Yabaitech Tokyo Vol.6

まだ休んじゃダメですよ。

ばいなり　MasWag　wasabiz

YABAITECH.TOKYO

vol.6

2020

目次

テキサスホールデムの役判定に見る高速化テクニック

ばいなり

1. はじめに

はじめまして、yabaitech.tokyo の新メンバーのばいなりです。yabaitech.tokyo のメンバーの一部でテキサスホールデムポーカーが流行っており、一緒にポーカーで遊んでいたらこちらに加入させられてしまいました。せっかくなのでポーカーに関する技術記事を書こうということで、今回はテキサスホールデムの役判定を爆速で行う話についてでも書かせてもらえればと思います。

所詮はポーカーの役判定ごとき、もともと愚直に書いてもマイクロ秒単位で処理できるため、もともと高速じゃないか、そんなものを高速化して何が面白いのかと思われるかもしれませんが、次章で説明するようにテキサスホールデムポーカーでは使えるカードが明らかになっていない段階でアクションを起こす必要があり、その時点での勝率を計算しようとすると大量にあり得るハンドの役判定を行わねばならず、この処理の高速化が要求されるわけです。限定的な条件下ではありますが、中には 1 秒間に 22 億回もの評価が行えるプログラムまで登場しますので、楽しみに読み進めていただければ幸いです。

2. テキサスホールデムのルール

テキサスホールデム (Texas hold'em) は、数あるポーカーの変種の中でも世界的に見て最もポピュラーなものと言えるゲームです。日本ではまだ馴染みの薄い感じもありますが、世界的には数多くのプレイヤーがおり、最も権威ある世界大会ではなんと 10,000,000 ドル (!) もの優勝

賞金が懸けられるほどです。この章では、テキサスホールデムとは何ぞやという方に向けて、そのルールを簡単に説明します（テキサスホールデムを既に知っている方は読み飛ばしていただいても構いません）。

2.1. 役の一覧

テキサスホールデムにおける役は 5 枚のカードの組み合わせで決まり、強い順に以下の 9 つがあります。さまざまなポーカーの変種で共通なため、馴染みのある方も多いでしょう。

(1) **ストレートフラッシュ (Straight flush):** フラッシュとストレートの複合役。すべての役の中で最も強い。例 : A♠ K♠ Q♠ J♠ 10♠

(2) **フォーカード (Four of a kind):** 同じランク（数字）のカードを 4 枚集めた役。例 : 6♠ 6♥ 6♦ 6♣ J♦

(3) **フルハウス (Full house):** 同じランクのカード 3 枚と、別の同じランクのカード 2 枚で構成される役。例 : K♥ K♦ K♣ 3♥ 3♣

(4) **フラッシュ (Flush):** 同じスート（柄）のカードを 5 枚集めた役。例 : Q♥ 10♥ 7♥ 5♥ 4♥

(5) **ストレート (Straight):** 数字が連続した 5 枚のカードで構成される役。A は 14 または 1 として扱うことができる。A、K、2 の 3 枚を含むものはストレートとは見なされない。例 : J♥ 10♣ 9♣ 8♥ 7♠

(6) **スリーカード (Three of a kind):** 同じランクのカードを 3 枚集めた役。例 : 5♠ 5♦ 5♣ A♥ J♣

(7) **ツーペア (Two pair):** 同じランクのカード 2 枚の組を 2 組集めた役。例 : 8♠ 8♣ 3♠ 3♦ Q♦

(8) **ワンペア (One pair):** 同じランクのカードを 2 枚集めた役。例 : 7♦ 7♣ 9♠ 4♠ 2♦

(9) **ハイカード (High card):** 上記のいずれにも当てはまらないもの。いわゆる “ブタ”。例 : A♣ J♠ 8♦ 4♣ 2♥

役が同じ場合は、カードのランクをもとにタイブレークがなされます（A が最も強く、2 が最も弱い ; ストレートで A を 1 として扱った場合のみ A が最も弱い）。カードのスートはハンドの強さに影響しません。このようにハンドの強さを定義した場合、ハンドは 7462 種の同値類に分類されることが知られています。

2.2. テキサスホールデムの進行

テキサスホールデムでは、**ホールカード (Hole cards)** と呼ばれる各プレイヤーに 2 枚ずつ配られるカードと、**コミュニティカード (Community cards)** と呼ばれるプレイヤー全員が使用できる 5 枚の共通のカードの計 7 枚の中から、最も強い 5 枚の組み合わせを用いてハンドの強さを競います。よくあるクローズドポーカーとは異なり、配られた手札を交換することはできません。最終的に手札を公開してハンドの強さを競うことになった場合（これを**ショーダウン (Showdown)** と呼ぶ）、より強いハンドを持っていたプレイヤーがそれまでに賭けられてきた**ポット (Pot)** を総取りします。

ただし、実際の進行ではショーダウンが行われることはそれほどありません。というのも、他のプレイヤーを全員 “降ろす” ことができれば、その時点でポットは残った 1 人のプレイヤーのものとなるためです。ベッティングラウンドでは、賭け金の合意が取れるまで各プレイヤーは定められた順序で以下のいずれかのアクションを取る必要があります :

- **ベット (Bet) / レイズ (Raise)**: 現在の賭け金からさらに賭け金を上乗せするアクションです。賭け金がゼロの状態から行うアクションをベット、既にベットされている状態から行うアクションをレイズと言います。
- **チェック (Check) / コール (Call)**: 現在の賭け金に同意し、同額を支払ってプレイを継続するアクションです。追加の賭け金が必要無い状態で行うアクションをチェック、賭け金を追加で支払う必要のある状態で行うアクションをコールと言います。
- **フォールド (Fold)**: 現在の賭け金を支払わず、手札を捨てて勝負から降りるアクションです。フォールドを行うと、それ以降のベッティングラウンドにも参加できず、ポットを獲得する権利を完全に失います。

テキサスホールデムでは、ショーダウンまでに 4 回のベッティングラウンドを行います :

- **プリフロップ (Pre-flop)**: 2 枚のホールカードが配られた直後に行われるラウンドです。コミュニティカードは 1 枚も公開されません。プリフロップでは、**スモールブラインド (Small blind)** と**ビッグブラインド (Big blind)** と呼ばれる 2 人のプレイヤーは定められた額のベットを強制的に行わなければなりません。
- **フロップ (Flop)**: コミュニティカードが 3 枚公開された状態で行われるラウンドです。
- **ターン (Turn)**: 4 枚目のコミュニティカードが公開された状態で行われるラウンドです。

- **リバー (River):** 5 枚目のコミュニティカードが公開され、すべてのコミュニティカードが明らかになった状態で行われるラウンドです。

特にプリフロップの段階では、 最終的に使用できる 7 枚のカードのうち 2 枚しか明らかになっていません。この時点での勝率の計算などを厳密に行おうとすると、大量にあり得るハンドの役判定が必要になるという冒頭の主張に繋がるわけです。

3. レギュレーション

さて、以上に説明したテキサスホールデムのルールを踏まえて、作りたいプログラムの要件を定義していきましょう。今回の目標は「7 枚のカードの組が与えられたときの役の判定」を行うプログラムであって、なるべく高速なものを作ることです。もう少し厳密に書くと、次の要件を満たすようななるべく高速な関数の実装を目指します。

- 7 枚のカードで構成されるハンドを入力として受け取って非負整数を返す関数であって、2.1 節に示したようにハンドの強さを定義したとき、 より強いハンドが入力として与えられた場合はより大きい整数を返し、 強さが同じハンドが入力として与えられた場合は同じ整数を返すようなもの（特に、役が同じであっても多くの場合はタイブレークによる強弱が存在することに注意する）。

なお、各カードは 0 〜 51 の整数で表現されているものとし、0 〜 3 が2♣ 2♦ 2♥ 2♠、4 〜 7 が3♣ 3♦ 3♥ 3♠、…、48 〜 51 がA♣ A♦ A♥ A♠に対応しているものとします。

今回は勝率計算などに用いるサブルーチンとしての役判定を想定しているため、プログラムの高速化のために次のような機能は含めなくてよいものとします：

- ハンドが 7 枚未満の場合の役判定を行う機能。
- 7 枚のカードのうちどの 5 枚が用いられたかを示す機能。
- 入力のバリデーションチェック。 すなわち、 入力長は 7 であり、 入力の各要素は 0 〜 51 の整数であって相異なることを仮定する。

プログラムは以下の 3 点によって評価されます :

- **シーケンシャル速度** : ${}_{52}C_7$（= 133,784,560）通りの 7 枚のカードの組み合わせについて辞書順にハンドの評価を行い、掛かった時間を計測する。 この測定では、ハンドを表現する何らかのデータ構造にカードを追加する操作は最内ループの外で行っても良いものとする。すなわち、以下のようなプログラムが認められる :

```
Hand hand = Hand();
for (int card1 = 0; card1 < 46; ++card1) {
  // add_card() の複雑さに関わらず、この操作を以降のループ中に含めなくてよい
  Hand hand1 = hand.add_card(card1);  // hand に card1 を追加した結果を返す
  for (int card2 = card1 + 1; card2 < 47; ++card2) {
    Hand hand2 = hand1.add_card(card2);
    // 3 枚目以降のカードの処理 ...
  }
}
```

- **ランダムアクセス速度** : 1 億通りのランダムなハンドについて評価を行い、掛かった時間を計測する。 この測定では、0 〜 51 の整数からなる要素数 7 のランダムな配列のみが予め用意される。 なお、配列の要素はソートされているとは限らない。ハンドを表現する何らかのデータ構造にカードを追加する操作が必要な場合は毎回ゼロから行う必要があり、 その時間も計測結果に反映される。
- **メモリ使用量** : これから紹介するプログラムには、 予め前計算したテーブルを用いるものが含まれるため、 そういった静的なテーブルの大きさを示す（実行時のメモリ使用量を計測したものではない）。静的なテーブルを用いない場合は「極小」と表記する。

プログラムの記述言語には C++17 (g++ 10.2.0 -O3 -march=native）を採用します。また、実行時間の計測には筆者のデスクトップマシン（Ryzen 7 3700X CPU）を用います。

それでは、次章でさまざまな実装について見ていきましょう！

4. さまざまな実装

4.1. ナイーブな実装

- **シーケンシャル速度**（1.33 億ハンド）: 58.0 秒（231 万評価 / 秒）
- **ランダムアクセス速度**（1 億ハンド）: 50.0 秒（200 万評価 / 秒）
- **メモリ使用量** : 極小

何はともあれ、まずはベースラインとなるナイーブな実装について見ていきましょう。7 枚のカードから 5 枚を選ぶ ${}_7\mathrm{C}_5 = 21$ 通りの選び方を予め定数として用意しておき、その組み合わせをすべて試して最も強かった結果を返すというのがナイーブな実装の大枠になるかと思います。

5 枚のカードからなるハンドの評価を行う関数 `evaluate_naive_5()` については後ほど見ていくことにして、プログラムを記述していくと次のようになるでしょう :

```
// 7 choose 5 の組み合わせを予め定数として持っておく
const int comb_7_5[21][5] = {
  { 0, 1, 2, 3, 4 }, { 0, 1, 2, 3, 5 }, (中略), { 2, 3, 4, 5, 6 }
};

// 7 枚のカードからなるハンドの強さを評価する
int evaluate_naive(const vector<int> &hand) {
  int best = 0;
  vector<int> subhand(5);
  for (int i = 0; i < 21; ++i) {
    for (int j = 0; j < 5; ++j) {
      subhand[j] = hand[comb_7_5[i][j]];
    }
    best = max(best, evaluate_naive_5(subhand));
  }
  return best;
}
```

それでは、関数 `int evaluate_naive_5(const vector<int> &hand);` を続いて定義していきましょう。ナイーブとは言えども、ビット演算を利用した効率的な実装を目指すこととし、ま

ずは出現したスートとランクを管理するビットセットと、各ランクの出現回数を管理する変数を準備します。

```
int suitset = 0;
int rankset = 0;
int rankcount[13] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
for (int card : hand) {
  int suit = card % 4;
  int rank = card / 4;
  suitset |= 1 << suit;
  rankset |= 1 << rank;
  ++rankcount[rank];
}
```

さらに、出現回数からランクのビットセットを逆引きできる変数も準備します。

```
int rankset_of_count[5] = { 0, 0, 0, 0, 0 };
for (int rank = 0; rank < 13; ++rank) {
  rankset_of_count[rankcount[rank]] |= 1 << rank;
}
```

以上の変数が用意できれば、フラッシュとストレートの判定は以下のように行えます。

```
bool is_flush = count_ones(suitset) == 1;
int is_straight = rankset & (rankset << 1) & (rankset << 2)
                          & (rankset << 3) & (rankset << 4);
if (rankset == 0b1'0000'0000'1111) is_straight = 1 << 3;
```

ここで、`count_ones()` は 2 進数表記でのビットの立っている数を返す関数とします （gcc/clang における `__builtin_popcount()` と同様)。フラッシュの判定は明らかですが、ストレートの判定については解説が必要かもしれません。 ストレートが成立している場合、 ランクのビットセットは 5 桁連続でビットが立っていることになります（5432A の場合を除く)。このとき、 ビットを 0 〜 4 桁ズラしたもの同士でビット論理積を取るとその結果は非ゼロとなります。

```
  0000000111110 (x)
& 0000001111100 (x << 1)
& 0000011111000 (x << 2)
& 0000111110000 (x << 3)
& 0001111100000 (x << 4)
-----------------------
  0000000100000
```

逆に、ストレートが成立していない場合は、このビット論理積はゼロとなります。よって、唯一の例外である 5432A のストレートについて最後にケアしてあげれば、変数 `is_straight` は非ストレートの場合はゼロ、またストレートの場合は非ゼロでかつ最もランクの大きいカードの情報が入っている状態となります。

ここまで準備ができれば、あとはナイーブに役判定を行っていくのみです。ここで返り値のフォーマットについて考える必要がありますが、上位 4 ビットに役の情報を、下位 26 ビットにタイブレークに関する情報を埋め込んだ 30 ビットの整数を用いることにしましょう。

```
if (is_flush && is_straight) {
  // straight flush
  return (8 << 26) | is_straight;
} else if (rankset_of_count[4]) {
  // four of a kind
  return (7 << 26) | (rankset_of_count[4] << 13) | rankset_of_count[1];
} else if (rankset_of_count[3] && rankset_of_count[2]) {
  // full house
  return (6 << 26) | (rankset_of_count[3] << 13) | rankset_of_count[2];
} else if (is_flush) {
  // flush
  return (5 << 26) | rankset;
} else if (is_straight) {
  // straight
  return (4 << 26) | is_straight;
} else if (rankset_of_count[3]) {
  // three of a kind
  return (3 << 26) | (rankset_of_count[3] << 13) | rankset_of_count[1];
```

```
} else if (rankset_of_count[2]) {
  // two pair or one pair
  int num_pairs = count_ones(rankset_of_count[2]);
  return (num_pairs << 26) | (rankset_of_count[2] << 13) | rankset_of_count[1];
} else {
  // high card
  return (0 << 26) | rankset;
}
```

以上がナイーブな実装についてでしたが、これだけで予想外に 3 ページ強も費やしてしまいました。タイブレークまで正確に考慮した実装を簡潔に記述するのは案外簡単ではないのではということで丁寧に解説してみましたが、いかがでしたでしょうか。

4.2. 7 枚のカードを直接評価する実装（ナイーブ）

- **シーケンシャル速度**（1.33 億ハンド）: 2.77 秒（4830 万評価 / 秒）
- **ランダムアクセス速度**（1 億ハンド）: 3.14 秒（3190 万評価 / 秒）
- **メモリ使用量** : 極小

4.1 節で紹介したナイーブな実装は ${}_7C_5$ 通りのカードの選び方をすべて試すものでしたが、7 枚のカードが与えられたときにそれらを直接評価する戦略も考えられます。例えば、K♣ 10♦ 8♣ 4♣ 4♥ 4♠ 3♥というハンドが与えられたとき、 人間は ${}_7C_5$ 通りのカードの選び方をすべて考えなくとも、まず 4 のスリーカードを認識し、残ったカードのうちランクの高いK♣と10♦を加えた4♣ 4♥ 4♠ K♣ 10♦が最終的な 5 枚組になることを把握できるでしょう。

このような戦略に基づく実装を正確に書き下すのは、5 枚のハンドを評価する実装と比べてさらに注意力を要しますが、丁寧に考察をしていけばそれなりに簡潔に記述することが可能です。まずはユーティリティ関数として、立っているビットのうち上位から n ビットを保持する関数を定義しましょう：

```
int keep_n_msb(int x, int n) {
  int result = 0;
  for (int i = 0; i < n; ++i) {
    int m = msb(x);
```

```
    x ^= m;
    result |= m;
  }
  return result;
}
```

ここで、`msb(x)` は入力 `x` の立っているビットのうち最上位のビットのみが立った値を返す関数とします（gcc/clang では `int` 型に対しては `1 << (__builtin_clz(x) ^ 31)` として定義できます）。このような関数を用意できると、先ほどの例のようなスリーカードの処理は次のように書き下すことができます：

```
// スリーカードより強い役ではないことが確認できているとする
if (rankset_of_count[3]) {
  // three of a kind
  int remaining = keep_n_msb(rankset_of_count[1], 2);
  return (3 << 26) | (rankset_of_count[3] << 13) | remaining;
}
```

本節では実装のすべての紹介は行いませんが、この戦略に基づく実装を実際に書いてみようという場合は、ストレートとフラッシュが両方成立しているからといってストレートフラッシュとは限らない点（ストレートフラッシュは同じスート内でストレートを完成させる必要がある）、3 枚組が 2 つある場合の処理（フルハウス）、2 枚組が 3 つある場合の処理（ツーペア）などに気をつけてみてください。

筆者の実装では、4.1 節の実装と比べて 16 〜 21 倍程度の高速化を実現することができました。どの 5 枚が最終的に用いられたかの情報を復元する処理は自明には書けなくなってしまいましたが、それと引き換えにこれだけの高速化が施せれば十分でしょう。

このような記事を書いておいて言うのも微妙ですが、ポーカーの役判定が必要となるような場面のほとんどは、4.1 節と本節に示したようなナイーブな処理で正直間に合うようにも思います。ここからさまざまなテクニックを用いて処理を高速化していくわけですが、最終的に得られるプログラムは本節のものと比べてシーケンシャル速度で 17 倍、ランダムアクセス速度で 7.5 倍速いという程度に留まります。この 7.5 〜 17 倍を引き出す過程が面白いと感じたから

こそこのような記事を書いているわけではありますが、7.5 倍遅い程度で済むなら明快な処理の方が好ましいという状況も少なくないのではないでしょうか。また、ナイーブな実装は高速な実装の動作をテストする際にも役に立ちますので、ナイーブなものが悪いということでは決してない点は留意していただければと思います。

4.3. 前計算に基づく巨大なテーブルを用いる実装（その１）

- **シーケンシャル速度**（1.33 億ハンド）: 9.11 秒（1470 万評価 / 秒）
- **ランダムアクセス速度**（1 億ハンド）: 33.0 秒（303 万評価 / 秒）
- **メモリ使用量** : 約 5.5GB

繰り返しですが、52 枚のカードのうち 7 枚を取り出す場合の数は ${}_{52}C_7 = 133{,}784{,}560$ 通りです。これらの約 1.33 億通りの場合について予め前計算を行ってテーブルとして保存してしまうことを考えると、各要素を 2 バイト整数で表現した場合（前述したようにハンドの強さは 7462 種類の同値類に分類できるため 2 バイト整数で表現可能）、高々 256MB の容量に収まることが分かります。この 256MB という容量は、例えばアプリに組み込むことを考えるとこれだけですでに許容できない容量で、後述するようにハッシュテーブル等を用いてしまうとさらに定数倍が掛かってしまいますが、ローカルで計算を行う分には何とかなるでしょう。ここからは、このように予め約 1.33 億通りの場合をすべて前計算してしまって、その結果を利用することで高速化が図れないかを考えていきます。

まず問題となるのは、0 〜 51 の整数で表現されたカード 7 枚の組が与えられたときに、どうやってテーブルを引くのかという点です。つまり、カード 7 枚の組を何かしらのフォーマットに正規化する方法を考える必要があります。ここでは、カードを表す整数 7 つをソートして順序を一意にした上で、それらを 8 ビット整数として並べた 56 ビット整数として表現することにしてみましょう。このようにフォーマットを定めてさえしまえば、あとは探索木なりハッシュテーブルなりで検索を行うことが可能になります。例えば、ハッシュテーブルを用いた実装は次のようになるでしょう :

```
// ハッシュテーブルは前計算しておく
unordered_map<uint64_t, uint16_t> hash_table;

uint16_t evaluate_with_hash_table(const vector<int> &hand) {
```

```
  int sorted[7];
  copy(hand.begin(), hand.end(), sorted);
  sort(sorted.begin(), sorted.end());
  uint64_t lo = sorted[0] | (sorted[1] << 8) | (sorted[2] << 16) | (sorted[3] << 24);
  uint64_t hi = sorted[4] | (sorted[5] << 8) | (sorted[6] << 16);
  return hash_table[lo | (hi << 32)];
}
```

さてこれにて一件落着 … としたいところなのですが、せっかく高速化の記事ですのでこの実装をさらに高速にすることを考えましょう。 ここで注目するのは `sort()` を呼び出している部分です。C++ STL（標準ライブラリ）のソート関数は汎用的に使用できるものですが、今回は要素数が 7 であることが分かっているため、このことを利用します。

小さい配列のソートのような処理においてボトルネックとなるのは、主に条件分岐です。CPU はヒューリスティクスによって条件分岐命令が実行される以前から分岐を予測し、投機的に命令を実行していますが、 この予測が失敗した際にオーバーヘッドが生じます。 小さい配列のソートにおいては、条件分岐が何回も行われる上にその予測も難しいため、このオーバーヘッドが問題になるわけです。

そこで、 ここでは**ソーティングネットワーク**を用いることにします。 ソーティングネットワークに関する詳細な説明は Wikipedia 等に譲りますが、簡単に言えば 2 要素のソートを定められた順序で複数回繰り返すことでソートを実現するという手法です。 この “定められた順序で” というのが重要で、比較の順序がそれまでの比較結果によらないため、比較の順序に関する分岐を廃することが可能で、並列実行に向いたアルゴリズムでもあります。

7 要素のソートの場合は、2 要素のソートを 16 回行うことでソートを行うことが可能で、 また 16 回という回数が下限である（＝最適である）ことも知られています（ソーティングネットワークでない一般の比較ソートでは比較回数の下限は 13 回となりますが、こちらはそれまでの比較結果に応じて比較対象が動的に変化します）。

```
void sort_7(int *p) {
  sort_2(p[0], p[4]); sort_2(p[1], p[5]); sort_2(p[2], p[6]);
  sort_2(p[0], p[2]); sort_2(p[1], p[3]); sort_2(p[4], p[6]);
```

```
  sort_2(p[2], p[4]); sort_2(p[3], p[5]);
  sort_2(p[0], p[1]); sort_2(p[2], p[3]); sort_2(p[4], p[5]);
  sort_2(p[1], p[4]); sort_2(p[3], p[6]);
  sort_2(p[1], p[2]); sort_2(p[3], p[4]); sort_2(p[5], p[6]);
}
```

続いて、2 要素のソートを行う関数を定義する必要がありますが、SSE 4.1 (または AVX) 拡張命令の (v)pminsd, (v)pmaxsd 命令を利用することで条件分岐を完全に廃することを目指します。手元の環境（gcc 10.2.0 -O3 -march=native）では、次のように記述することで vpminsd, vpmaxsd 命令が発行されました。

```
void sort_2(int &a, int &b) {
  int minv = a < b ? a : b;
  int maxv = a >= b ? a : b;
  a = minv;
  b = maxv;
}
```

ここまでのソートの高速化によって短縮された計算時間は、ランダムアクセス（1 億ハンド）で 30 秒程度でした。高速化後のランダムアクセス（1 億ハンド）の全体の計算時間は 33.0 秒ですから、計算時間を半分近くにまで削減できたことになりますが、残りの半分は普通にハッシュテーブルを引く処理であって、 方針を変えない限りはこれ以上の高速化は難しそうです。前計算した結果がメモリに収まるからと言って、 安直にハッシュテーブルを用いるだけでは、4.2 節の実装より 10 倍以上も遅くなってしまうことが分かりました。

4.4. 前計算に基づく巨大なテーブルを用いる実装（その２）

- **シーケンシャル速度**（1.33 億ハンド）: 3.70 秒（3610 万評価 / 秒）
- **ランダムアクセス速度**（1 億ハンド）: 3.86 秒（2590 万評価 / 秒）
- **メモリ使用量** : 256MB

4.3 節のハッシュテーブルを用いた実装は、 ハッシュテーブルを引くのが遅い上にメモリ使用量も定数倍が大きく 5GB 以上を消費し、 良いことがありませんでした。 カード 7 枚の組が与えられたときに、0 〜 133,784,559 の範囲の一意な整数を何らかの方法で直接対応させるこ

とができれば、ハッシュテーブルではなく単に配列を用意すれば良くなるのですが、そのようなことは可能なのでしょうか。

ここで **Combinatorial number system** の出番です。実は、カード $c_7 > c_6 > \cdots > c_1 \geq 0$ が与えられたときに、カードの組 $(c_7, c_6, \cdots, c_1)$ が辞書順で何番目に現れるかというと、

$$ {}_{c_7}\mathrm{C}_7 + {}_{c_6}\mathrm{C}_6 + \cdots + {}_{c_1}\mathrm{C}_1 $$

番目であることが知られています。この式を利用して、ハッシュテーブルを用いない効率的なテーブルの参照を行ってみましょう。

まずは、${}_n\mathrm{C}_k$ を毎回計算するのを避けるため、これらを変数に代入します：

```
// comb[i][j] は i choose (j + 1) を表す
vector<vector<int>> comb(52, vector<int>(7));

void prepare_comb_table() {
  for (int i = 0; i < 52; ++i) {
    comb[i][0] = i;
    for (int j = 1; j < 7; ++j) {
      comb[i][j] = comb[i][j - 1] * (i - j) / (j + 1);
    }
  }
}
```

変数 `comb` を前計算したら、実際にテーブルを引く処理は次のように記述できます：

```
// 配列は前計算しておく
vector<uint16_t> vec;

uint16_t evaluate_with_vector(const vector<int> &hand) {
  int sorted[7];
  copy(hand.begin(), hand.end(), sorted)
  sort_7(sorted);
  int key = comb[sorted[0]][0] + comb[sorted[1]][1] + comb[sorted[2]][2]
```

```
            + comb[sorted[3]][3] + comb[sorted[4]][4] + comb[sorted[5]][5]
            + comb[sorted[6]][6];
    return vec[key];
}
```

このように、ハッシュテーブルを用いていた部分を単なる配列に差し替えることで、メモリ使用量は 5.5GB から 256MB にまで減少し、ランダムアクセス速度（1 億ハンド）も 33.0 秒から 3.86 秒にまで短縮されました。ただし、それでも 4.2 節の 7 枚のカードを直接評価する実装に対しては 1.3 倍程度の計算時間となっており、ナイーブな実装の優秀さが際立っていますね。多少の工夫では前計算を施しても太刀打ちできないナイーブな実装に対して、どのように高速化を考えていけば良いのでしょうか。

4.5. 前計算に基づく巨大なテーブルを用いる実装（その３）

- **シーケンシャル速度**（1.33 億ハンド）: **0.06 秒（22.0 億評価 / 秒）**
- **ランダムアクセス速度**（1 億ハンド）: 6.91 秒（1450 万評価 / 秒）
- **メモリ使用量** : 124MB

4.3 節と 4.4 節で見てきた前計算に基づくテーブルを用いる実装では、入力の組を正規化するためにソートを行っていましたが、今度はこのソートを行わずにテーブルを引く方法を模索します。鍵となる観察は、例えば「K♠ → 8♥」と読んだ状態と、「8♥ → K♠」と読んだ状態は同じ状態であるという点です。つまり、**有向非巡回グラフ (Directed acyclic graph; DAG)** を辿っていくことを考えると、「K♠ → 8♥」を読んだ場合と「8♥ → K♠」を読んだ場合が同じ頂点（ノード）に辿り着くようにグラフを構成すれば良さそうなことが分かります。

さらにこのグラフを最適化することを考えると、例えば「K♠ → 8♥ → 6♦ → 9♣」と「K♠ → 8♥ → 6♣ → 9♦」は後半の 6 と 9 のスートが異なりますが、どちらもフラッシュの可能性が無いため、役判定の観点からはこれまでに出現したランク（K、9、8、6）の情報のみを保持すればよく、これらも同じ状態として扱って良いことが分かります。このようにして構成されたグラフは実は 124MB の容量に収まることが知られており、2006 年頃に Two Plus Two というフォーラムで考案されたことから “Two Plus Two evaluator” などと呼ばれています。

```
// 構成されたグラフを予めロードしておく
vector<int> hand_ranks(32487834);

int evaluate_two_plus_two(const vector<int> &hand) {
  int p = 53;
  for (int i = 0; i < 7; ++i) {
    p = hand_ranks[hand[i] + p + 1];
  }
  return p;
}
```

評価関数は以上のように極めて単純なコードで記述することができ、 シーケンシャル速度（1.33 億ハンド）は驚異の 0.06 秒を記録しました。それに対してランダムアクセス速度（1 億ハンド）は 6.91 秒とスループットが 1/150 以下になっており、ランダムアクセスには弱いことが分かります。 これは、 処理は極めて単純であるものの、 ランダムアクセスの場合はキャッシュヒット率が非常に悪いことに起因しているものと思われます。巨大なテーブルを用いる方法が根本的に抱えている問題が浮き彫りになってしまった形で、ランダムアクセス速度を高速化するには、テーブルの大きさをキャッシュに乗る大きさに収めないと意味が無さそうなことが分かってきましたね。

4.6. Cactus Kev による実装

- **シーケンシャル速度**（1.33 億ハンド）: 10.3 秒（1300 万評価 / 秒）
- **ランダムアクセス速度**（1 億ハンド）: 10.2 秒（979 万評価 / 秒）
- **メモリ使用量** : 48KB

紹介する順序と歴史的な順序が前後してしまいましたが、 本節では Cactus Kev というポーカープレイヤー兼プログラマーが 2002 年頃に発明し、2006 年頃に Paul Senzee というプログラマーによって改良された実装を紹介します。

まず第一の発明として、各カードを 0 ～ 51 の整数で表現するのではなく、次に示す 32 ビット整数として表現することを考えました :

```
32-bit representation = |xxxAKQJT|98765432|cdhsrrrr|xxpppppp|

p: r番目の素数(2->2, 3->3, ..., A->41)
r: ランクの4ビット表現(2->0, 3->1, ..., A->12)
cdhs: スートのビットフラグ(例えばスペードなら0001となる)
AK...2: ランクのビットフラグ(例えば4なら00000|00000100となる)
```

このように表現を行うと、5枚のカードからなるハンドの役判定の大枠は次のように記述することができます:

```
uint16_t evaluate_cactus_kev_internal(int c1, int c2, int c3, int c4, int c5) {
  // ランクのビットフラグの論理和(最大値は0b1'1111'0000'0000 = 7936)
  int q = (c1 | c2 | c3 | c4 | c5) >> 16;

  // フラッシュかどうかの判定
  if (c1 & c2 & c3 & c4 & c5 & 0xf000) {
    // 変数qでテーブルを引く
    return flushes[q];
  }

  // ストレートおよびハイカードの場合はテーブルを引いて判定 / 処理
  if (unique5[q]) {
    return unique5[q];
  }

  // フラッシュ / ストレート / ハイカードのいずれでもない場合の処理
  return ???;
}
```

フラッシュ / ストレート / ハイカードの場合の処理はたった7,937要素のテーブルを引くだけで行えるようになり、これまでのキャッシュヒットの問題は解決することができました。問題は残る場合の処理ですが、すでにフラッシュではないことが分かっているため、ランクの組み合わせのみによって役が決まることを利用します。ここで、32ビット表現の下位6ビットに相異なる素数を割り当てたことを思い出すと、これらの積がランクの組み合わせを表現するのに最適です。

それでは、ランクの組み合わせを表現する積は何通りの値を取り得るかを考えてみましょう。まず、簡単のため各ランクのカードが無限に存在すると仮定すると（実際には 4 枚ですが）、それらの中から 5 枚を選ぶ場合の数は ${}_{13+5-1}\mathrm{C}_5 = 6{,}188$ 通りです。この中には同じランクを 5 枚選んでいる場合が 13 通りだけ含まれてしまっているので、実際には $6{,}188 - 13 = 6{,}175$ 通りとなることが分かります。さらに、今回は 5 枚のランクが相異なる場合を除外しているので、それら ${}_{13}\mathrm{C}_5 = 1{,}287$ 通りを差し引くと、最終的に 4,888 通りが残ることになります。

よって、理論的には 4,888 要素のテーブルを用意すれば、そのテーブルを引くことで役判定が行えることになります。問題はそのテーブルをどのように引くかですが、ここに**完全ハッシュ関数 (Perfect hash function)** を用いることを考えたのが Paul Senzee というプログラマーです。完全ハッシュ関数とは、定義域が定まっているときにそれらの入力に対して単射であるような（つまり衝突が無い）ハッシュ関数のことで、Paul Senzee は具体的に以下のようなマジカルな実装を与えました：

```
unsigned find_fast(unsigned u) {
  u += 0xe91aaa35;
  u ^= u >> 16;
  u += u << 8;
  u ^= u >> 4;
  unsigned a = (u + (u << 2)) >> 19;
  unsigned b = (u >> 8) & 0x1ff;
  return a ^ hash_adjust[b];
}
```

ここで、`hash_adjust` は要素数 512 のマジカルな配列ですが、 この関数を用いると前述した 4,888 通りの積を 13 ビットの一意な整数に変換することができます。 よって、8,192 要素の配列 `hash_values` を前計算することで、 フラッシュ / ストレート / ハイカードのいずれでもない場合の処理は次のように記述できるようになりました。

```
return hash_values[find_fast(
  (c1 & 0x3f) * (c2 & 0x3f) * (c3 & 0x3f) * (c4 & 0x3f) * (c5 & 0x3f)
)];
```

ここまでに用意したテーブルの容量は 48KB 程度と、 これまでに見てきたテーブルとは桁違

いに容量を削減できており、これならキャッシュヒット率も問題ありません。7 枚のカードからなるハンドを評価するには、4.1 節に示した最もナイーブな実装と同じように ${}_7C_5$ 通りの 5 枚組を評価しなければなりませんが、ランダムアクセス速度（1 億ハンド）は 10.2 秒と、4.1 節の実装と比べると 5 倍程度高速になっています。7 枚のカードを直接評価する 4.2 節の実装と比較してしまうとやはり遅いですが、7 枚のうちどの 5 枚が用いられたかを復元する必要がある場合などは、こちらを使用した方が簡潔に処理できるかもしれません。

4.7. これまでの考察を活かした実装

- **シーケンシャル速度**（1.33 億ハンド）: 0.16 秒（8.19 億評価 / 秒）
- **ランダムアクセス速度**（1 億ハンド）: **0.41 秒（2.41 億評価 / 秒）**
- **メモリ使用量** : 145KB

さて、ここまで前置きが長くなってしまいましたが、これまでの考察を活かしてオレオレ最強爆速評価器を作ることを考えていきましょう。

4.6 節の Cactus Kev による実装は、5 枚のカードからなるハンドにしか直接適用できませんでしたが、以下のような重要なテクニックを与えてくれました :

- 52 枚のカードを 0 ～ 51 の整数ではなく特殊なフォーマットで表現する
- フラッシュとそれ以外に処理を分割することで、テーブルサイズを大幅に削減する
- 完全ハッシュ関数の導入

本節では、これらのテクニックを引き続き用いながら、7 枚のカードからなるハンドを直接評価する方法を考えていきます。まずは、7 枚のカードの組が与えられたときに、フラッシュかどうかをどのように高速に判定するかを考えていきましょう。4.6 節ではどのように判定していたかというと、各カードにスートを表す 4 ビットのビットフラグを持たせて、それらのビット論理積を取って判定をしていました。ハンドが 7 枚のカードで構成される場合では、7 枚のうちの 5 枚が同じスートなら良いわけですから、全く同じ戦略を取ることはできません。

そこで、今度は各スートの出現回数をカウンターを用いて記録することにしましょう。具体的には、各スートの出現回数は最大で 7 回ですから、8 進数を用いて記録してあげれば良さそうです。例えば、「0o4102」はクラブが 4 枚、ダイヤモンドが 1 枚、ハートが 0 枚、スペードが 2 枚現れたことを示すといったようにです。このようにフォーマットを定めれば、フラッシュかどうかの判定は「5」以上の桁が存在するかどうかという問題に変換することができます。

ただ、「5」以上の桁が存在するかどうかという判定を実際に行おうとすると、愚直に処理を書き下すしかなく、あまり高速には判定が行えません。そこで、8 進数と言わず 16 進数を使うことにし、各桁に「3」だけ下駄を履かせることを考えます。つまり、これまで「`0o4102`」だった表現は、16 進数に変換した上で各桁に「3」が足されて、「`0x7436`」に変換されることになります。このように表現すると何が嬉しいかというと、フラッシュかどうかの判定は「8」以上の桁が存在するかどうかという問題となり、ビットフラグ「`0x8888`」とビット論理積を取るだけで判定が行えるようになっているのです。ビット演算に不慣れな方には理解しづらいかもしれませんが、その場合は実際に手を動かしてみると理解できるようになるかもしれません（それでも良く分からない場合はそういうものだとして受け入れましょう）。

それでは、続いてランクに関する表現を考えていきましょう。4.6 節では、ランクの組み合わせを素数の積で表現していましたが、カードが 7 枚となるとその最大値は

$$41^4 \times 37^3 = 143{,}133{,}271{,}933$$

となってしまい、この後に完全ハッシュ関数を適用するとはいえ、値がちょっと大きくなりすぎてしまいます。そこで、ランクの組み合わせをある基底の「積」ではなく「和」で表現することを考えてみましょう。最も明快な方法は、スートの出現回数を 16 進数で表現したように、ランクの出現回数も 5 進数で管理するというものです（同じランクのカードは最大でも 4 枚しか出現しないため、5 進数で一意な表現が可能です）。つまり、基底として $\{5^0, 5^1, \cdots, 5^{12}\}$ を採用した場合に対応しており、カード 7 枚の組が与えられたときの最大値は

$$5^{12} \times 4 + 5^{11} \times 3 = 1{,}123{,}046{,}875$$

となります。この時点ですでに素数の積による表現よりも最大値を小さくできていますが、さらに最適化された基底として、OMPEval [*1]というライブラリでは `{0x2000, 0x8001, 0x11000, 0x3a000, 0x91000, 0x176005, 0x366000, 0x41a013, 0x47802e, 0x479068, 0x48c0e4, 0x48f211, 0x494493}` という非自明な基底が用いられています。この基底を用いた場合、カード 7 枚の組が与えられたときの最大値は `0x494493` × 4 + `0x48f211` × 3 = 33,548,415 となり、表現に必要なビット数を 25 ビットにまで落とし込めています[*2]。最適な基底の生成方法については筆者自身よく理解できていないのですが、ともあれ今回はこの基底を流用することにしましょう。

---

1 https://github.com/zekyll/OMPEval

2 表現に必要なビット数を削減するのみなら、23 ビットにまで落とし込んだ基底を自前でも生成できましたが、後述の完全ハッシュ関数を構築する段になるとこちらの基底の方が優れた結果となります。

ランクの組み合わせに関する表現が定まったので、今度は続いてこの表現に適用する完全ハッシュ関数を自前で定義していきましょう。 カード 7 枚を用いたランクの組み合わせは 49,205 通り存在するため（${}_{13+7-1}C_7 = 50{,}388$ で上から抑えられることはすぐに分かります）、完全ハッシュ関数の値域は 0 〜 49,204 に収まるのが理想的と言えます。

ここからは、Single-displacement 法[*3]と呼ばれる手法に基づくハッシュ関数を構築することを考えていきます。Single-displacement 法では、 次に示すような非常に単純なプログラムでハッシュ値を計算します :

```
hash_key = offset[input_value / t] + (input_value % t);
```

ここで、`offset` はマジカルなテーブルでこれから構築を目指す対象、t はハッシュ関数の挙動を定めるパラメータですが、これだけでは何のこっちゃという感じだと思いますので、ここからは定義域を 48 未満の素数、また $t = 8$ として具体的に構築の例を見ていこうと思います。

$t = 8$ ですので、あり得る入力を長さ 8 の行に分割して次のような表を得ます :

```
.. ..  2  3 ..  5 ..  7
.. .. .. 11 .. 13 .. ..
.. 17 .. 19 .. .. .. 23
.. .. .. .. .. 29 .. 31
.. .. .. .. .. 37 .. ..
.. 41 .. 43 .. .. .. 47
```

このハッシュ関数の構築において、多くの場合に優れたパフォーマンスを発揮することが知られている First-fit-decreasing 法[*4]というヒューリスティクスでは、次にこれらの行を要素数が多い順に並び替えます :

```
.. ..  2  3 ..  5 ..  7
.. 17 .. 19 .. .. .. 23
.. 41 .. 43 .. .. .. 47
```

3 A. V. Aho and J. D. Ullman. **Principles of Compiler Design**. Addison-Wesley. 1977.

4 R. E. Tarjan and A. C. C. Yao. **Storing a Sparse Table**. Communications of the ACM, 22(11):606-611. 1979.

```
.. .. .. 11 .. 13 .. ..
.. .. .. .. .. 29 .. 31
.. .. .. .. .. 37 .. ..
```

最後に、各列で衝突が発生しないよう、上の行から順に可能な限り小さい値でシフトします：

```
shift(-2) .. .. |  2  3 ..  5 ..  7
shift(+1)       |    .. 17 .. 19 .. .. .. 23
shift(+6)       |                   .. 41 .. 43 .. .. .. 47
shift(+7)       |                      .. .. .. 11 .. 13 .. ..
shift(+9)       |                            .. .. .. .. .. 29 .. 31
shift(+1)       |    .. .. .. .. .. 37 .. ..
----------------+-------------------------------------------------
          .. .. |  2  3 17  5 19  7 37 41 23 43 11 .. 13 47 29 .. 31
```

このシフト量が先ほどマジカルなテーブルと言った `offset` に対応していて、今回の場合は並び替えた行を元に戻して `offset[] = { -2, 7, 1, 9, 1, 6 };` と定義すると、48 未満の素数を定義域とした場合に 0 〜 16 を値域とする完全ハッシュ関数を構築することができました。

それでは定義域をランクの組み合わせを表現する値に差し替えて、同様に完全ハッシュ関数を構築することを試みてみましょう。手元の実装では、$t = 2^{12}$ としたときに、First-fit-decreasing 法によって値域が 0 〜 49,599 となる完全ハッシュ関数を構築することができました。

さて、ここまで準備できればいよいよ実装に取り掛かることができます。まずは、ランクの基底となる定数を定義しましょう：

```
const uint32_t rank_bases[13] = {
  0x002000, 0x008001, 0x011000, 0x03a000, 0x091000, 0x176005, 0x366000,
  0x41a013, 0x47802e, 0x479068, 0x48c0e4, 0x48f211, 0x494493,
};
```

続いて、ハンドのデータを表現する構造体を作ります。SIMD 命令による高速化を可能にするため、構造体は共用体の中に定義することにします：

```
union HandData {
  struct {
    uint32_t rank_key;
    uint32_t suit_key;
    uint64_t bit_flag;
  } s;
  __m128i simd_reg;
};
```

ここで、simd_reg は構造体を 128 ビットレジスタとしても扱えるようにするためのメンバ変数です。rank_key では先ほど定義したランクの基底の和を管理し、suit_key の上位 16 ビットではスートの出現回数を管理し、bit_flag では出現したカードのビットセットを 64 ビット整数として管理することにします（64 ビット整数を 16 ビットごとに分割し、下位からクラブ、ダイヤモンド、ハート、スペードの 2 〜 A に対応させます）。

このように HandData 共用体を定義したら、各カードを次のように定数として定義できます：

```
const HandData cards[52] = {
  /* 2c */ {{ rank_bases[0], 0x1000'0000, 0x0000'0000'0000'0001 }},
  /* 2d */ {{ rank_bases[0], 0x0100'0000, 0x0000'0000'0001'0000 }},
  /* 2h */ {{ rank_bases[0], 0x0010'0000, 0x0000'0001'0000'0000 }},
  /* 2s */ {{ rank_bases[0], 0x0001'0000, 0x0001'0000'0000'0000 }},
  /* 3c */ {{ rank_bases[1], 0x1000'0000, 0x0000'0000'0000'0002 }},
  (中略),
  /* As */ {{ rank_bases[12], 0x0001'0000, 0x1000'0000'0000'0000 }},
};
```

さらに、実装の肝となる Hand 構造体も続けて定義していきましょう：

```
class Hand {
public:
  Hand() : data{{ 0, 0x3333'0000, 0 }} {}
  Hand(__m128i simd_reg) : data{.simd_reg = simd_reg} {}
  Hand add_card(int card) const;
```

```
  uint16_t evaluate() const;

private:
  HandData data;
};
```

デフォルトコンストラクタでは、suit_key をカウンタの初期値が 3 となるよう 0x3333'0000 で初期化し、 他のメンバは 0 で初期化しています。 また、simd_reg を引数とするコンストラクタも定義しておきます。

残るは Hand::add_card() と Hand::evaluate() の実装のみです。Hand::add_card() については簡単で、 自分自身に cards[card] を足し合わせたものを返せば良く、SIMD 命令を用いるように実装すると次のようになります:

```
Hand Hand::add_card(int card) const {
  return Hand{_mm_add_epi64(data.simd_reg, cards[card].simd_reg)};
}
```

最後に、Hand::evaluate() の実装を先に与えてしまうと次のようになります:

```
uint16_t Hand::evaluate() const {
  uint32_t is_flush = data.s.suit_key & 0x8888'0000;
  if (is_flush) {
    uint16_t flush_key = data.s.bit_flag >> (4 * leading_zeros(is_flush));
    return lookup_flush[flush_key];
  } else {
    uint32_t hash_key = offset[data.s.rank_key >> 12] + data.s.rank_key;
    return lookup[hash_key];
  }
}
```

ここで、leading_zeros() は 2 進数表記での leading zero の個数を返す関数です（gcc/clang における __builtin_clz() と同様)。以下簡単に Hand::evaluate() の実装を解説すると、2 行目における is_flush の定義では、先述したように suit_key に「8」以上の桁が存在するかどう

かをチェックした結果が代入されています。もしフラッシュの場合は、bit_flag を適切にシフトして、フラッシュとなっているスートで出現したランクのビットセットを取得し（4 行目）、そのビットセットでフラッシュ用のテーブルを引いています（5 行目）。フラッシュでなかった場合は、Single-displacement 法によるハッシュ値を計算し（7 行目）、そのハッシュ値でテーブルを引いています（8 行目）。

offset、lookup、lookup_flush はいずれも前計算を必要とするテーブルです。まず offset については、rank_key の最大値が 33,548,415、また $t = 2^{12}$ としたことを思い出すと、その要素数は $\left\lfloor 33{,}548{,}415 / 2^{12} \right\rfloor + 1 = 8{,}191$ 個となります。また、完全ハッシュ関数の値域は 0 〜 49,599 でしたので、lookup の要素数は 49,600 個です。最後に lookup_flush については、ビットセットの最大値は 0x1fc0 = 8,128 ですので、要素数は 8,129 個となります。これらのデータサイズは合計しても 145KB に収まっており、例えば Web アプリ等に組み込むことも可能でしょう。

記事の最後に紹介する実装だけあって、本節はここまでで 6 ページ超に及んでいますが、苦労した甲斐はちゃんとあり、シーケンシャル速度は 0.16 秒と 4.5 節の Two Plus Two evaluator に次いで速く、ランダムアクセス速度は最速の 0.41 秒を記録しました。特に、ランダムアクセス速度は 4.2 節のナイーブな実装が高い壁として立ちはだかっていましたので、ここまで工夫することでようやく打破できたという喜びを一緒に感じていただければ幸いです。

5. まとめ

テキサスホールデムの役判定を題材に、プログラムの高速化にまつわるさまざまなテクニックを紹介してみました。ニッチな題材なだけあって、ビット演算のテクニックやソートの高速化、完全ハッシュ関数についてなど、紹介したテクニックもニッチなものが多かったかもしれませんが、数学力、アルゴリズム力、実装力、アーキテクチャへの理解などさまざまな分野にまたがった力が求められる内容でもあったのではないかと思います。

今回ベンチマークに使用したプログラムは https://github.com/b-inary/yabai-vol6-src に公開していますので、実装の細部を確認したい方はそちらを参照ください。また、5 〜 7 枚のカードからなるハンドに対応した Rust 製ライブラリ https://github.com/b-inary/holdem-hand-evaluator も公開していますので、興味のある方はそちらもぜひ参照ください（こちらには 4.7 節で用いられているテーブルの生成スクリプトも含まれています）。

ゲーム自体も奥が深く、数学的にもプログラマ的にも興味深いテキサスホールデム、みなさんもぜひプレイしてみてください！

GNU Emacs で LaTeX 文書を書く話

MasWag

1. はじめに

1.1. Emacs、GNU Emacs とは

> メロスには Lisp がわからぬ。

時は 200X 年、*nix 業界では vi/Vim と Emacs 系というテキストエディタ界の二大巨塔で争う、エディタ戦争というエクストリームスポーツがあった[*5]。vi は遥か遠い昔に ed というラインエディタから派生して登場したスクリーンエディタで Vim はその vi の派生、Emacs 系のテキストエディタというのはこれまた遥か遠い昔に TECO というラインエディタと関係して生まれたスクリーンエディタであるオリジナルの Emacs やその代替のエディタであり、vi/Vim と Emacs 系エディタの両者の派閥が云々 ...

という話は本記事の本題ではないのでこれ以上深追いしないでおきます。本来単に Emacs と書いた場合 GNU Emacs 以外にも例えば MicroEMACS などを含む Emacs 系のエディタ全般を指しますが、本記事では GNU Emacs 以外の話はしないので、以下 GNU Emacs のことを単に Emacs と書きます。 [*6] GNU Emacs は Emacs Lisp という Lisp 方言で拡張を行うことができ様々な拡張機能が実装されてはいるものの、筆者を含め Emacs Lisp や Emacs の拡張機能のエコシステムに詳しくない人にはなかなか取っ付き辛いです。

---

5 2000 年代にはまだ Atom も VSCode もなく、やっと 2008 年に Sublime Text が登場した頃なのであった

6 というのは変な突っ込みが入らないための単なるおまじないであり、今時 Emacs と言われて GNU Emacs 以外のエディタの事を考える人はほぼいないでしょう

1.2. LaTeX とは

> 吾輩はてふである。名前はまだ無い。
>
> どこで生れたか頓と見當がつかぬ。何でも薄暗いじめじめした所でニヤーニヤー泣いて居た事丈は記憶して居る。吾輩はこゝで始めて組版処理システムといふものを見た。然もあとで聞くとそれは TeX といふ組版処理システム中で一番獰悪な種族であつたさうだ。

LaTeX は組版処理システムである。筆者は LaTeX のことは良くわからないですし今回の記事の本題でもないので、これ以上の LaTeX の説明は Wikipedia や TeX wiki を参照してください。SATySFi という型も付いていて比較的全うな組版処理システムを使っておきながらなぜ LaTeX を使う話を書いているのか、という気もしますが、出版社や学会の定型フォーマットに従わなければならない場合、今日 SATySFi のテンプレートが提供されているケースはまずないので結局 LaTeX を使わなければならない場面はかなり多いです。仮に文書の書式が自由であっても、複数人で文書を書く場合には、全員が SATySFi で文書を書ける弊サークルの様な例外を除いては、皆がある程度扱うことのできる組版処理システムと云うことで LaTeX を採用することも少なくないでしょう。[*7]

1.3. この文書は何？

本記事では Emacs 系エディタの中でもかなり長い間主流として使われている GNU Emacs の上で LaTeX の文書を書く際の設定について扱います。前述の通り GNU Emacs では Emacs Lisp を使って様々な拡張機能を書くことや設定を行うことができますが、Emacs Lisp に詳しくない人にとっては自分で一から書くのは大変です。また、どんな便利なパッケージがあるかなどの情報を得るのも頑張って調べる必要があります。この記事では「とりあえずこういう設定をするとそれなりに良いぞ」というものを見せることを目的とします。これが最善の方法ではないと思いますし、そもそも各人で最適な設定は異なると思いますが、自分にとって使いやすい設定を探す上で役立つと幸いです。

2. おことわり

本記事では LaTeX 及び BibTeX というものを聞いたことがあったり、最低限の使い方を知って

7 実は弊サークルでも最初期には LaTeX を使うという案もありました

いることを前提にしています。これらの使い方がわからない場合は TEX Wiki の LATEX 入門 [1] の記事等を参考にしてください。

また、本記事では Emacs Lisp の細かい言語機能などの知識は仮定しませんが[*8]、Emacs の使い方を最低限理解していることは前提としています。例えばメジャーモードとマイナーモード等の用語や、`M-x` や `C-c` などの良く Emacs の説明で使う表記は前置きなく使います。Emacs の初心者向けの説明は、例えば Emacs JP にある入門記事 [2] を参照してください。

本記事の内容は GUI 版の GNU Emacs 27 で動作確認を行っています。古い版の Emacs では動かないものもあるかもしれませんし、Spacemacs 等での動作確認も行っていません。また、mac OS と Linux で動作確認をしていますが、Windows では (持ってないので) 動作確認を行っていません。

また、本記事の内容は後日別途無償公開する可能性が多いにあります。また、本記事で紹介している設定を纏めた `init.el` は https://bit.ly/382lFAB [*9]からダウンロードできます。

3. 下ごしらえ

3.1. package.el

Emacs のパッケージのインストールは package.el[3] を使います。package.el は Emacs 24 以降では標準搭載されているパッケージ管理ツールです。一応 Emacs 23 でも自分でインストールすることで package.el を使うことは可能ですが、特殊な事情がある場合を除いて新しい Emacs を使う方が良いと思います。

Emacs 24 以降向けの package.el の設定方法は以下の様になります。

```
(require 'package)
(add-to-list 'package-archives
             '("melpa" . "https://melpa.org/packages/") t)
(package-initialize)
```

8 そもそも筆者自身もが Emacs Lisp の詳細をわかっていないです

9 短縮していない URL は以下になります : https://gist.github.com/MasWag/35bf58e27b36dc3f45263d7c247ca418

```
(package-refresh-contents)
```

3.2. use-package

Emacs のパッケージの設定には use-package[4] を使います。(use-package の登場以前の様に) `(require ...)` `(autoload ...)` の様な生の Emacs Lisp を使って設定をすることも可能ですが、use-package を使うとマクロを使ってより簡潔に設定を書くことができます。[*10]

use-package のインストール及び設定方法は以下の様になります。

```
(package-install 'use-package)
(require 'use-package)
```

4. YaTeX (野鳥): 筆者おすすめの LaTeX 用メジャーモード

YaTeX [6](Yet Another TeX mode for Emacs、野鳥) は筆者が長年使っている LaTeX 用のメジャーモードです。Emacs には標準で TeX mode も搭載されていますが、YaTeX の方がより高機能です。YaTeX と並んで AUCTeX も人気がある LaTeX 用メジャーモードの様ですが、筆者は使ったことがありません。

4.1. 補完

YaTeX は様々な場面で LaTeX のコマンドを補完することができます。詳細は公式ドキュメント [7] にありますが、以下ではその中でも特に良く使うものを紹介していきます。

section 型補完 : `\foo{...}`: C-c C-s

section 型補完は `\` コマンド名 `{...}` の形式の LaTeX コマンドの補完に用います。名前の通り `\section{...}` や、`\documentclass{...}`、`\ref{...}`、`\cite{...}` などの例があり

10 筆者はまだ使ったことがないですが、今時は use-package ではなく [[https://github.com/conao3/leaf.el][leaf.el]] の方が良いかもしれないです。この様に流行のパッケージが移り変るのも最適に Emacs を設定することを難しくしています。

ます。section 型補完を行う場合は、C-c C-s を打つと補完すべき LaTeX コマンドを聞かれるので、 入力します。\section{...} の様に単にコマンド名 + 内容を入力するものの場合はこれで終了ですが、\documentclass{...} の様にオプションを指定できる LaTeX コマンドや \ref{...} の様にこれまでに設定したラベルを選択するものの場合、追加でオプションを聞かれたり既存のラベルを選択する画面が表示されたりします。

また、section 型補完ではありませんが、例えば \section{foo} を \subsection{foo} に変更するなど、 既に使われているコマンドを別のものに変更したい場合には、\section の部分で C-c C-c を打つことでコマンドの置き換えができます。

begin 型補完 : \begin{foo}...\end{foo}: C-c C-b

begin 型補完は \begin{環境名}...\end{環境名} の形式の入力の補完に用います。\begin{document}...\end{document} や、\begin{itemize}...\end{itemize}、\begin{figure}...\end{figure}、\begin{tabular}...\end{tabular} などの例があります。begin 型補完を行う方法は、C-c C-b に続けて環境名に対応した一文字を打つ方法と、C-c C-b SPC を打った後に補完すべき環境名を聞かれるので、適宜入力する方法の二種類があります。前者の方がタイプ数が少ないので一見便利そうですが、 予め指定された少ない数の環境名にしか対応していないため、実際には C-c C-b SPC を使う場面の方が多いです[*11]。

また、section 型補完と同様に、例えば \begin{figure}...\end{figure} を \begin{wrapfigure}...\end{wrapfigure} に変更するなど、 既に使われている環境を別のものに変更したい場合には、\begin{foo} や \end{foo} の部分で C-c C-c を打つことでコマンドの置き換えができます。

maketitle 型補完 : \foo : C-c C-m

maketitle 型補完は \foo の形式の入力の補完に用います。\maketitle や \newpage などの例があります。section 型補完や begin 型補完と比べると使用頻度は少ないですが、C-c C-s の代わりに C-c C-m を打つことで、概ね section 型補完と同様に使うことができます。

数式記号・ギリシャ文字補完

数式中で ; や : を打つことでそれぞれ数式記号やギリシャ文字を補完することができます。

11 個人の感想です

特に数式記号については数式記号の形に即した入力で補完することができ、例えば ; に続けて o を打つと \circ が、oo を打つと \infty が、x を打つと \times が補完されます。詳細なコマンドについては ; や : に続けて <tab> を打つことで表示することができます。

4.2. プロセス起動 : C-c C-t

C-c C-t を打つことで latex や bibtex などの組版やプレビューアのプロセスを起動することができます。詳細は公式ドキュメント [8] にありますが、以下が良く使うと思います。

- LaTeX の起動 : C-c C-t j
  - 筆者は platex や pdflatex を直接呼ぶのではなく latexmk を呼ぶ様に設定することで、別途 dvipdfmx 等を呼ばずに PDF をコンパイルでき、さらに latexmk に変更を監視させることで自動で再コンパイルされる様にしています。
- PDF や dvi 等のプレビュー : C-c C-t p
  - latex や platex 等で dvi を生成してプレビューする場合は xdvi 等を起動する必要がありますが、筆者は PDF をプレビューしているので Skim や evince を起動する様に設定しています。

4.3. 対応する場所へのカーソルジャンプ : C-c C-g

YaTeX では \begin{foo}...\end{foo} の \begin{foo} の部分と \end{foo} や \ref{foo} と \label{foo} の様に対応した部分の行き来を C-c C-g で行うことができます。なお、\ref{foo} と \label{foo} のジャンプはできますが、どうやら \cref{foo} と \label{foo} のジャンプは上手く動かない様です。

4.4. use-package による設定法

use-package による YaTeX のインストール及び設定は以下の様になります。

```
(use-package yatex
  ;; YaTeX がインストールされていない場合、package.el を使ってインストールする
  :ensure t
  ;; :commands autoload するコマンドを指定
```

```
:commands (yatex-mode)
;; :mode auto-mode-alist の設定
:mode (("\\.tex$" . yatex-mode)
       ("\\.ltx$" . yatex-mode)
       ("\\.cls$" . yatex-mode)
       ("\\.sty$" . yatex-mode)
       ("\\.clo$" . yatex-mode)
       ("\\.bbl$" . yatex-mode))
:init
(setq YaTeX-inhibit-prefix-letter t)
;; :config キーワードはライブラリをロードした後の設定などを記述します。
:config
(setq YaTeX-kanji-code nil)
(setq YaTeX-latex-message-code 'utf-8)
(setq YaTeX-use-LaTeX2e t)
(setq YaTeX-use-AMS-LaTeX t)
(setq tex-command "/Library/TeX/texbin/latexmk -pdf -pvc -view=none")
(setq tex-pdfview-command "/usr/bin/open -a Skim")
(auto-fill-mode 0)
;; company-tabnine による補完。company については後述
(set (make-local-variable 'company-backends) '(company-tabnine)))
```

5. RefTEX: 参照挿入のためのマイナーモード

RefTEX[9] は `\ref{...}` / `\cref{...}` [*12]や `\cite{...}` といった参照関係のコマンドを検索して挿入するためのマイナーモードです。個人的な経験としては、複数のファイルに分割して文書を書く場合に検索に失敗することがありますが、単一ファイルで文章を書く際は概ね問題なく動いている様です。

以下のコマンドを良く使います。詳細は公式マニュアル [10] を参照してください。

- `\ref{...}` / `\cref{...}` の挿入 : `C-c (`
- `\cite{...}` の挿入 : `C-c [`

12 ちなみに cleveref パッケージを使うことで cref を使った賢い相互参照ができます。

5.1. use-package による設定法

use-package による RefTEX の設定は以下の様になります。RefTEX は Emacs 24.3 以降では Emacs に同梱されているので別途インストールせずに使うことができます。

```
(use-package reftex
  :ensure nil
  :hook (yatex-mode . reftex-mode)
  :bind (:map reftex-mode-map
              ("C-c (" . reftex-reference)
              ("C-c )" . nil)
              ("C-c >" . YaTeX-comment-region)
              ("C-c <" . YaTeX-uncomment-region))
  :defer t
  :custom
  ;; \ref ではなく \cref を使うための設定
  (reftex-ref-style-default-list '("Cleveref") "Use cref/Cref as default"))
```

6. biblio.el: 書誌情報の Web 検索機能

biblio.el[11] を使うことで書誌情報を Web 上で検索して、例えば BibTEX[12] のエントリーをダウンロードしてファイルに挿入することや `kill-ring` にコピーすることができます。それ以外にも (個人的にはほぼ使いませんが) 検索結果の画面をブラウザで開くこともできます。検索サイトとして arXiv、CrossRef、DBLP、HAL、IEEE Explore に対応している様ですが、個人的には専ら DBLP を使っています。

6.1. 個人的に良く使う使い方

以下では一度 `biblio-lookup` を呼ぶことで使いたい検索サイトの選択画面を表示させていますが、`M-x biblio-dblp-lookup` 等で直接検索サイトを指定することもできます。

(1) BibTEX のエントリーを挿入したい場所に移動する。

(2) `M-x biblio-lookup` で `biblio-lookup` を呼び、使いたい検索サイトを選択。

(3) 検索文字列を聞かれるので入力。

(4) 検索結果が表示されるので `i` または `I` で挿入する。

6.2. package.el によるインストール方法

biblio.el は特別な設定をしなくても動くので、個人的には use-package を使わずに単に package.el を使ってインストールしています。 例えば `M-x package-list-packages` から biblio を選択してインストールすれば大丈夫です。Emacs Lisp を使って自動でインストールする場合は以下の様になります。

```
(package-install 'biblio)
```

7. company-mode: 補完用マイナーモード

company-mode[13] は Emacs の補完用マイナーモードです。IDE によく付いてくる補完機能と同じ様なものだと思って差し支えないでしょう。 一昔前は auto-complte が良く使われていましたが、現在は company-mode の方が良く使われている様です。

プログラムを書く場合に補完するのは変数名や関数名ですが、LaTeX の文章を書く場合には主に英単語を補完したいです。英単語の補完方法には幾つかありますが、ここでは TabNine という人工知能による補完を `company-tabnine` を使います[*13]。

7.1. use-package による設定法

use-package による company-mode 及び company-tabnine の設定は以下の様になります。なお、 初回起動時には `M-x company-tabnine-install-binary` で TabNine のバイナリをインストールする必要があります。

```
(use-package company
  :ensure t
  :config
  (global-company-mode)
```

13 TabNine を使った補完以外にも、 より以前から使われてきたスペルチェッカーの辞書を使った補完 (company-ispell) も良く動きます。

```
  ;; 遅延なしにする。
  (setq company-idle-delay 0)
  ;; デフォルトは 4。より少ない文字数から補完が始まる様にする。
  (setq company-minimum-prefix-length 2)
  ;; 候補の一番下でさらに下に行こうとすると一番上に戻る。
  (setq company-selection-wrap-around t)
  ;; 番号を表示する。
  (setq company-show-numbers t)
  :bind (:map company-active-map
              ("C-n" . company-select-next)
              ("C-p" . company-select-previous)
              ("C-s" . company-filter-candidates)
              ("<tab>" . company-complete-selection))
  :bind (:map company-search-map
              ("C-n" . company-select-next)
              ("C-p" . company-select-previous)))
(use-package company-tabnine
  :ensure t
  :config
  (add-to-list 'company-backends #'company-tabnine))
```

8. Ispell: 対話的スペルチェック

Emacs では対話的なスペルチェック [14] を行うこともできます。 古典的には Ispell が使われていたために、Emacs では対話的スペルチェックのコマンドも Ispell と呼ばれていますが、現代では後継の Aspell や Hunspell が主流となっています。

8.1. Ispell の使い方

Ispell には幾つかの使い方がありますが、`M-x ispell` で `ispell` を呼ぶのが基本的な使い方です。ispell はスペルチェックの結果辞書にない単語を見つけると、置換候補を画面上方に表示するので、選択すると置き換えられます。無視したい場合は `<SPC>` を入力します。それ以外にも当該バッファのみの辞書に追加する等も可能なので、詳細な使い方は置換対象の選択画面で `C-h` か `?` を入力して確認してください。`M-x ispell` では基本的にバッファ全体に対してスペ

ルチェックを行いますが、リージョンがアクティブな場合はアクティブなリジョンに対してのみスペルチェックを行います。また、1 単語に対してスペルチェックを行いたい場合は M-$ を打ちます。

8.2. use-package による設定法

use-package による ispell の設定は以下の様になります。ispell は Emacs に同梱されているので別途インストールせずに使うことができます。ここではスペルチェッカとして aspell を使う設定をしているので、システムにインストールされていない場合は別途 aspell のインストールも必要です。

```
(use-package ispell
  :init
  ;; スペルチェッカとして aspell を使う
  (setq ispell-program-name "/usr/local/bin/aspell")
  :config
  ;; 日本語の部分を飛ばす
  (add-to-list 'ispell-skip-region-alist '("[^\000-\377]+")))
```

9. FlySpell: リアルタイムスペルチェッカ

Ispell では対話的なスペルチェックを行うことができますが、FlySpell[15] を使うことで他の多くのテキストエディタと同様にリアルタイムのスペルチェックも行うことができます。筆者は試したことがないですが、多分 ispell か aspell がシステムにインストールされていないと動かないと思います。

9.1. use-package による設定法

use-package による FlySpell の設定は以下の様になります。

```
(use-package flyspell
  ;; flyspell をインストールする
  :ensure t
```

```
;; YaTeX モードで flyspell を使う
:hook (yatex-mode . flyspell-mode))
```

10. Ace Jump Mode: カーソルジャンプ用マイナーモード

Ace Jump Mode [16] を使うことでバッファ中の「任意の場所」に一瞬でジャンプできる様になります。「任意の場所」というのは具体的には : 1) 単語の先頭の文字、2) 単語の先頭以外も含む文字、3) 行のどれかになります。これらを数タイプで指定することでジャンプできるため、カーソル移動の時間がかなり短かくなります。但し英文を書いている場合は結構本当に 2 ストロークで画面中のほぼ全ての場所に移動できますが、日本語を書いている場合は文字指定に難があるので、画面中のほぼ全ての「行」に飛べるくらいが本当のところになります。英文を書く場合には、単語の先頭以外も含む文字だと候補が多くなりすぎて必要なタイプ数が増えてしますので、単語の先頭の文字を指定するのが使い勝手が良いです。

10.1. 使い方

Ace Jump Mode では、以下のコマンドに続いて飛びたい先の文字を入力すると、その文字に指定するための文字が表示されるので、その文字を入力して位置を指定することでジャンプすることができます。なお行指定については最初から各行に文字が表示されるのでそれらの文字を入力することで行を指定することができます。

- 単語の先頭の文字 : C-c j [key]: (または M-x ace-jump-word-mode)
- 単語の先頭以外も含む文字 : C-u C-c j [key] : (または M-x ace-jump-char-mode)
- 行 : C-u C-u C-c j [key] : (または M-x ace-jump-line-mode)

前述の様に英語の文書を書く場合は C-c j の単語先頭の文字指定が重宝します。一方で日本語の場合文字指定が大変なので C-u C-u C-c j の行指定によるジャンプが重宝します。

10.2. use-package による設定法

use-package による Ace Jump Mode 設定は以下の様になります。ここでは C-c j で ace-jump-mode を呼び出す様に設定していますが、他のキーバインドが良い場合は適宜設定してく

ださい。

```
(use-package ace-jump-mode
  :ensure t
  :bind (("C-c j" . ace-jump-mode)))
```

11. Position Registers: 多分無名な標準機能

Emacs には標準の機能としてレジスター [17] があります。 レジスターを使うことで、 各文字 (数字や英字) に一つ、テキストやウインドウの設定などを一時的に保存できます。ここでは特にレジスターにバッファ上の位置を保存すること使い方を紹介します。

レジスターではバッファ上の位置を上手く記憶することができるので、例えば次の様な使い方ができます。

(1) LaTeX 文章を書いている途中でプリアンブルを編集したくなる

(2) 現在編集している場所をレジスターに保存する

(3) 適宜プリアンブルを編集する

(4) レジスターに保存されている場所に戻る

また、Emacs の Position Registers ではバッファ上の位置を保存して、別のバッファからでも移動することができます。そのため例えば書誌情報を編集するために .bib ファイルを編集した後でメインの文書ファイルの特定位置に戻る、といった使い方もできます。

なお、Position Registers は Emacs の標準機能なので特に設定を行う必要はありません。

11.1. Position Registers の使い方

Position register の使い方は以下の様にとてもシンプルです。

- `C-x r SPC [レジスター]` で指定したレジスターに現在位置を記録
- `C-x r j [レジスター]` でレジスターに記録されている位置にジャンプ

12. Flycheck: Emacs の構文チェッカ

Flycheck は Emacs の構文チェッカです。プログラムを書く際は型チェックを通したり lint に構文を確認させたりするために使いますが、ここでは chktex や lacheck など (La)TEX 用の lint の設定を紹介します。正直なところ Flycheck が無くてもあまり困ることはありませんが、たまに括弧対応などの有益な指摘をしてくれることがあります。一方で数式中で半開区間が表われた場合の様に意図して括弧を対応させていない場合にも lint が指摘するため、lint の結果の扱いには注意が必要です。

12.1. 使い方

以下の設定を行った場合、`M-n` や `M-p` で前後のエラー箇所にジャンプすることができます。また、エラー上にカーソルがある場合、エラーの説明文が表示されます。

- `M-n` (`flycheck-next-error`): 直後のエラー箇所にジャンプ
- `M-p` (`flycheck-previous-error`): 直前のエラー箇所にジャンプ

12.2. use-package による設定法

```
(use-package flycheck
  :ensure t
  :hook (after-init . global-flycheck-mode)
  :config
  (flycheck-add-mode 'tex-chktex 'yatex-mode)
  (flycheck-add-mode 'tex-lacheck 'yatex-mode)
  ;; chktex が自動で見付からない場合は以下の様に指定する。lacheck についても同様
  (setq flycheck-tex-chktex-executable "/Library/TeX/texbin/chktex")
  :bind (:map flycheck-mode-map
              ("M-n" . flycheck-next-error)
              ("M-p" . flycheck-previous-error)))
```

13. まとめ

いかがだったでしょうか !!

今回は GNU Emacs で LaTeX 文書を書くための設定の一例を紹介しました。今回の方法が最適解であるかはさておき、GNU Emacs を LaTeX 文書を書くために設定する際のたたき台になると幸いです。

参考文献

[1] TeX wiki contributors. *LaTeX 入門 - TeX Wiki*. https://texwiki.texjp.org/?LaTeX%E5%85%A5%E9%96%80, 2020.

[2] conao3. *2020 年代の Emacs 入門 | Emacs JP*. https://emacs-jp.github.io/tips/emacs-in-2020, 2020.

[3] syohex. *package: パッケージ管理ツール | Emacs JP*. https://emacs-jp.github.io/packages/package, 2020.

[4] John Wiegley. *jwiegley/use-package: A use-package declaration for simplifying your .emacs*. https://github.com/jwiegley/use-package, 2020.

[5] conao3. *conao3/leaf.el: Flexible, declarative, and modern init.el package configuration*. https://github.com/conao3/leaf.el, 2020.

[6] Hirose Yuuji. *Yet Another LaTeX mode for Emacs*. https://www.yatex.org/, 2019.

[7] Hirose Yuuji. *Info Node: (yatexj)Completion*. https://www.yatex.org/~yuuji/bin/info2www.cgi?(yatexj)Completion, 2019.

[8] Hirose Yuuji. *Info Node: (yatexj)Invocation*. https://www.yatex.org/~yuuji/bin/info2www.cgi?(yatexj)Invocation, 2019.

[9] AUCTeX project. *RefTeX - References, labels, citations*. https://www.gnu.org/software/auctex/reftex.html, 2013.

[10] Free Software Foundation. *RefTeX Manual - GNU Project - Free Software Foundation (FSF)*. https://www.gnu.org/software/auctex/manual/reftex.index.html, 2009.

[11] Clément Pit-Claudel. *cpitclaudel/biblio.el: Browse and import bibliographic references from CrossRef,*

DBLP, HAL, arXiv, Dissemin, and doi.org from Emacs. https://github.com/cpitclaudel/biblio.el, 2020.

[12] Alexander Feder. *BibTeX*. http://www.bibtex.org/, 2006.

[13] Dmitry Gutov, company-mode contributors. *company-mode for Emacs*. https://company-mode.github.io/, 2020.

[14] Ayanokoji Takesi. *GNU Emacs Manual(Japanese Translation): Spelling*. https://ayatakesi.github.io/emacs/24.5/Spelling.html, 2016.

[15] Emacs Wiki contributors. *[Home] Fly Spell*. https://www.emacswiki.org/emacs/FlySpell, 2019.

[16] winterTTr. *winterTTr/ace-jump-mode: a quick cursor jump mode for emacs*. https://github.com/winterTTr/ace-jump-mode, 2014.

[17] Ayanokoji Takesi. *GNU Emacs Manual(Japanese Translation): Registers*. https://ayatakesi.github.io/emacs/25.1/Registers.html, 2016.

[18] Sebastian Wiesner, Flycheck contributors. *Flycheck — Syntax checking for GNU Emacs — Flycheck 32-cvs documentation*. https://www.flycheck.org/en/latest/, 2017.

高速 SAT ソルバーを支える技術

wasabiz

1. はじめに

yabaitech.tokyo vol.6 をご覧の皆さんおはようございます。wasabiz です。

これまでの yabaitech.tokyo では定理証明支援系を自作したり (vol.2)、粘菌を愛でたりして (vol.3) 計算の本質に迫る試みを行ってきました。 言い換えるとこれまでは 「どういった問題が計算機で解けるのか」に注目していたわけですが、第六回となる今回は少し趣向を変えて「どうやって**速く**問題を解くか」 に注目したいと思います。 具体的には、 本記事では 「速い SAT ソルバー」を作る方法について解説していきます。

1.1. 実装について

この記事で紹介するコードは全て以下のリポジトリにまとまっています。

https://github.com/nyuichi/yabai-sat

節ごとの内容が 1 コミットに対応しており、各コミットの差分も必要最低限になるように注意深く書かれています。(めっちゃ歴史修正頑張りました。) この記事ではコードの全体像を追うことはせず核となるアルゴリズムの部分の解説しか行っていないので、実際に動くコードを確認しながら理解したい・記事に書かれていない細かい部分が分からなくて混乱するという場合にはぜひ上記リンク先と照らし合わせながら読み進めてみてください。

1.2. NP 完全問題と SAT ソルバー

SAT ソルバーと聞くと、大半の人は「NP 完全問題とかいう問題を解くソルバーらしい」とい

うぐらいの感想が思い浮かぶのではないでしょうか。この言明自体は間違っていないのですが、これが意味することがなんなのかを正確に理解するためには NP 完全問題とは何かを知る必要があります。では NP 完全問題とはなんなのでしょうか。

NP 完全問題とは何かを説明するためにはまず計算量理論の基本的な概念である P や NP といったクラスについての理解が必要になります。P とは多項式時間で解くことができる判定問題のクラスです。オーダーの記法を使えば入力のサイズ n に対して $O(1)$、$O(n)$、$O(n^3)$、$O(n \log n)$ などの時間で判定できる問題はいずれも P に属します。P に属する問題は計算量理論においては「最も基本的」で「簡単な」問題だと考えられてます。ただしこれは実用上簡単に解けると言う意味ではなく、例えば $O(n^{10000})$ のように指数の肩が大きい計算量であっても P に属する問題です。このような問題を簡単と言われてしまうとやや驚きますが、世の中にはこれらより遥かに難しい問題がたくさんあるのでそれらより相対的に難しいと言う意味で簡単であるとみなされています。

P にさらに難しい問題を付け加えたクラスの一つが NP です。NP とは判定問題の答えが yes であることの「証拠」が与えられたときにその証拠が正しいかを多項式時間で判定できる問題のクラスです。少しややこしいですが、平易に言い換えれば「自力で正解を見つけるのは難しい (かもしれない) が、ある答案が正しいか間違っているかは簡単にわかる」ような問題のクラスです。例えばぷよぷよに関する以下の問題は NP だそうです (実は NP 完全)。[松金 & 武永 05]

> 縦と横に十分広いぷよぷよを考える。ぷよの初期配置 B とこれから降ってくるぷよの列 P、自然数 k が与えられたとき、このぷよぷよをプレイして k 連鎖できるか？

この問題における「証拠」はぷよの操作の列です。操作の列が与えられれば実際にそれをプレイしてみるだけで本当に k 連鎖できるかどうかがすぐわかります。

ここで P、NP いずれの場合も正式な定義はチューリング機械を用いることに注意してください。通常の直観ではついランダムアクセスメモリやランダムアクセスストレージがついた普通の計算機で考えてしまいますがそれは不正確です。入力のサイズもビット数を表します。今あげたぷよぷよの問題が NP 問題であるという定理も、証明を全て展開すれば最終的にはチューリングマシン上でぷよぷよを実行しています。

数ある NP 問題の中でも普遍的なものを NP 完全問題といいます。どういうことかというと、「それさえ解ければ他のどんな NP 問題も簡単に解ける」ような NP 問題を NP 完全であると呼

ぶという意味です。(正式には NP に属していてかつ、 任意の NP 問題を多項式帰着できるような問題のことです。) つまり、 何らかの NP 完全問題とそのソルバーがあればあらゆる NP 問題が「簡単に」解けることになります。そしてそのような問題の一つが CNF-SAT です。

CNF-SAT (より短く SAT) は以下のような問題です。いくつか単語を定義します。

- 真 ($\top$) か偽 ($\bot$) かのどちらかが割り当てられる変数を論理変数と言う。
- 論理変数 (V) か論理変数に否定をつけたもの ($\neg V$) をリテラルと言う。
- 0 個以上のリテラルを有限個「または」で繋いだものを節と呼ぶ。(例 : $(V_1 \vee \neg V_2 \vee V_3)$)
- 0 個以上の節を有限個「かつ」で繋いだものを CNF と呼ぶ。(例 : $(V_1 \vee \neg V_2) \wedge (\neg V_1 \vee V_3 \vee V_4)$)

SAT とは CNF が一つ与えられた時に、 各論理変数に真か偽の割り当てることによって CNF 全体を真にする (充足する) ことができるか？という問題です。

> CNF Δ が与えられたとき、Δ を充足させる割り当ては存在するか？

非常に有名な事実として SAT は NP 完全問題です。 よって、SAT の高速なソルバーを作ることであらゆる NP 問題を高速に解くための手段が得られることになります。

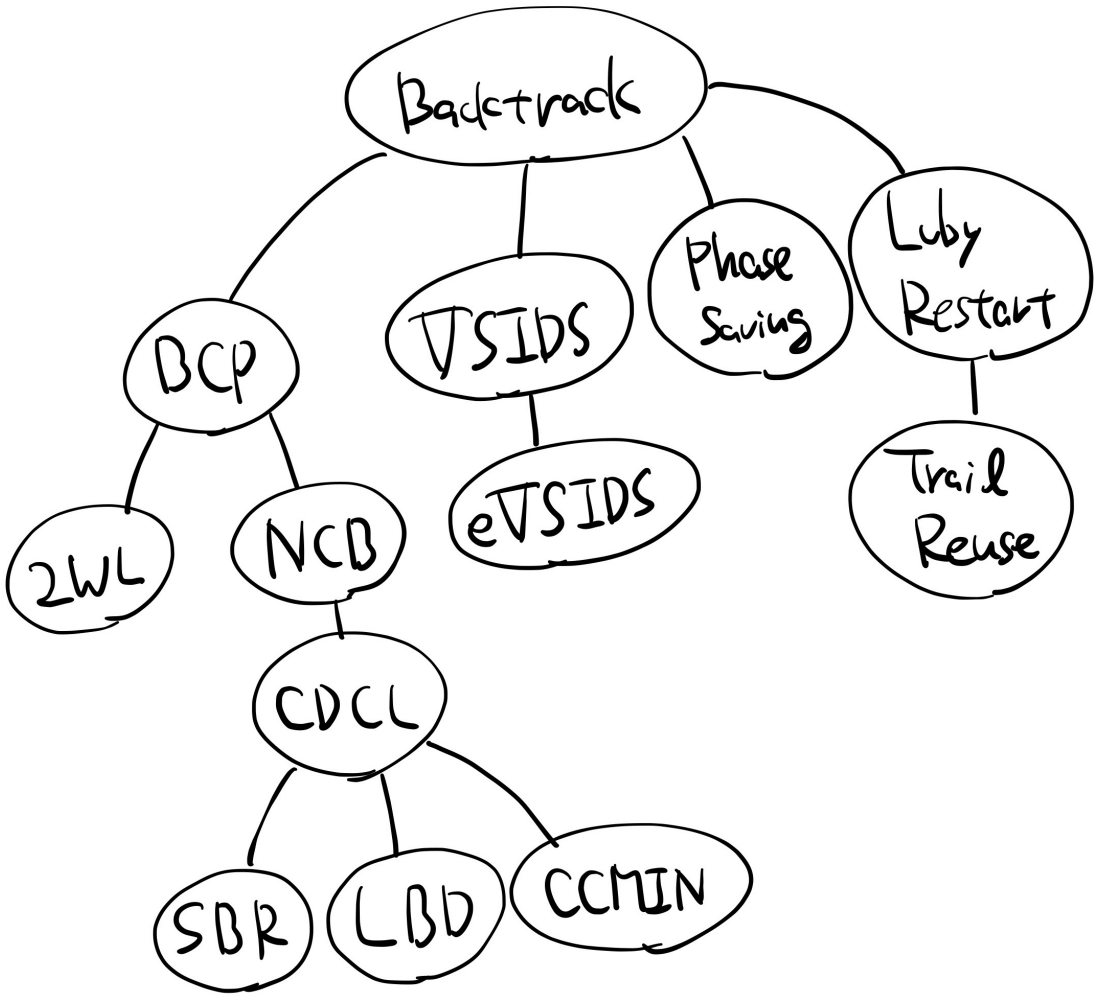
1.3. SAT ソルバーを作ろう！

解くべき問題が明らかになったところで、さっそく実際に SAT ソルバーを作っていくことにしましょう！

この記事の残りの部分では SAT ソルバーを高速化する際に必要な要素技術について順を追って解説していきます。現代の高速な (いわゆる state-of-the-art の)SAT ソルバーというのは一朝一夕で出てきた突拍子もない技術によって成り立っているわけではありません。 古くは 1960 年ごろから続く地道な改良によって成り立っています。 この記事ではその歴史に倣い、ナイーブなアルゴリズムを徐々に改良していくことで現代的な SAT ソルバーを構築していきます。

SAT ソルバーの高速化については毎年たくさんの論文が出ており、多くのテクニックが提案されています。この記事ではそれらの中でも概ね評価が安定し、実績があり、理論的にも性能向上が保障され、多くのソルバーで採用されているような技術に絞って解説を行います。各技術は独立性が高いのでそれぞれだけを実装することも可能ですが、解説の都合上、他の技術を実装済みのコードベースの上にさらに新しい技術を実装していく流れにしています。 これから解説

する各技術の間の依存関係は以下の通りです。



解説に移る前に一点コメントを残しておきます。この記事で紹介するのはいわゆる**系統的ソルバー (systematic solver)** と呼ばれるものです。このタイプのソルバーは入力の CNF が充足可能であればその割り当てを返し、充足不能 (充足する割り当てが存在しない) であればその事実を報告します。他にも**確率的ソルバー (stochastic solver)** と呼ばれるタイプのソルバーも存在します。このタイプのソルバーは確率的な探索を行うため問題が充足可能な場合にはその割り当てを返すことはできますが、充足不能だった場合にはそれを判定することができません。確率的ソルバーも近年は研究が進み高速なものが登場しているようですが、今回の記事ではそれらについては全く扱いません。ただし、近頃は系統的ソルバーに確率的ソルバーの技術を取

り入れたソルバーも登場してきているそうなので、系統的ソルバーを究極まで高速化するためには確率的ソルバーの技術を避けては通れないでしょう。

2. ナイーブなアルゴリズム

最も簡単な SAT ソルバーのつくり方は全探索です。与えられる CNF の大きさが有限なので論理変数への真偽の割り当てを全通り調べる方法で解を求めることができます。この方法では変数の数 n に対して 2^n 回の反復が必要になってしまうので、例えば 100 変数の問題を解くのは絶望的です。

これは全探索というアルゴリズムが悪いのではなく、NP 完全問題自体がそもそも最悪ケースでは絶対にこれぐらいの計算量がかかってしまうものなのが原因です。残念ながら、理論的な観点から言えばこの指数アルゴリズムを改良するような手法は存在しません。(もし P=NP とかだったりするとまた別です。)

しかし、これはあくまで最悪ケースの話です。現実的な入力の場合はほとんどの場合最悪ケースとは違って、2^n の広さの探索空間全てを探索することなく充足可能性を判定することができます。

例えば、CNF の中に一つのリテラルだけからなる節を見つけることができれば、それだけで探索空間が半分になります。以下の CNF は解を持つとすればその解の変数 V_1 の割り当ては必ず偽になるはずです。(そうでないと 2 個目の節が充足されない。) そのため 2^n 個の割り当てのうち V_1 を真とするような 2^{n-1} 個の割り当ては調べる必要がないということがすぐにわかります。

$$(V_1 \vee \neg V_2) \wedge (\neg V_1) \wedge (\neg V_2 \vee V_3 \vee \neg V_4)$$

あるいは、入力の CNF の全ての節が高々二つのリテラルで構成されている場合その問題は入力に対して線形時間で解くことができます。(そのような問題を 2SAT と呼びます。)

このようにして、2^n 個の解の候補のうち絶対に解にならない割り当てを探索する前に判断する (つまり枝刈りする) ことで、非常に大きな問題であっても現実的な入力であれば (典型的には) 高速に解くことができます。以降の節では、そのような最適化を少しずつ実装していきます。

3. バックトラック

ここでは最も基本的な最適化であるバックトラックを実装します。今バックトラックを「最も基本的な最適化」と表現しましたが、これは少し不思議な表現です。一般的には「バックトラック」と言う単語は単なる反復の実装方法を指すからです。本当のことをいうと、今から実装するバックトラックは通常の意味のバックトラックの上にある種の枝刈りを実装したものです。その具体的な枝刈りとは「割り当てを途中まで決めた段階で矛盾が見つかったら、その割り当てをやめて次の割り当てを探索する」と言うものです。

例えば $(L_1 \vee L_2) \wedge (L_3 \vee L_4)$ と言う問題を考えます。(L_1 から L_4 は何らかのリテラルを表しています。) ここでバックトラック中に L_1 と L_2 に共に偽が割り当てられ L_3 と L_4 は未割り当てだったとしましょう。(リテラル L を割り当てる、というのはここでは L を真にするように変数を割り当てる、ということです。例えば $L = \neg V$ なら、V に偽を割り当てることを指します。) この時点で一つ目の節である $(L_1 \vee L_2)$ が偽に評価されるので L_3 と L_4 の割り当てを決めずとも今の割り当てでは解が見つからないことがわかります。そこで全ての割り当てを決める前に現在の探索空間を捨てて他の空間 (L_1 と L_2 に対する他の割り当て) に移る、というのが今回実装する枝刈りです。

この節で紹介するソルバーはアルゴリズムとしては以上で終わりです。つまり、単なるバックトラックに単純な枝刈りを加える以外は何もしません。

この方法がうまくいくためには反復のたびに矛盾を見つけるコストよりもそれによって探索空間を削減できるメリットの方が大きい必要があります。これについてはもちろん問題にもよるのですが、実際に動かしてみると**極めて良い性能を示します**。この節では矛盾を見つける処理を非常にナイーブに実装しますが、そのような実装であってもなお非常に高速にしてくれます。

さて、アルゴリズムについてはこれで終わりですが、この節のもう一つのポイントは実装テクニックです。この節で紹介するソルバーの実装はかなり最適化されており、実装テクニックだけでもいくつものアイデアが詰め込まれています。また、コード全体の構造は記事全体を通してほとんど変わらず、性能評価の点でもここでの実装は今後の全ての最適化のベースラインとなるものです。これから記事を読み進めていくにはまずはこの節を理解しておくことが必要でしょう。

3.1. 実装

それでは具体的なコードを見ていきます。まず先に、大まかな構造を把握するため探索のメインループを見ます。

探索の本体は以下のコードです。

```
while (1) {
    while (find_conflict()) {
        if (decision_level == 0)
            return false;
        backtrack();
    }
    if (! decide())
        return true;
}
```

ソルバーはメインループの一番初めに `find_conflict()` を呼んでいますが、これはソルバーの入力として与えられた CNF の中に**空節 (empty clause)** があるかどうかを調べる関数です。ただしここで空節と呼んでいるものは「現在の割り当ての下で全てのリテラルが偽に評価される節」のことです。節が空ではないのに空節と呼ぶのは変な気がしますが、これは数学的には全てのリテラルが偽に評価される節と空節が同じだとみなされるからです。例えば $V_1 = \bot, V_2 = \top, V_3 = \bot$ という割り当ての下では $(V_1 \vee \neg V_2 \vee V_3)$ という節は $(\bot \vee \neg \top \vee \bot)$ と同じであり、節の評価が偽になります。一方で空節 () も同じく評価が偽になるため、この二つは同一視されます。

ところで、空節というのは残る未割り当ての変数にどのような割り当てを行っても絶対に充足できない節です。そのため、探索中に見つけた空節を「今探索中の割り当てが正しくない割り当てだった」という意味で**矛盾 (conflict)** と呼ぶことがあります。

メインループの話題に戻りましょう。`find_conflict()` の呼び出しで空節が見つからなかった場合の処理は簡単で、そのまま未割り当ての変数から新たに適当なものを選んで適当な割り当てを行い (`decide()`)、探索を続けます。このように任意に新たな割り当てを一つ選ぶ操作を**決定 (decision)** と言います。

さて、問題は `find_conflict()` で空節が見つかった場合です。空節が見つかったということは現在の割り当てが問題の解になっていない / なり得ないということを意味します。もしなんらかの決定を経て今の割り当てに至っている場合はその決定が間違っていたということになります。その場合は、一番最後に行った決定まで探索を巻き戻して他の選択肢を試します。(`backtrack()`) あとで再度解説しますが、この SAT ソルバーの `backtrack()` は他の選択肢を試す際に、最後の決定の割り当ての真偽を単に反転させます。

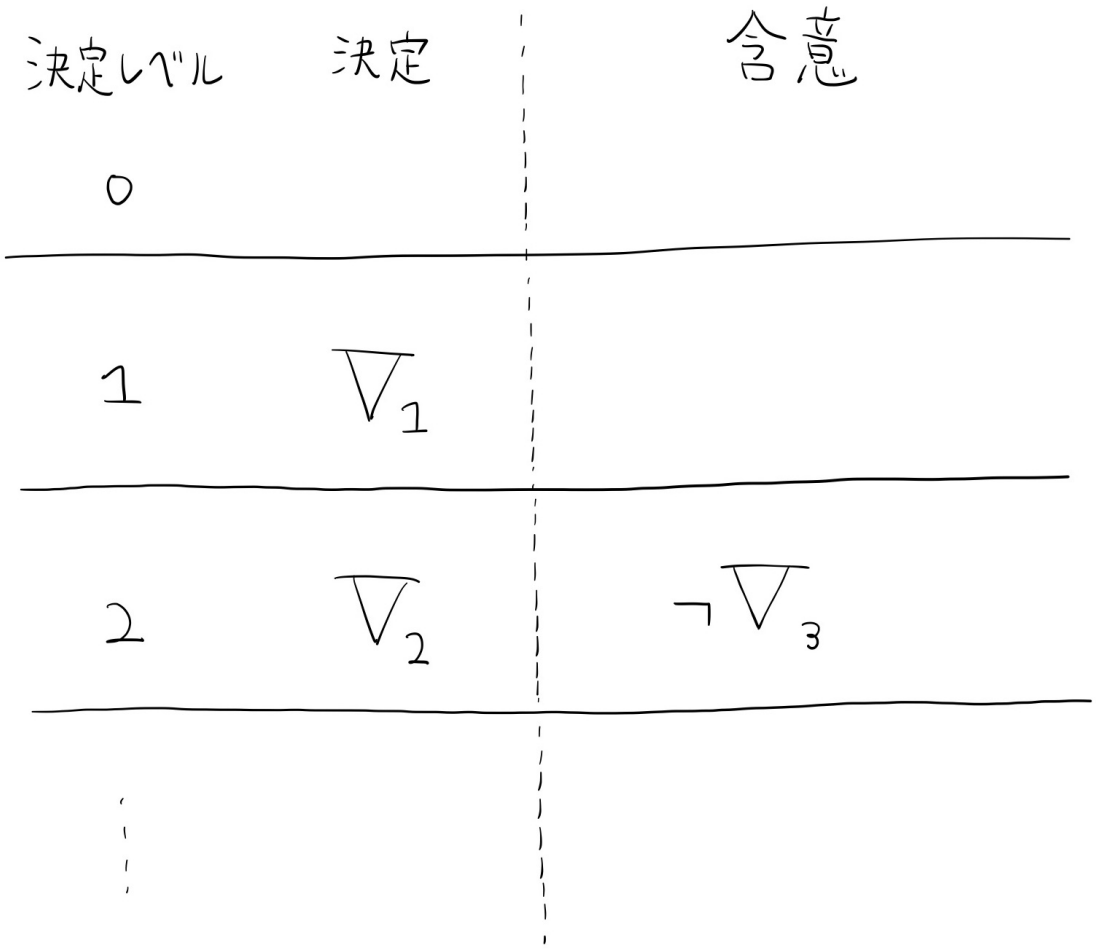
具体例で考えてみます。一回目の決定で $V_1 = \top$, 二回目の決定で $V_2 = \top$ を割り当て、三回目の決定で $V_3 = \top$ を割り当てたところ空節が見つかったとしましょう。ここで空節が見つかったということは $V_1 = \top$ かつ $V_2 = \top$ ならば $V_3 = \bot$ でなければならないということを意味します。そこで、この実装では $V_3 = \top$ という決定を巻き戻し、真偽を反転させて $V_3 = \bot$ を割り当てます。

この例での最後の割り当て ($V_3 = \bot$) はそれ以前の二回の決定 (V_1 と V_2 への割り当て) から導かれる論理的帰結だったことに注目してください。つまり $V_3 = \bot$ という割り当ては決定による割り当て (ソルバーが好きに選んだ割り当て) ではなく他の割り当てからの含意による割り当てであると言えます。このように、変数の割り当ては「決定によって割り当てられたもの」と「含意によって割り当てられたもの」の二種類に分類することができます。

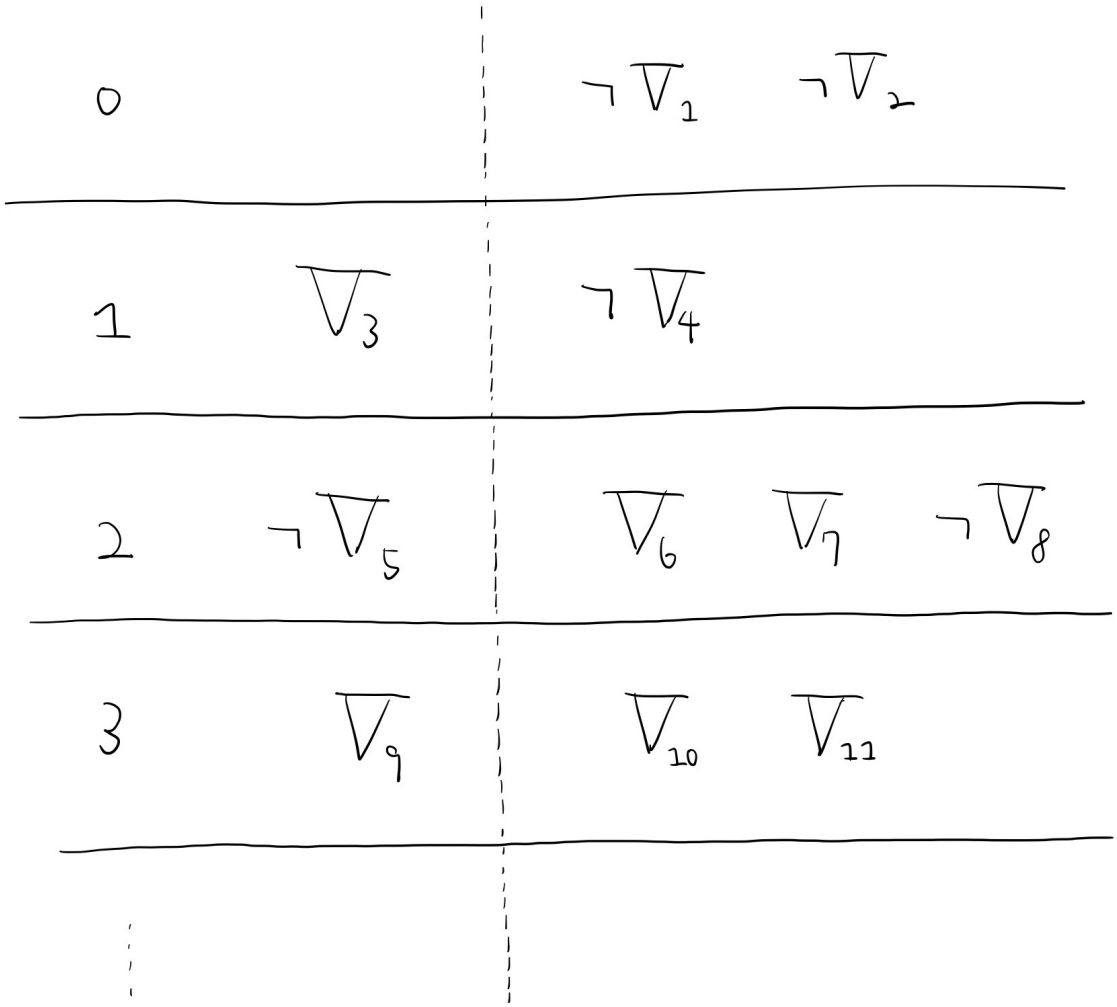
この二種類の区別は非常に重要なので、後々のために言葉を整理します。変数 V が割り当て済みだとします。その割り当てが決定による割り当ての時、その変数を**決定変数 (decided variable)** と言い、そうでない場合 (つまり、他から論理的に導かれた場合) その変数を**含意変数 (implied variable)** と言います。また、V が決定変数の時、その割り当て自体を**決定 (decision)** と呼び、V が含意変数の時、その割り当て自体を**含意 (implication)** と呼びます。同様に、リテラル L が割り当て済み (つまり L を構成する変数 V が割り当て済み) でかつ L が真に評価される時、V が決定変数なら L を**決定リテラル (decided literal)** と呼び、V が含意変数なら L を**含意リテラル (implied literal)** と呼びます。

決定と含意の区別は「決定レベル」という言葉を用いるとさらにわかりやすくなります。探索中のある時点において、その時の決定変数の数を**決定レベル (decision level)** と呼びます。また、割り当て済みの変数 V について、V の割り当てが得られた時の決定レベルを V の決定レベルと呼びます。例えば、初めての決定によって $V = \top$ が割り当てられた場合、V の決定レベルは 1 です。(0 ではないのがポイント。)

先程の例を決定レベルを用いて図示すると以下のようになります。決定レベル 1 で $V_1 = \top$ が、決定レベル 2 で $V_2 = \top$ が決定され、その含意として $V_3 = \bot$ が割り当てられています。(変数 V_3 に偽を割り当てることはリテラル $\neg V_3$ を割り当てることと同じ (というか「リテラルを割り当てる」の意味をそう定義した) なので、図ではリテラルを書いて割り当てを表現しています。)



これは小さな例を図示しましたが、一般的には一つの決定が複数の含意を導くことがあります。例えば、以下の場合では V_1 から V_7 までの割り当てによって V_8 が含意されていることを意味します。加えて、図にあるように決定レベル 0 の含意が生まれる可能性もあります。例えば極端なケースとして、与えられた CNF の中に $(\neg V_1)$ という節が存在すれば、ソルバーが決定が一切行わない状態でも論理的帰結として $V_1 = \bot$ が得られます。



このように、探索が進み空節が導かれるに従って含意による割り当てがどんどん増えていき、それによりさらに空節が見つかりやすくなる、ということが繰り返されていきます。そして、割り当て方が 1 通りしかあり得なくなった変数は決定レベル 0 の含意として登録されていきます。決定レベル 0 の含意が増えていけばそれにより空節が見つかる可能性も増えていきます。そして、もし決定レベル 0 の含意だけから空節が導かれた場合、それはもう他の選択肢がない、つまり問題自体が充足不能であった、ということを意味します。

大まかな構造を説明したところでより細かい部分に焦点を当てていきます。

まずは必要なデータ構造についてです。

変数は 1 から N までの数字で表されています。0 は変数を表しません。 リテラルは変数に符号をつけたもので、ここでは否定がついたリテラルを負数で表しています。一般に、L をリテラルとした時 L の**補リテラル (complement)** $\overline{L}$ は以下で定義されます。

$$\overline{L} = \begin{cases} \neg V & (L = V) \\ V & (L = \neg V) \end{cases}$$

コード上で `lit` の補リテラルを得るには単に `-lit` とします。

ソルバーのソースコードの先頭には `N`、`M`、`F` というグローバル変数が用意されていて、`N` が変数の数、`M` が節の数、`F` が問題を表す CNF です。(formula の頭文字のつもり。) これらは一度初期化された後は変更されることはありません。 このように、ソルバー全体を通してグローバル変数を多用しています。(いわゆる競プロ的なコード。)

ちなみに、`uint` は `unsigned` のことです。後で `uchar` というのも出てきますが `unsigned char` のことです。

```
uint N; // number of variables
uint M; // number of initial clauses
vector<vector<int>> F; // problem
```

SAT ソルバーの実装方針には、新たな割り当てが発生するたびに CNF をどんどん書き換えていく方法と、CNF を固定したまま割り当ての情報だけを更新していく方法の二種類があります。教科書的な方法は前者ですが、速度を求める場合は後者を採用します。 よって今回も後者の方法で実装します。 今回作ったソースコードでは `model` というグローバル変数が現在の割り当てを表します。`model` は長さが `N+1` のベクタで、各要素はビットフラグになっています。(0 番目の要素 `model[0]` は使われません。) そのフラグの内容は `MODEL_DEFINED` がその変数が割り当て済みであることを表していて、`MODEL_PHASE` は変数が割り当て済みであるときに真が割り当てられていることを表します。

`defined` と `phase` は `model` から変数の状態を取り出す関数です。 関数 `ev` は今の割り当てによって変数がどう評価されるかをリテラルの形で返します。 変数 `v` に真に割り当てられていれば `ev(v)` は `v` に、 偽に割り当てられていれば `-v` に、 未割り当てなら `0` になります。 こうするとリテラル `lit` について `ev(abs(lit)) == lit` と書くことで `lit` が真に評価されるか

どうかを確認できるようになるので便利です。(他のソルバーでこのやり方を採用しているのをみたことがないのですがめっちゃ便利なのでもっと広まればいいなと思ってます。)

```
enum {
    MODEL_DEFINED = 1,
    MODEL_PHASE = 2,
};
vector<uchar> model;

bool defined(uint var) {
    return (model[var] & MODEL_DEFINED) != 0;
}
bool phase(uint var) {
    return (model[var] & MODEL_PHASE) != 0;
}
int ev(uint var) {
    return ! defined(var) ? 0 : phase(var) ? (int) var : -(int) var;
}
```

探索に使うデータは以下の通りです。バックトラックの過程でどう言う経路を辿ってきたかを `trail` というスタックに格納しています。`trail` は先程の決定レベルの図の中のリテラルを一列に並べたものだと思ってください。なので、割り当てを行うたびに伸び、バックトラックで割り当てを未割り当てに戻すたびに縮みます。`decision_level` は現在の決定レベルを表します。

`push()` と `pop()` はそれぞれ割り当てを行ったり、割り当てを巻き戻すための関数です。それぞれ `trail` を伸ばしたり縮めたりします。

`push(lit)` はリテラル `lit` を割り当てます。(解説パートですでにちらっと言いましたが、「リテラル L を割り当てる」というのは L が真に評価されるように変数の割り当てを行うという意味です。) 例えば `push(-42)` を実行すると変数 42 に偽が割り当てられます。`pop()` は一回呼ばれるごとに一番最後の割り当てを一つ巻き戻します。

バックトラックを行うときは `pop()` を呼び出して割り当てを巻き戻すことを繰り返します。このとき直近の決定までたどり着いたら処理を終了する必要があります。これを実現するため、

trail の中のどれが決定だったかを判定できるように決定を行うたびに trail には 0 を積むことにしています。(すぐ後に出てくる decide() と backtrack() の実装を参照。)なので、trail のなかの 0 の数と decision_level は一致します。

```
vector<int> trail; // 0 for decision mark
int decision_level;

void push(int lit) {
    uint var = abs(lit);
    model[var] = lit > 0 ? MODEL_DEFINED | MODEL_PHASE : MODEL_DEFINED;
    trail.push_back(lit);
}
int pop() {
    int lit = trail.back();
    uint var = abs(lit);
    model[var] &= ~MODEL_DEFINED;
    trail.pop_back();
    return lit;
}
```

空節を探す find_conflict() は単に全ての節についてループを回しています。ここは特に面白いことはありません。

```
bool find_conflict() {
    for (auto & c : F) {
        int num_undef = 0;
        for (auto lit : c) {
            if (! defined(abs(lit))) {
                ++num_undef;
            } else if (ev(abs(lit)) == lit) {
                goto next;
            }
        }
        if (num_undef == 0)
            return true; // conflict found
```

```
    next:;
    }
    return false; // no conflict found
}
```

backtrack() は先程の解説通りに実装されています。直近の決定まで trail を巻き戻したあと、割り当ての真偽値を反転させています。一番最後の push() は含意のため、決定レベルは増加されません。

```
void backtrack() {
    int lit = 0;
    for (uint i = trail.size() - 1; trail[i] != 0; --i)
        lit = pop();
    trail.pop_back(); // remove the mark
    --decision_level;
    push(-lit); // flip the decision
}
```

決定をおこなう decide() もここまでの解説を読めば特に難しくはないでしょう。 適当な変数とその真偽を選び(choose())、trail に決定を行ったことを記録します。決定を行う際は trail に 0 を積んでそれを記録し、決定レベルを一つ増やします。もし選ぶべき変数がなければ、 現在の割り当てが解になっているということを意味します。 この段階では変数選択(choose())は非常にナイーブな実装になっていて、線形に未割り当ての変数を探索した上で真と偽のうち常に真を選ぶようになっています。もし真の割り当てが間違っていたことがわかるとバックトラックの中で決定をひっくり返して再度探索が行われるのでそのうち偽の割り当ても試されることになります。

```
int choose() {
    for (uint v = 1; v <= N; ++v) {
        if (! defined(v))
            return (int) v;
    }
    return 0;
```

```
}

int decide() {
    int lit;
    if ((lit = choose()) == 0)
        return false; // sat
    trail.push_back(0); // push mark
    ++decision_level;
    push(lit);
    return true;
}
```

コード全体は以下の通りです。実はこの段階でも適当に作ったソルバーと比べて遥かに（本当に遥かに！）速いのですが、これをさらにさまざまなテクニックを使って高速化していきます。

```
uint N; // number of variables
uint M; // number of initial clauses
vector<vector<int>> F; // problem

enum {
    MODEL_DEFINED = 1,
    MODEL_PHASE = 2,
};
vector<uchar> model;
vector<int> trail; // 0 for decision mark
uint decision_level;

bool defined(uint var) {
    return (model[var] & MODEL_DEFINED) != 0;
}
bool phase(uint var) {
    return (model[var] & MODEL_PHASE) != 0;
}
int ev(uint var) {
```

```
    return ! defined(var) ? 0 : phase(var) ? (int) var : -(int) var;
}

void push(int lit) {
    uint var = abs(lit);
    model[var] = lit > 0 ? MODEL_DEFINED | MODEL_PHASE : MODEL_DEFINED;
    trail.push_back(lit);
}
int pop() {
    int lit = trail.back();
    uint var = abs(lit);
    model[var] &= ~MODEL_DEFINED;
    trail.pop_back();
    return lit;
}

void backtrack() {
    int lit = 0;
    for (uint i = trail.size() - 1; trail[i] != 0; --i)
        lit = pop();
    trail.pop_back(); // remove the mark
    --decision_level;
    push(-lit); // flip the decision
}

bool find_conflict() {
    for (auto & c : F) {
        int num_undef = 0;
        for (auto lit : c) {
            if (! defined(abs(lit))) {
                ++num_undef;
            } else if (ev(abs(lit)) == lit) {
                goto next;
            }
        }
        if (num_undef == 0)
```

```
            return true; // conflict found
    next:;
    }
    return false; // no conflict found
}

int choose() {
    for (uint v = 1; v <= N; ++v) {
        if (! defined(v))
            return (int) v;
    }
    return 0;
}

int decide() {
    int lit;
    if ((lit = choose()) == 0)
        return false; // sat
    trail.push_back(0); // push mark
    ++decision_level;
    push(lit);
    return true;
}

bool solve() {
    model.resize(N + 1);
    trail.reserve(2 * N);
    decision_level = 0;

    while (1) {
        while (find_conflict()) {
            if (decision_level == 0)
                return false;
            backtrack();
        }
        if (! decide())
```

```
            return true;
        }
    }
```

4. Boolean Constraint Propagation

記事の冒頭でも言及したように、 現代の高速な SAT ソルバーたちはどれも基本的な構造が同じです。 と言うのも、いずれも 1960 年代に開発されたアルゴリズムをベースに改良を加えていったものだからです。 そのアルゴリズムというのが **DPLL アルゴリズム**と呼ばれるものです。

DPLL アルゴリズムは以下の三つの規則を節集合に対して適用して解を求めるアルゴリズムです。

(1) 単位伝搬 (unit propagation) (Boolean Constraint Propagation (BCP))

(2) 純リテラル除去 (pure literal elimination)

(3) 分割規則 (splitting rule)

このうち 2 の純リテラル除去は現代的には採用されないことが多いです。(それでもアルゴリズムの完全性は失われません。) そして 3 は実は先程のバックトラックの節で説明したアルゴリズムと同じものです。残る 1 がこの節での主題です。

それでは単位伝搬 (あるいは BCP) がどのような処理なのかを説明します。 とはいっても単位伝搬の考え方は非常に簡単です。 単位伝搬ではまず一つのリテラルだけからなる節を見つけます。 そのような節を**単位節 (unit clause)** と呼びます。 実装上は、 一つのリテラルが未割り当てで残りのリテラルが全て偽に評価されるような節のことだと考えて下さい。(2 個以上のリテラルを持つ節を単位節と呼ぶのは不思議な感じもしますが、 空節が全てのリテラルが偽に評価される節だったことを思い出せば違和感はないでしょう。) その節が仮に (V_1) だったします。 この場合、 その節を充足させるには V_1 に真を割り当てるしかありません。 一方でその節が $(\neg V_1)$ だったとすると、その節を充足させるには V_1 に偽を割り当てるしかありません。なのでどちらにしても含意が一つ得られることになります。 この観察に基づき、 バックトラック中に単位節を見つけ、 それを構成する変数の割り当てを確定させると言うことを繰り返せば探索空間が小さくなることがわかります。 これが単位伝搬です。

先程の決定レベルを用いた図で言うと、単位伝搬は含意変数の獲得を促進する処理だといえます。単位伝搬がない実装だと、`trail` に含意変数が積まれるためには一度その変数の否定が決定されてそこから矛盾が導かれる必要がありました。一方、単位伝搬があるとその矛盾を導く処理をショートカットして直接新たな含意を獲得することができます。

4.1. 実装

先程のソルバーからの変更点だけを示します。というのも `find_conflict()` が変わるだけで残りの部分は同じだからです。

基本的な構造はこれまでと同じく、単に節全体をループで回して矛盾を探しているのですが、その際同時に単位節を探すように変更しています。単位節が見つかればそれを伝搬させます。単位節を伝搬させたことによりさらに新しい単位節が見つかることがあるので、単位節が見つかるたびに関数全体を再度実行して単位伝搬を繰り返すようにしています。

```
bool find_conflict() {
    bool retry;
    do {
        retry = false;
        for (auto & c : F) {
            int num_undef = 0;
            int undef_lit;
            for (auto lit : c) {
                if (! defined(abs(lit))) {
                    ++num_undef;
                    undef_lit = lit;
                } else if (ev(abs(lit)) == lit) {
                    goto next;
                }
            }
            if (num_undef == 0)
                return true; // conflict found
            if (num_undef == 1) {
                push(undef_lit);
                retry = true;
            }
```

```
            next:;
            }
        } while (retry);
        return false; // no conflict found
    }
```

5. Two Watched Literals

ここまで実装してきたようなバックトラック + BCP をベースとしたソルバーを DPLL 型のソルバーと呼ぶのですが、 この種のソルバーにおいて最も重要な操作が `find_conflict()` です。`find_conflict()` は空節と単位節の発見の両方を担っており、ソルバーの心臓部とも言えます。 しかしこの最も重要な部分がまだナイーブなループによる実装になっています。 ここをある種の遅延データ構造によって高速化する方法が知られており、**Two Watched Literals (2WL)** などと呼ばれています。日本語では単に**監視リテラル**と呼ぶことが多いようです。(ちなみに空節を見つけることだけに特化した **One Watched Literal (1WL)** という手法もあります。)

2WL の考え方を理解するために `find_conflict()` について思いを馳せてみます。 以下では話を簡単にするために全ての節は長さが 2 以上であるとしましょう。また、一つの節の中に同じ変数は二回現れないものとします。さて、今から最適化しようとしている `find_conflict()` 内のループは単位節と空節を探すためのものでした。もし絶対に単位節や空節になり得ない節をこの対象から除くことができれば `find_conflict()` を高速化することができます。そのために以下の事実を使います。

> 節 C が単位節または空節であるためには C は偽でないリテラルを高々一つしか持たないことが必要。

偽でないリテラルとは真に評価されるかあるいは未割り当てのリテラルを意味します。 つまり、 この主張にある「偽でないリテラルを高々一つしか持たない」 節というのは要するに空節か、 単位節か、 一つのリテラルが真で残り全てが偽であるような節です。 なので、 この主張が成立するのは自明です。

この事実から、以下のどちらかの条件を満たす節は探索の対象外としても関数の挙動が変わらないことがわかります。

(1) (WI-1) 充足されている

(2) (WI-2) 偽でないリテラルを少なくとも二つ持つ。

この二つ条件を合わせて **Watching Invariant (WI)** と呼びます。

実は `find_conflict()` が `false` を返した直後では全ての節が WI を満たします。 なぜなら単位節も空節も存在しないからです。つまり、ソルバーのメインループは WI の観点から見れば

(1) `find_conflict()` の結果全ての節が WI を満たすようになる。

(2) `decide()` で変数を割り当てる。どれかの節の WI が壊れる。

(3) `find_conflict()` で全ての節が WI を満たすまで伝搬を行う。

(4) `decide()` で変数を割り当てる。どれかの節の WI が壊れる。

(5) ...

という構造になっていることがわかります。(「WI が壊れる」と書いた部分では実際には WI が壊れないこともあります。) ここでは `find_conflict()` が `false` を返す場合のみを考えましたが、`find_conflict()` が `true` を返す場合もほぼ同じで、バックトラックによって最後の決定を巻き戻した瞬間では全ての節が WI を満たしています。その直後に `backtrack()` は最後の決定の真偽を反転させたものを `trail` に積むので、この瞬間にどこかの節の WI が壊れます。(壊れないこともあります。)

先程の議論から、`find_conflict()` は WI が壊れた節だけを探索すれば全ての単位節・空節を探すことができることが分かっているので、「割り当てによって WI が壊れたかもしれない節」だけを調べる方法があれば `find_conflict()` を高速化することができます。

これを実現するために、各リテラルに対して「それが割り当てられた時に WI が壊れるかもしれない節の集合」 を紐づけたいです。 これを**監視リスト (watch list)** と言います。 ただし、テクニカルというか、 歴史的な事情で、 リテラル L の監視リストというと、 リテラル $\overline{L}$ が割り当てられたときに WI が壊れるかもしれない節の集合を指します。 リテラル L の監視リストを watchlist(L) と書くことにします。

監視リストがまともに使えるためには探索の過程で以下の不変条件が守られている必要があります。

- (I-WLIST) L を割り当てた結果 WI が壊れるかもしれない節は全て watchlist($\overline{L}$) に入っている

問題はこの不変条件をどのようにして保ち続けるかです。そこで、監視リストの定義を以下のようにします。

(1) 各節に対してその中の異なる二つのリテラルを選ぶ。それらを**監視リテラル (watched literal)** と呼ぶ。節 c の監視リテラルを $W_0(c), W_1(c)$ と書くことにする。

(2) 監視リストは次のように定義する : watchlist(L) = $\{c \mid W_0(c) = L \vee W_1(c) = L\}$

その上で、以下の不変条件を考えます。

- (I-WL) (WI-1) を満たさないが (WI-2) を満たす (つまり充足されていないが異なる二つの偽でないリテラルを持つ) 節について、その監視リテラルはどちらも偽でない。

すると、(I-WL) が成り立てば (I-WLIST) が成り立ちます。WI を満たす節 c がリテラル L の割り当てによって WI を満たさなくなったとします。WI を満たさなくなったということは c は (WI-1) を満たさないが (WI-2) を満たす節だったということです。すると (I-WL) より L の割り当て前の二つの監視リテラルはどちらも偽ではなかったということになります。仮に L が c の監視リテラルでないとします。この場合 L の割り当てによって c の監視リテラルの値は変わりません。よって c は L の割り当て後も (WI-2) を満たすはずです。しかし、L の割り当てによって c が WI を満たさなくなった、つまり、(WI-2) が満たされなくなったことがわかっているので、$\overline{L}$ が c の監視リテラルだったということになります。これはつまり、$c \in$ watchlist($\overline{L}$) ということです。

よって、不変条件 (I-WLIST) を維持するためには (I-WL) を維持すればいいことがわかりました。そこで次に、リテラル L を割り当てた結果 (I-WL) が壊れた場合にそれを修正できるかを考えます。L の割り当てによって監視リストの状態が変わりうる節は watchlist(L) の中の節か watchlist($\overline{L}$) の節のどちらかです。しかし、watchlist(L) の節の (I-WL) が壊れることはありません。なぜなら監視リテラル L が未定義から真に変わってもどちらも「偽でない」ため (I-WL) の条件を満たし続けるからです。なので watchlist($\overline{L}$) だけに注目すれば良いです。L を割り当てた後の watchlist($\overline{L}$) の中の節の状態は

(1) 空節になったもの

(2) 単位節になったもの

(3) (WI-1) を満たす、つまり、充足済みのもの (これは L の割り当て以前から節が充足済みだったことを意味します)

(4) (WI-1) を満たさないが (WI-2) を満たすもの

の 4 種類のいずれかに分類できます。1、2 の場合は WI を満たしていないので自動的に (I-WL) を満たします。よってこの場合は何もしなくて良いです。3 の場合もやはり (I-WL) を満たすので何もしなくて良いです。大事なのは 4 の場合で、この場合には (I-WL) が満たされるように監視リテラルを選び直すという操作を行います。つまり、節の中の他の偽でないリテラル L' を見つけ、$\overline{L}$ の代わりにそれを新たな監視リテラルとします。(それに合わせて監視リストも更新します。)

このやり方の素晴らしいところは割り当てを巻き戻すときに特に何もしなくていいことです。上記のアルゴリズムでは 4 の場合で監視リテラルの選び直しを行っていますが、監視リテラルを選び直した後の節が (I-WL) を満たしていれば割り当て L を巻き戻しても (I-WL) をそのまま満たします。つまり、L の割り当てを行う前から L' が監視リテラルとして選ばれていたと考えても特に矛盾しないということです。これは割り当て L だけに限らず他の割り当てを巻き戻しても同様です。(ただしきちんと割り当てたのと逆の順に巻き戻す必要はあります。$(L_1 \vee L_2)$ を $L_2 = \top$、$L_1 = \bot$ の順に割り当てると割り当て前も後も (I-WL) を満たしますが、これを $L2$ から巻き戻すと (I-WL) を満たさない節ができてしまいます。)

以上のアイデアを疑似コードに落とすと以下のようになります。

```
let Q be a queue.
function find-conflict(L)
  assign L
  push L into Q
  while Q is not empty
    M := pop from Q
    for each clause c in watch-list(-M)
      if c is satisfied
        continue
      elif all literals are false
        report conflict
```

```
      elif only one literal N is unassigned
        assign N
        push N to Q
      else
        make c watch some other non-false literal
```

`find-conflict` が引数としてリテラルを一つとるようになっていますが、これは `find_conflict()` が呼ばれる前に `trail` に積まれたリテラルと同じものだと思って下さい。つまり、`decide()` によって決定されたリテラルか、あるいは `backtrack()` によって真偽が反転されたリテラルです。

単位伝搬の際に同時にキューにリテラルを積み、そのキューを舐めながら (I-WL) を満たすように監視リストを更新していきます。キューに入れているのは単位伝搬によって獲得した割り当てそれぞれについて監視リストを更新する必要があるためです。

この実装だと先ほどの説明の時と違ってリテラル L が割り当てられた瞬間には watchlist($\overline{L}$) は更新されず、L がキューから取り出されたタイミングで更新されます。そのため、watchlist($\overline{L}$) の更新のタイミングでは L 以外にも監視リストが未更新の割り当て済みリテラルが複数存在する可能性があります。この場合も特に特別な処理をする必要はなく、伝搬によって獲得した割り当て全てに対して最終的に監視リストが更新できれば (I-WL) は維持されます。

疑似コードではキューを使いましたが、キューにしていることに特に意味はなく、全部の割り当てについて監視リストを舐められれば良いのでここはスタックでも構いません。ただし、`assign N` を行った瞬間に `-N` の監視リストを更新するようなコード (例えば `find-conflict()` を再帰で書く) は難しいでしょう。というのも、条件分岐の最後で監視リストの付け替えを行っているために監視リストのループを回している最中に監視リストの中身が書き変わってしまう (いわゆる iterator invalidation が発生する) からです。

5.1. 実装

監視リストでは長さが 1 以下の節や同じ変数が複数回が現れる節の扱いが特殊になるため、入力の CNF を前処理する工程を導入してそのような節を事前に排除しておくことにします。具体的には、

(1) 入力に空節が含まれていたら充足不能であると報告して終了する

(2) 入力に単位節が含まれていたら決定レベル 0 の含意として `trail` に積み、そこからの単位伝搬も行っておく

(3) V と $\neg V$ の両方を持つ節はトートロジーなので節ごと除去する

(4) 節に同じリテラルが複数回現れる場合は吸収律によりそれらを一つに潰す

という処理をメインループに入る前に行います。(コードは省略)

追加のデータ構造は以下の通りです。

節を表す構造体 `clause` を新たに導入しています。`clause` 型の値は前処理済みの節です。`lits` の先頭二つが監視リテラルを表します。

`clause` は flexible array member を使用しています。`clause` に `vector<int>` のフィールドを生やすよりも間接参照が一回減るからです。しかし flexible array member は C++ では非標準の機能で、clang でも本当にサポートされているかとても怪しいです。(しかしコンパイルしても警告は出ない。) もしかすると undefined behavior を踏んでいる可能性がありますが今のところこれ由来と思われるバグには遭遇していません。

また、`lits` の先頭二つを監視リテラルとするのをやめて、`clause` の中に `lits` の何番目が監視リテラルかを覚えておくフィールドを生やすという手もあります。むしろ自然な発想はこちらで、先頭二つを監視リテラルとするというやり方はややトリッキーなのですが、コード量としてはむしろ今のやり方の方が少なくなって見通しも良くなるのでこうなっています。

`pos_list` と `neg_list` はそれぞれ positive/negative なリテラルに対してその監視リストを紐づけます。

```
struct clause {
    uint num_lit;
    int lits[]; // lits[0] and lits[1] are watched literals
};
vector<vector<clause *>> pos_list, neg_list;
```

`pos_list` と `neg_list` は以下の関数を通して操作されます。リテラル `lit` からそれに紐づいた監視リストを得るには以下の `watch_list(lit)` を実行します。

```
auto & watch_list(int lit) {
    return lit > 0 ? pos_list[lit] : neg_list[-lit];
}
```

それでは 2WL を用いた `find_conflict()` の実装を見ていきます。基本的な処理は上記のアルゴリズム解説通りなのですが、いくつか細かい点についてコメントしておきます。

まず、この実装では疑似コードにあったキューがなくなっています。代わりに `trail` をそのままキューとして使うようにして効率化を行っています。(コード中 1.)

疑似コードでは `find-conflict()` は引数を一つ受け取るようになっていましたが、その代わりに `trail` の先頭から直接リテラルを取り出しています。`find_conflict()` は `decide()` か `backtrack()` の後の、 リテラル一つだけ単位伝搬が始まっていない状態で呼ばれるのでこのようなコードになっています。ただし、メインループの初回の反復では `trail` が空になり得るので、 それを防ぐためにメインループの開始直前に、`trail` が空の時に先に一つ決定を行うというコードを入れています。(コードは省略)

コード中 2. では伝搬の結果もう片方のリテラルを取り出す際に `lits[0]` と `lits[1]` をスワップする処理が入っています。 これにより、割り当てが伝搬してきたリテラルが節の 1 番目に、もう片方が節の 0 番目に入ります。

コード中 3. ではこの節が充足済みの場合の処理をしています。ここも先程の疑似コードとは微妙に違っているところで、節の全体を眺めて充足済みかどうか判定するのではなく、もう片方の監視リテラルを見て充足済みかを判定しています。 これだと充足済みの節を見逃してしまう場合があるのですが、実はこれでも正しく動きます。そもそも充足済みであっても監視リテラルが二つとも偽でなければ充足済みでないケースと同じように扱って問題ありません。問題は、充足済みでかつ監視リテラルを二つとも偽以外にするのが無理なケースです。今、節の監視リテラルが L_0, L_1 だとして L_0 が偽になった状態だとすると、2 通りの場合があり得て、

(1) L_1 が真 (つまり監視されていないリテラルが全て偽)

(2) L_1 は偽 (つまり監視されていないリテラルのうち一つだけが真で残りが偽)

のどちらかです。1 の場合はコード中 3. によってすでにハンドルされています。2 の場合はそのまま L_0 が真なリテラルと付け替えられます。 この時点で L_1 への偽の割り当てはまだハンド

ルされていないはず (\(L_0\) の方が \(L_1\) より先に割り当てられてキューに積まれた) なので、\(L_0\) の付け替えが終わった後にキューから \(L_1\) が取り出され、そのタイミングでやはりコード中 3. によってこの節は充足済みであると判定されます。 結果的に、 この実装では (I-WL) よりほんの少し強い不変条件を維持していることになります。(関連する考察が [Fleury+, 2018] にあります。)

残りの部分 (4.,5.,6.) は概ね疑似コードの通りです。

```
bool find_conflict() {
    // 1. trail をキューとして使っている
    for (uint prop = trail.size() - 1; prop < trail.size(); ++prop) {
        int lit = trail[prop];
        auto & wlist = watch_list(-lit);
        for (uint i = 0; i < wlist.size(); ++i) {
            auto c = wlist[i];
            // 2. c->lits[1] が必ず偽になるようにする
            if (c->lits[0] == -lit)
                swap(c->lits[0], c->lits[1]);
            int lit = c->lits[0];
            // 3. c は充足済み
            if (ev(abs(lit)) == lit) // satisfied
                continue;
            for (uint k = 2; k < c->num_lit; ++k) {
                int lit = c->lits[k];
                // 4. c は (WI-2) を満たす
                if (ev(abs(lit)) != -lit) { // update watch list
                    watch_list(lit).push_back(c);
                    swap(c->lits[1], c->lits[k]);
                    wlist[i] = wlist.back();
                    wlist.pop_back();
                    --i;
                    goto next;
                }
            }
            // 5. c は空節
            if (defined(abs(lit)))
                return true; // conflict found
```

```
            // 6. c は単位節
            push(lit);
        next:;
        }
    }
    return false; // no conflict found
}
```

2WL が実行時間を削減する要因は二つあります。

ひとつは単純に既知の全ての節を舐めるのをやめたことによるものです。これまでの実装では、単位伝搬の際、一つ割り当てを調べるごとに全ての節を舐めて単位節を探していました。一方 2WL では割り当てごとにその監視リストを舐めるという実装です。監視リストは通常は節全体と比べればとても小さいため、ループの反復回数が少なくなります。

もう一つはバックトラックとの相性の良さです。DPLL 型のソルバーはバックトラックにより探索を行いますが、特に探索中に同じ割り当てをバックトラックを挟んで繰り返すことがよく発生します。(あるリテラル L を割り当てて、空節を発見しバックトラック、その後探索がするんでもう一度 L の割り当てが発生する、と言う状況です。) 2WL の実装では与えられたリテラルの単位伝搬を計算する際に監視リストを更新しますが、この際監視リテラルを割り当て直すおかげでそのリテラルを監視している節の数が少なくなります。これにより、同じリテラルを再度割り当てた時に一回目と比べて舐める節の数が減ります。またこれらの要因によって単に処理量が減るだけでなく、単位伝搬の一回の反復の中で触るメモリの範囲も非常に小さくなり、結果的にキャッシュヒット率が向上します。

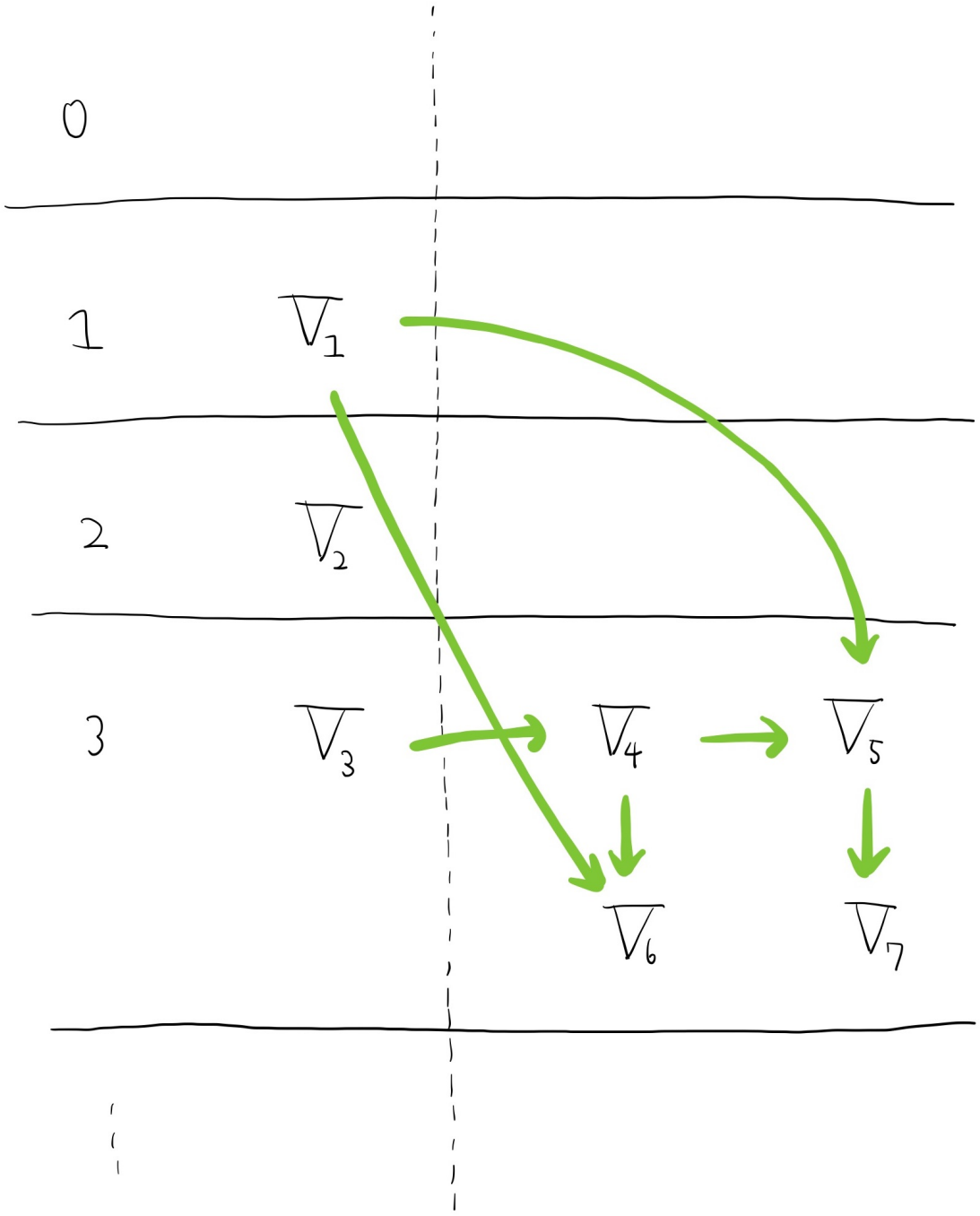
2WL は非常に優れたテクニックで、まだまだその巧妙さについて語り尽くせていない部分がたくさんあります。とはいえ長くなるのでここではひとまずこれくらいにしておきましょう。

6. Non-Chronological Backtracking

ここまでで DPLL 型のソルバーの心臓である `find_conflict()` 関数を高速化してきました。次に紹介するのはどうやって矛盾を高速に見つけるかではなく、矛盾を見つけた後に何を行うか、についてのお話です。

単位伝搬は DPLL 型のソルバーにおける最も重要な操作ですが、それがなぜ重要なのかというと「リテラルを決定して、矛盾を導いて、バックトラックする」という一連の操作を「単位伝搬する」という一回の操作でショートカットできるからでした。つまり、無駄なバックトラックを省けるというのがポイントでした。

実は、現状のソルバーにはまだ無駄なバックトラックが発生している場所があります。それは例えば以下のような状況です。



$$(\neg V_3 \vee V_4) \wedge (\neg V_1 \vee \neg V_4 \vee V_5) \wedge (\neg V_4 \vee V_6) \wedge (\neg V_1 \vee \neg V_4 \vee V_6) \wedge (\neg V_5 \vee V_7) \wedge (\neg V_6 \vee \neg V_7)$$

今、V_1, V_2, V_3 が決定されていて、V_4, V_5, V_6, V_7 が単位伝搬によって割り当てられている状態を考えています。V_4, V_5, V_6, V_7 はどれも決定レベル 3 の含意変数です。(この四つを横に並べるとごちゃごちゃしてしまうので改行しています。) 太い矢印は単位伝搬の流れを表しています。このようなグラフを**含意グラフ (implication graph)** と呼びます。 含意グラフは必ず DAG です。

この状況では、問題の中の $(\neg V_6 \vee \neg V_7)$ という節が空節になるため矛盾が導かれています。

前節までの実装ではここで V_3 の決定がよくなかったと判断して、 決定レベル 2 の含意リテラルとして $\neg V_3$ を積んでいました。 しかし、 この絵をよく見てみると、 今回と同じ矛盾を導くためには V_1 と V_3 だけがあればいいことに気づきます。つまり、この矛盾は V_2 の割り当てがどうなっているかに関わらず発生する矛盾だということです。 このことから、$\neg V_3$ を決定レベル 2 ではなく決定レベル 1 で積んでも良いということがわかります。

このように、矛盾が起こった理由を解析して、可能であれば二つ以上の決定レベルを巻き戻すことによって探索空間を減らすというのが**矛盾解析 (conflict analysis)** の基本的な考え方です。

SAT の世界ではこのようなバックトラックを Non-Chronological Backtracking (NCB) と呼びます。日本語だと**非時間順バックトラック**とか訳したりするようです。また、もう少し一般的な文脈だと、探索木を複数段一気に上がる操作のことを (バックトラックと対比して) **バックジャンプ (backjump)** と呼ぶようです。

NCB が探索空間を削減するのをイメージするために上記の問題が (実はもっと他に節があって) 充足不能だったとします。 話を簡単にするために変数選択は番号順に行われて、 まず真に割り当てられるものとします。 絵の状況から始めて「決定レベル 1 に $V_1, \neg V_2, \neg V_3$ が積まれている」という割り当てにたどり着くまでのステップ数を考えると、もし NCB がなければ

(1) $\neg V_3$ が決定レベル 2 に積まれる。

(2) 矛盾が導かれる。

(3) $\neg V_2$ を決定レベル 1 に積む。

(4) V_3 を決定レベル 2 で決定する。

(5) 矛盾が導かれる。

(6) $\neg V_3$ を決定レベル 1 に積む。

という 6 ステップですが、NCB があれば

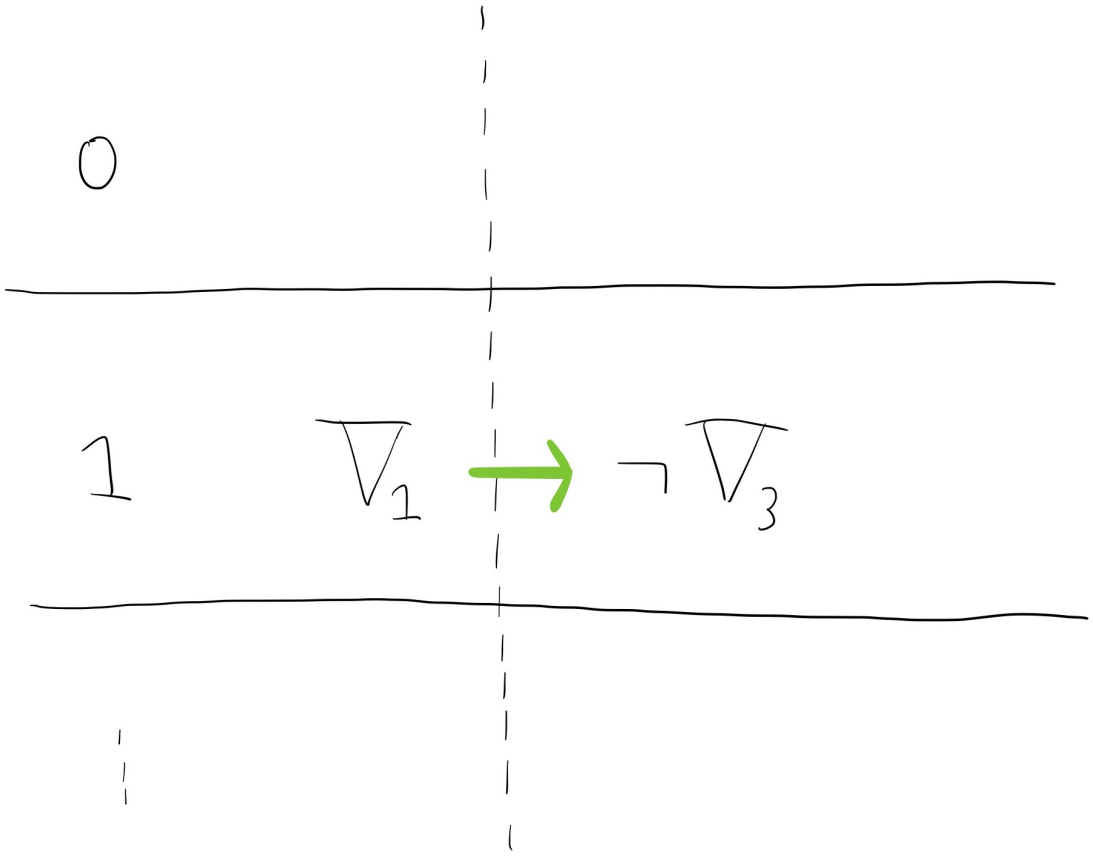
(1) $\neg V_3$ を決定レベル 1 に積む。

(2) 矛盾が導かれる。

(3) $\neg V_2$ を決定レベル 1 に積む。

という 3 ステップで済みます。(実は NCB は後で説明する phase saving と組み合わせないと効果が薄い (か場合によっては逆効果になる) のですが、その話はまた後で ...)

ここまでの例では `trail` の現在のレベルには決定と、そこからの単位伝搬による割り当てだけがある状態を考えていました。実際の探索中ではここに NCB によって得られた割り当てが加わります。例えば、上図の状態から NCB を行うと $\neg V_3$ が決定レベル 1 に積まれます。この時の含意グラフは V_1 から $\neg V_3$ に向かって太い矢印が生えた状態になります。もし V_3 よりも前に割り当てられたリテラルで V_1 以外にも矛盾導出に関わるリテラルがあればそこからの矢印も生えます。



ただし、これに関して実装上の注意点があります。

実装上は含意グラフのグラフ構造は「各変数にそれを割り当てた節」を紐づけることで保持します。例えば、V_1 と $\neg V_2$ が割り当てられている時にそこから $\omega = (\neg V_1 \vee V_2 \vee V_3)$ という節による単位伝搬で V_3 が割り当てられた場合は、V_3 に ω 由来の割り当てであるという情報を記録しておきます。このように、含意変数が割り当てられた理由となる節を**原因節 (reason clause)** と呼びます。

問題は、バックトラックによって割り当てられた変数には原因節が存在しないということです。もし矛盾を導出するのに寄与した (現在のレベルの) 割り当てが単位伝搬によるものか決定によるものだったのなら含意グラフは原因節だけから完全に復元できます。しかし、そうでない場合は含意グラフを復元できず、バックジャンプ先を正しく計算できなくなります。

この問題に対する解決案は二つあります。

一つは、保守的にバックジャンプ先を計算するというものです。もし矛盾に原因節を持たない (現在のレベルの) 含意リテラルが関与していた場合はそのリテラルに対して過去割り当てられた全てのリテラルから矢印が生えている、と考えます。結果的に、そのようなケースではNCB は行われず、一つ上の決定レベルに戻る (つまりこれまでのバックトラックと同じ挙動をする) ことになります。

もう一つのやり方は、仮想的な節を生成してしまうことです。先程の例だと、$\neg V_3$ を割り当てるに至った原因は V_1 だったので、その情報から動的に $(\neg V_1 \vee \neg V_3)$ という原因節を生成します。この節は入力の問題には存在しなかった節で、あくまで含意グラフを保持するためだけの仮想的な節です。こちらの方法では探索中に malloc/free が頻繁に呼ばれることになりますが、そのおかげでバックジャンプ先を正確に計算することができます。

今回は後者の方法を実装します。

6.1. 実装

メインループの本体は以下のようになります。単位伝搬で空節が見つかった場合に今までは `true` を返していましたが、この時にどの節が空節になったかの情報を `optional` 型を使って返すようにします。(これを実現するための `find_conflict()` への変更は非常に簡単なのでコードは省略します。) そして、`analyze()` の中でその解析と NCB を行います。

```
while (1) {
    while (auto conflict = find_conflict()) { // ココが変わった
        if (decision_level == 0)
            return false;
        analyze(*conflict); // ココが変わった
    }
    if (! decide())
        return true;
}
```

`analyze()` のコードを見る前に幾つか準備をします。

追加するデータ構造のうち本質的なものは以下の二つです。

reason は各変数 v が含意変数だった場合に原因節を保持します。 もし v が決定変数だった場合には nullptr が入ります。 変数が未割り当ての場合の値は不定です。reason は次の意味で単射的です : 二つの割り当て済み変数 v1,v2 について、reason[v1] != nullptr && reason[v2] != nullptr ならば reason[v1] != reason[v2] が成り立つ。 これは、 直観的には、一度原因節になった節は (バックトラックが起こらない限り) 他の変数の原因節にはならないということです。

level は変数 v が割り当て済みだった場合にその決定レベルを保持します。こちらも変数が未割り当ての場合は値は不定です。

```
vector<clause *> reason; // nullptr for decision
vector<uint> level;
```

他にも analyze() の実装上の都合で使用するデータ構造が追加されます。 詳細は後述します。

```
vector<bool> seen; // only used in `analyze`
vector<int> learnt; // only used in `analyze`
```

push() は level と reason を更新するように変更されていますがそれ以外は一緒です。pop() は動的に生成した原因節が使われなくなったタイミングでそれを開放する処理が追加されています。 これを実現するためには reason[var] が問題由来の 「ちゃんとした節」 か矛盾解析由来の「節もどき」かを判定する必要があるため、clause に flags というフィールドを追加し、CLAUSE_LEARNT というフラグを定義しています。(コードは省略。)

```
void push(int lit, clause * c) {
    uint var = abs(lit);
    model[var] = lit > 0 ? MODEL_DEFINED | MODEL_PHASE : MODEL_DEFINED;
    level[var] = decision_level; // ココが変わった
    reason[var] = c; // ココが変わった
    trail.push_back(lit);
}
void pop() {
```

```
    int lit = trail.back();
    uint var = abs(lit);
    model[var] &= ~MODEL_DEFINED;
    auto c = reason[var];
    if (c && (c->flags & CLAUSE_LEARNT) != 0) {
        free(c);
    }
    trail.pop_back();
}
```

NCB を実現するため、これまで使っていた決定をひとつ分巻き戻す backtrack() の代わりに複数個分の決定を巻き戻す backjump() を導入します。backjump() は backtrack() と異なり最後の決定をひっくり返す処理は行いません。(代わりに analyze() のなかでそれを行います。)

```
void backjump(uint level) {
    while (decision_level != level) {
        for (uint i = trail.size() - 1; trail[i] != 0; --i)
            pop();
        trail.pop_back(); // remove the mark
        --decision_level;
    }
}
```

さて、肝心の analyze() 関数を見ていきましょう。

先程の解説パートの通りに実装しようとすると、含意グラフを実際に構成して全ての頂点を見て回る処理が必要に思えます。普通ならこういう状況ではキューかスタックかを用意して、頂点を踏むたびにそこからつながる頂点をそこに積むというループを書くことになります。しかし、実際にはそのように明示的に探索用のキュー / スタックを用意する必要はありません。というのも、trail が実質的にそのデータ構造の役割を果たしてくれるからです。

trail に並ぶリテラルは下から順に推論の時系列通りに並んでいます。つまり、trail の中身は含意グラフがトポロジカルソートされたものだと考えることができます。(含意グラフが必

ず DAG になっていることが効いています。 ちなみにこのトポロジカルソートはレベル順でのソートになっています。 これは 2WL で `trail` をキューとして利用していたためです。) ただし、ここでの含意グラフはこれまで割り当てられた全てのリテラルからなる巨大なグラフです。実際に矛盾解析の中で興味があるのは矛盾 `conflict` から逆向きに辿っていける頂点からなる部分グラフなので、含意グラフ全体の中からうまくそのような部分グラフを抜き出す必要があります。

といってもこれを実装するのは簡単で、`trail` を上から下に向かって積んだ順とは逆順に辿るときに、矛盾に寄与したリテラルに印をつけていくだけです。トポロジカルソートされているおかげで `trail` の上にあるリテラルの原因となった割り当ては必ずそのリテラルより下の位置にあるので、これだけでうまくいきます。

今回グローバル変数として新たに追加した `seen` の用途の一つがこの印をつける操作です。`seen` は長さが `N+1` の `bool` の配列です。`analyze()` 関数の呼び出しの前後では必ず `seen` の全ての要素が `false` になっています。`seen` には変数の決定レベルに応じて二つの役割があります。矛盾が発生した時の決定レベルを l とします。

(1) 決定レベルが l であるような変数 `v` については、`seen` は今説明した用途に使われます。 つまり、`seen[v]` は含意グラフのうち現在の決定レベルの頂点だけからなる部分グラフを表すために使われます。 初期状態では `conflict` からたどれる全ての頂点 (変数) の `seen` をまず `true` にし (コード中 1.)、探索中には `seen` が `false` なリテラルを無視することで、含意グラフのうち適切な部分グラフだけを辿ります。(コード中 2-1.)

(2) 決定レベルが l 未満であるような変数 `v` については、`seen[v]` は動的に生成する原因節の中に入れるリテラルが重複しないようにする役割を持ちます。

`analyze()` が開始した時点では必ず `seen` の全ての要素が `false` になっている必要があるので、上記二つのケースそれぞれについてどうやって `seen` を初期化するか考えます。

前者の場合は、`trail` を探索する過程で貪欲に初期化を行っています。(コード中 2-2.) `trail` は因果グラフがトポロジカルソートされているので、 変数 `v` の原因節のリテラルが `trail` の中で変数 `v` より下の位置に現れることはありません。 そのため、 探索のループ内で `seen` を `false` に戻しても問題なく動きます。

後者については、学習節の中に現れている変数の `seen` を `false` にすることで対応していま

す。(コード中 3.) 後者に該当する変数は全て `learnt` に入れられているので、`learnt` についてループを回すだけで全ての `seen` が `false` に戻ります。

含意グラフの頂点から新たな頂点を辿るところでは 2WL の性質をうまく使っています。(コード中 2-3.) 2WL の実装のおかげで、割り当て済みの変数 `v` について `reason[v] != nullptr` のとき必ず `abs(reason[v]->lits[0]) == v` が成り立ちます。これのおかげで、単位伝搬を逆向きに辿る際に添字を 1 から始めるだけで自分自身を除外することができます。

矛盾に寄与したリテラルがリストアップできたら、バックジャンプ先を決定します。(コード中 4.) もし矛盾解析を行った結果、矛盾に寄与するリテラルが全て現在のレベルの割り当てだった場合はレベル 0 にバックジャンプします。

以上が終わると最後に現在の決定をひっくり返したものを割り当てますが、この時に原因節を生成して登録します。(コード中 5.) ただし、原因節の長さが 1 の場合は特殊です。原因節の長さが 1 ということは決定レベル 0 へのジャンプが起こっているはずですが、決定レベル 0 に積まれたリテラルの原因を辿ることはないので (なぜなら `analyze()` は必ず決定レベル 1 以上で呼ばれるからです)、その場合は節は生成せずに `nullptr` にしておきます。

```
void analyze(clause * conflict) {
    learnt.push_back(0); // reserve learnt[0] for decided literal
    // 1. 含意グラフ全体の中で conflict からたどれる部分だけ seen をセット
    for (uint i = 0; i < conflict->num_lit; ++i) {
        int lit = conflict->lits[i];
        uint v = abs(lit);
        seen[v] = true;
        if (level[v] < decision_level) {
            learnt.push_back(lit);
        }
    }
    int decision;
    // 2. 幅優先探索の本体
    for (uint i = trail.size() - 1; true; --i) {
        int lit = trail[i];
        uint v = abs(lit);
        // 2-1. 矛盾に関与していないリテラルは無視
```

```
        if (! seen[v])
            continue;
        // 2-2. seen の初期化はループ内でやっても問題ない
        seen[v] = false;
        auto c = reason[v];
        if (! c) {
            decision = lit;
            break;
        }
        // 2-3. c->lits[0] は lit なので除外
        for (uint i = 1; i < c->num_lit; ++i) {
            int lit = c->lits[i];
            uint v = abs(lit);
            if (seen[v])
                continue;
            seen[v] = true;
            if (level[v] < decision_level) {
                learnt.push_back(lit);
            }
        }
    }
    learnt[0] = -decision;
    // 3. 決定レベルが現在のレベルより小さい変数の seen の初期化
    uint num_lit = learnt.size();
    for (uint i = 1; i < num_lit; ++i)
        seen[abs(learnt[i])] = false;
    // 4. バックジャンプ先の決定
    uint max_lv = 0;
    for (uint i = 1; i < num_lit; ++i)
        max_lv = max(level[abs(learnt[i])], max_lv);
    backjump(max_lv);
    // 5. 原因節の生成
    if (num_lit == 1) {
        push(-decision, nullptr);
        learnt.clear();
        return;
```

```
        }
        push(-decision, make_clause(learnt, CLAUSE_LEARNT));
        learnt.clear();
    }
```

7. Conflict-Driven Clause Learning

NCB では矛盾が発生するたびにそれを解析して動的に原因節を生成していましたが、この節はあくまで含意グラフを保持するためのもので、通常の節とは異なるものでした。特に、生成した原因節を監視の対象にしていないため、`find_conflict()` でこの節が利用されることはありません。実は、この節を単位伝搬や矛盾の発見に利用しても問題がないどころか、うまく利用することで大幅にソルバーの性能を向上させることができます。

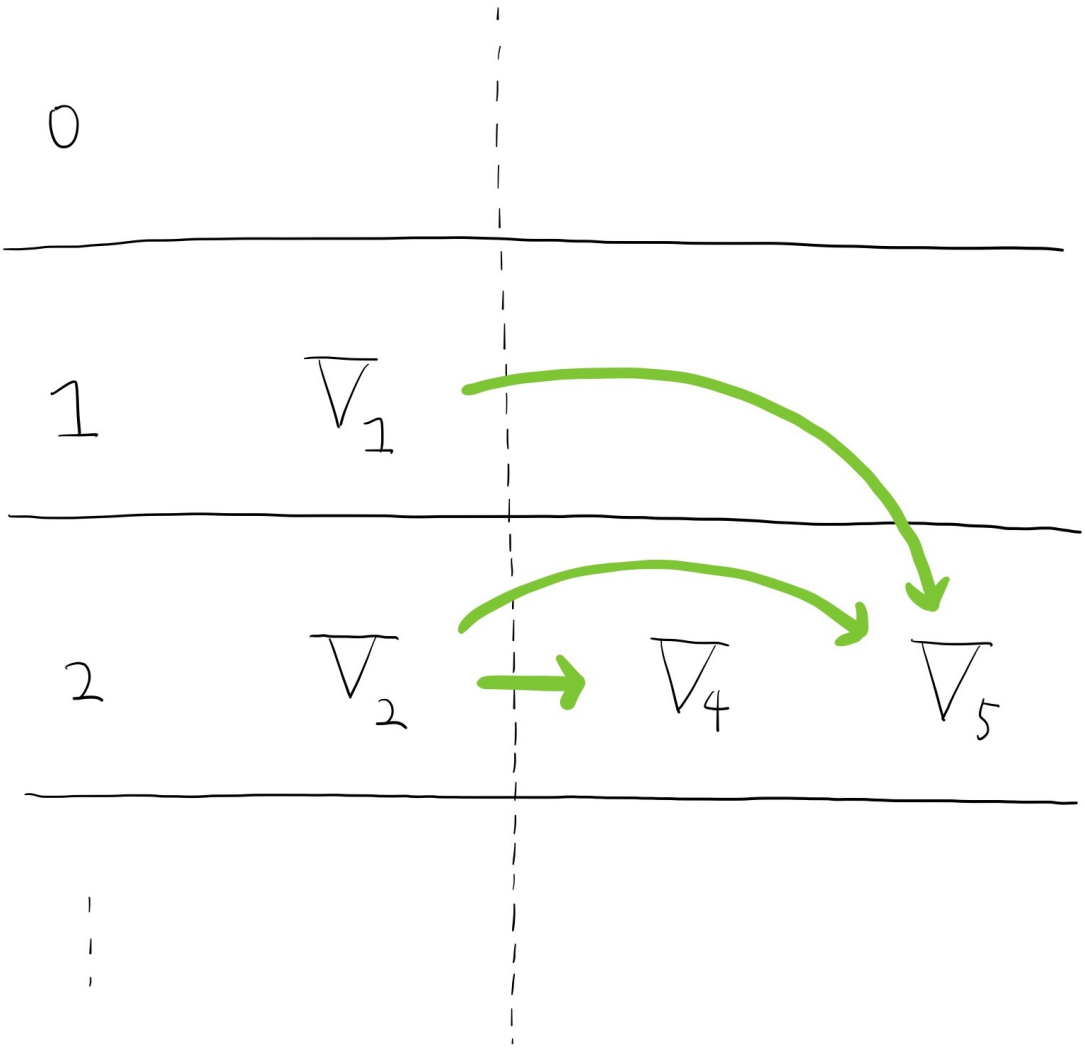
ここまでは動的に生成した原因節はあくまで「含意グラフを構成するための部品」だったのですが、これを本当に「節」だと見ることにすると、どういう節だと思えるでしょうか。結論から言うと、そのような節は「元々の問題からの論理的帰結」だと捉えることができます。論理的帰結ということは、その節を問題の節集合に追加しても充足可能性を変えないということです。(解の集合も変えません。) よって、生成した節を「最初から問題にあった」「本当の節」と見なしても特に問題がないです。

このように生成した原因節を本当の節と見なすことを「節を学習する」あるいは「節を獲得する」と言います。そして、学習した節を学習節と言います。矛盾解析によって学習節を獲得する手法を **Conflict-Driven Clause Learning (CDCL)** と言います。

学習節を獲得することがなぜ重要かというと、 それが探索空間の削減につながるからです。例えば、以下のような例を考えます。

$$(\neg V_1 \vee \neg V_2 \vee V_5) \wedge (\neg V_2 \vee V_4) \wedge (\neg V_3 \vee V_4) \wedge (\neg V_3 \vee \neg V_4) \wedge (\neg V_5 \vee \neg V_4)$$

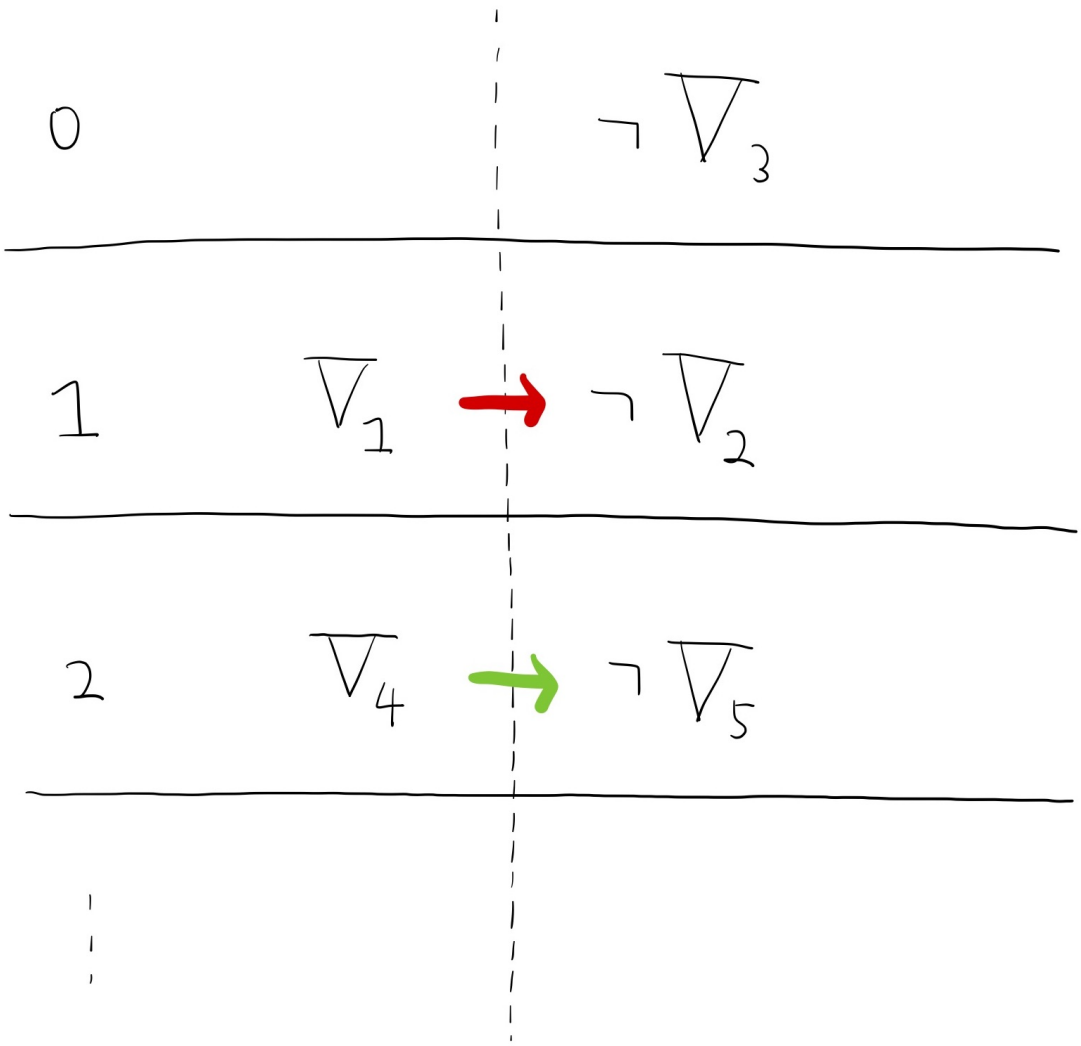
これをソルバーで回した時の動作を考えます。V_1, V_2 を順に割り当てた場合を考えます。V_1 を割り当てても何も伝搬が起こらず、V_2 を割り当てたときに伝搬が発生して V_4, V_5 への割り当てが行われます。しかしこれは CNF の最後の節が空節になるので矛盾です。



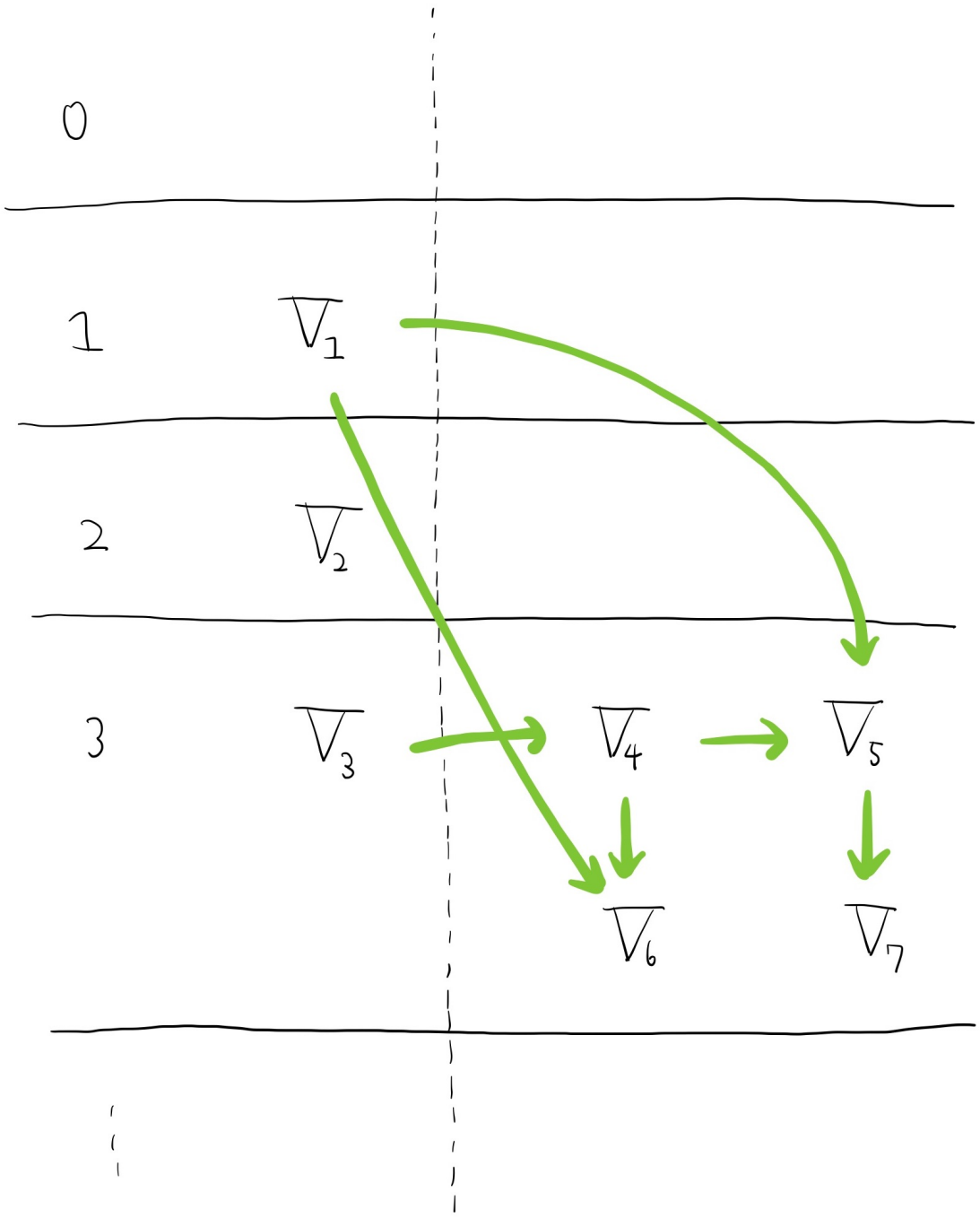
このとき得られる原因節 $(\neg V_1 \vee \neg V_2)$ がここでの学習節になります。この矛盾解析の結果 $\neg V_2$ が決定レベル 1 に積まれますが、そこからの単位伝搬はなく、そのまま V_3 が決定されます。すると、単位伝搬によりもう一度矛盾が導かれます。この時の矛盾には V_3 以外のリテラルが寄与しないので、決定レベル 0 にまでバックジャンプした上で $\neg V_3$ が積まれます。

$\neg V_3$ からの伝搬はないのでそのまま次の決定で V_1 が選ばれたとします。この時もし学習節があれば $\neg V_2$ が含意されます。そして、その次に V_4 が決定されれば $\neg V_5$ が伝搬して解が見つかります。これまでのように学習節がない場合は再度 V_2 を決定して矛盾を導くプロセスをや

り直すことになってしまいます。



さて、ここまでの全ての例ではバックジャンプ後には必ず現在の決定レベルの決定をひっくり返すという想定でした。前節の例でいえば、$\neg V_3$が積まれる想定です。しかし実は、そこで代わりに$\neg V_4$を積んでも良いことに気が付きます。というのも、V_1, V_4の二つが割り当てられたとしても、やはり同じ矛盾が導かれるからです。



(前節の例の再掲)

これは、V_3 も V_4 もこの含意グラフにおける Unique Implication Point (UIP) だからです。リテラル L が UIP であるというのは、L が現在の決定レベル l のリテラルであって、 決定レベルが l 未満の他のリテラルと合わせると矛盾が導けるうようなときにそう呼びます。例えば V_4 は V_1 と合わせればそこから矛盾が導けるので UIP です。現在の決定レベルの決定リテラルは (矛盾解析が発生したときはいつでも)UIP です。

矛盾解析によって原因節・学習節を獲得する、という定性的な観点から言えば実は獲得する節は UIP であればどれでも良いです。 一方定量的な観点から UIP 一つを選ぶというのであれば、一番良いのは矛盾に最も近い UIP です。これを first UIP (1-UIP) と呼びます。(今、最も矛盾に近い、という表現をしましたが、同じ矛盾から得た UIP はその矛盾からの近さで全順序がつけられることに注意してください。 これは絵を描くと明らかです。) というのも、矛盾から近い UIP ほどリテラルの数が少なくなるからです。矛盾から遠い方の UIP は矛盾から近い UIP を (単位伝搬によって) 含意するので、論理的には 1-UIP は UIP の中でも一番弱いのですが、リテラルが少ないおかげでこれ以降の探索で単位伝搬に寄与しやすくなります。そこで、アルゴリズムの方も 1-UIP を獲得するように変更することにします。NCB の時は含意グラフを構成できるかだけに注目していましたが、これからは学習節が単位伝搬に有用かどうかを考えなければならなくなったので、選ぶ UIP を変えるというわけです。

さて、これらを実際に実装するためにもう一つ考えなければならないのは学習節の増えすぎです。

これまで動的に生成した原因節はバックトラックのタイミングで削除されていました。しかし、今後はバックトラックが起こっても節は保持したままになります。そうすると矛盾を見つけるたびに節がどんどん増えていってしまい、すぐにシステムのリソースを食いつぶしてしまいます。また、節の数が多すぎると単純にソルバー自体の性能も低下させてしまいます。それを防ぐためには定期的に学習節を削除する必要があります。学習節はあってもなくても問題自体の充足可能性や解の集合を変えるものではないので消してしまっても問題がないのです。

しかし、節の削除のアルゴリズムを具体化するためにはいくつも考慮しなければならない要素があります。少なくとも、以下の 3 点は明確にしなければなりません。

(1) いつ節削除を行うか

(2) どの節から削除していくか

(3) どれぐらいの数の節を削除するか

残念ながら節の削除については決定版と呼べる方法はなく、基本的にはこれらのパラメータをヒューリスティクスで決めていきます。本記事ではどのようなヒューリスティクスが優れているかについては深入りはせず、実装の枠組みを作ることを目標にします。実際に様々なヒューリスティクスを比較するには多くの実験が必要になるでしょう。

7.1. 実装 (節の学習)

まずは節の学習だけを行う (節の削除はしない) バージョンを作成します。

主な変更点は `analyze()` の中だけですが、`pop()` の方も動的に生成された原因節を `free()` する処理を消すという変更を入れてあります。(コードは省略)

`analyze()` のうち前半、つまり、1-UIP を探す部分は少しの変更だけで実装できます。`count` は含意グラフの中で今現在探索しているリテラルのうち現在のレベルのものの数です。これが一時的に 0 になったタイミングで UIP を見つけたということになります。

```
void analyze(clause * conflict) {
    learnt.push_back(0); // reserve learnt[0] for UIP
    uint count = 0; // ココが変わった
    for (uint i = 0; i < conflict->num_lit; ++i) {
        int lit = conflict->lits[i];
        uint v = abs(lit);
        seen[v] = true;
        if (level[v] < decision_level) {
            learnt.push_back(lit);
        } else {
            ++count; // ココが変わった
        }
    }
    int uip;
    for (uint i = trail.size() - 1; true; --i) {
        int lit = trail[i];
        uint v = abs(lit);
        if (! seen[v])
            continue;
        seen[v] = false;
```

```
            // ココが変わった
            --count;
            if (count == 0) {
                uip = lit;
                break;
            }
            auto c = reason[v];
            for (uint i = 1; i < c->num_lit; ++i) {
                int lit = c->lits[i];
                uint v = abs(lit);
                if (seen[v])
                    continue;
                seen[v] = true;
                if (level[v] < decision_level) {
                    learnt.push_back(lit);
                } else {
                    ++count; // ココが変わった
                }
            }
        }
        learnt[0] = -uip;
        uint num_lit = learnt.size();
        for (uint i = 1; i < num_lit; ++i)
            seen[abs(learnt[i])] = false;
        // 後半へ
        ...
    }
```

1-UIP が求まったら、そこで得た原因節を単位伝搬に利用できるようにします。ここもコード上は本質的には二箇所の変更だけです。特に、最も重要なのが学習節を `watch_clause()` で監視対象にする部分です。(コード中 2.) コード上は単に関数を呼んでいるだけなのですが、なぜこれだけで正しく動くのかについてはかなりちゃんと考えないといけません。というのも、2WL では節の扱いがかなり特殊なため、獲得した節を問題由来の節と同等に扱うためには、節の長さや監視リテラルの選び方についてしっかり考える必要があるのです。

まず、節の長さについてですが、2WL では長さが 1 以下の節の扱いが特殊になるのでした。

それに対応するため問題由来の節についてはソルバーを回す前に問題を前処理してそのような節を除外したり、先に単位伝搬させるということを行いました。学習節についても同様の扱いが必要で、学習節を追加するときに長さが 1 の節については監視の対象にしません。(ここはたまたま NCB から何も変更しなくても動いています。)

また、監視リテラルを二つ選ぶ方法についても、果たしてそんなものが選べるのかというところから考える必要があります。結論としては、監視リテラル二つは選べます。

今、獲得した学習節が $(L_1 \vee L_2 \vee \cdots \vee L_n)$ だったとして、リテラルが決定レベルで降順に並んでいたとします。 このとき、L_1 と L_2 をこの節の監視リテラルとして選び、 管理リストを更新します。この状態で `analyze()` を終了すると、バックジャンプは L_2 の決定レベルに対して行われ、L_1 は一旦未割り当てになったあと真が割り当てられます。 この時点で L_2 から L_n は偽になっています。この状態は (I-WL) を満たします。ここからさらにバックジャンプが発生すると L_1 と L_2 が未割り当てになるのでやはり (I-WL) を満たします。よって、学習節の中でレベルが最も高い二つのリテラルを監視リテラルとすれば 2WL の枠組みとうまく組み合わせられるということがわかります。(コード中 1.)

```
void analyze(clause * conflict) {
    // 前半から
    ...
    uint max_lv = 0;
    for (uint i = 1; i < num_lit; ++i) {
        uint lv = level[abs(learnt[i])];
        if (lv > max_lv) {
            max_lv = lv;
            swap(learnt[1], learnt[i]); // 1. learnt[1] にバックジャンプ先のレベルのリテラルが入るようにする。
        }
    }
    backjump(max_lv);
    if (num_lit == 1) {
        push(-uip, nullptr);
        learnt.clear();
        return;
    }
```

```
    auto c = make_clause(learnt, CLAUSE_LEARNT);
    push(-uip, c);
    learnt.clear();
    watch_clause(c); // 2. 監視を行う
}
```

7.2. 実装 (節の削除 ; size-based randomization)

解説パートでは節の削除には三つの考えるべきことがあるという話をしました。 実装に当たってこれらを先に決めておきます。

(1) いつ節削除を行うか

(2) どの節から削除していくか

(3) どれぐらいの数の節を削除するか

まず、1 については決定を行うたびに学習節の数が閾値に達しているか確認し、 閾値に達していれば削除を行うこととします。 この閾値はこれまで見つかった空節の数に応じて指数で大きくしていくことにします。 また、長さが 2 の学習節は単位伝搬を促進するのに非常に有用だと考えられるので削除しません。2 については [Jabbour+ 2014] で提案されている Size-based randomized strategy を採用することにします。 この戦略では節ごとに以下で計算されるスコアを割り当てて、よりスコアが大きいものから削除していきます。ただし k は事前に設定されたパラメータで、ω はランダムに選ばれます。

$$\mathrm{score}_k(c) = \begin{cases} |c| & (|c| < k) \\ |c| + \omega & (\omega \in [0,1)) \end{cases}$$

3 については MiniSAT に倣って一度に学習節を半分に削減するという戦略を採用します。

以上がこれから実装するものになります。実装上の注意点は、 探索中の節削除によって状態が壊れないようにする点です。 具体的には、`reason` から参照されている節を削除しないようにしなければなりません。

実際のコードを見ていきます。

まず、全ての節を保持する `db` という `deque` を導入します。最初の `db_num_persistent` 個

の節は絶対に削除しない節、残りの節は削除しても良い節ということにします。入力由来の節や、長さが 2 以下の節は persistent 扱いになります。

```
deque<clause *> db; // all clauses; first `db_num_persistent` clauses are persistent
uint db_num_persistent;
```

メインループに対する変更は以下の通りです。矛盾を見つけるたびに閾値を更新しつつ、決定を行うたびに reduce() を呼んで節の削減を行います。 閾値の更新のロジック (追加した 6 行) はおまじないだらけですがこれは MiniSAT のものをそのままクローンしているからです。今回はヒューリスティックスの詳細 (実験的な結果) には立ち入らないことにしたので、ここはこういうものだと受け入れてください。

```
while (1) {
    while (auto conflict = find_conflict()) {
        if (decision_level == 0)
            return false;
        analyze(*conflict);
        // 以下の 6 行を追加
        ++backoff_timer;
        if (backoff_timer >= backoff_limit) {
            backoff_timer = 0;
            backoff_limit *= 1.5;
            db_limit = db_num_persistent + (db_limit - db_num_persistent) * 1.1;
        }
    }
    if (! decide())
        return true;
    reduce(); // この行を追加
}
```

節を表す構造体に score というフィールドを追加しています。score が小さいほど節が削除の対象になりにくいです。flags と CLAUSE_LEARNT は先程言及したもののコードは見せなかったものです。ここでは新しく CLAUSE_LOCK というビットフラグを追加しています。(用途は後述。)

```
enum {
    CLAUSE_LEARNT = 1,
    CLAUSE_LOCK = 2,
};
struct clause {
    uint num_lit;
    int flags;
    double score;
    int lits[];
};
```

CLAUSE_LOCK は以下のように変数を割り当てたり割り当てをキャンセルするタイミングで設定されます。

```
void push(int lit, clause * c) {
    ...
    reason[var] = c;
    if (c)
        c->flags |= CLAUSE_LOCK;
}
void pop() {
    ...
    clause * c = reason[var];
    if (c)
        c->flags &= ~CLAUSE_LOCK;
}
```

最も重要な reduce() 関数の実装です。現在 reason として使用されている節は削除しないようになっています。このせいで正確には節の数を半減することはできないのですが、これでよしとします。

```
void reduce() {
    if (db.size() < db_limit)
        return;
```

```
    sort(db.begin() + db_num_persistent, db.end(), [](auto x, auto y) {
        return x->score < y->score;
    });
    uint new_size = db_num_persistent + (db.size() - db_num_persistent) / 2;
    for (uint i = new_size; i < db.size(); ++i) {
        if ((db[i]->flags & CLAUSE_LOCK) != 0) {
            db[new_size++] = db[i];
            continue;
        }
        free(db[i]);
    }
    db.resize(new_size);
}
```

最後に学習節のスコアの計算です。計算アルゴリズムは論文のものをそのまま使用しました。今回はパラメータとして論文中で最も良い性能を出したとされる 12 を選択しました。(このソルバーはまだ大きい問題を解けないのでこのパラメータが今のソルバーにとって良い値かは不明です。)

```
#define SBR_BOUND 12

void analyze(clause * conflict) {
    ...
    // learn new clause
    double score;
    if (num_lit < SBR_BOUND) {
        score = num_lit;
    } else {
        score = SBR_BOUND + rand() * (1 / ((double) RAND_MAX + 1));
    }
    auto c = make_clause(learnt, CLAUSE_LEARNT, score);
    if (num_lit == 2) {
        db.push_front(c);
        ++db_num_persistent;
    } else {
```

```
            db.push_back(c);
        }
        ...
    }
```

8. Variable State Independent Decaying Sum

ここまででは単位伝搬やそれに関わる部分の高速化に取り組んできましたが、決定に関わる部分の高速化は行ってきませんでした。つまり、決定の際の変数選択にはまだ改良の余地が残されているということです。現在の `choose()` は単なるループなので、ここを改善することをこの節の目標にします。

```
int choose() {
    for (uint v = 1; v <= N; ++v) {
        if (! defined(v))
            return (int) v;
    }
    return 0;
}
```

節削減での話と同様、変数選択も基本的にはヒューリスティクスによるものです。ただし、この部分についてはある種の定石があり、多くのソルバーがそれを実装しています。それが、**Variable State Independent Decaying Sum (VSIDS)** と呼ばれる手法です。

VSIDS では各変数にスコアを割り当て、変数選択時にはスコアが最も優れた変数を選びます。その際のスコアとして VSIDS では「コンフリクトの解析でその変数が出現した回数」を採用しています。出現回数が多い変数ほどより多くの含意を持ち、それを選択することによって多くのリテラルの値が自動的に決まると考えられるからです。ただし実際に愚直に回数を数えてしまうと桁溢れを起こしてしまうことが想定されるので、定期的に全ての値を定数値で割ります。これによる副作用として、より最近のコンフリクト解析で出現した変数のスコアがより高くなります。(VSIDS といえば普通はこのような解析時の変数の出現回数を数えるアルゴリズムを指すのですが、VSDIS の枠組み自体はかなり一般的で何の回数を数えるかはそれなりに自由に選ぶことができます。)

8.1. 実装

まず各変数に対してスコアを保持するようにデータを用意します。コード中では変数のスコアを activity と呼んでいます。

```
vector<double> activity;
```

スコアの操作には二種類のクエリがあります。一つが、ある変数のスコアを上昇させる操作、もう一つが全てのスコアを定期的に定数で割る操作です。コード上はそれぞれ bump_activity()、decay_activity() が対応します。analyze() は変数出現を一回見つけるたびに bump_activity() を呼び、スコアを +1 します。ソルバーのメインループである solve() は空節が見つかるたびに decay_activity() を呼び、定期的に全てのスコアを半減させます。

```
void bump_activity(uint v) {
    activity[v] += 1.0;
}
void decay_activity() {
    ++num_conflict;
    if ((num_conflict % ACTIVITY_DECAY_PERIOD) == 0) {
        for (uint v = 1; v <= N; ++v)
            activity[v] *= 0.5;
    }
}
```

例えば、ソルバーのメインループは以下のように変更されます。(コードの見やすさのために CDCL にまつわるコードを消したものを示しています。)

```
while (1) {
    while (auto conflict = find_conflict()) {
        if (decision_level == 0)
            return false;
        analyze(*conflict);
        decay_activity(); // ココ
```

```
    }
    if (! decide())
        return true;
}
```

以上でスコアの計算は終わりです。あとは choose() をこのスコアに沿って変数を選ぶように変更するだけです。

```
int choose() {
    double max_score = 0;
    int max_lit = 0;
    for (uint v = 1; v <= N; ++v) {
        if (! defined(v)) {
            double s = activity[v];
            if (s >= max_score) {
                max_score = s;
                max_lit = (int) v;
            }
        }
    }
    return max_lit;
}
```

8.2. 実装 (ヒープ)

今の実装では choose() は単なるループのままでした。 変数の集合の中から最もスコアが高い変数を選ぶ、ということをしているので、 これを Priority Queue を使って高速化したいと考えるのは自然な発想でしょう。 ただし、 少し注意する点があります。 変数のスコアが bump_activity() が呼ばれるたびに変化してしまうという点です。つまり変数のスコアが変わるたびにその変数の Priority Queue の中での位置を再調整できる必要があります。 このために、普通の二分ヒープに加えて、変数からその変数の二分ヒープ内での位置を逆引きするための配列を用意します。

```
vector<uint> heap; // priority queue for variable selection
vector<uint> heap_index; // variable to index in heap; 0 if variable not in heap
```

ヒープに対する操作 (heap_push(), heap_pop(), heap_top(), heap_up() など) は普通通りに実装します。(コードは省略) ただし、ヒープの中の値を移動させた場合は heap_index の方も変更するようにしておきます。heap_up() があれば bump_activity() は以下のように実装できます。これで、ヒープの状態と実際のスコアが整合するようになります。

```
void bump_activity(uint v) {
    activity[v] += 1.0;
    if (heap_index[v] != 0)
        heap_up(heap_index[v]);
}
```

ヒープの中には未定義の変数が入っていることが期待されるので push() では変数をヒープから削除し、pop() では変数をヒープに追加することが期待されます。 ただし、push() の中で変数をヒープから削除するのはコストが大きいので、push() で変数をヒープから削除することは諦めてヒープの中に定義済みの変数が入ってしまうことを許容します。 その代わりに、choose() の中では heap_pop() するたびにそれが現在未定義かどうかを確認することにします。

```
void pop() {
    ...
    if (heap_index[var] == 0)
        heap_push(var);
}
```

```
int choose() {
    while (! heap_empty()) {
        uint v = heap_top();
        heap_pop();
        if (! defined(v)) {
```

```
            return v;
        }
    }
    return 0;
}
```

8.3. 実装 (exponential VSIDS / eVSIDS)

VSIDS はそれだけでも十分軽い処理です。しかし、空節が見つかるたびに定期的とはいえ全ての変数を舐める `decay_activity()` はそれなりのコストになります。 そこで MiniSAT では exponential VSIDS (eVSIDS) と呼ばれるテクニックを用いて `decay_activity()` を高速化しています。

eVSIDS の鍵となる観察は、定期的に全ての変数のスコアを半減させることと、定期的にスコアの上昇度合いを倍増させることが同じであるということです。変数のスコアは優先度としての意味しかなく、相対的な大小にしか意味がありません。今までは `bump_activity()` では必ずスコアを `+1` して `decay_activity()` で定期的に全スコアを半減させていましたが、 その代わりに `bump_activity()` でスコアを `+activity_increment` することにして `decay_activity()` で定期的に `activity_increment` を倍にすれば、スコアの大小関係を保ったままで `decay_activity()` の中のループを無くすことができます。ただし、これではスコアが急激に大きくなってしまうので今度は `bump_activity()` の中で定期的にスコアの切り下げを行います。

以上のアイデアを実装したのが以下です。実際のコードでは `ACTIVITY_DECAY_FACTOR` は `0.5` ではなく (MiniSAT に倣って)`0.9` にしています。

```
void bump_activity(uint v) {
    activity[v] += activity_increment;
    if (activity[v] > ACTIVITY_RESCALE_LIMIT) { // rescore
        activity_increment *= (1 / ACTIVITY_RESCALE_LIMIT);
        for (uint i = 1; i < activity.size(); ++i)
            activity[i] *= (1 / ACTIVITY_RESCALE_LIMIT);
    }
```

```
    if (heap_index[v] != 0)
        heap_up(heap_index[v]);
}
void decay_activity() {
    activity_increment *= (1 / ACTIVITY_DECAY_FACTOR);
}
```

9. Phase Saving

ここまでの VSIDS に関する節では決定を行う際にどの変数を選択するかを議論してきました。一方で、どの変数を選択するかが決まった時にその変数に真と偽のどちらを割り当てるかについては何も考えてきませんでした。(今のところ常に真を割り当てるようになっています。) 実際のところ、どちらを割り当てるのがより効率的なのでしょうか？実はこれについてはある程度答えが出ていて、Phase Saving [Pipatsrisawat & Darwiche, 2007] と呼ばれる手法が有効であることが知られています。

Phase saving (あるいは Progress saving, 部分解キャッシュ) の考え方は以下の通りです。

CNF、つまり節集合 Δ の部分集合 $C \subseteq \Delta$ が $\Delta \setminus C$ と変数を共有しないとき、C を Δ の component と呼びます。明らかに、component C の解は $\Delta \setminus C$ の解には依存しません。ソルバーの探索中には component の解を探索することが度々発生します。もちろん最初に与えられた問題が複数の component を持つこともあり得ますが、むしろ探索中の特定の時点での (部分) 割り当ての下で問題が複数の component に分割できるというような状況の方が支配的でしょう。このような状況では、ソルバーは component C とその補集合 $\Delta \setminus C$ の解を同時に探索することになります。その際、C の解は見つかったものの、$\Delta \setminus C$ の解の探索中にコンフリクトが見つかる、というようなことが発生します。こうなるとコンフリクト解析を経てバックトラックが行われたときに、すでに見つかっている C の解が消えてしまうということが発生します。これを防ごうというのが Phase saving の問題意識です。

では Phase saving はどのようにしてこれを解決するのかというと、決定をする際に単に前回の割り当て時の phase を復元するということを行います。もしその変数に一度も割り当てが行われたことがなければ真偽値は適当に選びます。これによってすでに解が見つかっていた component の解をバックトラック後も復元することができ、探索を効率化することができま

す。

9.1. 実装

Phase saving の実装は非常に簡単で、一行変更するだけです。これまでの実装では phase はバックトラックの後も消去されず `model` の中に保存されたままになっているのでそれを取り出すだけです。

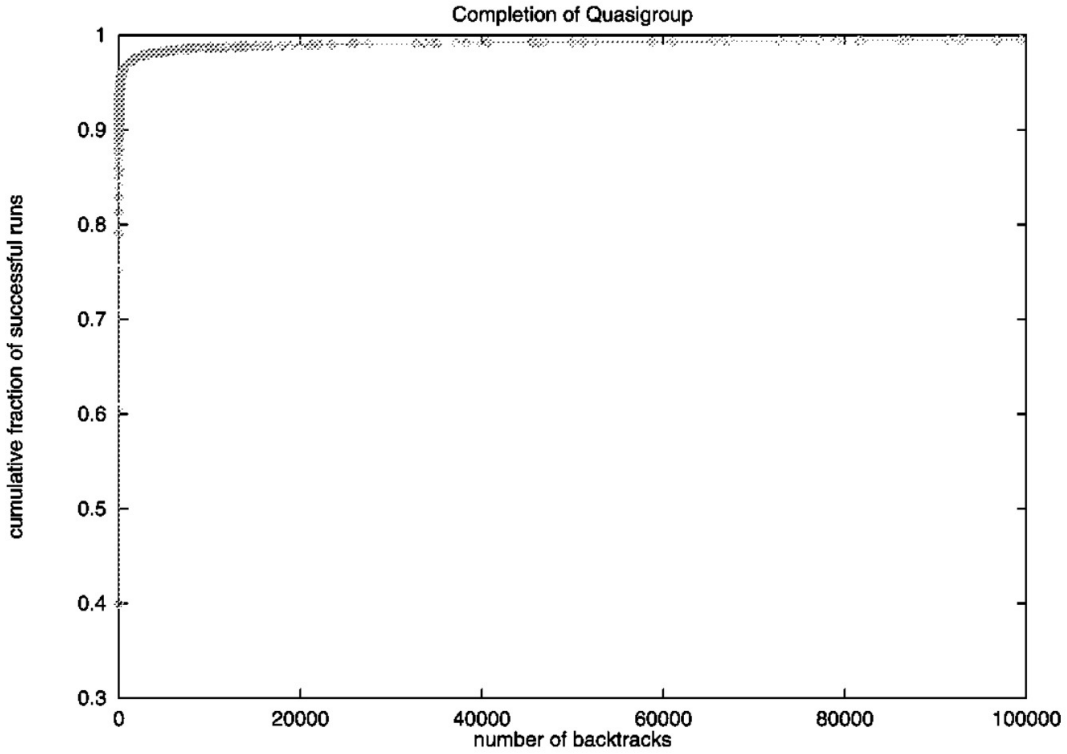
```
int choose() {
    while (! heap_empty()) {
        uint v = heap_top();
        heap_pop();
        if (! defined(v)) {
            return phase(v) ? (int) v : -(int) v; // 前は `(int) v` だった
        }
    }
    return 0;
}
```

10. Restarts

これまでの種々のテクニックの導入の中でいくつかのヒューリスティクスを導入してきました。 具体的には節削除と変数選択です。 これらの操作はある種の自由度があり、 いくつもの選択肢の中からできるだけ最良のものを選ぶべくヒューリスティクスが導入されています。 ではもしヒューリスティクスが選んだ選択肢が「ハズレ」だった場合は何が起こるのでしょうか？ もっと言えば、どれくらいのペナルティがあるのでしょう、あるいはハズレはどのくらいあるのでしょうか。

このような問いについての研究が [Gomes+, 2000] です。 この研究では DPLL ソルバーにランダムな挙動を許した場合にどのような結果になるかを調査したものです。ランダムな挙動を許す、というのはこの場合、例えば「変数選択で X_1 と X_2 のどちらも同じぐらいのスコアだった場合にどちらかをランダムに選択する」というような改造を施したという意味です。この論文の結果は非常に示唆的です。例えば論文中の以下の図を見てください。この図はある特定の問題 (ここでは Quasigroup completion problem という NP 完全問題を SAT にエンコードした

もの) をソルバーで 10000 回実行した時に、解を導けた割合をプロットしたものです。横軸はバックトラックの最大回数を何回に設定したかを表しており、バックトラックの数を増やせば増やすほど求解に至る割合が増えています。しかし、この図が示すようにバックトラックの回数をどれだけ大きくしても縦軸が 100% になっていません。これはこの問題に限らずあらゆる問題に言えるようで、いずれの場合もこのように裾野が非常に広い分布を示すようです。



このように同じソルバーであっても選択肢に自由度がある時に「ハズレ」を引いてしまうと、そうでなければすぐに解けていたはずの問題が解けずじまいになってしまいます。このような事態を防ぐために現代的なソルバーはいずれも**リスタート**と呼ばれる処理を行います。

リスタートはアイデア自体は非常に簡単で、しばらく探索を続けて答えが分からなければ最初から探索をやり直す、というものです。これをアルゴリズムにする際に考えなければいけないのは

(1) 「しばらく」とはどれくらいか

(2) 最初から探索をやり直す、とはどういうことか

の 2 点です。まず二つ目の点については、単に決定レベル 0 にバックジャンプすることを意味します。この際学習節や変数のスコアは変更しません。このおかげで前回決定レベル 0 から探索を開始した時とは違うパスが探索されることになります。(節削除や変数選択のヒューリスティクスがあまりに悪いとリスタートの前後で全く同じことになるかもしれませんが普通はまず起こらないと思います。) 一つ目の点については **Luby リスタート**と呼ばれるやり方を採用します。まず、自然数 k について以下で定義される自然数列 (t_i) を **Luby 数列**と呼びます。

$$t_i = \begin{cases} k2^{j-1} & (i = 2^j - 1 \text{ for some } j) \\ t_{i-2^{j-1}+1} & (2^{j-1} \leq i \leq 2^j - 1) \end{cases}$$

$k = 1$ の時、この数列は以下のような値になります。

$$1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,\cdots$$

リスタートはこの Luby 数列に従って発火させます。i 回のリスタート後のソルバーは t_i 個のコンフリクトを見つけるとその探索を諦めてリスタートを行います。

なぜこのような不思議な数列を使うかというと、これがある意味で最適であることが証明できるからです。[Luby+, 1993] 証明は追っていないのですが、とりあえずこの論文の主張を翻訳すると以下のようになります。

> A をラスベガスアルゴリズム (必ず停止して答えを出すが、答えに至るまでの時間がランダムであるようなアルゴリズム) とする。戦略 $S = (t_0, t_1, \cdots)$ を時間の列として、A を t_0 ステップ回して、答えがでなければ次に t_1 ステップ回して ... ということを繰り返すとする。この時、答えが出るまでの時間の期待値を最小化したい。結論としては、A の実行ステップ数の分布が不明な場合、戦略 S として Luby 数列を採用することが任意の A について最適である。

ここでの A が我々の SAT ソルバーです。この問題の設定でいうところのアルゴリズム A を 1 ステップ回すことが、SAT ソルバーで k 個のコンフリクトを見つけることに相当します。

10.1. 実装

さて、実装にあたります。ソルバーのメインループが以下のように変更されます。(見やすさのために VSIDS と database reduction のコードを除去していますが本質は変わりません。)

```
restart_timer が (前回のリスタート以降に) コンフリクトを見つけた数です。先程の説明で
```

は k 個のコンフリクトを見つけ次第リスタートを行う、という説明でしたが、この実装では k 個のコンフリクトを見つけた後も単位伝搬が一通り終わるまではリスタートをしません。こうすると単位伝搬のループがなかなか終了しない場合にリスタートができず「ハズレ」から抜け出せない気もしますが、それは話が逆で「ハズレ」のケースというのはほとんどが単位伝搬がなかなか起きず枝刈りが全然できない状態であるという経験則があります。そのため単位伝搬が長く続くようであればむしろそれは「ハズレ」を引いていないということだろう、という推測ができます。そのようなケースでは最後まで伝搬を行うことで有用な学習節を得たり解が求まる期待があります。

```
while (1) {
    while (auto conflict = find_conflict()) {
        if (decision_level == 0)
            return false;
        analyze(*conflict);
        ++restart_timer; // ココを追加
    }
    if (restart()) // ココを追加
        continue;
    if (! decide())
        return true;
}
```

さて、`restart()` を実装するにあたって Luby 数列を実装する必要があります。先程紹介した数式通りに実装するとかなり重い処理になってしまうのですが、幸いビット演算を使った非常に高速な計算方法が知られています。ちなみにこのアルゴリズムを考えたのは Donald Knuth だそうです。(またお前か)

$$
\begin{aligned}
(u_0, v_0) &= (1,1) \\
(u_{i+1}, v_{i+1}) &= \begin{cases} (u_n + 1, 1) & (\mathrm{bitwiseAND}(u_n, -u_n) = v_n) \\ (u_n, 2v_n) & \text{otherwise} \end{cases}
\end{aligned}
$$

とすると、(v_i) は Luby 数列。

ここまでの説明があれば `restart()` 関数の実装はすぐ読めます。リスタートの基準を満た

していればLuby数列を計算し、決定レベル0へのバックジャンプを行います。コード中のRESTART_BASE_INTERVALが先程の解説にあったkのことです。

```
array<int, 2> luby_seq { 1, 1 }; // reluctant doubling

bool restart() {
    if (restart_timer < restart_limit)
        return false;
    luby_seq = {
        (luby_seq[0] & -luby_seq[0]) == luby_seq[1] ? luby_seq[0] + 1 : luby_seq[0],
        (luby_seq[0] & -luby_seq[0]) == luby_seq[1] ? 1 : 2 * luby_seq[1]
    };
    restart_timer = 0;
    restart_limit = RESTART_BASE_INTERVAL * luby_seq[1];
    backjump(0);
    return true;
}
```

11. Partial Restarts

さて、先程の実装ではリスタートの際には必ず決定レベル0に戻っていました。決定レベル0に戻った後はその時点での学習節や変数のスコア、保存されたphaseの情報に応じて変数選択と単位伝搬が行われ探索が進みます。しかしもしリスタート後にリスタート前と同じ変数選択が行われた場合、同じような導出を繰り返してしまうことになります。もちろん、学習節の状態が違うので全く同じ単位伝搬が行われるわけではないのですが、リスタート前に行った伝搬の結果はリスタート後であっても論理的帰結としては正しいままです。また、phase savingにより、決定の際に選ばれるphaseもリスタートの前後で同じものになるはずです。

以上の観測から、リスタートを行うときに必ず決定レベル0にバックジャンプするのではなく、リスタート前とは異なった推論が行われはじめる決定レベルまで巻き戻すのにとどめれば、計算の重複を減らせることがわかります。それが**部分リスタート (Partial restart, Trail reuse)**です。[Ramos+, 2011]

11.1. 実装

アイデアは上で説明した通りなので早速実装してみます。まず、準備として決定レベルからその決定がどのようなものであったか（どのリテラルを選択したか）の情報が辿れるようにするためにグローバルな vector を一つ追加します。

```
vector<uint> decision; // for parital restarts
```

実際に改造するのは restart() 関数のみです。先程 backjump(0) としていた部分が少し複雑なコードに置き換わっています。日本語で処理の内容を説明すると、以下のような処理になっています。

(1) リスタート前の状態で次に選ぶことになるであろう変数を得る。(next_var)

(2) 決定変数をレベル 0 から順に舐めていき、next_var の方がスコアが大きければそこへバックジャンプを行う。

これで通常の決定レベル 0 に戻るリスタート (フルのリスタートと呼ぶことにします) と同じ効果が得られているかについては少しばかり考えておく必要があります。そこで、これまでのフルのリスタートで何が起こっていたかを考えてみます。まず大前提として restart() は単位伝搬が完了した後にのみ呼ばれます。つまり、その時点での (部分) 割り当ては直接は矛盾を導きません。もちろん探索を進めれば矛盾していることがわかるかもしれませんが、そのためには一度は何らかの新たな変数決定を行いそこから空節を導いて新たな節を学習することが必要です。加えて、単位伝搬が完了した後だということは全ての含意リテラルはすでに trail に登録されているということを意味しています。 よって、 リスタート前に割り当てられていた変数をリスタート後にどのような順で再度選択しても、 同じ割り当てが復元されるだけです。(phase も同じになります。) これはそこから単位伝搬を行っても変わりません。加えて、空節が導かれないので変数のスコアもリスタート前からは変わりません。つまり、リスタート前に未割り当てだった変数のどれかが決定されて初めて探索木の中の新たなパスの探索が始まるということです。しかも、そこで選択される変数はリスタート前の段階で (未割り当ての変数の中で) もっともスコアが高かったものと同じものです。

もう少しこの状況について考えてみます。今がリスタート直後でいくつかの決定を終えているとします。さらに、これまで決定した変数は全てリスタート前に割り当て済みだったものだ

とします。先程説明した理由からこの時点での割り当てはどう頑張ってもリスタート前の割り当ての部分集合にしかなっていません。ただし、どの変数が決定変数でどの変数が含意変数かについてはリスタートの前後で変わっている可能性があります。よって、どのタイミングで初めてリスタート前に未割り当てだった変数が決定されるかは、 リスタート前に割り当て済みだった決定変数と含意変数のうちどの変数がリスタート後の決定変数として選ばれたかに依存します。

この意味で、以下で実装した部分リスタートはフルのリスタートよりもアグレッシブなリスタートを行います。どういうことかというと、 リスタート後に next_var が決定されるより前のタイミングでは、フルのリスタートであればリスタート前に含意変数だったものが決定変数として選択される可能性がありますが、部分リスタートではリスタート前に決定変数だったものしか決定変数になりえず、しかも決定変数内の順番の入れ替わりもありません。結果的に next_var より前に割り当てられる変数の数 (つまりリスタートの前後で引き継ぐ知識の量) はフルのリスタートよりも部分リスタートの方が同じか少なくなるため、部分リスタートの方が未探索の領域を探索しやすいことになります。

```
bool restart() {
    if (restart_timer < restart_limit)
        return false;
    luby_seq = {
        (luby_seq[0] & -luby_seq[0]) == luby_seq[1] ? luby_seq[0] + 1 : luby_seq[0],
        (luby_seq[0] & -luby_seq[0]) == luby_seq[1] ? 1 : 2 * luby_seq[1]
    };
    restart_timer = 0;
    restart_limit = RESTART_BASE_INTERVAL * luby_seq[1];

    // 以下のコードを追加
    uint next_var;
    while (1) {
        if (heap_empty())
            return false;
        next_var = heap_top();
        if (! defined(next_var))
            break;
```

```
            heap_pop();
        }
        auto next_activity = activity[next_var];
        for (uint level = 0; level < decision_level; ++level) {
            uint var = decision[level + 1];
            if (activity[var] < next_activity) {
                backjump(level);
                return true;
            }
        }
        return false;
    }
```

12. その他の最適化

ここまで様々な最適化を施してきましたが、これでもまだ MiniSAT には遠く及ばない程度の性能です。小さなインスタンスならしばしば勝つことがある、ぐらいです。

本当はここで解説を終わらせるつもりだったのですが、ソルバーをいじっているうちに勝ちたくなってきたので、ここまでのソルバーの上にさらに以下を実装しました。

(1) conflict clause minimization (ccmin)

(2) in-process simplification

(3) literal block distance (LBD)

1 は CDCL に関連するものです。 学習節はしばしば不要なリテラルを含むことがあります。不要というのは、ここでは他のリテラルからの単位伝搬によって割り当てが (偽に) 決まるという意味です。conflict clause minimization は `analyze()` のなかで学習節を `db` に登録する前に節から不要なリテラルを取り除く処理のことを指します。

ccmin のアルゴリズムには松と竹があって、竹の場合は動作が軽量ですが削除するリテラルの量が少ないです。一方松の場合は処理が重くなりますがその分リテラルを多く削除できます。今回は両方実装しました。

2 は探索中に問題そのものの単純化を行うというものです。 この処理は決定レベルが 0 の割

り当てが積まれた時におこないます。通常 simplification というと CNF を充足可能性を保ったまま別の CNF に書き換える操作のことで、非常に重い可能性があるが、その分 CNF を大きく単純化できる操作を指します。通常はソルバーの実行前に一回動作させます。(preprocessing) simplification は重い操作なので探索中に頻繁に回すことはできないのですが、決定レベルが 0 の割り当てが増えた時にはその割り当てが巻き戻されることがないため、多少重い操作でも問題が簡単になるメリットの方が大きいことが多いです。今回は充足済みの節の削除と、偽なリテラルの削除を実装しました。

3 は今回最も効いた変更です。LBD は節の長さの一般化のようなもので、全てのリテラルが割り当て済みの節に対して定義される値です。その定義はというと、節の中のリテラルをそれが割り当てられた決定レベルでクラスタリングし、そのクラスタの数を LBD と呼ぶ、というものです。特に LBD が 2 の学習節は長さが非常に長くても単位伝搬を促進する可能性が高いです。LBD を節のスコアとして節の削除を行うことで、有用な節が残りやすくなります。

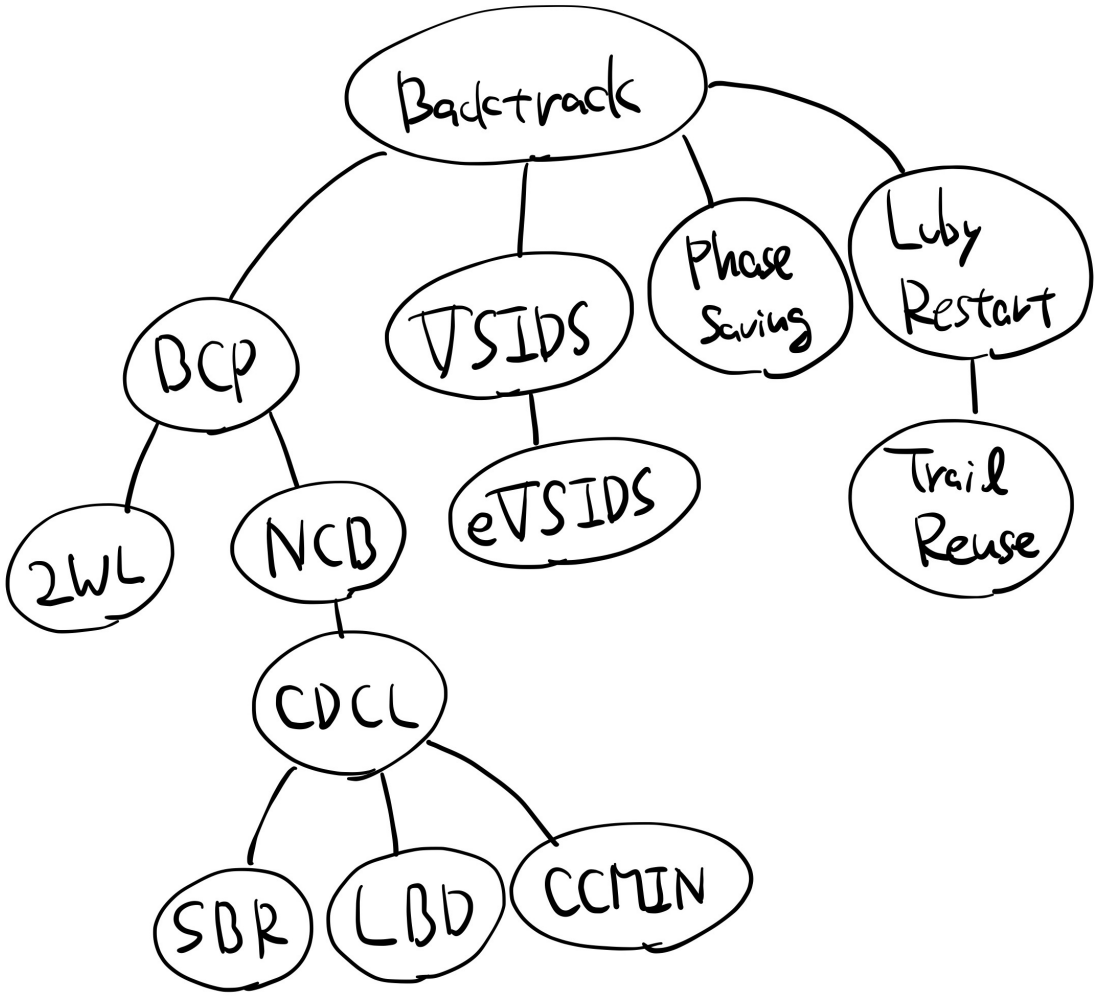
この三つを実装したところ、大抵のケースで MiniSAT に勝てるようになりました！ (やったね！)

ただし、これは MiniSAT のプリプロセスをオフにした場合の話で、これをオンにすると余裕で負けます。(相当難しい問題なら勝てるかもしれない) MiniSAT に堂々と勝ったと宣言できるためにはプリプロセスも実装しないといけません。

この三つについては具体的なコードの解説はここではしませんが、リポジトリにはそれぞれの実装がちゃんと上がっています。ここに書いた動機を理解した上でコードを読めばすぐに理解できると思うのですが、ccmin(松) だけはちょっとややこしいです。

13. まとめ

ここまでで高速 SAT ソルバーを実装するのに必要なテクニックを学んできました。記事冒頭で紹介した節の依存関係の絵をもう一度みてみましょう。大体概観がつかめたでしょうか？



SAT ソルバーのテクニックを解説した文章はいくつもネット上に転がっていますが、本記事のように、

(1) 動くコードと一緒に見ながら

(2) ここ数年で誕生した新しいテクニックも網羅しつつ

(3) コードがなぜ動くのかを徹底的に解説した

記事はそうないのではないでしょうか？というよりも、無いのが不満だったのでこの記事を書いたという動機があるので、この記事が上の三つを達成できていれば本望です。特に 3 については正直「ここまでコードのインバリアントや最適化が有用になる具体例について丁寧に文

章で書き下したものは世の中に存在しないんじゃないか」と思うぐらいかなり丁寧に説明したつもりです。(締め切り駆動特有のハイテンションな結びの言葉)

特に 2WL のインバリアントについてはとても力を入れました。原論文 [Moskewicz+, 2001] を含めてかなり幅広く論文や講義資料を漁ったのですが、2WL がなぜ動くかをちゃんと本記事ほど丁寧に書いている文章を見つけられませんでした。(英語・日本語どちらも見当たりませんでした。) Coq で 2WL の形式化を行っている [Fleury+, 2018] はかなり欲しいものに近かったのですが、すでに動いているアルゴリズムに対してインバリアントを天下り的に与えている (それが理由か本記事で議論したインバリアントとは違うインバリアントを採用している) のが微妙だったので、本記事では違う方針を取りました。そのおかげというかそのせいで形式的な議論に慣れていない人には少し追いづらかったかもしれません。(自分がハッピーなので OK の精神。)

他にも、NCB についての記述は本記事特有だと思います。 実は、NCB を単体で実装しているソルバーというのはかなり調べても全然見つかりません。(もしかしたら世の中にないものを生み出してしまったのかもしれない。) 見つけた範囲では全ての実装が NCB と CDCL を同時に実装していました。 例えば、[Ketabi+, 2011] では今回説明したような個別の最適化技法がそれぞれどのくらいソルバーの性能を向上させているかを測っているのですが、 ここでも CDCL と NCB は両方有効にするか両方無効にするかの二択になっています。(これについては [Nadel&Ryvchin, 2018] でも同じことが指摘されています。) [Lynce & Marques-Silva, 2002] には

> It would be interesting to distinguish between results obtained with non-chronological backtracking and with clause recording, since these techniques do not necessarily have to be jointly applied.

という注釈があり、当初から CDCL と NCB はあまり分離して考えられてはいなかったようです。ただし、最近は無条件に NCB を行うのは弊害が大きいという話も出ており [Nadel&Ryvchin, 2018]、近年見直しが進んでいるところのようです。

また、本記事 (というかリポジトリ) のように各実装技術がお互いに干渉しないように注意深く実装されているものも世の中にほとんど無いと思います。実のところ、 この記事のために 2 週間ぐらいひたすらソルバーのコードの歴史修正を繰り返していて、 何回も心が折れそうになったのですが、 おかげでコミットの差分がとてもわかりやすくなったと思います。 歴史修正を繰り返しまくった結果、 実はベースラインの実装が当初と比べてめちゃくちゃ速くなっ

てしまい、 それに合わせて記事を書き直す ... というサイクルが発生してしまっていたのもつらいポイントの一つだったのですが、 実装上の高速化とアルゴリズムそのものに起因する高速化を切り分けられたのは結果的にはよかったなと思います。 とにかく歴史修正が大変だったので、 ぜひ github にその成果を見に行ってください。URL はこちら：https://github.com/nyuichi/yabai-sat

さて、言及するのがめちゃくちゃ遅くなってしまいましたが、なんとこの記事は高速化の記事であるにもかかわらずベンチマークが一つも載っていません。もちろん当初の予定では各節にベンチマークの結果を載せるつもりだったのですが、実際のところまるで間に合いませんでした。すいません。マイクロベンチマークの結果はあるはあるのですがデータとしてまとめるには非常に不便です。 というのも、 この記事の実装を通して SAT ソルバーが高速化されすぎていて、一番最初の実装と一番最後の実装でまともな時間で解ける問題のサイズがあまりに変わっているのです。加えて、SAT の問題というのは色々な特性があり、さまざまな問題を解いてみてやっと全体の性能がわかるというのがあります。今回の実装範囲でまともな時間で動くさまざまな問題セットを集める、というのがなかなか難しく、色々後回しにしていたら締め切りになってしまいました。いつかそのうちちゃんと計測したいと思います。

この種の記事では最後の方に参考文献がまとまっているのが普通です。今回もソルバーを作るに当たって結構な数の論文を読んだんですが、残念ながら締め切りがやばすぎてあまりまとめきれませんでした。本当に重要なものについては略式ですが参照を文中に載せたので活用してみてください。

ちなみに、実装に当たって参考にしたコードは以下の通りです。

- li-sat-solver https://github.com/necavit/li-sat-solver/ ソルバーを作り始めた時に参考にしたやつ。コードが綺麗。機能は少ない。
- MiniSAT https://github.com/niklasso/minisat コードはあまり綺麗じゃない (個人の感想) ですが、アルゴリズムが参考になります。
- Glucose https://github.com/mi-ki/glucose-syrup (非公式ミラー ?) あまりみてないですが LBD の実装の確認とかに使いました。

あとこの節を書いてる途中に見つけた Rust の SAT ソルバーが割と良さそうだったので紹介しておきます。精読はしてません。

- https://github.com/shnarazk/splr

そろそろ書きたいこともなくなってきたのでこの辺りで記事を締めたいと思います。この記事をきっかけに SAT ソルバーを作ろうと思った方がいれば光栄です。この文章を読めば少なくとも MiniSAT をちぎるソルバーぐらいなら簡単に書けると思います。あるいはこの記事がすでに SAT ソルバーを作っている人にとって良い刺激になったり、より深い理解につながれば幸いです。多分そういう人は自分と感性が似ていると思うので今度飲みにいきましょう。それでは。

YABAITECH.TOKYO vol.6

2020 年 12 月 26 日 技術書典 10 版 (電子版)

2021 年 7 月 10 日 技術書典 11 版 (書籍版)

発行者　yabaitech.tokyo

Web サイト　http://yabaitech.tokyo

連絡先　admin@yabaitech.tokyo