり、無限の計算資源・時間があったとしても鍵が無ければ解読不可能」
というものです。
まるで究極の暗号ですが、どのような方法なのでしょうか。
それは、排他的論理和です。排他的論理和とは、２つの、０と１の２種類の入力に対しもし同じ種類なら０を、違うなら１を返すものです。まずデータと同じビット長の乱数を用意して鍵にします。そしてデータと鍵の対応するビットの排他的論理和をとることで暗号化できます。元の平文のビットは鍵によって左右されるため、暗号文からは何もわかりません。だから鍵が無くては平文を当てられる確率はｎビットだと２のｎ乗分の１です。これは暗号の安全面から考えてこれ以上ない理想値です。
しかし、Vernam 暗号にはその利点を遥かに上回るような欠点があります。
平文と同じ長さの鍵を用意し安全に送信しなければならないのです。

つまり、平文と同サイズの鍵を安全に送る技術があるのなら、
直に平文を送ればいいわけで、、、
というわけで現在はほぼ役に立っていません。
ですが、近年は後述の量子暗号における応用で再び日の目を見れそうです。
例）101010101111 ⇒ 000100011001（鍵：101110110110）

AES

代表的な共通鍵暗号といえばこれでしょう。米国の標準暗号でもある AES は無線 LAN の通信の暗号化などで大活躍しており、今この文章を読んでいる人でその恩恵を受けていない人は一人もいません。

以前の標準暗号である DES は残念ながらムーアの法則に忠実に進化してしまったコンピュータのせいで計算量的に寿命が来てしまいお役御免になりましたが、AES は量子コンピュータとかいうチートが実用化されない限り少なくとも 10 年以上は現役でいられるといわれています。

AES はブロック暗号と呼ばれる種類の暗号で、一定のサイズにデータを分けて暗号化するものです。詳しく説明すると長くなるしそもそも僕が完全に AES を理解しているわけでもないので簡単に手順を説明します。

1　　S ボックスという換字表に基づく 1 バイトごとの置換
2　　4 バイト単位の行を一定の規則で左シフト
3　　ビット演算による 4 バイト単位の行列変換
4　　ラウンド鍵との排他的論理和

もう訳がわかりませんね。訳が分からないからこそ安全性が高いのですが。

ですが、大活躍の AES も共通鍵暗号であるので、「鍵をどうやって送るか」という問題に悩まされます。鍵を新たに共通鍵暗号で送るとその新しい鍵は、、、と無限ループになり、最終的には直接送る以外に手段が無くなります。それを解決するのが次説明する公開鍵暗号なのです。

RSA

RSA 暗号は最も有名といえる公開鍵暗号です。
「暗号の分類」でも述べた通り、公開鍵暗号は「暗号に使う鍵と復号に使う鍵が違う」暗号です。一体どういうことなのでしょうか。RSA 暗号を例にみてみましょう。

1　適当に正の整数 e を決定する。通常は 65537 を用いる。
2　大きな素数 p と q を決定する。
3　p と q の積 n を求める。
4　e にかけると「(p-1) × (q-1) で割った余り」が 1 になるような整数 d を求める。
5　e と n が誰でも知ることができる公開鍵、d が二人だけしか知らない秘密鍵である。
6　平文 m を暗号化するときは、「m の e 乗を n で割った余り」を求める。
7　暗号文 c を復号するときは、「c の d 乗を n で割った余り」を求める。

なぜ、暗号化と復号で鍵が違うのに正しく復号できるのでしょうか。まず、暗号化を数式で表すと
$c \equiv m^e \pmod{n}$
($b \equiv a \pmod{n}$ は「b を n で割った余りが a を n で割った余りと等しい」のこと)
になります。また、復号を同様に表すと
$m \equiv c^d \pmod{n}$
になります。平文 m を暗号化しさらにそれを復号したものを m' とすると、
$m' \equiv m^{de} \pmod{n}$ となります。
ここで、「オイラーの定理」というものを使います。これは、
$a^n \equiv a^{\phi(n)} \pmod{n}$ （$\phi(n)$ は「n 未満の n と互いに素な自然数の個数」）
という定理です。これを使うことで、
$m' \equiv a^{de \pmod{\phi(n)}} \pmod{n}$　と表すことが可能になります。さて、$\phi(n)$ はいったいどのような値なのでしょうか。まず、n は半素数すなわち 2 つの素数の積です。
つまり、「n 未満の自然数で p の倍数でも q の倍数でもないものの個数」が $\phi(n)$ です。
これは、集合を考えることで、
$\phi(n) = n - n/p - n/q + n/(pq) = pq - p - q + 1 = (p-1)(q-1)$　と表せます。
おや、どこかで見かけた式が出てきましたね。そう、手順 4 の「秘密鍵 d の定義」で出てきました。ここにはさらに興味深いことが書いてあります。
「d と e の積は $(p-1)(q-1)$ で割ると余りが 1 になる」のです。

これを使うことで、
$m' \equiv a^1 \pmod{n}$　になります。
さらに $m, m' < n$ とすると $m = m'$ になります。よって暗号文が正しく復号されました。

さて、正しく暗号化・復号ができることはわかりました。では、なぜこうすると安全なのでしょうか。それは、「素因数分解」を解くことは人間にとってだけではなくコンピュータにとっても非常に困難だからです。素因数分解は皆さんもご存じの通り (そのはず) 自然数を素数の積で表すことですが、この素数がくせ者なのです。素数の出現は今のところ法則性がわかっていないのです。近いのはリーマン予想ですが、いまだ証明されていません。気になる人はググってください。素数に法則性が無い、これはつまり素因数分解は当てずっぽう・総当たりで答えを探すしかないということです。これは我々人間にとってだけでなくコンピュータにとっても大きな負担です。現在のコンピュータの性能では、現在 RSA で使われている 1024bit すなわち約 310 桁以上の半素数の素因数分解は数十億年以上かかるためとても現実的ではありません。

どうして素因数分解が RSA の安全性につながるのか疑問をもっている人は前ページをよく読んでみてください。わかりましたか？そう、秘密鍵 d を計算するためには p と q が必須なのです。知っている数が n だけでは素因数分解をする以外に方法はありません。ですが、p または q を知っているなら一瞬で d が求まります。

実際の運用の際には、受信側が鍵を生成し、送信側に公開鍵を送ります。送信側は受け取った公開鍵で暗号化し、暗号文を受信側に送信します。そして受信側が秘密鍵で復号します。この過程で秘密鍵は外部へ一切出ていないので、安全なのです。

これだとまるで共通鍵暗号がいいとこなしのように聞こえます。ですが、公開鍵暗号にもかなりヤバい欠点があるのです。それは、計算時間が共通鍵暗号よりとても長いのです。これでは数秒で世界中を情報が駆け巡る安全なネットワークを構築することはできません。どうするのか。そこで、共通鍵暗号でデータを暗号化し、公開鍵暗号でその共通鍵暗号の鍵を送ります。共通鍵暗号は鍵の送受信が保障されれば安全であるし、鍵の長さはたかが知れているので公開鍵暗号も高速に動作できます。このように組み合わせて使ったものを「ハイブリッド暗号」といいますが、２つの方式が互いに短所を補い長所を伸ばしあっており、欠点らしい欠点が無い素晴らしい暗号なのです。

量子暗号

さて、これまでで述べた AES や RSA などの現代暗号は全て「計算量的に」安全であるといわれています。つまり、今後技術が進歩すれば既存の暗号方式は一瞬で解かれてしまう日が来てしまいます。そのためにさらに新しい暗号方式を開発してもしばらくするとまた引退する日が来ます。まるでいたちごっこですね。これに終止符を打つことはできないのでしょうか。実は、あります。それが量子暗号です。これは量子力学という、ミクロの世界の法則を応用した暗号方式です。最近 (2020 年秋) 東芝が事業化したらしいので、聞き覚えのある人もいるでしょう。さて、この量子暗号ですが、AES・RSA と違い「情報理論的安全性」を持っています。おや、見覚えがありますね。そう、古典暗号で述べた Vernam 暗号も情報理論的安全性を持っているのです。つまり、量子暗号はたとえ無限の計算リソースと無限の時間を持っていても破ることが不可能なのです。そして、量子暗号は Vernam 暗号と違い、理論上は有効な使い方をすることができます。現時点ではまさしく「夢の暗号」と言えるでしょう。どのように暗号化しているのでしょうか。

実は、これは厳密には量子「暗号」ではなく量子「鍵配送」です。つまり、この技術自体は暗号技術ではないんですね。そこで、前述した Vernam 暗号などを使います。Vernam 暗号も情報理論的安全性を持っているため、理論上は無限のリソースをもってしても突破することはかないません。では、量子鍵配送のうちの 1 種類について詳しくみていきましょう。

まず、基本的な暗号方式は「ワンタイムパッド」です。「ワンタイム」とあるように、一回限りの使い捨てです。平文の 1 単位それぞれに乱数を割り当てて使うのです。Vernam 暗号もその一種です。さて、量子鍵交換ではこのパンタイムパッドをどのように送るのでしょうか。現在のデータの媒体は基本的に電子、または光ファイバーの光子です。量子通信では、光子を使用します。ただ、普通の光ファイバーと違い 1 ビットのデータはたった 1 粒の光子だけで運びます。どのように情報を光子に保持するのか。それは、偏光方向です。偏光方向とは、粒子でも波でもある光の、波の振動方向のことです。偏光方向の傾きで情報を伝えられるのです。ですが、これも結局は盗聴されるように思えます。そう、盗聴「は」できてしまうのです。量子鍵交換の本質は、「盗聴を防ぐ」のではなく、「重要な情報の漏洩を防ぐ」ことにあります。量子暗号では、「量子のもつれ」という、２つの粒子が強い相関関係にあり常に互いに影響を及ぼすことと、量子の状態は不確定で観測して初めて定まるという性質が応用されています。(僕もチンプンカンプンなので詳しい説明

はしません)

1　第三者が光子を盗聴すると、不確定な状態だった光子の状態 (位置と運動量のどちらか) が確定する。
2　量子もつれによって、送信した光子が変化したことがわかり、
想定外に観測された＝盗聴者の存在がわかる。
3　同時に盗聴された情報の内容も把握できるので、その部分を廃棄し、新たに別の偏光でまた送信する。
4　これによって最終的には全てのデータを送信できる。

要は「盗聴された？データを捨てればいいじゃない」ということですね。非常に低速になりますが、安全性は飛躍的に高まるので欠点を補ってあまりあるでしょう。東芝でも事業化が開始されたことですし技術革新の日は近いのかもしれませんね。

終わりに

皆さんは授業で個人情報の漏洩などの危険性と対処について「習う予定だ」「習っている」「習った」と思いますが、情報漏洩と暗号は非常に深い関係があります。ブラウザを起動してインターネットを使うとき、現代の情報化社会の根底を支えている暗号技術について思いを馳せてみてください。

駄文を長らくお読みいただきありがとうございました。

機会があれば来年また会いましょう。

01-BFS の問題を BFS で解いてみる

74th Cyanmond

はじめに・自己紹介

初めまして。74th(中一ですね) Cyanmond です。競技プログラミングは、6 月の終わりごろから始めています。丁度 4 か月ほどですね。感慨深い。

なぜか突然部誌を書きたくなってきたので、書くことにします。

僕の競技プログラミング経験が少ないこともあり、(未経験の方はともかく) ある程度の知識があれば楽に読めるレベルの内容だと思います。

この記事の内容が役に立つかはさておき、一人でも読んでくれる人がいたら Cyanmond が泣いて喜びます。

ちなみに、執筆開始現在締め切り 3 日前ですね。急いで書くので、クオリティはお許しください。

なぜに BFS・01-BFS

はじめ、Fenwick tree についての解説をしようと思ったのですが、想像していたより自分の理解が浅く、また奥が深い (セグ木ほどではありませんが) データ構造なので、部誌の中で詳しく解説するのは難しいかな、と思いました。

それに比べれば BFS や 01-BFS は比較的簡単な題材であり、また今回面白い検証を思いついたからです。

決して僕が BFS が好きだからではないです。

導入...BFS とは

BFS とは、グラフの一点対全点最短経路を求めるアルゴリズムのうち「全ての辺が重み 1 のグラフ」に限定して用いることのできるアルゴリズムです。

詳しい解説は、僕の睡眠時間と部誌のページ数インフレ防止のためにここではしません。

インターネットを探せばとても分かりやすいものが数多くあるので、詳しい解説を知りたい方はそちらをご覧ください。

サンプルソースだけ、記載しておきます。サンプルソースは、入力が 0 indexed として与えられるとしています。

```
#include <bits/stdc++.h>
using namespace std;

queue<int> que;
void dfs(vector<vector<int>> &graph, vector<int> &ans) {
    que.push(0);
    ans[0] = 0;//0 スタートなので、頂点 0 は距離 0 とします
    while(!que.empty()) {
        int s = que.front();
        que.pop();
        for (int i = 0; i < (int)graph[s].size(); ++i) {
            if (ans[graph[s][i]] > ans[s] + 1) {
                ans[graph[s][i]] = ans[s] + 1;
                que.push(s);//更新できれば更新し、queue に push する
            }
        }
    }
    return;
}

int main(void) {
    int n, m;//n は頂点数、m は辺の数を表します
    //今回は、有向辺だとして実装します。無向辺の場合は有向辺を双方向に作ります。

    cin >> n >> m;
    vector<vector<int>> graph(n);//隣接リスト形式
    for (int i = 0; i < m; ++i) {
```

```
        int a, b; cin >> a >> b;//a → b の有向辺を表します
        graph[a].emplace_back(b);
    }
    vector<int> ans(n, 1000000);
    dfs(graph, ans);//ans に、頂点 0 からの最短距離を記録
    for (int i = 1; i < n; i++) {
        cout << ans[i] << endl;//0 から それ以外の頂点への最短距離を出力
    }

    return 0;
}
```

01-BFS is 何

01-BFS は、グラフの一点対全点最短経路問題のうち、グラフの重みが 0 or 1 のグラフに限って使える、効率的なアルゴリズムです。通常の BFS と同じく、一度更新した値は変更されることがありません。BFS との違いを説明すると、

- BFS...(queue などのデータ構造を用いて)、暫定最短距離の小さい順に探索する手法
- 01-BFS...(deque などのデータ構造を用いて)、重み 0 の辺をたどって探索できる頂点を優先的に探索することで最短距離を求める手法

探索する頂点を記録するとき、「その頂点にたどり着いたときに使った辺の重みが 1 なら deque の最後に挿入、0 なら deque の最初に挿入」という風にすることで、「暫定最短距離が更新されることはない」というのを実現しています。

こちらも、詳しい解説はインターネット上にわかりやすいものが数多くあるので、詳しい解説は省きます。サンプルソースを記載しておきます。こちらも 0 indexed です。

```
#include <bits/stdc++.h>
using namespace std;
```

```
deque<int> deq;

void BFS01 (vector<vector<pair<int,int>>>& graph, vector<int> &ans) {
    deq.push_front(0);
    ans[0] = 0;
    while(!deq.empty()) {
        int s = deq.front();
        deq.pop_front();
        for (int i = 0; i < (int)graph[s].size(); ++i) {
            if (graph[s][i].second == 0) {//cost = 0
                if (ans[graph[s][i].first] > ans[s]) {
                    ans[graph[s][i].first] = ans[s];
                    deq.push_front(graph[s][i].first);
                }
            }
            if (graph[s][i].second == 1) {//cost = 1
                if (ans[graph[s][i].first] > ans[s] + 1) {
                    ans[graph[s][i].first] = ans[s] + 1;
                    deq.push_back(graph[s][i].first);
                }
            }
        }
    }
    return;
}

int main(void) {
    int n, m; cin >> n >> m;//変数の意味は前述の BFS と同じです
    vector<vector<pair<int,int>>> graph(n);
    for (int i = 0; i < m; ++i) {
        int a, b, cost;//cost は辺の重みを表します。0 or 1 です。
        cin >> a >> b >> cost;;
        graph[a].push_back(make_pair(b, cost));
```

```
    }
    vector<int> ans(n, 1000000);
    BFS01(graph, ans);
    for (int i = 1; i < n; ++i) {
        cout << ans[i] << endl;
    }

    return 0;
}
```

ここで、あることに気が付きました。 ~~だれでも気が付くことをもったいぶるんじゃない~~ 。

01-BFS で解ける問題は、効率を考慮しなければただの BFS と同じように解くことができる!!

まあ、できなければ、直感的に嫌な気持ちになると思います。
どうせだから、検証をしてみましょう。それは、

01-BFS が想定解の問題で、BFS で AC することができてしまう問題はあるのか?

ということです。それでは、さっそく検証していきます。

01-BFS を使う問題

ここから先、AtCoder の問題を載せるので、一部ネタバレになってしまうかもしれません。

器物損害！高橋君

Link ... https://atcoder.jp/contests/arc005/tasks/arc005_3

Small Multiple

Link ... https://atcoder.jp/contests/abc077/tasks/arc084_b

Wizard in Maze

Link ... https://atcoder.jp/contests/abc176/tasks/abc176_d

3 つ挙げてみました。すべて、deque を使った 01-BFS で一度 AC した問題です。これらについて、deque の部分を queue に変え、BFS 解法に直してそれぞれ提出してみました。

結果

3 問それぞれ、01-BFS バージョンと BFS バージョンで提出した結果です。それぞれの ver で変わっているのは、queue と deque の部分のみです (ただ、wizard in maze は実装の都合上コードにかなり変更があります)。

全て単位は msec

品目	器物損害！高橋君	small multiple	wizard in maze
01-BFS	34	32	210
BFS	430	46	TLE(2000)
問題で一頂点から出る辺の最大本数	4	2	25

https://atcoder.jp/contests/arc005/submissions/17561937
https://atcoder.jp/contests/arc005/submissions/17561961
https://atcoder.jp/contests/abc077/submissions/17562543
https://atcoder.jp/contests/abc077/submissions/17562545
https://atcoder.jp/contests/abc176/submissions/17560598

https://atcoder.jp/contests/abc176/submissions/17560608

あれ、普通に 2 つも AC しちゃってる...

正直、検証対象の絶対数が少ない (僕の精進不足のせいですね... 笑) ので確かなことは言えませんが、一頂点から出る辺の本数が多いと 01-BFS と BFS での所要時間の差が大きいのは、(一頂点から出る辺の本数が多い問題だと)、暫定最短距離が更新されたとき (01-BFS をただの BFS で解くと暫定最短距離の更新が発生し、修正に余計な計算が発生する) に修正する頂点数も多くなるのが原因だと考えています。

また、テストケースにも依ります。例えば、wizard in maze の BFS 解法では、TLE が出たのは 1 ケースだけで、残りのケースは 1sec 以内に収まっています。

(ちなみに、この問題は適切な定数倍高速化を施すと多少実行時間が改善しましたが、TLE はそのままです。)

テストケース名を見るとランダムケースのようなので、テストケースが弱ければ、BFS 解法でも普通に AC ができたかもしれません。実行時間がケースによってここまで変わるのは、グラフの構成によっても、前述したような時に修正する頂点数が大幅に変わり、計算量もそれに伴って変わるからだと考えています。

で、お前は何が言いたいの?

今回検証してみて気が付いたこと・感想は、「意外と 01-BFS って BFS の実装でも間に合うことがある」ということと、「今後もし自分で 01-BFS の問題を作るときがあれば、できれば BFS での解法が落ちるようなテストケースを入れよう！」ということです。(もちろん、後者は問題設定によっては不可能な場合もあると思います)

残念ながら、大して労力も変わらない割に時間計算量面でかなり危険なので、自分が実際に contest で実装するときは普通に 01-BFS を書くことになるでしょう。唯一、グリッド上での 01-BFS 問題は、普通の BFS で実装することで結構実装が軽くなるので、早解きが必要な場面では使えるかも (???) しれないですね！もちろん、TLE には気を付けてください。

おわりに

こんな中一 and LaTeX 記法初挑戦の人が作った 2 時間 クオリティの記事を温かい目で最後まで読んでくださり、ありがとうございます！
記事の内容に致命的なミス・欠陥があったら、教えてくださると助かります (ただ、Paken の部誌公開形態を詳しく把握していないので、修正ができるかはまだわかりません)。

来年は、もっとガッチガチの数学だったり競プロだったりの記事を書きたいですね！書けるほど成長していればいいですが。

最後に、この記事を書く機会を提供してくださった Paken 先輩のみなさん、検証対象として使わせていただいた問題を作っていただいている AtCoder 社のみなさんに、感謝します。
それでは、また来年！僕に部誌を書く元気があればお会いしましょう！

クエリ問題は一つの世界です。

70th Thistle

はじめに

もう高校二年生になってしまった Thistle です。

もうすぐパ研を引退することになりそうですが、多分居座るのでまだ感慨とかはありません。

今回の部誌のテーマは競プロのクエリ問題に使えるテクニックについてです。

私が問題を解くときの思考を出来るだけ言語化してみました。あくまでも私独自の解き方であり、これに絶対の根拠があるというわけではないのでご注意ください。

クエリ問とは

クエリ問とは、まず何らかの情報が大量に与えられた後、それに関する情報を求める回答クエリが大量に投げられる競プロの問題の種類のことです。

クエリ系の問題は大きく 2 つにわけることができます。
『前処理で出来るだけの情報を集め、クエリ 1 つ 1 つでは少ない計算量で解く問題』と、
『クエリをあらかじめ先読みすることで一気に早く解く問題』です。
この記事では主に前者を説明します。
後者はクエリソートをしたりすれば普通の問題と同じように解くことができます。

まず制約を読もう

多分、こんなのあたりまえだと思った人が多いと思いますが、これは考察するうえで非常に大事です。例えば、ある問題の制約が $1 \leq N \leq 10^5, 1 \leq Q \leq 10^5$ (N は要素数、 Q はクエリ数) だったとしましょう。

この時、制約を見ただけで、「大体前処理に $O(N \log N)$ くらいかけて、クエリ解答を $O(\log N)$ か $O(\sqrt{N})$ か $O(1)$ 程度に抑えるんだろうな」というあたりが付きます。そうなると、$O(\log N)$ なら大体セグメント木などの二分木を使うことになりますし、$O(\sqrt{N})$ なら平方分割することになりますし、$O(1)$ なら答えが一つの数式で求まることが分かります。

そして、ここまでわかれば前処理で何が必要になるかが見えてきます。平方分割と $O(1)$ はもう考察することが少ないですし、$O(\log N)$ でもセグメント木を上手く使ってあげる、ということがほぼ確定となり、考察の道筋がかなり絞られます。考察の概形が決まっていることはとても有利です。

Q の制約が比較的大きい状況を例に出して説明しましたが、Q の値が小さい (100 程度など）の時にも応用することができます。（ただし、取れる選択肢が増えるので、Q の値が大きいときほどの有用性は感じられません）

クエリ計算に必要な情報をエスパーする

前のセクションで、クエリ当たりにかけることのできる計算量から、どんな前処理をすればよいかが絞られるという話をしました。
このセクションはその続きで、計算量から大体絞ったクエリ計算方法をより具体的に特定する方法についてです。
$O(1)$ は特にやることがないですし、$O(\sqrt{N})$ も平方分割するということが分かれば後は実装をすればよいです。また、クエリ当たり $O(N)$ の問題などは出来ることが多すぎて体系化が出来ません。
なので、ここでは $O(\log N)$ でクエリを処理するときの考え方についてです。

そもそも、競技プログラミングで log が出てくるとき、その原因は数えるほどしかありません。二分探索、三分探索、ソート、分割統治、データ構造をマージする一般的なテク、調和級数、GCD 等の繰り返される掛け算、LCA、重心分解、SparseTable、ダブリング、セグメント木、平衡二分探索等です。（挙げてみたら思ったよりありましたね）

この中で、ダブリング、SparseTable、セグメント木以外はクエリ問では出てこないか、出てきてもあまり本質的でないことが多いので説明はしません。（重心分解、LCA は出て

くることもありますが、これは特殊な場合だけなので割愛します）
また、ダブリング、SparseTable もほとんどの場合セグメント木で同様の操作が出来るので、ここでは説明しません。
なので、セグメント木を使う場合だけの説明を行います。

セグメント木について本質的なことは、区間に対する演算ができるということです。基本的にプログラミングで少ない計算量で取得できる情報は一要素アクセス程度なのですが、セグメント木では連続する区間の取得ができます。これを出来るだけ利用したいので、問題を考えるときは如何にして連続する区間に対する処理の集合にクエリ計算を落とし込むかを考えることになります。

ここから先は、具体的な問題名を出して説明します。問題のネタバレが嫌いな人は注意してください。

JOI2008 Typhoon

問題概要

地点 l_i から r_i までを襲う、という台風の情報が N 個与えられます。
この時、Q 個のクエリそれぞれについて、「台風 p_i から q_i のうち、地点 a_i を襲った台風の数」を求めなさい。

制約：
$1 \leq N, Q \leq 10^5$

さて、まず制約の確認をしましょう。Q は 10^5 以下なので、クエリ当たり大体 $O(\log N)$ 程度で解ければいいことが分かります。
log を使うアルゴリズムが軒並み壊滅するので、使えるのはセグメント木程度です。
というわけで、セグメント木で解くために、この問題を区間の処理の集合に分解します。
まず思うことは、与えられる台風の情報と答えるクエリの情報がともに区間だということです。

これをうまく使いたいのですが、2 つあるのでまずクエリで与えられる方、つまり台風

番号の区間を活かすことを考えます。
区間を活かすためにとりあえず「l_i 以上 r_i 以下の要素の数」を求めるようなセグメント木に台風番号を乗せます。そうすると、いま足りない情報は地点 a_i を含む台風かどうか、ということです。
これは、クエリを a_i の昇順でソートすることで現在どの台風を考える必要があるかを列挙することが出来るので、あとはセグメント木のクエリ処理を使うことによりこの問題を解くことが出来ました。

では次に、クエリで与えられないほう、つまり台風の襲う区間を活かすことを考えます。
適当に区間を座標圧縮しておき、セグメント木に乗せます。
この時に、セグメント木の節にはソートした頂点番号の列を持たせます。
こうすることで、クエリを計算するときに、見る節それぞれについて頂点番号列を二分探索することで、$l_i \leq x \leq r_i$ なる番号の台風の数を求める事が出来ます。

また、この問題は平方分割をする事によりもっと楽に解くこともできます。

JOI2017 Long Mansion

問題概要

連続する N 個の部屋それぞれについて、入るために必要な鍵の番号とその部屋に転がっている鍵の番号の一覧が与えられます。
Q 個のクエリそれぞれについて、部屋 l_i から r_i まで移動できるかどうかを求めなさい。

制約：
$1 \leq N, Q \leq 10^5$

さて、まず制約の確認をしましょう。
Q は 10^5 以下なので、クエリ当たり大体 $O(\log N)$ 程度で解ければいいことが分かります。
log を使うアルゴリズムが軒並み壊滅するので、使えるのはセグメント木程度です。
というわけで、セグメント木で解くために、この問題を区間の処理の集合に分解します。

まず、$l_i < r_i$ のクエリのみだと仮定します。逆の場合は左右反転することで解くことができます。とりあえず、l_i から r_i に向かうとき、それぞれの部屋は左から入れるかどうかだけ考えればよいです。（右から入る場合、すでに左から入っているはずなので）

では、何の情報が分かればクエリが求められるかを考えます。
$O(\log N)$ で出来ることは区間を定数個見る程度です。
これで得られる情報の内、クエリに役立つものを考えると、
「ある部屋 x に入るためには始まりの部屋がどこより左であればよいか」という情報を思いつきます。（思いついて）
これが分かれば後はセグメント木の区間最小値クエリで、$l_i < x \leq r_i$ の x についての値の最小値が l_i 以上かどうかを判定すればよいです。

それでは、この値を求めます。
ここからはクエリ問としての特性が薄れるので簡単な説明に留めます。
全ての部屋 x について、その部屋に左から入るためにはどこより左の部屋まで行けていれば良いかを求めます。
この部屋を $A[x]$ と呼ぶことにします。
同様に、全ての部屋 y についてその部屋に右から入るためにどこより右の部屋まで行けていればよいかを求めます。
この部屋を $B[y]$ と呼ぶことにします。
そうすると、 $A[x] \leq y < x$ であり、$B[y]$ が x 以上である最も左の部屋 y が、部屋 x に入るために開始地点である必要のある最右の部屋です。
これはセグメント木上で二分探索をする事で求める事が出来ます。

まとめ

以上の説明をまとめます。
・制約を読み、クエリあたりに掛けられる計算量から、使えるツールを列挙する。
・ツールを使うことを前提に、どんな情報があればクエリを求められるかをエスパーする。
・このエスパーした情報をどうやって前計算から求めるかを考える。

以上となります。
AtCoder ではクエリ問は出ないから役に立たないと思うかもしれませんがそれは違います。
クエリ問をマスターすればクエリ問に強くなれる上に、他の最適化問題や数え上げも、何か 1 つのパラメータを固定した上でクエリ問だと認識し直すことによって解くことが出来るようになります。
私はマスターできていないので、皆さんは頑張ってください。（丸投げ）

分割統治とデータ構造

70th autumn_eel

はじめに

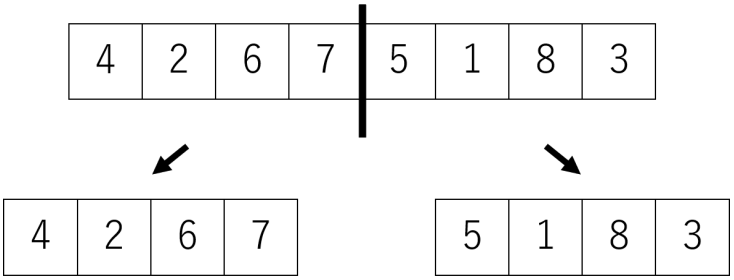
審査に間に合いませんでした。デコ責さん本当にごめんなさい。
分割統治とデータ構造の関係について書きます。

分割統治とは?

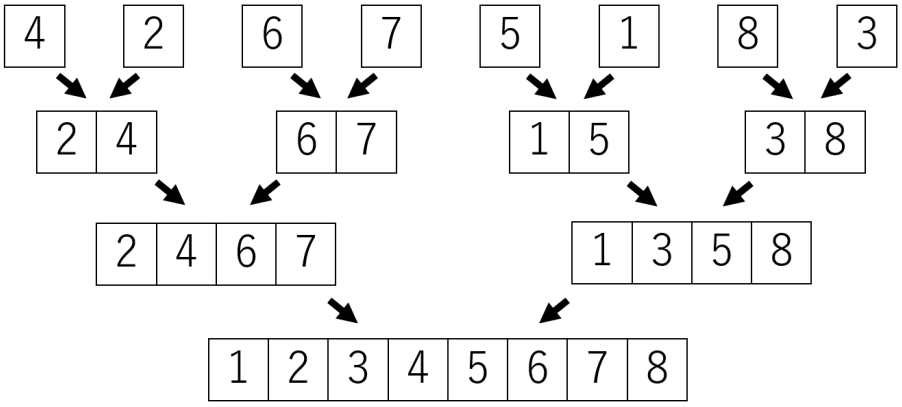
分割統治とは、1 つの大きな問題を複数の小さな問題に分割し、それらを再びまとめることで大きな問題を解く手法です。これだけだと何を言っているのか分からないと思うので、次の「列の分割統治」で具体例を説明します。

列の分割統治

ソートアルゴリズムの 1 つにマージソートというものがあります。詳細な手順は aspi 君の記事に書いてあるのでそれを読んでほしいのですが、このアルゴリズムは「1 つの問題を 2 つの小さな問題に分割していくことを繰り返したあと、それらをまとめていく」という分割統治法を用いていると考えることができます。例えば長さ 8 の列をソートしたい場合、長さ 4 の列をソートするという小さな問題 2 つに分割し、それを長さ 2 の列をソートするという小さな問題 4 つに分割し、さらに長さ 1 の列をソートするという 8 つの問題に分割します。



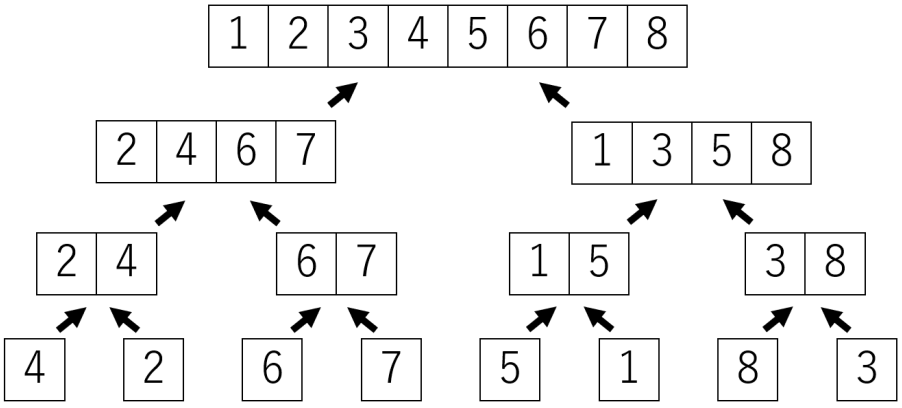
長さ 1 の列をソートした答えというのはもとの列そのものなので、8 つの小さな問題は既に解けていることになり、今度はこの結果をまとめ、長さ 2 の列に対する問題、長さ 4 の列に対する問題... を解いていくことになります。大きな問題の答えは、まとめる前の 2 つの小さな問題の答えをマージしていくことで求めることができます。



まとめる過程を図示すると上の図のようになります。

「列の分割統治」とデータ構造

先ほど、分割した列をまとめていく過程の図を出しましたが、この図を見て何かを連想した方もいるのではないでしょうか。分かりやすいように、さっきの図を上下ひっくり返して見てみましょう。



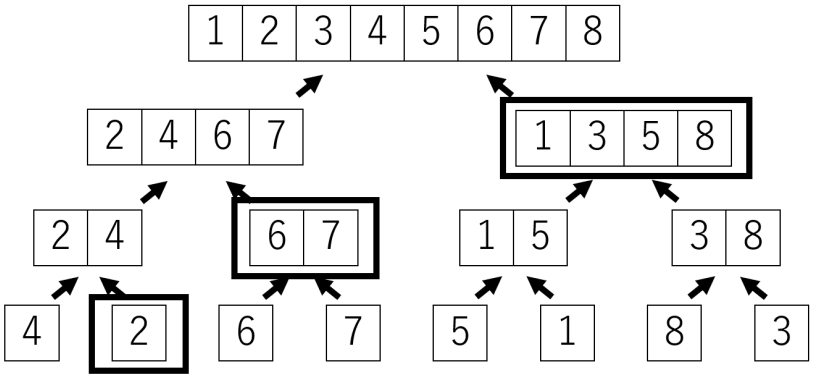
これ、セグ木に見えませんか?

それぞれの列を 1 つの頂点、矢印を辺として見ると、完全二分木となっており、確かにセグ木っぽいです。さらに、各頂点における列の要素はソート前の列におけるある連続した区間の要素に対応していることも分かります。これもセグ木と似ています。

では、もし実際に完全二分木に各列を保持させたデータ構造を作ってみることにしたら、どうなるでしょうか。
実はこんなクエリを処理することができます。

> ソートを行う前の (つまり、葉の順序で並んでいる) 列の区間 $[l, r)$ に、
> k 以下の数はいくつ含まれているか?

例えば、区間 [1, 8) (0-indexed) に含まれる 6 以下の数を数えたいとしましょう。この場合、セグ木と全く同じようにクエリの区間を分割し、それぞれの区間において二分探索で 6 以下の数を求め (ここでそれぞれのノードの列がソート済みであることが効いてきます)、足してやれば良いです。$O(\log N)$ 個の区間で二分探索をするので、計算量は $O(\log^2 N)$ になります。



このデータ構造は領域木というよく知られたものであり、蟻本にも載っています。マージソートの過程をデータ構造にのせる、という変わったアプローチでこのデータ構造が現れてくるのは、少し興味深いですね。海外では Merge Sort Tree とも呼ばれているらしく、こちらの方が今回の説明に合った名前かもしれません。

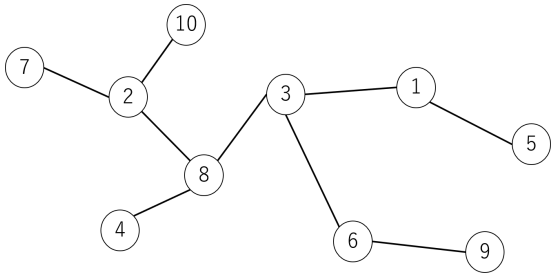
ここまでの説明から、分割統治の過程をデータ構造として保持するとよく分からないが面白いことができそう、ということを感じ取っていただけるとありがたいです。

木の分割統治

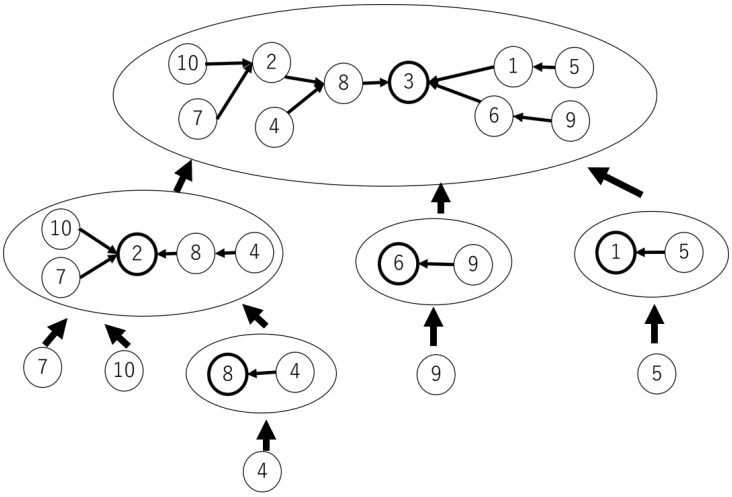
今度は列ではなく木の分割統治について見ていきたいと思います。
本当は重心分解について詳しく紹介したいところなのですが、ここに書こうとすると長くなってしまう上にうまく説明できる気がしないので、知らない方は蟻本などを読んで戻って来てください。~~戻ってきて下さる方はどれくらいいるのでしょうか...?~~

「木の分割統治」とデータ構造

ここからは重心分解を知っている前提で話を進めます。
こんな感じの木を重心分解で処理することを考えます。



この木における、重心分解の「分解した後まとめていく過程」をマージソートのときと同じように図示すると、こんな感じになります (太線の頂点が各ステップでの重心です)。



さきほどは図を眺めるとセグ木が見えてきましたが、今度は何が見えるでしょうか。

実は、重心分解の再帰の各ステップを頂点としてみると、これは深さが高々 O(log$N$) の根付き木になっています。セグ木は構造的には深さが高々 O(log$N$) の完全二分木なので、その一般形と言うこともできます (そもそも木は多くの場面において列の一般形とみることができます)。深さが O(log$N$) というのは、重心分解の再帰の深さが O(log$N$) であることから来ています。

では、根付き木の頂点にもとの処理する木の各頂点を持たせたデータ構造を実際に作ると、どのようなことができるのでしょうか。それを考えるために、次の問題を見てみましょう。

> N 頂点、すべての辺の長さが 1 の木があり、それぞれの頂点の重みは最初全て 0 である。次の 2 つのクエリを処理せよ。
>
> - 頂点 X_i からの最短距離が Y_i 以下であるすべての頂点 (頂点 X_i を含む) の重みに Z_i を加算する。
> - 頂点 X_i の重みを出力する。
>
> 出典: yukicoder No.1038 TreeAddQuery (一部改題)
> (`https://yukicoder.me/problems/no/1038`)

まずは 1 つ目のクエリについて考えてみます。
ある頂点 X_i のクエリを処理する時、その頂点を含む「根付き木の頂点」すべてを見ていきます。これは、「根付き木の根」から「頂点 X_i を重心とするステップを表す根付き木の頂点」までのパス上の頂点集合に等しいので、個数は高々 O(log$N$) 個であることが分かります。気持ちとしては、それぞれの根付き木の頂点に含まれる木の頂点 V で、

$$\text{頂点 } X_i \text{と頂点 } V \text{ を結ぶ最短パスが重心 } C \text{ を通る}$$

$$\text{かつ}$$

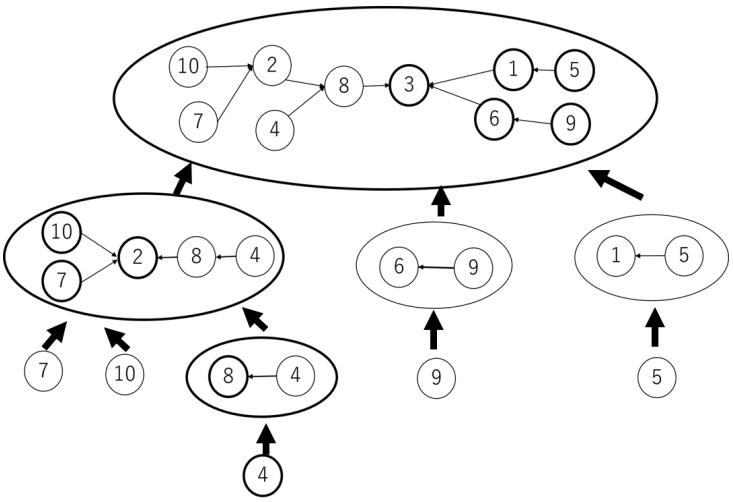
$$(\text{頂点 } X_i \text{から重心 } C \text{ までの距離}) + (\text{頂点 } V \text{ から重心 } C \text{ までの距離}) \leq Y_i$$

を満たすものについて重みを加算できれば良さそうで、 2 番目の条件は

$$(\text{頂点 } V \text{ から重心 } C \text{ までの距離}) \leq Y_i - (\text{頂点 } X_i \text{から重心 } C \text{ までの距離})$$

と変形できることから、C からそれぞれの (根付き木の頂点に含まれる、木の) 頂点までの距離を前計算し短い順にソートしておけば、BIT などを使って処理することができそう

です。



$X_i = 4$ の時に見なければならない頂点 (太線)

しかし、1 つ目の条件が少し厄介です。この条件をそのまま扱うのは厳しそうなので、これを満たさないものを後から引くことを考えます。

頂点 X_iと頂点 V を結ぶ最短パスが重心 C を通らない

というのは、上の図のように

頂点 X_iと頂点 V がその「根付き木上の頂点」の「子の頂点」で同じ場所に属している

と言い換えることができます。そのため、子の頂点によって分類した頂点集合に対応する BIT を保持し、これを使って先ほどのような処理を行うことで引き算ができるようになります。

これができてしまえば 2 つ目のクエリは簡単で、頂点 X_i が含まれている根付き木の頂点を全て見て、そこに BIT を用いて足されている値を求めれば良いです。

よってこの問題を解くことができました。
(解答ソースコード: `https://yukicoder.me/submissions/565602`)
説明が少し分かりにくかったかもしれません。ソースコードなども参考にしながら考えてみてください。

今回の問題では根付き木の頂点に含まれる頂点集合を BIT という形で保持しましたが、この問題 (https://yukicoder.me/problems/no/1197) のように、実際に頂点集合を持たなくて良い場合もあります。このような問題では、実際には (頂点集合ではなく) 重心同士を結ぶ木を構築しているような形になります。ただ、いずれにせよ「根付き木の頂点には頂点集合が含まれている」という考え方をもっていると、考察がしやすくなるのではないかと思います。

このように、木の分割統治においても、その過程をデータ構造に落とし込むと様々な処理ができるようになることが分かりました。

まとめ

ここまで、列の分割統治、木の分割統治について、その過程をデータ構造として見てクエリを処理する、という例を見てきました。では一体なぜ、このようなことができるのでしょうか。正直筆者もよく分かっていないのですが、「分割統治」と「データ構造」というのはどちらも再帰的であるという意味で共通しているから、分割統治の過程をデータ構造としてみるとうまく行く場合がある、ということなのではないかなあと思っています。データ構造の持ち方は天才的で知らないとどうしようもないものが多いですが、このように分割統治からデータ構造を考えると自分で発明することができて、なかなか面白いのではないかと思いました。

組み合わせゲーム理論と群論

71st oliverx3

はじめに

こんにちは。高校一年生の oliverx3 です。

部誌の原稿の締め切りまであと 1 週間ぐらいしかないのに誰も書き始めていません。大丈夫なのでしょうか。ダメみたいですねえ.....

とはいえ、さすがにそろそろ書かないとやばそうなので、一人で寂しく書いておきます。

今回は組み合わせゲーム理論について話し、群論との関係について触れたいとおもます。といっても、群論に関しては最後にちょこっと触れるだけです。数学ガチ勢の方、すみません。

自己紹介

自己紹介なんていらない？ ~~知ってますただの自己満足です~~

名前　oliverx3

年齢　16 歳 (高 1)

競プロ　競プロ歴 5 ヶ月緑コーダー (部誌が発行される頃には水色になっていたい気持ち)。

好きな分野　グラフ理論、組み合わせゲーム理論など。

twitter　@oliverx3_

自分が高 1 という事実に改めて驚きました。ずっと中学生でいいのに (よくない)

そろそろ本題に入る

Alice と Bob がいます。山がいくつかあり、それぞれの山には石がいくつかあります。二人はそれぞれのターンで山を一つ選び、その中からいくつかの石を取ります (必ず一個以上の石を取らなくてはいけません)。最後の石を取った人の勝ちです。さて、どちらが勝つでしょうか？

~~そんなこと知らねーよゲーム楽しめよ~~ もしゲームが始まる前に、山の状態だけを見てどちらが勝つか判定できたらすごいと思いませんか？思いますよね。思いますね。

今回は、そもそもゲームとは何か。ゲームをどうやって解析するのか。様々なゲームに共通する性質は何か。このようなことを話していきたいと思います。

部誌というものを初めて書く上、LaTeX を使った文書作成もほとんど初めてなので、拙い点もあると思いますが、よろしくお願いします。

前提：組み合わせゲームとは

正直あまり本質ではないと思うので、丁寧には書きませんが、組み合わせゲームとはそもそもどういうものなのかの感覚的な説明を最初にしたいと思います (厳密な定義はまだしません)

組み合わせゲームの定義

組み合わせゲームとは、以下の条件を満たすゲームである:

- 二人の対局者が交互に手を打つ
- どちらかの対局者が自分の手番に規則に従った手を打てなくなるまで続く
- 偶然に左右される要素がない (確定)
- 二人の対局者はゲームの状態についてのすべての情報を知っている (完全情報)

具体例を挙げると、まるバツゲームや将棋などが組み合わせゲームとなります。一般的なゲームと思ってもらってかまいません。

ゲームの基本的な用語

正規形・逆形

最後の手を打った対局者を勝ちとするゲームを正規形、最後の手を打った対局者を負けとするゲームを逆形と呼ぶ。

短いゲーム

短いゲームとは、同じ局面が繰り返されることがなく、その局面から到達できる局面の数が有限であるゲームのことです。(つまり、必ず有限回の手番でゲームが終了します)
以下、ゲームと言ったら組み合わせゲームかつ短いゲームのことを指すものとします。

対局者

二人の対局者をそれぞれ左 (L) と右 (R) と呼ぶ。(他にも様々な呼び方がある)

選択肢

ある局面 G から左が遷移可能な局面の集合を左選択肢 ($\mathcal{G}^L$), 右が遷移可能な局面の集合を右選択肢 ($\mathcal{G}^R$) と呼ぶ。$\mathcal{G}^L$ と $\mathcal{G}^R$ の和集合 (の要素) を局面 G の選択肢と呼ぶ。

帰結類

あるゲームを考えたとき、仮に二人の対局者が完璧な手を打ち続けたら、そのゲームで左が勝つか右が勝つかが一意に定まります。これを**組み合わせゲームの基本定理**と呼びます。以下、証明していきます。

定理 1.1 (組み合わせゲームの基本定理)

左と右が対局するゲーム G が与えられて、左を先手番とするとき、このゲームは、左が必ず勝つか、右が必ず勝つかのどちらかで、その両方でない。

証明：

左と右はそれぞれ自分が勝つための最善の手を打つ。つまり、帰納法により、左が手を打った結果の局面は、先手の右 (左の手が終わったので、新たなゲームの局面では右が先手となる) が必勝となるか後手の左が必勝となるかのどちらかに分類される。もし、後手の左が必勝となる局面を作り出せる手が存在するならば、左はその手を打つことで必ず勝つことができる。逆に、そのような手が存在しないならば、右が必ず勝つ。(証明終)

4 つの帰結類

基本定理によって、任意のゲームの局面はどちらが勝つかに注目して 4 つの帰結類に分類することができます:

$\mathcal{N}$ (ファジー) ... 先手番 ($\mathcal{N}$ext:次) の対局者が必ず勝つ
$\mathcal{P}$ (零) ... 後手番 ($\mathcal{P}$revious:直前) の対局者が必ず勝つ
$\mathcal{L}$ (正) ... どちらが先手番でも左 ($\mathcal{L}$eft) が必ず勝つ
$\mathcal{R}$ (負) ... どちらが先手番でも右 ($\mathcal{R}$ight) が必ず勝つ

帰結類と選択肢

ゲーム G の帰結類は、以下のようにそのゲームの選択肢の帰結類から決定することができます:

	ある $\mathcal{G}^R \in \mathcal{R} \cup \mathcal{P}$	全ての $\mathcal{G}^R \in \mathcal{L} \cup \mathcal{N}$
ある $\mathcal{G}^L \in \mathcal{L} \cup \mathcal{P}$	$\mathcal{N}$	$\mathcal{L}$
全ての $\mathcal{G}^R \in \mathcal{L} \cup \mathcal{N}$	$\mathcal{R}$	$\mathcal{P}$

一つだけ証明してみることにします。今回は、左下を証明していきます。

証明：

$G \in \mathcal{R}$ ならば、右は先手番で勝ちとなる。すなわち、$\mathcal{G}^R$ の中に右が後手番で勝ちになる選択肢があるということである。よって、$\mathcal{R} \cup \mathcal{P}$ に含まれる $\mathcal{G}^R$ が存在することが示せた。また、右は後手番でも勝つことができるので、左が勝つ選択肢はない。よって、すべての $\mathcal{G}^L$ は $\mathcal{R} \cup \mathcal{N}$ に含まれることが示せた。

一方、$\mathcal{R} \cup \mathcal{P}$ に含まれる $\mathcal{G}^R$ があるならば、右は先手番で勝つことができる。また、$\mathcal{G}^L$ がすべて $\mathcal{R} \cup \mathcal{N}$ に含まれるならば、左は先手番で勝つことができない。すなわち、$G \in \mathcal{R}$ が示せた。(証明終)

ゲームと数学

今回のメインとなる (予定の) 章です。今まで、ゲームについての厳密な定義をしないまま議論をしてきました。今、皆さんはゲームを数学的に厳密に考えるために必要なツールをすべて持っています。ゲームと数学が交わるところへ、行ってみましょう……。

基本的定義

ようやくゲームの定義ができます。今回定義するのは、五つの概念です。つまり:ゲームとは何か、二つのゲームの直和とは何か、ゲームの (符号の) 反転とは何か、二つのゲームはいつ等しくなるのか、左にとってあるゲームが別のゲームよりも好ましくなるのはいつか、です。

ゲーム G は $\{\mathcal{G}^L \mid \mathcal{G}^R\}$ と定義する。
$G + H \overset{\text{def}}{=} \{\mathcal{G}^L + H, G + \mathcal{H}^L \mid \mathcal{G}^R + H, G + \mathcal{H}^R\}$
$-G \overset{\text{def}}{=} \{-\mathcal{G}^R \mid -\mathcal{G}^L\}$
$G = H \overset{\text{def}}{\Leftrightarrow} (\forall X)\, G + X$ は $H + X$ と同じ帰結類に属する。
$G \geq H \overset{\text{def}}{\Leftrightarrow} (\forall X)\, H + X$ において左が勝つときはつねに $G + X$ において左が勝つ。

よくわかりませんね。これから一つ一つ詳しく解説していきます。

ゲームの定義

まずは一番上、ゲームの定義についてです。これは直感的にもわかりやすいのではないでしょうか。あるゲームはそのゲームの選択肢によって成り立っている、ということです。

このような定義をすることで何が嬉しいでしょうか。実は、ゲームに関する命題を証明する時にとても簡単になります。

この定義は見てのとおり再帰的です。そして、$G = \{\emptyset \mid \emptyset\}$ はもちろんゲームです (零ゲームと呼びます)。

よって、ゲームに関する命題を帰納的に簡単に解くことができます (零ゲームによって再帰が終結するところを場合わけしなくていいことになります)。これはかなり嬉しいですね。

ゲームの直和

新しい記法が出てきてるので、順に説明します。

$\mathcal{G}^L + H$ とはどういう意味でしょうか。ゲームの集合とゲームの局面の和はまだ未定義です。そこで、G をゲームの局面、S をゲームの集合として、以下のように定義します:

$$G + S \stackrel{\text{def}}{=} \{G + X\}_{X \in S}$$

お気持ちとしては、ゲームの局面とゲームの集合の和は、そのゲームの局面とゲームの集合のそれぞれの要素との和によって得られたゲームの集合である、という感じです。

また、直和の定義中にカンマが出てきますが、これは $\cup$ と同じ意味です (見やすいのでこちらを使います)。

直感的にはどういうことでしょうか。具体例を考えるとわかりやすいので、是非やってみてください。少し説明してみると、G と H からできた成分で、左は G に対して手を打ち、H には手を打たない $(\mathcal{G}^L + H)$, H に対して手を打ち、G には手を打たない $(G + \mathcal{H}^L)$ の二つの選択肢があるということです。右に関しても同様です。

ここで,3 つの重要な定理を証明してみます。このあと話す群論との関係にも関わってきます。

定理 2.1 (単位元の存在)

$\forall G : G + 0 = G$

証明

$G + 0 = \{\mathcal{G}^L \mid \mathcal{G}^R\} + \{|\} = \{\mathcal{G}^L + 0, G + \emptyset \mid \mathcal{G}^R + 0, G + \emptyset\}$
ここで、任意の集合 S に対して $S+\emptyset = \emptyset$ であるから、帰納法により $\mathcal{G}^L+0 = \mathcal{G}^L, \mathcal{G}^R+0 = \mathcal{G}^R$ である。よって、前述の式の右辺は $\{\mathcal{G}^L \mid \mathcal{G}^R\} = G$ とまとめられる。(証明終)

定理 2.2 (交換法則の成立)

$\forall G, \forall H : G + H = H + G$

証明

$$\begin{aligned} G + H &= \{\mathcal{G}^L + H, G + \mathcal{H}^L \mid \mathcal{G}^R + H, G + \mathcal{H}^R\} \\ &= \{H + \mathcal{G}^L, \mathcal{H}^L + G \mid H + \mathcal{G}^R, \mathcal{H}^R + G\} \ (\because \text{帰納法により}) \\ &= H + G \ (\text{証明終わり}) \end{aligned}$$

定理 2.3 (結合法則の成立)

$\forall G, \forall H, \forall J : (G + H) + J = G + (H + J)$

証明

左選択肢に関して

$$\begin{aligned} [(G + H) + J]^L &= \{(G + H)^L + J, (G + H) + \mathcal{J}^L\} \\ &= \{(\mathcal{G}^L + H) + J, (G + \mathcal{H}^L) + J, (G + H) + \mathcal{J}^L\} \\ &= \{\mathcal{G}^L + (H + J), G + (\mathcal{H}^L + J), G + (H + \mathcal{J}^L)\} \ (\because \text{帰納法により}) \\ &= [G + (H + J)]^L \end{aligned}$$

右選択肢に関しても同様である (証明終)

符号の反転

次に符号の反転です。部誌の締め切り 45 分前なので素早く行きます (おい)

符号の反転は対局者の役割の交換を意味します。定義より、$-\mathcal{G}^L = \left\{-G^L\right\}_{G^L \in \mathcal{G}^L}$ であるので、再起的に左選択肢と右選択肢が入れ替わります。

ゲームの等価性

定義から見ても明らかなとおり、ゲームが等しいとはゲームの直和に対して同じ振る舞いをするということです。

ここで、$=$ は同値関係、つまり、反射法則、対称法則、推移法則が成り立ちます。本当は証明しなければいけない事柄ですが、時間の都合上省略させていただきます.....。読者への課題ということで。

いくら時間がないとはいえ、重要なものまで省略することはできないです。以下の定理を証明してみましょう。

定理 2.4(帰結類とゲームの値)

$$G = 0 \Leftrightarrow G \in \mathcal{P}$$

証明

$\Rightarrow$: $G = 0$ ならば任意のゲーム X に対して $G + X$ は $0 + X = X$ と同じ帰結類に属する必要があります。

$\Leftarrow$: G を $\mathcal{P}$ 局面とする時、任意の X に対しても $G + X$ が X と同じ帰結類に属することを示す必要があります。

ここで、X において左が後手番で勝つならば、$G + X$ に関しても左は必ず後手番で勝つことができます。右が $G + X$ の 1 番目の直和に対して手を打ったならば、左はその直和での必勝戦略 (つまり G の必勝戦略) で応手します。一方、2 番目の直和に対して手を打ったならば、左はその直和での必勝戦略 (つまり X での必勝戦略) で応手します。

$G \in \mathcal{P}$ であるため、左は $G + X$ の 1 番目の直和で最後の手を打ち、2 番目の直和でも最後の手を打ちます。同様にして、X において左が先手で勝つ場合、右が後手で勝つ場合、右が先手で勝つ場合も証明できます。

以上より、任意の X に対して $G + X$ は X と同じ帰結類に属することがわかりました。(証明終)

ゲームの比較

ついに最後の定義の説明です。比較はかなり難しいので、お気持ちだけの説明となります..... すみません.....

定義の意味を説明すると、いかなる状況でも H を G に置き換えることが左にとって不利でないならば、$G \geq H$ であるということです。

群構造

いよいよ最後です。ゲームの集合が二項演算 $+$ について群をなしていることを紹介していきます。

なお、群に関する基本的な事柄は既知として議論を進めます。

群であるためには、演算について閉じている、結合法則が成り立つ、単位元が存在する、逆元が存在する、の 4 つが必要です。また、$G + H = H + G$ を満たすならばそれは可換群と言います。以下、ゲーム全体の集合が可換群をなすことを証明して部誌を締めたいと思います。

定理 2.5(ゲームと群)

ゲーム全体の集合は可換群となる。

証明

演算について閉じていること:

$G + H \overset{\text{def}}{=} \left\{\mathcal{G}^L + H, G + \mathcal{H}^L \mid \mathcal{G}^R + H, G + \mathcal{H}^R\right\}$ は、帰納法によりゲームの集合の対なので、$G + H$ もまたゲームとなる。

結合法則:

定理 2.3 より

単位元の存在:

定理 2.1 より

逆元の存在:

$\forall G : G + (-G) = 0$ となることを示す。
後手の対局者は必ず $G + (-G)$ でかつ。すなわち、先手が一方の直和の成分に対して手を打ち $G + (-H)$ にしたとすると、後手は先手と対応する手を打つことで $H + (-H)$ の状態にすることができる。帰納法により、先手の番でゲームの値が 0 になるため、後手が勝つ。
$G + (-G) \in \mathcal{P}$ であるため、定理 2.4 より、$G + (-G) = 0$ となる。

交換法則:

定理 2.2 より

以上より、ゲーム全体の集合は二項演算 $+$ に関して可換群をなす (証明終)

最後に

ふう、ついに書き終わりました。現在 10/25 の 12:12:00 です。部誌の締め切り 12 分遅れです。なんとか間に合いました (は？)

今回、組み合わせゲーム理論について話させていただきました。もともと競プロでゲームの問題を解く時に、Nim についてや Grundy 数についての知識が必要だったため、そこでこの理論に出会いました。

その時見たサイトに、ゲームの遷移の様子をグラフで表していたり、Nim のとても美しい必勝法が載っていたりして、とても興奮したのを覚えています (グラフ理論はもともと大好きでした)

そこからは、不偏ゲームについて調べたり、組み合わせゲーム全般について調べたりしました。是非ともみなさんにこの理論を知っていただきたいという思いと、自分でも復習したいという思いで、今回部誌に書かせいただきました。

今回、具体的なゲームでの説明なしに書きました。少々わかりにくかったと思います。すみません....。また、証明もなるべく厳密に書くようにはしましたが、紙面の関係や僕の理解不足もあって途中経過を省略したものもありました。申し訳ないです。

群の話をする時に、半順序を持つことも話したかったのですが、ゲームの比較に関する説明を適当に終わらせてしまったためできませんでした.... 悲しいです.....。

色々と謝ってばかりですが、個人的には、組み合わせゲーム理論が初めての人でも結構深いところまで学べる内容にできたと思います。

今回紹介できなかったものの中にも面白いものはたくさんあります。特に不偏ゲームなどはかなり面白いです。

今回の部誌を読んで組み合わせゲーム理論を好きになってくれる人が一人でもいたら、嬉しい限りです。

なんか達成感もあってか泣きたくなってきました。このままだと永遠に感謝の言葉を書いてしまいそうなので、そろそろ締めたいと思います。

組み合わせゲーム理論をここまで発展させてくださった先人の皆様、このような世界に入らせてくれた競技プログラミング、部誌を書く機会を提供してくれた Paken の方々、本当にありがとうございました！

参考文献

Michael H.Albert, Richard J.Nowakowski, David Wolfe 著, 川辺治之 訳, 『組み合わせゲーム理論入門 勝利の方程式』, 共立出版 (株), 2011 年

この chmin セグ木に区間和クエリを！

72nd define

はじめに

こんにちは、中三の define（本名は平田誠治）です。一応中学部長をしていますが、最近コンテストで同級生に負け気味なのでそろそろ辞めさせられそうです。
申し訳程度に部長っぽい事を書くと、今年は新入生である 74 期が非常に強くて、まだ始めて半年くらいしか経ってないのに青コーダーがいたりします。パ研の未来は明るい！

ところで、このタイトルの元ネタを知っていますか？そう、「この素晴らしい世界に祝福を！」です。最近読み始めたんですが、とても面白いのでそこの貴方も~~部誌なんて読んでないで~~本屋へ GO！

さて、前置きはこれくらいにしておいて、早速本題に入って行きましょう。

目標

この記事では、以下のようなデータ構造を完成させる事を目標にします。
以下、対象となる整数からなる配列を A 、 A の要素数を N とします。
また、全てのクエリについて対象となる半開区間を $[l, r)$ 、i は $i \in [l, r)$ を満たす全ての整数とします。

次のようなクエリを各々ならし $O(log^2\ N)$ で処理する。

- $A_i \leftarrow min(x, A_i)$
- $A_i \leftarrow max(x, A_i)$
- $A_i \leftarrow x$
- $A_i \leftarrow A_i + x$
- $min(A_i)$ を求める
- $max(A_i)$ を求める

- $sum(A_i)$ を求める

つまり、区間に対して
更新：chmin, chmax, add, assign
求値：min, max, sum
の処理ができるデータ構造を作ろう！という事です。

Segment Tree 概略

Segment Tree を知っていますか？僕は知っています。←様式美
Segment Tree とは、モノイドについて区間演算を高速に行える完全二分木データ構造です（以下、セグ木）。

~~締切まで時間がないので~~ Segment Tree は既知として詳細は割愛しますが、例えば次のようなクエリを各々 $O(log\ N)$ で処理できます。

- $A_i \leftarrow min(A_i, x)$
- $min(A_i)$ を求める

このようなセグ木は、操作が chmin と同じである事から chmin セグ木とも呼ばれます。
セグ木は他にも区間に対する操作は色々とでき、超便利なデータ構造です。

しかし！

この chmin セグ木では区間和を求める事はできません。なぜでしょう？
遅延伝播セグ木の場合、操作は要素と要素の二項演算、要素と作用素の二項演算、作用素と作用素の二項演算によって成り立っています。

区間和の場合、要素と作用素の二項演算が上手く行きません。なぜなら、区間内でもいくつ減るかが違うため、要素（区間和）が作用素によってどれだけ増減するかを計算できないからです。
ここで登場するのが、**Segment Tree Beats** です！！

Segment Tree Beats、爆誕...!

Segment Tree Beats とは、Segment Tree の応用版です。名前の由来はアニメ Angel Beats!だそうです。驚きですね！~~見たことないけど~~

このデータ構造を使うと、セグ木以上に様々なクエリをこなす事ができます。
まず、更新クエリを chmin , 求値クエリを min , max , sum に絞って考えてみましょう。
テンプレートを以下のように定義します。

```
void update(int k){

}
void updateNodeMax(int k,ll x){

}
void push(int k){

}
void updateMin(int a,int b,ll x,int k=0,int l=0,int r=-1){
    if(r==-1)r=size;
    if(break_condition)return;
    if(tag_condition){
        updateNodeMax(k,x);
        return;
    }
    push(k,r-l);
    updateMin(a,b,x,k*2+1,l,(l+r)/2);
    updateMin(a,b,x,k*2+2,(l+r)/2,r);
    update(k);
}
```

メインアイデアは、最小値/最大値だけでなく、2 番目の最小値と最小値の個数（最大値も同様）も持つようにする事です。
変数にすると、以下です。

mx[k] = 最大値 , smx[k] = 2 番目の最大値 , mxc[k] = 最大値の個数
mn[k] = 最小値 , smn[k] = 2 番目の最小値 , mnc[k] = 最小値の個数
sum[k] = 和

このようにする事で、(2 番目の最大値) $< x \leq$ (最大値) の時に数の種類を最大値の 1 種類に絞って更新する事ができます。
具体的には、区間の和を $(x-$ (最大値)) $\times$ (区間に含まれる要素数) だけ増やした上で最大値を x にすれば良くなります。
コードにすると以下です。二項演算の場合分けが面倒ですね...

```
void update(int k){
    sum[k]=sum[k*2+1]+sum[k*2+2];
    mx[k]=max(mx[2*k+1],mx[2*k+2]);
    if(mx[2*k+1]<mx[2*k+2]){
        mxc[k]=mxc[2*k+2];
        smx[k]=max(mx[2*k+1],smx[2*k+2]);
    }else if(mx[2*k+1]>mx[2*k+2]){
        mxc[k]=mxc[2*k+1];
        smx[k]=max(smx[2*k+1],mx[2*k+2]);
    }else {
        mxc[k]=mxc[2*k+1]+mxc[2*k+2];
        smx[k]=max(smx[2*k+1],smx[2*k+2]);
    }
}
void updateNodeMax(int k,ll x){
    sum[k]+=(x-mx[k])*mxc[k];
    mx[k]=x;
}
void push(int k){
    if(k>=size-1)return;
    if(mx[k*2+1]>mx[k])updateNodeMax(k*2+1,mx[k]);
    if(mx[k*2+2]>mx[k])updateNodeMax(k*2+2,mx[k]);
}
```

```
break_condition = r <=a || b <= l || mx[k] <= x
tag_condition = a <= l && r <=b && smx[k] < x
```

求値クエリについては、通常のセグ木と同じなので省略します。

chmin と chmax 混合の更新クエリの場合は、chmin クエリ時に最大値を x にするだけでなく、最小値や 2 番目の最小値を更新する必要がある場合がある事に注意です。

```
void updateNodeMax(int k,ll x){
    sum[k]+=(x-mx[k])*mxc[k];
    if(mx[k]==mn[k]){
        mx[k]=mn[k]=x;
    }else if(mx[k]==smn[k]){
        mx[k]=smn[k]=x;
    }else {
        mx[k]=x;
    }
}
```

加算クエリや代入クエリが絡んでくるともう少し複雑になりますが、本質は変わりません。また、これらの更新クエリは通常の遅延伝播セグ木と実装も大して変わりません。
参考までに、更新クエリ：chmin , chmax , add , assign 求値クエリ：min , max, sum までの実装例を貼っておきます。

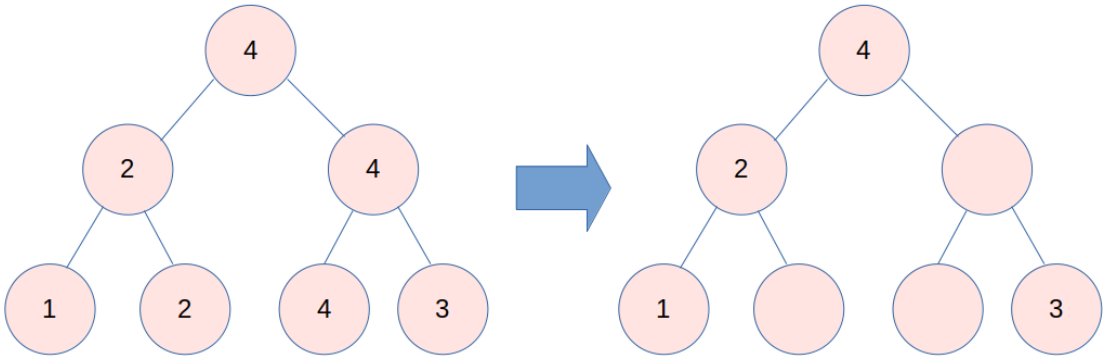
https://github.com/defineProgram/Library/blob/master/structure/SegmentTreeBeats.cpp

200 行近いですが、ほとんど同じような関数が並んでいるだけなのでそこまで大変ではありません。

Segment Tree Beats の計算量

Segment Tree Beats の計算量解析は結構難しくて、私もフィーリングでの理解にしか至っていませんが、各クエリがならし $O(log^2\ N)$ である証明の一例を紹介します。

まず、「タグ」という概念を定義します。この「タグ」は各ノードにおける最大値です（前項における mx に相当します）。もし、親ノードとタグの値が同じだったら、自身のタグを削除します。



また、通常ノードと余分ノードを定義します。通常ノードとは、通常のセグ木で訪れるような最低限のノードを指し、余分ノードとは Beats の厳しい tag_condition によって訪れたノードを指します。
次に、$\Phi(x) =$ 全てのタグの深さの合計と定義します。操作前、$\Phi(x) = O(N\ log\ N)$ です。
この時、次の事が言えます。

1. 更新クエリの後、余分ノードにはタグが存在しない。よって、新しいタグは通常ノードのみから生まれると考える事ができる。また、通常ノードから子ノードへの push がされると、更新前の最大値タグが子ノードに伝播し、タグが 1 つ増える。よって、新しいタグと合計して $\Phi(x)$ が $O(log\ N)$ 増える。通常ノードの個数は $O(log\ N)$ 個あるため、合計で $O(log^2\ N)$ 増える。
2. 余分ノードを訪れる度に、タグが増える代わりに自分より深いノードで消えるタグがあるので、毎回 $\Phi(x)$ が 1 減る。

よって、$\Phi(x)$ の合計増分は $O(Q\ log^2\ N)$ であり、余分ノードを訪れる度に $\Phi(x)$ が 1 減る事より余分ノードを訪れる回数は $O(Q\ log^2\ N)$ 、時間計算量は合計で $O(Q\ log^2\ N)$ である（各クエリならし $O(log^2\ N)$ である）。

... まぁザックリ言うとこんな感じです。微妙に間違っている所があるかもしれません

(最悪)。以上より、目標を達成するデータ構造を完成させる事ができました！

いかがでしたか？

やっぱり計算量解析難しいですね...。~~まぁ競プロでは通れば良いので既知のデータ構造の計算量解析はどうでもいい~~

実はまだ Segment Tree Beats を実際のコンテストで使った事はないんですが、すごく強力なデータ構造ですよね！実装多いので、JOI で出たら破滅する気がします。

この記事を作成するにあたり、こどふぉブログを読むために必要な英語力を付けてくれた鉄緑会英語科にはこの場を借りて深く御礼申し上げます。

参考文献

- Segment Tree Beats の実装メモ (基本まわり)
  https://smijake3.hatenablog.com/entry/2019/04/28/021457
- A simple introduction to ”Segment tree beats”
  https://codeforces.com/blog/entry/57319

フローアルゴリズム入門

72nd kaage

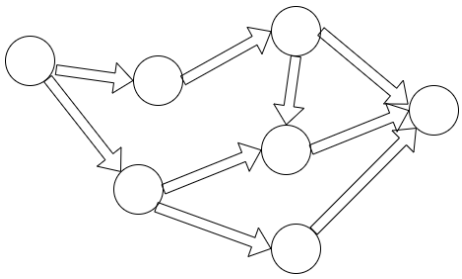
はじめに

みなさんは「ネットワークフロー」をご存知だろうか。[*1]ネットワークフロー、または単にフローとは、グラフ理論の一分野で、特に情報科学における数理最適化問題と深い関係がある。この記事では、ネットワークフローに関する問題を解くアルゴリズムについて扱う。

導入

ネットワークフローとは

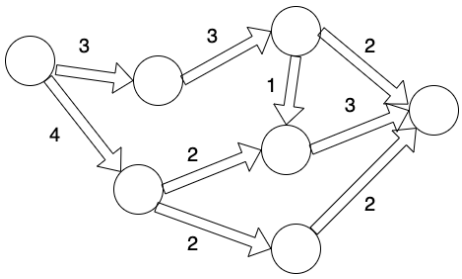
ネットワークフローでは、重み付き有向グラフの辺に「フロー」を流し、そのフローに関して最適化問題を解く。例えば、次の図のようなものが対象となるグラフになる。



フローの対象となるグラフの例

ここには、たとえば次のように「フロー」を流すことができる。

[*1] 「フローネットワーク」と呼ばれることもあるが、ここでは、フローアルゴリズムに関わる分野全般を「ネットワークフロー」、のちに定義する具体的なネットワークを「フローネットワーク」、フローの流量に関する情報のない有向グラフ単体を「ネットワーク」と呼ぶことにする。



フローの例

フローネットワークは、厳密には次のように定義される。

Definition 1. 有限な有向グラフ $G(V,E)$ の各辺 $(u,v) \in E$ に、非負実数の容量 $c(u,v) \in \mathbb{R}$ が定義されているとする。便宜上、$(u,v) \notin E$ の場合 $c(u,v) = 0$ とする。このとき、フローネットワークは実数関数 $f : V \times V \to \mathbb{R}$ で表され、次の要件を満たす。

$$f(u,v) \leq c(u,v) \tag{1}$$

$$f(u,v) = -f(v,u) \tag{2}$$

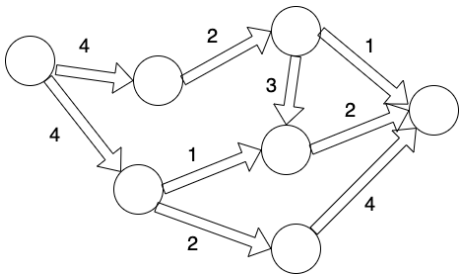
$$\sum_{w \in V} f(u,w) = 0\{u \neq s,t\} \tag{3}$$

$$\sum_{w \in V} f(s,w) = \sum_{w \in V} f(w,t) \tag{4}$$

これに加えて他の条件を満たすフローネットワークの中で、流量やコスト（後述）を最適化するのがフローアルゴリズムである。このようなネットワークフローのアルゴリズムは実社会でそのまま使えることも多い。以下では、ネットワークフローに関する有名な二つの最適化問題を扱う。第 2 節では、これらの最適化問題の解法について扱い、第 3 節では応用を扱う。

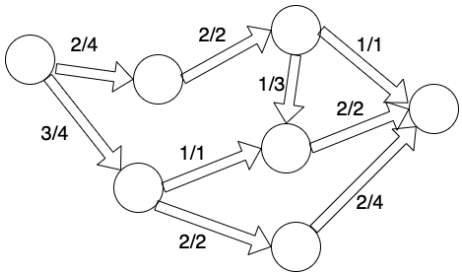
最大流問題

グラフの辺に最大流量の制限が定められ、始点から終点へ流れるフローの流量を最大化する問題を、最大流問題と呼ぶ。たとえば、次のような流量制限がついたグラフを考える。



最大流問題の例

このグラフの左端の頂点から右端の頂点へ流せるフローの最大値を考える。このグラフでは、次のようにすることでフローを 5 流せて、これが最大となる。

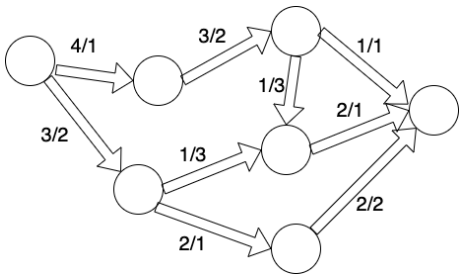


最大流問題の解

このようにして最大流を求めることは、たとえばデータの転送や物流の効率化などと直接関係する。また、他の最適化問題も最大流問題に帰着させることができる場合がある。（詳しくは第 3 節で扱う。）

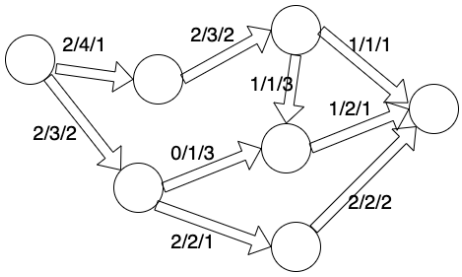
最小費用流問題

グラフの辺に最大流量とコストの基準値が定められ、ある流量 F を流すときのコストの総和を最小化する問題を、最小費用流問題と呼ぶ。ここでは、辺のコストの基準値にその辺におけるフローの流量をかけたものをすべての辺について足し合わせたものがフロー全体のコストとなる。たとえば、次のようなグラフで最小費用流問題を考える。（辺の上の数はそれぞれ流量制限とコストの基準値を順に示している）



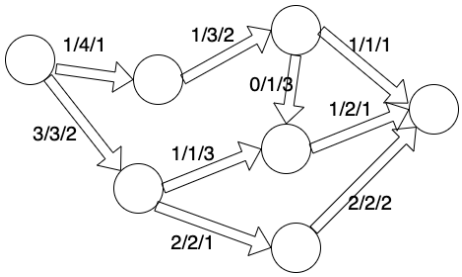
最小費用流問題の例

このグラフの左端から右端へ流量 4 流すとすると、たとえば次のような流し方が考えられる。（辺の上の数はそれぞれ流量、流量制限、コストの基準値を順に示している）



条件を満たすフローの例

この流し方だと、コストの総和は 21 となる。計算すれば明らかだろう。しかし、このグラフではこの流し方は最適ではなく、次のようなコストの総和 20 の流し方が最適となる。



最適なフローの例

配電や転送などの問題は直接最小費用流問題に帰着できることが多い。最大流問題と同様、他の最適化問題を帰着できる場合もある。（詳しくは第 3 節で扱う。）

考察・解法

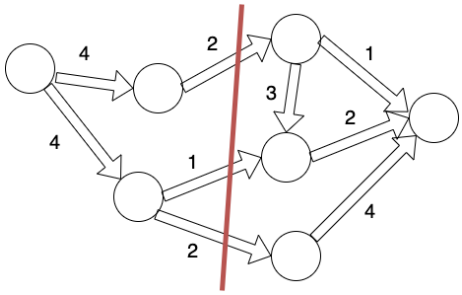
ここでは、先ほど挙げたふたつの問題に対する考察や解法を解説する。

最大流問題

最小カット問題

最大流問題を解くアルゴリズムはいくつも知られているが、その証明の多くには最小カット問題が利用される。まずは、最小カット問題について説明する。

最小カット問題とは、あるネットワークで、始点から終点へのパスを完全に取り除くために削除する必要のある辺の最大流量の最小値を求める問題である。たとえば、次のグラフでは最小カットは図のように切った時の 5 となる。



最小カット

最大フローと最小カットの間には、最大フロー最小カット定理と呼ばれる定理が成り立つ。

最大フロー最小カット定理

「残余グラフ」を定義する。

Definition 2. E にその逆辺も加えた辺集合を E' とすると、残余グラフ $G_f(V, E')$ とは、フローネットワーク f に対して、

$$c_f(u, v) = c(u, v) - f(u, v) \tag{5}$$

を満たすもののことをいう。

また、次の補題も示しておく。

Lemma 1. 任意のネットワークは最大フローを持つ。

Proof. ネットワークの条件は $\mathbb{R}^E$ の有界閉集合であり、最大値の定理より、コンパクト集合から実数値への連続関数は最大値をもつので、示された。 □

この証明では、ネットワークから最大フローへの関数が連続であることを自明として用いたが、証明はここでは省く。

さて、これで最大フロー最小カット定理を示す準備が整った。

Theorem 1. 任意のネットワークにおいて、最大フローと最小カットは等しい。（最大フロー最小カット定理）

Proof. あるカットが存在するとき、そのカットの容量を超えてフローを流すことはできないので、ネットワーク $G(V,E)$ における任意のフローについてその容量を f とし、任意のカットの容量を c とすると、

$$f \leq c \tag{6}$$

が成り立つ。すなわち、特に

$$\max f \leq \min c \tag{7}$$

となる。Lemma 1 より、任意のネットワークは最大フローを持つので、この最大フローを f とする。このフローに対する残余グラフ G_f に、始点 s から終点 t への重みが正の辺のみからなるパスが存在すれば、これは f が最大フローであることに矛盾するので、G' において s から重みが正の辺を通って到達できる頂点集合を S, $T = V \backslash S$ とすると、

$$S \cap T = \emptyset \tag{8}$$

となる。

ここで、$e(u,v) \in E$ のうち、$u \in S, v \in T$ のものについて、$c_f(u,v) = 0$ より $f(u,v) = c(u,v)$ が、$u \in S, v \in T$ のものについて、$c_f(v,u) = 0$ より $f(u,v) = c(u,v)$ が成り立つ。このとき、これらの辺上でのフローの総和は最大フローに等しく、また、これらの辺集合はネットワークのカットになっている。

よって、最大フローに対して、同じ容量のカットが存在するので、(7) と合わせて、最大フローと最小カットの容量が等しいことが示された。 □

また、この証明の (7) 以降では、f が最大フローである仮定は必要なく、すべての辺で重みが正の $s-t$ パスがなければ容量の等しいカットが存在するので、それが最大フローであることも直ちに示せる。

Lemma 2. すべての辺で重みが正の残余グラフに $s-t$ パスが存在しなければ、それに対応するフローネットワークはそのネットワークの最大フローである。

残余グラフ上のすべての辺で重みが正の $s-t$ パスは一般に「増加路」と呼ばれる（フローの流量を増加させるためである）。以降、簡単のためこのようなパスは「増加路」と呼ぶことにする。

さて、上の補題より、増加路をもたないフローネットワークを構築できれば最大流が求められることがわかった。以降ではそのようなアルゴリズムの例を挙げる。

Ford-Fulkerson algorithm

Ford-Fulkerson algorithm は非常に単純なアルゴリズムである。深さ優先探索で増加路を探し、見つけ次第フローを増加路に沿って流していく。Lemma 2 より、このアルゴリズムは停止すればそのとき最大流を導く。流量制限が有理数なら必ず停止することを容易に示せるが、そうでない場合停止しないこともある。特に流量制限が整数のとき、最大フローを F として、時間計算量は $\mathrm{O}(|F||E|)$ となる。

Edmonds-Karp algorithm

Edmonds-Karp algorithm は、Ford-Fulkerson algorithm の派生で、深さ優先探索ではなく幅優先探索で増加路を探す。これで、毎回最も使う辺の本数が少ない増加路を見つけられる。毎回の増加路について、最も容量が小さい辺（複数ある場合もある）を「ボトルネック」と呼べば、各辺がボトルネックになる回数は $\mathrm{O}(|V|)$ なので、これに辺の本数 $|E|$ をかけて、毎回の幅優先探索にかかる $\mathrm{O}(|E|)$ も合わせて、時間計算量は $\mathrm{O}(|V||E|^2)$ となる。

Dinic algorithm

Dinic algorithm では、Edmonds-Karp algorithm と同様に、長さ順に増加路を探していく。ただし、このとき、ある長さの増加路を探すときには、もうその長さの増加路に使

えない辺を使わないようにする。詳細は省くが、このアルゴリズムは $O(|V|^2|E|)$ で動作する。また、これも詳細は省くが、二部グラフの最大マッチングを求めるような場合では $O(|E|\sqrt{|V|})$ の時間計算量になることも知られている。

最小費用流問題

最小費用流問題は、グラフの条件によってアルゴリズムが大きく変わる。以下では、辺の重みが整数であり、コストの合計が負となる閉路をネットワークが含まないことを仮定する。負の閉路がある場合は、この閉路を除去するアルゴリズムを先に実行しなければならない。

Primal-Dual algorithm

Primal-Dual algorithm では、毎回コストが最小となる増加路を見つけて、それにフローの総量が F に達するまで流せるだけフローを流す貪欲法をする。負のコストの辺がある場合は Bellman-Ford 法を、ない場合は Dijkstra 法で最短路を見つければ、時間計算量はそれぞれ $O(F|V||E|), O(F|E|\log|V|)$ となる。

このアルゴリズムの正当性を示す。

Theorem 2. Primal-Dual algorithm は最小費用流を導く。

Proof. 流量 0 のフローは 1 通りしか存在せず、これは明らかに最小費用流である。Primal-Dual algorithm で得られた流量 i の最小費用流 f_i を仮定し、ここに増加路が存在して新たにフローを流し、流量 $i+1$ のフローネットワーク f_{i+1} を生成することを考える。これが最小費用流でないとすると、もっとコストの小さいフロー f'_{i+1} が存在する。アルゴリズムの定義より、始点と終点の間の最短経路は $f_{i+1}-f_i$ なので、$f'_{i+1}-f_i$ は最短経路にフローを流した場合よりもコストが小さく、よってコストの合計が負の閉路を含む。よって、f_i に負閉路が存在してここにフローを流せることになり、これは f_i が最小費用流であることに矛盾する。よって、背理法から f_{i+1} も最小費用流であり、数学的帰納法から、Primal-Dual algorithm で生成されるすべてのフローネットワークは最小費用流となる。 □

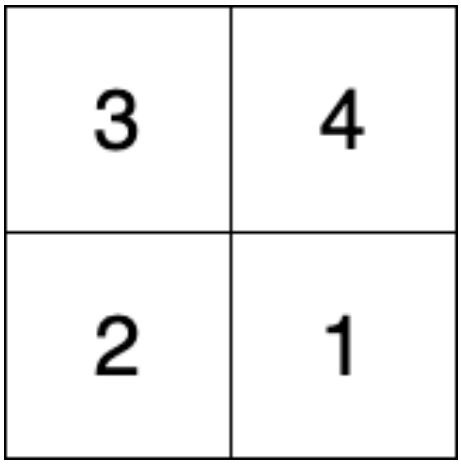
ところで、Primal-Dual algorithm は、負辺がある場合でも各頂点にポテンシャルを定義することで、Dijkstra 法を使えるようになって計算量が $O(F|E|\log|V|)$ になるが、ここでは詳細な説明は省く。

フローアルゴリズムの応用

次の問題を考えてみる。

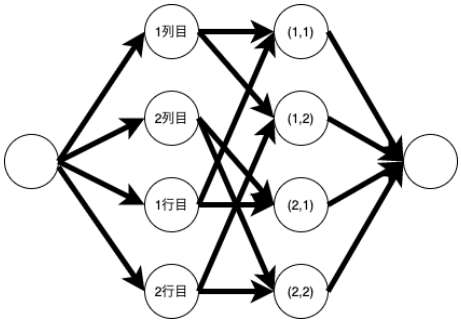
> 縦 N 列、横 N 行のグリッドがあり、N^2 個のマスがある。それぞれのマスには定数 $a_{i,j}$ が定まっている。
> 今からいくつかのマスを選んで、選んだすべてのマス (i, j) に対する $a_{i,j}$ の値の総和を最大化する。
> （出典: AtCoder Library Practice Contest - E MinCostFlow）

この最適化問題は、一見フローアルゴリズムとは関係なさそうに見える。しかし、実はこの問題は最小費用流問題に帰着できる。



グリッド

このようなグリッドに対しては、次のようなネットワークを考える。



対応するネットワーク

始点から行の頂点への辺の容量と列の頂点から終点への辺の容量は K、その他の容量はすべて 1 として、行 i から列 j への辺のコストは $-a_{i,j}$、その他はコスト 0 とし、このネットワーク上で NK 以内の流量を流した時の最小費用を求め、その符号を反転させれば答えとなる。

このように、ネットワークフローの問題に帰着できる最適化問題は多くある。

参考文献

Wikipedia - フローネットワーク https://ja.wikipedia.org/wiki/フローネットワーク
ネットワークフロー入門 (by hos_lyric) http://hos.ac/slides/20150319_flow.pdf
プログラミングコンテストチャレンジブック (マイナビブックス)
Dinic 法について (by TAISA_) https://ta1sa.hatenablog.com/entry/2020/04/13/123802

おわりに

70th capra314cabra

いかがだったでしょうか。

今年は部員 12 人が部誌を書いてくれたので、いつもよりも読み応えがあるものだったと思います。

~~部誌の締め切りに間に合った人、半数もいないのでは...?~~

とか色々思うところはあるんですが、まあ、結果的に「電脳」として完成させることができたので、結果オーライですね!

今年から部誌編集を Word から LaTeX に切り替えたのですが、慣れない部員も多く、彼らが混乱することもあったと思います。そんな状況の中でもタフに部誌を書いてくれた部員に感謝。読んでくれたそこのあなたにも感謝。そして、これを編集した自分にも感謝 (おい!)

感謝。感謝。感謝。この電脳は感謝の塊です。

なんかいい感じのことを言うことができたのでこれで部誌を終わりにします。最後まで読んでいただき、ありがとうございました。

デコレーション責任者 capra314cabra

以下、例年通りの話です。