SAT ソルバー入門

2015/03/23

JOI 2014/2015 春合宿

今日の内容

- 「難しい問題」を，実用的に解きたい
- 主に，「SAT ソルバー」を使って解く話をします
- 題材として，ペンシルパズルを解きます
  - 数独とか

なぜパズルか？

- ~~楽しい！！✌('ω'✌)三✌('ω')✌三(✌'ω')✌~~
- 現実的な制約問題などと比べて，ルールが比較的簡単に書ける
- また，多くのペンシルパズルは「答えの正当性のチェックは簡単」で扱い易い

「難しい問題」？

- 多項式時間で「解ける気がしない」問題
- NP 完全な問題などは多項式時間では解けないと考えられている
  - P≠NP 問題
  - 多項式で解けたら（解けないことを示しても）100 万ドルがもらえる
- 今回は「どうみてもやばそう」的なレベルくらいでも「難しい」と言うことにします

P，NP

- クラス　P，NP　は，判定問題(「～～かどうか？」という形の問題)たちのクラス
- クラス　P　の問題は，「多項式時間で解ける」問題
- クラス　NP　の問題は，「解ける証拠をくれれば，証拠の正当性を多項式時間で確かめられる」問題

NP 完全, NP 困難

- NP 困難問題は, どんな NP の問題よりもそれ以上に「難しい」問題
  - ある NP 困難問題が多項式で解ければ, NP のすべての問題が多項式で解ける
- NP 完全問題は, NP 困難かつ NP な問題

難しい問題を解く方法？

- まじめに解くのを諦める
  - ヒューリスティック的な解法
  - 近似解法
- がんばってまじめに解く
  - 全探索をがんばる
  - SAT ソルバーに投げる ←今回のメイン

ヒューリスティック

- 解けそうなところだけ解く
- パズルがやりたいならかなり有効な手法
  - 人間が解く方法をまねる
  - 自動生成にも便利
- 問題によっては解けない
  - が，探索と組み合わせるとだいぶ強くなる
- かわりに速度は速く（多項式にも）できる

SAT ソルバーで解く？

- 「SAT ソルバー」が解ける形にしてから SAT ソルバーで解く
- 「問題によっては（難しすぎて）解けない」，みたいなことは基本的にない

SAT ソルバーとは？

- 「充足可能性問題」を解くソルバー
- （ブール論理の）論理式を与えると，それを真にする変項の割り当てを探す

論理式とは？

- プログラミングにおいて，
  （x && y）|| !z
  みたいな式は日常的に書くと思います
- ここでは，bool 型の変数たちと，論理演算子 &&,||,! で構成される式と思ってもらえば十分です
- 今回は論理記号にもこれ（&&,||,!）を使います

充足可能性問題

- 論理式が与えられる
- 変数の値 true/false をうまく選んで，式の計算結果を true にできるか？を判定する
    - 判定するだけだと不便なので，SAT ソルバーは「可能」のときは変数の割り当ても教えてくれる

例

- `(!x || y) && (x || (y && z))`
- この場合は,
  - `x == false`
  - `y == true`
  - `z == true`
  - などが条件をみたす

例

- `x && y && (!x || !y)`
- これはどうやっても無理！
  - `x == true, y == true` が必要
  - すると `!x || !y` は `true` にならない

充足可能性問題の理論的性質

- NP 完全
  - 世界で初めて NP 完全だと示された問題
- なので，多項式で解ける気がしない
- 逆に言えば，充足可能性問題が速く解ければ他の問題にも応用ができそう

SAT ソルバーの特徴

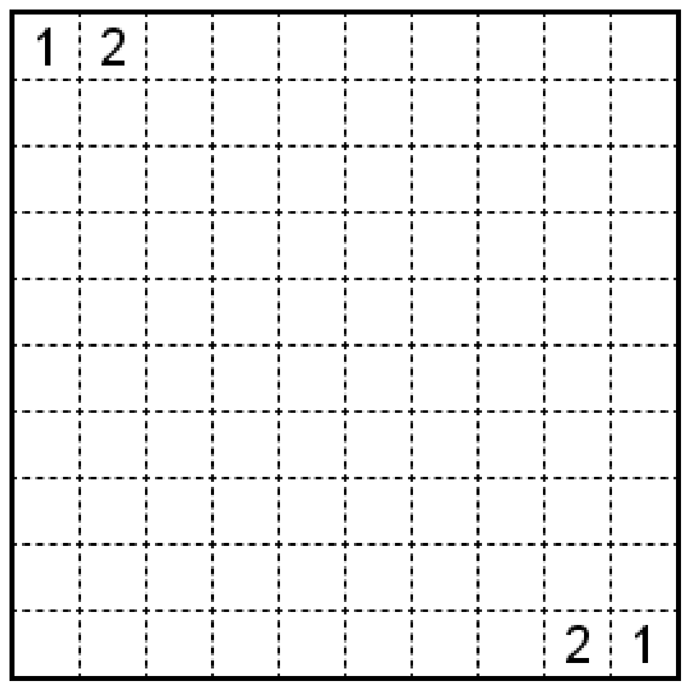
- 最近の SAT ソルバーは，速い！
  - 問題ごとに下手に専用ソルバーを作るよりも，SAT ソルバーを使ったほうが速いことも多い
- 連言標準形(CNF)で与えないといけないことが多い
  - (a || b || !c || …）みたいな節たちを && で結んでできる式
  - よく使われる CNF フォーマットがある

弱点

- 最適化問題は苦手？
- SAT ソルバー単体だと，表現のレベルが低すぎて扱いにくいことも
  - CSP（制約充足問題）に還元するともう少し扱いやすい
  - CSP だと整数なども扱える
  - それでも，かなり論理レベルに近い定式化が求められることには変わりない

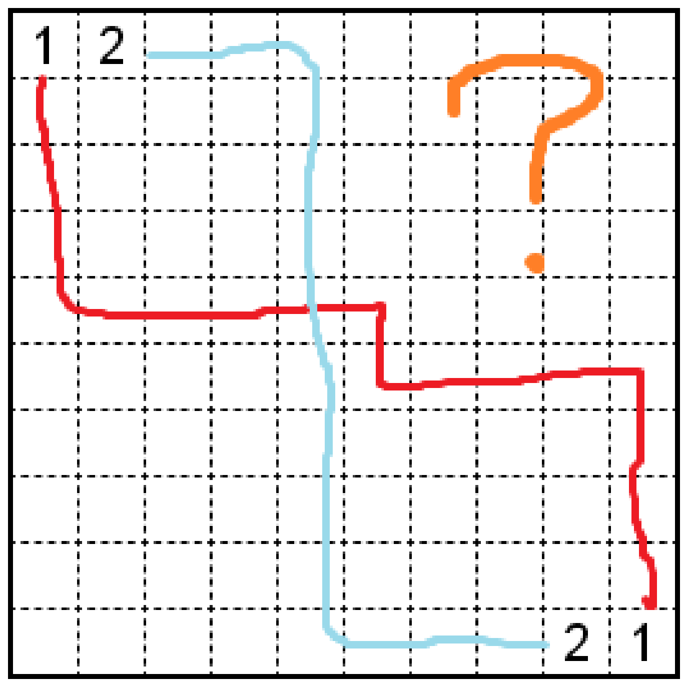
弱点

- 位置関係で絞り込みをするなどは苦手
- 下図の 1 同士，2 同士を交差させずに結ぶのは明らかに無理



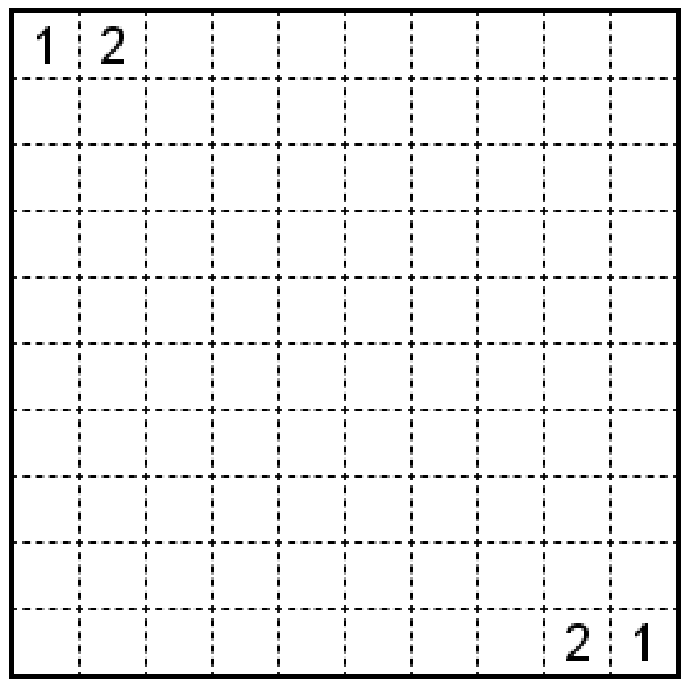
弱点

- 位置関係で絞り込みをするなどは苦手
- 下図の １ 同士，２ 同士を交差させずに結ぶのは明らかに無理



弱点

- しかし，SAT に帰着して解く(解がないことを確かめる)とかなり時間がかかる



CNF ファイル

- 1 行目に，変数の数と節の数を書く
  - p cnf (変数の数) (節の数)
- 2 行目以降は，各節を書く
  - 変数の番号は 1-origin
  - 1 以上の数を書くとその変数そのまま
  - 負の数を書くと，対応する変数の否定
  - 0 で節の終端
  - 1 2 -3 0 は $v_1||v_2||!v_3$ に対応

例

```
p cnf 4 10
1 2 3 0
-1 -2 0
-1 -3 0
2 -4 0
-2 4 0
-1 -3 -4 0
1 3 0
3 4 0
1 -4 0
-1 4 0
```

SAT ソルバーたち

- clasp
- Glucose
- MiniSat
- PicoSAT
- …

問題を解くために

- 分かりやすくするため，論理式に → 「ならば」という新たな記号を付け加えます
- A → B は !A || B と等価，と思う

新たな記号 →

- A → B とは !A || B
- これは「A ならば B」を表現できる
- A, B の値によって A → B は次のようになる

A	B	A → B
false	false	true
false	true	true
true	false	false
true	true	true

新たな記号 →

- 「A → B」が成り立つとき，
  - A が true なら B も true
  - A が false なら B はなんでもいい
- 数学の「ならば」とまったく同じ気分

→ を用いた例

- A == B の表現
  - これは「(A → B) && (B → A)」
  - 書き直せば (!A || B) && (A || !B)
- それ以外にも，普通に「～ならば…」という気分で論理式を書きたいときに使える

論理式の性質

- 交換法則
  - (A && B)==(B && A)
  - (A || B)==(B || A)

論理式の性質

- 結合法則
  - ((A && B) && C)==(A && (B && C))
  - ((A || B) || C)==(A || (B || C))
- この性質があるので，上の 2 つはそれぞれ
  - A && B && C
  - A || B || C
- と書けば十分（紛らわしくならない）

論理式の性質

- 分配法則
  - ((A && B) || C)==((A || C) && (B || C))
  - ((A || B) && C)==((A && C) || (B && C))
- 「→」についての分配法則
  - (A → (B && C))==((A → B) && (A → C))
  - (A → (B || C))==((A → B) || (A → C))

論理式の性質

- De Morgan の法則
  - `(!(A && B)) == (!A || !B)`
  - `(!(A || B)) == (!A && !B)`
- その他，当たり前のこと
  - `(!A || A) == true`
  - `(!A && A) == false`
  - `(!!A) == A`
  - …

SAT ソルバーを使う手順

1. 問題を論理式に変形する
2. 論理式を CNF に変形する
3. SAT ソルバーに論理式を解かせる
4. 得られた結果をもとに元の問題の解を復元する

問題 → 論理式

- 決まった手順はない
- 問題の「答え」となるパラメータは変数にすることが多い
- 答えには直接関係しないパラメータも変数にしないといけないことも多い

問題 → 論理式

- 規模が大きいと変換はプログラムじゃないと手に負えない
- プログラムの実装の簡単のため，あえて「無駄な」変数を用意するのも手
  - 必ず `true / false` な変数など
  - SAT ソルバーは頭がいいので，そういうことをしても性能にはほとんど影響しないはず

論理式 → CNF

- 最初から CNF を目指して論理式を書いてもよい
- 論理式の性質を使ってがんばって変形
- 新たな変数をおいて，節の数の爆発を抑えることも

例 1: 犯人は誰だ？

- 例題として，次のつまらない問題（自作）を考える
- A「B，C の少なくとも一方は犯人」
- B「D は犯人だ」
- C「A も D も犯人だ」
- D「A は犯人ではない」
- 犯人は必ず嘘をつき，犯人以外は必ず正直だとします
- 犯人は誰だ？（1 人とは限らない）

例 1: 犯人は誰だ？

- 問題を論理式で表現
- 変数は A，B，C，D とする
  - A，B，C，D のそれぞれが犯人かどうか
- 制約を 1 つずつ論理式にしていく

例 1: 犯人は誰だ？

- A「B, C の少なくとも一方は犯人」
  - (!A) == (B || C)
- B「D は犯人ではない」
  - (!B) == (!D)
- C「A も D も犯人だ」
  - (!C) == (A && D)
- D「A は犯人ではない」
  - (!D) == (!A)
- これらを CNF に変換していく

例 1: 犯人は誰だ?

- A「B, C の少なくとも一方は犯人」
  - (!A) == (B || C)
  - (!A → (B || C)) && ((B || C) → !A)
    - (!A → (B || C)) は, A || B || C
    - ((B || C) → !A)
    - !(B || C) || !A
    - (!B && !C) || !A
    - (!B || !A) && (!C || !A)
  - 結局,
  - (A || B || C) && (!A || !B) && (!A || !C)

例 1: 犯人は誰だ？

- B「D は犯人ではない」
  - (!B) == (!D)
  - (!B → !D) && (!D → !B)
  - (B || !D) && (!B || D)

例 1: 犯人は誰だ?

- C「A も D も犯人だ」
  - (!C) == (A && D)
  - (!C → (A && D)) && ((A && D) → !C)
    - (!C → (A && D)) は, 結局
    - (C || A) && (C || D) になる
    - ((A && D) → !C)は !A || !C || !D になる
  - 結局,
  - (!A || !C || !D) && (A || C) && (C || D)

例 1: 犯人は誰だ？

- D「A は犯人ではない」
  - (!D) == (!A)
  - (A || !D) && (!A || D)

例 1: 犯人は誰だ?

- 4 つの式を変形してできた論理式を && で結ぶと, 解くべき式が得られる
  - (A || B || C) && (!A || !B) && (!A || !C) && (B || !D) && (!B || D) && (!A || !C || !D) && (A || C) && (C || D) && (A || !D) && (!A || D)

例 1: 犯人は誰だ？

- この式を SAT ソルバーに与えると，答えが返ってくる
  - A, B, D が false, C が true
- これ以外に解がないか検証したい
- 求まった解の否定も条件に加えて解かせる

例 1: 犯人は誰だ？

- !(!A && !B && C && !D)
- これは A || B || !C || D になる
- この式も加えて解かせると，解なし

- よって，C のみが犯人

例 2: 数独

- 9×9 の盤面に 1〜9 の数を入れる
- 各行, 列, ブロックに 1〜9 の数が 1 個ずつ入るようにしないといけない

例 2: 数独

- 例題(自作)

		6	1	2				
		7	3					
		8				2	9	3
			7		4		2	5
7								6
4	1		5		3			
2	8	5				7		
					6	8		
				9	7	1		

例 2: 数独

- 例題(自作)の解答

3	9	6	1	2	8	5	4	7
5	2	7	3	4	9	6	8	1
1	4	8	6	7	5	2	9	3
8	6	9	7	1	4	3	2	5
7	5	3	9	8	2	4	1	6
4	1	2	5	6	3	9	7	8
2	8	5	4	3	1	7	6	9
9	7	1	2	5	6	8	3	4
6	3	4	8	9	7	1	5	2

例 2: 数独

- 「マス（i, j）に数字 n が入る」を v_{ijn} で表す
- 各マスにちょうど 1 個の数字が入る
  - $v_{ij1}, v_{ij2}, \ldots, v_{ij9}$ のうちちょうど 1 個が true
  - $v_{ij1} || v_{ij2} || \ldots || v_{ij9}$ が true
  - $v_{ij1} \&\& v_{ij2}$ などは false
    → $!v_{ij1} || !v_{ij2}$ が true

例 2: 数独

- 同じ行に同じ数は入らない
  - 1〜9 が 1 個ずつ入る，というのは明示的に指定しなくても従う
  - v_{i1n}&&v_{i2n} などが false
    → !v_{i1n}||!v_{i2n}
- 列，ブロックについてもまったく同様

例 2: 数独

- 最初から入っている数字の制約
  - (3, 4) に 5 が入るなら v_{345} が true
  - このとき「(3, 4) に 6 は入らない」とかは指定しなくても十分

例 2: 数独

- これらの条件をもとに解かせてみる
- 人力で CNF を書こうとすると疲れるのでプログラムで生成

- （ここでプログラムを動かす）

例 3: ナンバーリンク

- 同じ数字のペア同士を線で結ぶ
- 線は縦横に引く
- 同じマスに複数の線が通ってはいけない

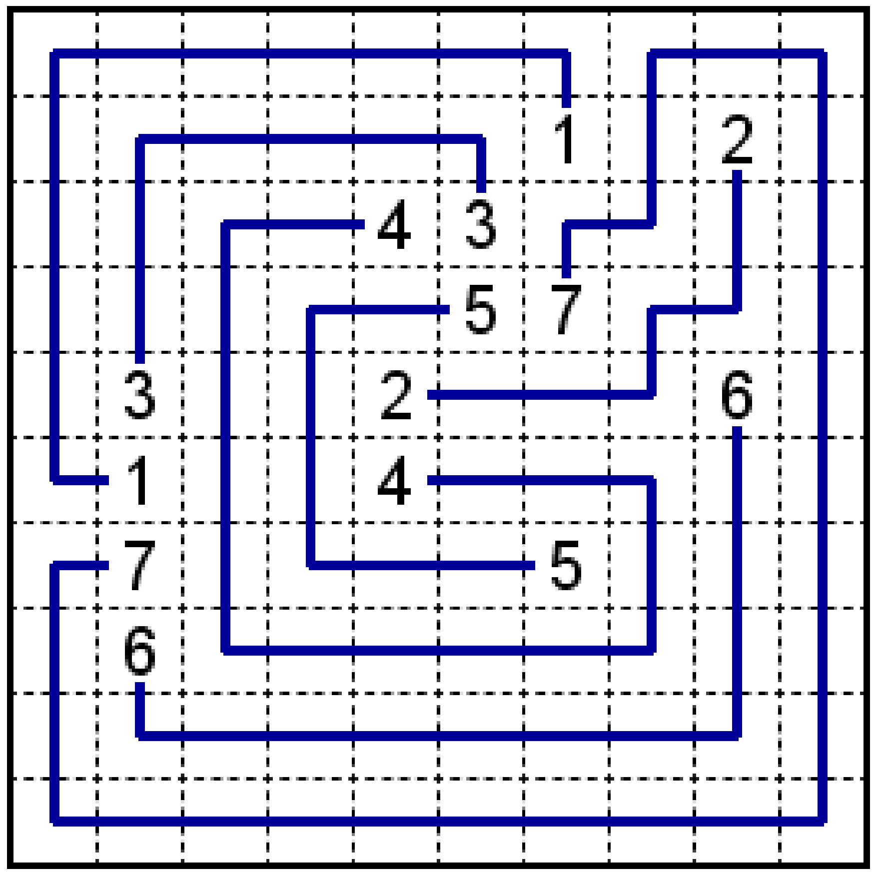
例 3：ナンバーリンク

- 例題（半自作）

						1		2	
				4	3				
					5	7			
	3			2				6	
	1			4					
	7					5			
	6								

例 3: ナンバーリンク

- 例題(半自作)の解答

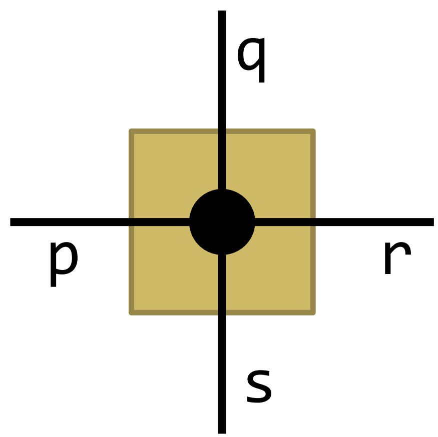


例 3: ナンバーリンク

- 使いうる線それぞれについて, それを使うかどうかを変数にする
- 各マスから出る線は 0 本か 2 本
  - 数字マスでは 1 本

例 3：ナンバーリンク

- あるマスを通る 4 つの線の候補について, この条件を考える
- 出る線を p，q，r，s とする



例 3: ナンバーリンク

- 数字マス(1 本だけ出る)なら簡単
- 1 本以上線が出る
  - p || q || r || s
- どの 2 本を選んでも, 両方の線を使うということはない
  - !(p && q) すなわち !p || !q
  - !p || !r, !p || !s, …

例 3: ナンバーリンク

- 普通のマス(0 or 2 本)は少し面倒
- 条件を単純に書き下すと

```
  (!p&&!q&&!r&&!s)
||( p&& q&&!r&&!s)
||( p&&!q&& r&&!s)
||( p&&!q&&!r&& s)
||(!p&& q&& r&&!s)
||(!p&& q&&!r&& s)
||(!p&&!q&& r&& s)
```

例 3: ナンバーリンク

- これを CNF にしなければいけない
- いきなり展開すると 4^7 個の項が出てきて大変
  - 実際はほとんど消える
- 条件の否定をベースに考えると見通しがよくなる

例 3: ナンバーリンク

- 出る線の数が 1 ではない
  - (p&&!q&&!r&&!s) ではない, すなわち !p|| q|| r|| s
  - p||!q|| r|| s
  - p|| q||!r|| s
  - p|| q|| r||!s
  - 結局上の 4 個の CNF 節にできる

例 3: ナンバーリンク

- 出る線の数が 3 ではない
  - `p||!q||!r||!s`
  - `!p|| q||!r||!s`
  - `!p||!q|| r||!s`
  - `!p||!q||!r|| s`
- 出る線の数が 4 ではない
  - `!p||!q||!r||!s`

例 3: ナンバーリンク

- 「論理圧縮」ができる
- (!p||!q||!r|| s)&&(!p||!q||!r||!s)
- (!p||!q||!r)&&(s||!s) と等価
- すなわち !p||!q||!r と等価
- 結局前ページの 5 節が 4 節になる:
  - !p||!q||!r
  - !q||!r||!s
  - !r||!s||!p
  - !s||!p||!q

例　3：ナンバーリンク

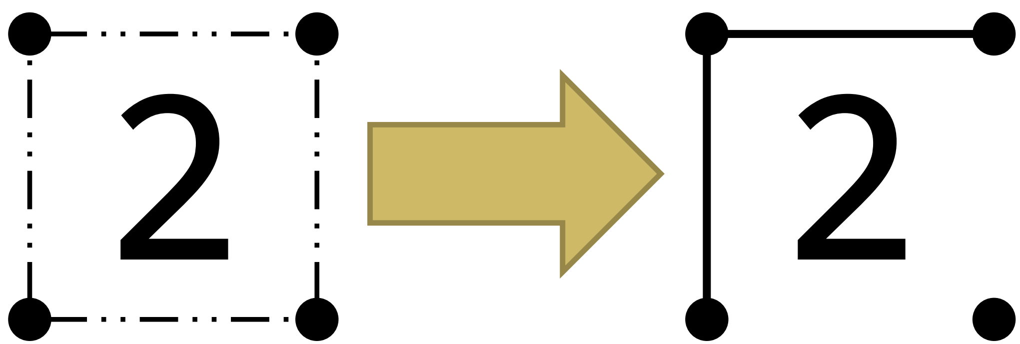
- 今までの条件だけだと，違う数字を結ぶ線ができてしまう
- 各マスについて，「線が通るとしたら，数字　n　の線かどうか」も変数にする
- 線　e　がマス　u，v　を結ぶとき，「e　→　(uの数字がn　==　vの数字がn)」が成り立つ

例 3: ナンバーリンク

- 「u の数字が n」を u_n で表します
- 「e → (uの数字がn == vの数字がn)」
  - e → (u_n → v_n)
  - !e || $!u_n$ || v_n
  - !e || u_n || $!v_n$
- 同じマスに複数の数字を割り当てない
  - n≠m なら, $!u_n$ || $!u_m$
- 数字 n が書いてあるマスは, 当然「数字 n の線が通る」としておく

例 4: スリザーリンク

- 盤面のいくつかの点を縦横に結んで 1 つのループを作る
- 書いてある数字は，その数字の周りを何箇所線が通るか



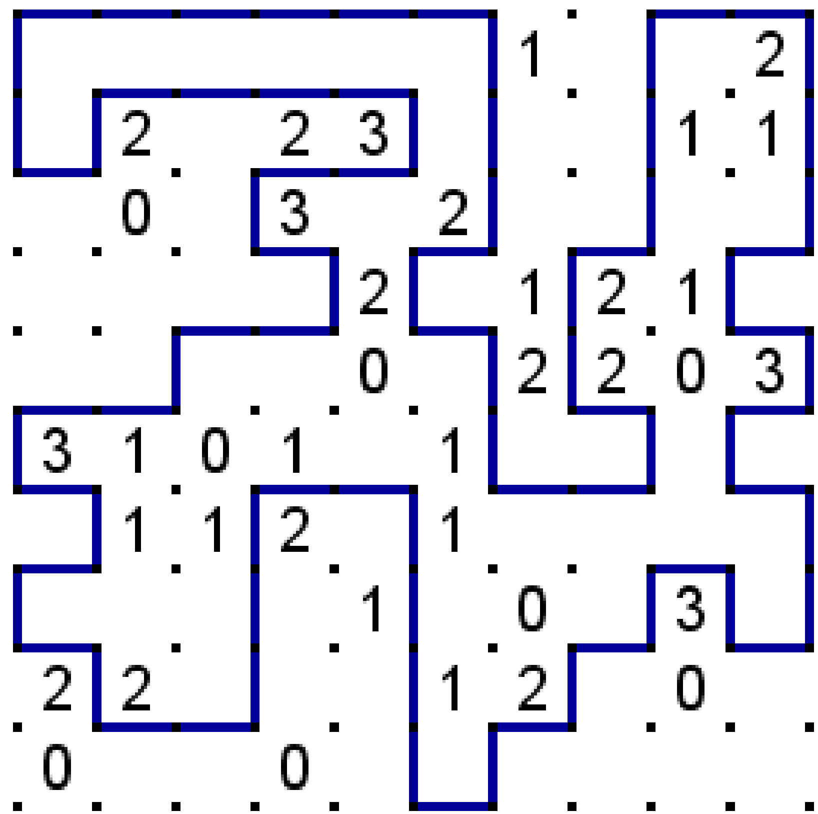
例 4: スリザーリンク

- 例題(自動生成)

						1			2
	2		2	3				1	1
	0		3		2				
				2		1	2	1	
				0		2	2	0	3
3	1	0	1		1				
	1	1	2		1				
				1		0		3	
2	2				1	2		0	
0			0						

例 4: スリザーリンク

- 例題(自動生成)の解答



例 4: スリザーリンク

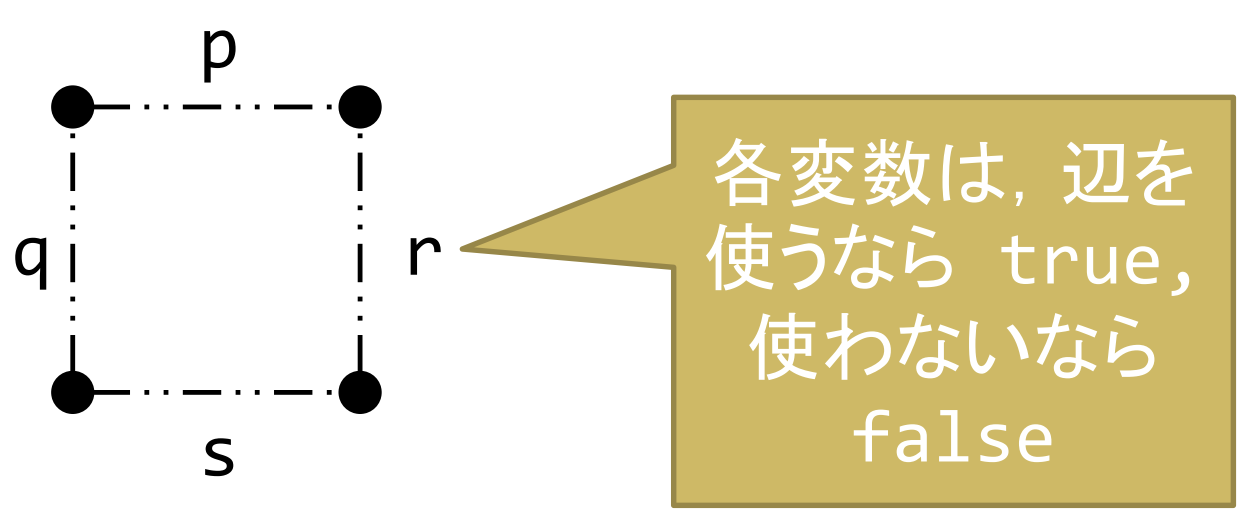
- ありうる線について，それが引かれるかどうかを変数にする
- 実はそれだけだと足りない（後述）

例 4: スリザーリンク

- 1 つのループを作るので，各点に対して，そこから出る線の数は 0 か 2
- ナンバーリンクのときとまったく同様に扱える

例 4: スリザーリンク

- 数字のまわりの 4 本の線(候補)のうち,使われるのはその数だけ
- 0,1,2,3 でそれぞれ別に考える



例 4: スリザーリンク

- 0 のときは明らか
  - !p && !q && !r && !s
- 1 のとき
  - p || q || r || s
  - !p || !q, !p || !r, …
- 3 のとき
  - 1 のときの逆
  - !p || !q || !r || !s
  - p || q, p || r, …

例 4: スリザーリンク

- 2 のとき
  - 「true の数は 0, 1, 3, 4 個ではない」という発想で論理式を組み立てる
  - p || q || r
  - q || r || s
  - r || s || p
  - s || p || q
  - !p || !q || !r
  - !q || !r || !s
  - !r || !s || !p
  - !s || !p || !q

「0, 1 個ではない」

「3, 4 個ではない」

例 4： スリザーリンク

- ループが 1 つにならないといけない，という条件がある
  - これが大変
- 連結性を SAT で扱うためのテクニックがあります

連結性を扱う方法

- 全域木を表すように変数を定める
- さらに，全域木は勝手な根を定めて有向木とする
  - 根かどうかも変数にする

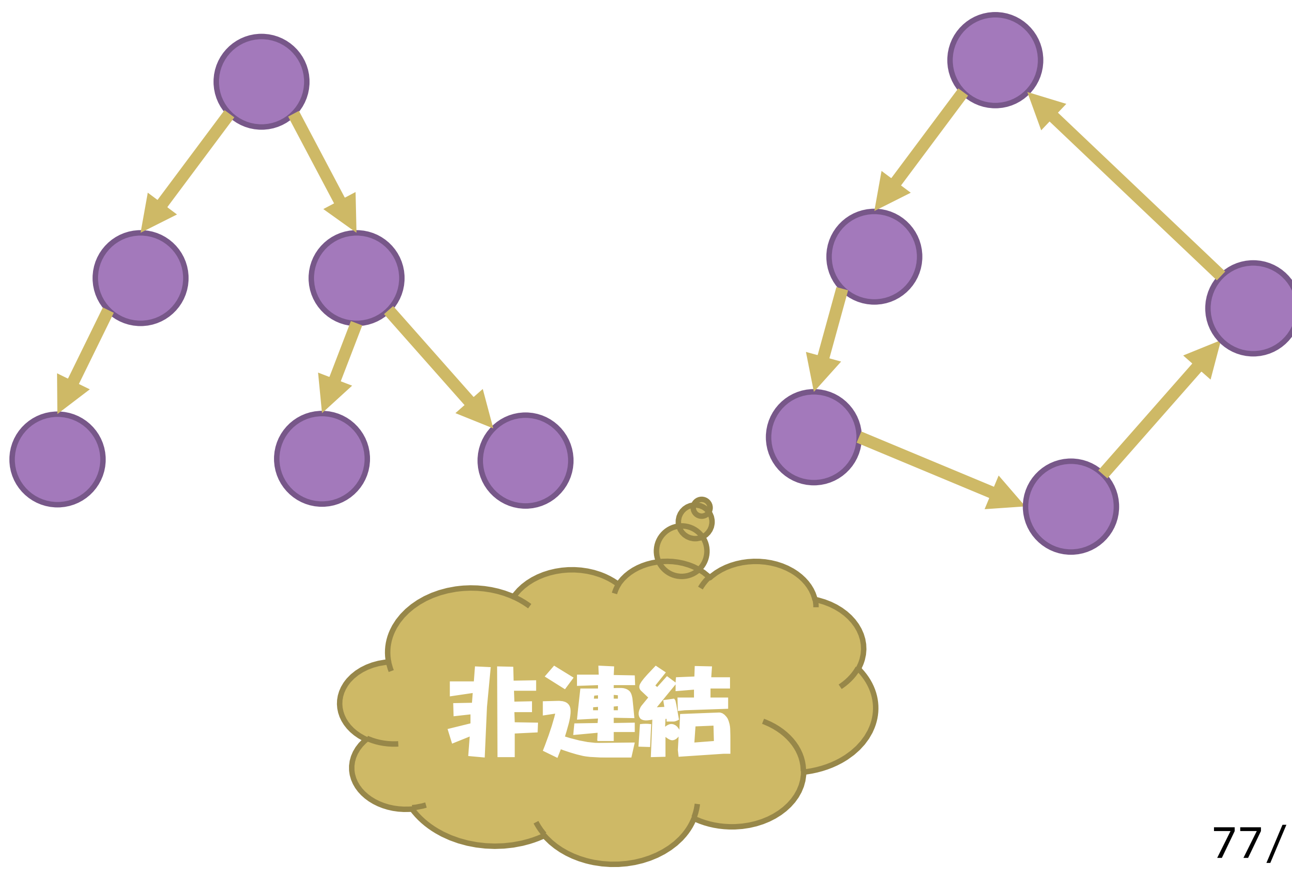
連結性を扱う方法

- 根には入ってくる辺があってはいけない
- 根以外には，入ってくる辺がちょうど 1 本なければならない
- 全体が連結なら，根がちょうど 1 個でなければならない

- 実は上の条件だけだと，ループを回避できない

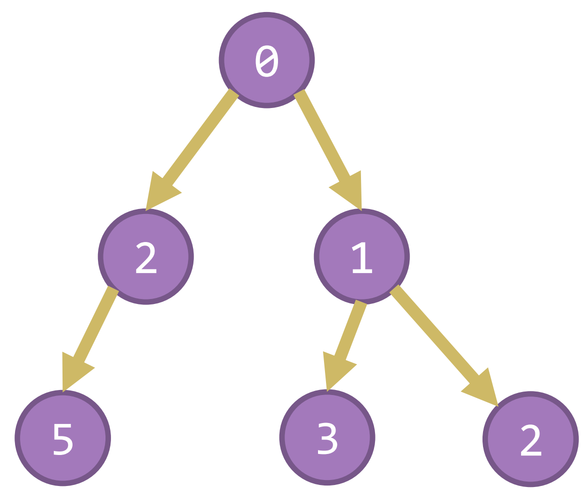
連結性を扱う方法

- だめな例



連結性を扱う方法

- ループを回避するために，各点に「高さ」の値を持たせる
  - 必ずしも根からの距離に一致する必要はなく，根から葉に向かって単調増加になっていればよい



連結性を扱う方法

- 結局持つものは以下のようになる
  - 有向辺それぞれについて，使うかどうか
  - 各点の高さ
  - 各点が根かどうか（高さが 0 かどうかで代用できる）

連結性を扱う方法

- 条件は次のようになる
  - 根に入ってくる辺は存在しない
  - 根以外の点については，その点に入ってくる辺はちょうど 1 個存在する
  - 辺 a->b について，(aの高さ)＜(bの高さ)

- 高さ（整数）の扱い方は後で述べます

例 4: スリザーリンク

- スリザーリンクの話に戻ります
- 各辺を使うかどうかは変数にするが，辺にも「向き」をつける
    - 使われる辺がどちら向きかを表す変数も用意
- 頂点には，整数の「高さ」の値を持たせる
    - 頂点の数を v として，高さは 0〜v-1 の範囲とすれば十分
- 無向辺としての条件は，今まで述べた通り

例 4: スリザーリンク

- 各点について，向き付き辺としての制約
  - ある点から複数の辺が出ない
  - ある点に複数の辺が入らない
- 高さの制約
  - 向き付き辺 a->b があったとき，b の高さは a の高さより大きいか 0
  - 高さ 0 は「ループの代表」なので，高さ 0 の点は複数あってはいけない

整数のエンコーディング

- 2 進エンコーディング？
- M 通りの値を表すために，「値が n である」のフラグを各 n に対して用意？
  - 等しいという条件は簡単
  - 大小比較が大変

- 「順序符号化」というものを使う

整数のエンコーディング

- M 通りの値を表すために,「値が n 以下である」のフラグを各 n に対して用意
  - 等しいという条件はやっぱり簡単
  - 大小比較も簡単
- CSP ソルバー「Sugar」で用いられている方法です
- 2 進エンコーディングと比べて, 変数の数は増えるが, 本質的な複雑さは変わらないので問題ではない

整数のエンコーディング

- 0〜9 の値を表す変数 v を考える
- n=0,1,…,9 に対して「v≦n かどうか」を表す変数 v_n を用意
  - v_9 はなくてもよい（常に true）

- v≦n なら v≦n+1 なので，n=0,1,…,7 に対して「$v_n \rightarrow v_{n+1}$」が成り立つ
- これで，ちょうど 10 通りの値が表現できる

整数のエンコーディング

- 0〜9 の範囲の整数 u,v がこの形で与えられている
- u≦v という条件を論理式にしたい

整数のエンコーディング

- u≦v なら，任意の n に対して「n≦u ならば n≦v」が成り立つ
  - そして，これが必要十分条件
  - n は 1〜9 を動かして考えればよい
- 整数なので，これは「!(u≦n-1) ならば !(v≦n-1)」と言い換えられる
- すなわち「$!u_{n-1} \rightarrow\ !v_{n-1}$」
  - $u_{n-1}\ ||\ !v_{n-1}$

整数のエンコーディング

- 結局, u≦v の条件は, 「n=0,1,…,8 に対して u_n || !v_n」となる
- これを応用すると, 「b → u≦v」なども簡単に表現できる
  - 各 n に対して !b || u_n || !v_n となる

参考文献

- http://bach.istc.kobe-u.ac.jp/sugar/puzzles/
「パズルを Sugar 制約ソルバーで解く」
    - いろいろなパズルを CSP(制約充足問題)ソルバーで解く方法が紹介されています
    - 今回紹介した例でも，ここに書いてある方法を参考にしました