Lean で正規表現エンジンをつくる

そして正しさを証明する

井山梃子歴史館 (pandaman64)

2025-06-15

発表の目的

- 内容
  - Lean とは？
  - なぜ Lean で定理を証明するのか？
  - プログラムの正しさを証明するとは？
- 仲間を探しに来た
  - みんなも定理証明、やろう！
  - （あわよくば）lean-regex にコントリビュート、しよう！

Lean とは？

Lean とは何か

- Lean の二面性
  - 純粋関数プログラミング言語
    - 依存型: 型の中に値が含められる
    - モナドを使った手続き型プログラミング
    - 自由なマクロシステム
      - 正規表現、HTML、SQL
  - 定理証明支援系
    - 数学の定理やプログラムの性質、それらの証明を記述する言語
    - Lean の**カーネル**が証明が成立することを厳密にチェックする

Lean のコード例

```
def fib (n : Nat) : Nat :=
  match n with
  | 0 | 1 => 1
  | n + 2 => fib n + fib (n + 1)
def main : IO Unit := do
  IO.println s!"fib 10 = {fib 10}"
```

プログラムの例

```
theorem reverse_reverse (xs : List α) :
  xs.reverse.reverse = xs := by
  induction xs with
  | nil => rfl
  | cons x xs ih =>
    simp [ih]
```

証明の例

```
def sumAt {n} (xs ys : Vector Nat n) (i : Nat) : Option Nat :=
  -- `h` is a proof that `i < n` holds
  if h : i < n then
    some (xs[i]'h + ys[i]'h)
  else
    none
```

証明を使うプログラムの例

Lean で正規表現を実装する

- lean-regex: 自作の正規表現ライブラリ
  - 正規表現をオートマトンにコンパイルして実行
  - Lean 上で実装の正しいことを検証済み
- 「実装が正しい」とは？
  - 正規表現のマッチ結果を厳密に定義する
    - `inductive Captures : Iterator → Iterator → CaptureGroups → Expr → Prop`
  - 検索関数 `def Regex.find : Iterator → Regex → Option CaptureGroups` について
    - ✅ 健全性: 見つかったマッチは `Captures` を必ず満たす
    - ✅ 完全性: `Captures` を満たすマッチが存在するなら必ずマッチを見つける
    - これらを示す Lean の証明を書いた

なぜ正規表現？

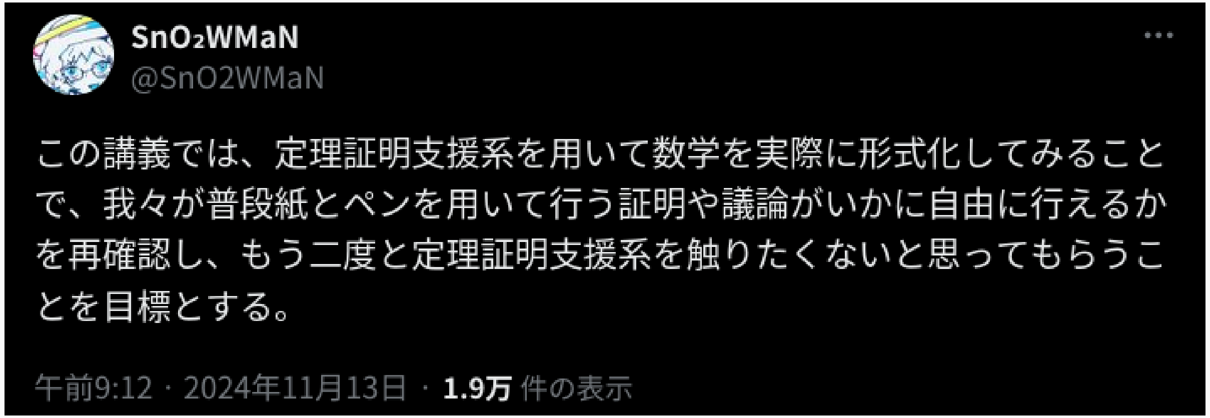
1. 正規表現は広く使われている
    - テキスト処理の場面でよく出てくる
    - 実用的なプログラミング言語には正規表現実装がつきもの
2. 仕様・実装がほどよく複雑
    - 検索関数の正しさを数学的に明確に定式化できる
    - 実装はオートマトンへのコンパイルや探索など、そこそこ複雑
    - エッジケースも含めて定理証明支援系で厳密に表現・検証する価値あり
3. パフォーマンスが重要
    - 大量のテキストを効率よく処理したい
    - Lean の最適化の出番！

Lean の最適化

- 2 つの実行モデル
  1. カーネルによるインタプリタ: 証明の検証・エディタでの実行など
  2. C へのコンパイル: Lean を C 言語に変換してネイティブコードを生成
- C へのコンパイル時はオブジェクトを参照カウンタで保持
  - 正格な純粋関数型言語なので（基本的には）参照サイクルが発生しない
- 参照カウンタを見るとデータ構造の更新を**破壊的変更**に最適化できる
  - 例: `let xs' := Array.set xs i v` のような操作が実質 O(1)で実行
    - 証明の検証時は `xs` と `xs'`の両方が同時に存在するかのように扱える
  - オートマトンベースの正規表現エンジンに最適

なぜ定理証明するのか？

なぜ定理証明するのか？

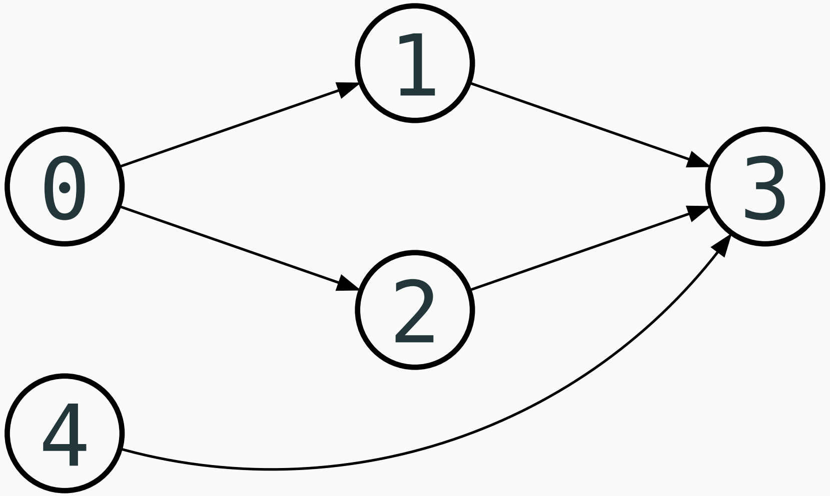
- 定理証明は苦しい…
  - 証明のコード量は実装の 2〜20 倍
  - 定理証明支援系のご機嫌取りでボイラープレートが増える
    - 

- それでもなぜやるのか？
  - 信頼性の保証: 暗号処理など、信頼性が要求される領域で確実な保証を得る
  - 実装の品質向上: 証明過程でバグを発見
  - パズル的な面白さ: 証明が通った瞬間の達成感は中毒性がある
  - **深い理解**が得られる: これが最も重要！

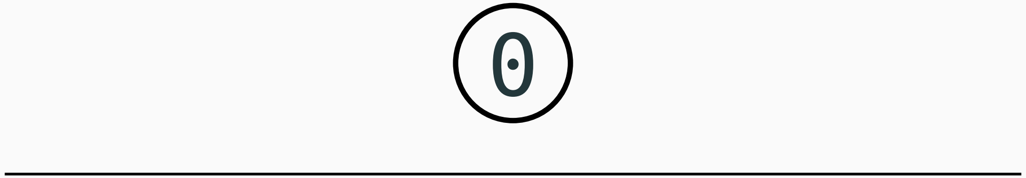
定理証明で得られる「深い理解」とは

- 証明を書くには**なぜ定理が成立するか**を理解しなければならない
  - 書いたプログラムがなぜ正しく動くのか？を深く理解する必要がある
- プログラムはなぜ正しく動くのか？ = **よい不変条件**が成立しているから
  - 不変条件: プログラムの各ステップ前後で常に成立している性質
  - プログラムの性質を証明するには
    1. 不変条件を見つける
    2. 各処理が見つけた不変条件を**保存**することを示す
    3. 見つけた不変条件が所望の性質を**導く**ことを示す
  - どうやって不変条件を見つけるの？
    - ~~頑張る💪~~
    - 具体例を計算したり欲しい性質から逆算したりする

例: DFS で到達可能性を計算する



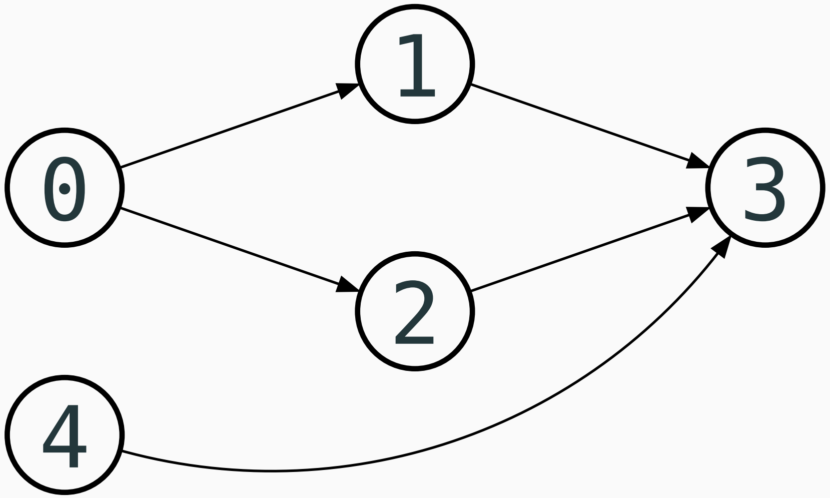
グラフの例。探索した頂点を赤く塗っている



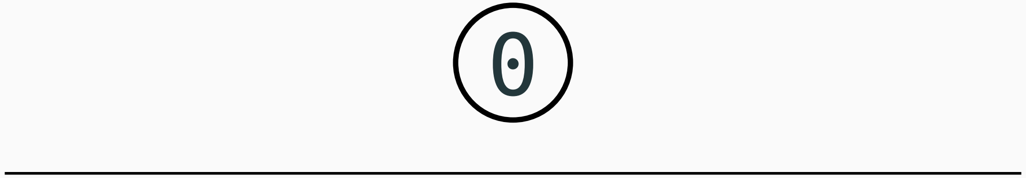
これから探索する頂点のスタック

- グラフの到達可能性: 頂点 `0` から到達できる頂点の集合は？
  - 正規表現のマッチ ≒ 正規表現をコンパイルしたオートマトンの到達可能性
- DFS（深さ優先探索）で到達可能性の判定ができる。なぜ？
  - DFS が**よい不変条件**を満たすから

DFS の不変条件



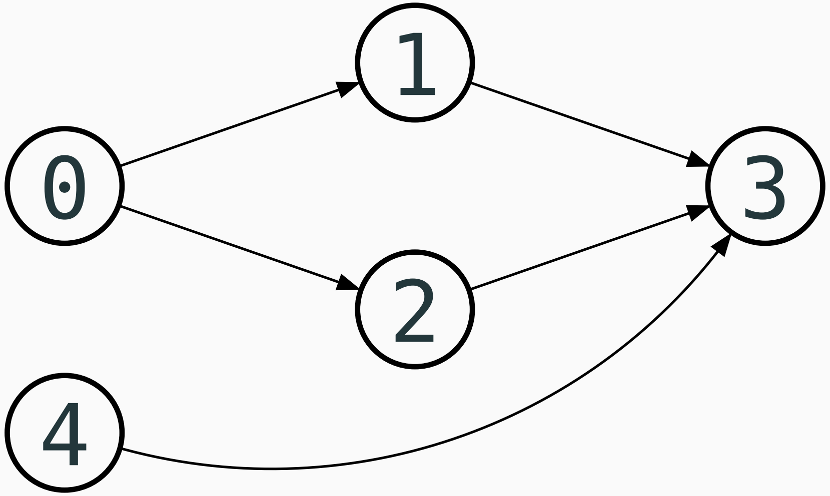
グラフの例。探索した頂点を赤く塗っている



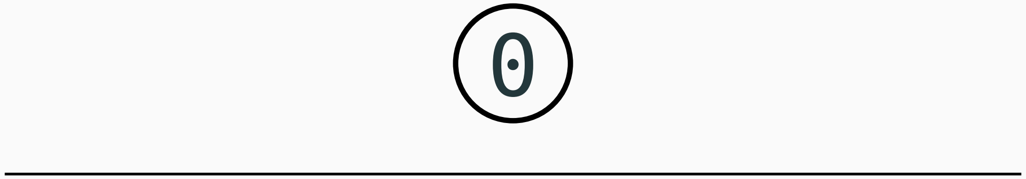
これから探索する頂点のスタック

- **不変条件**: 到達済みの頂点から 1 ステップ先の頂点は既に到達済み or スタック上

不変条件の保存



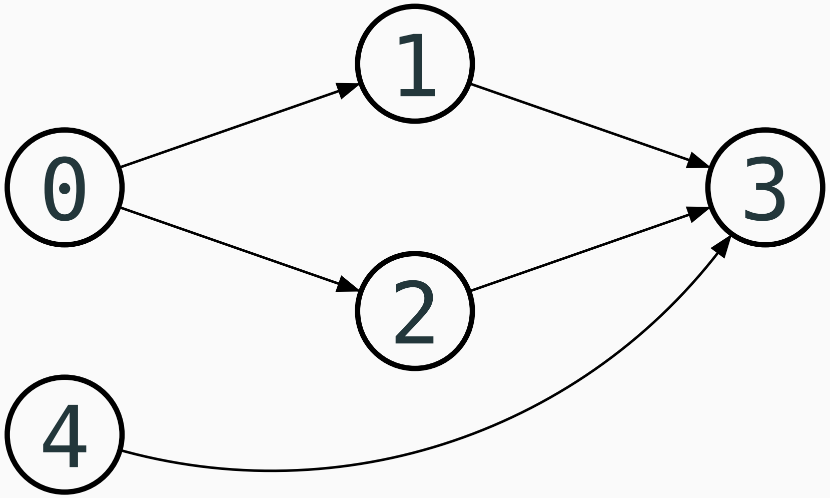
グラフの例。探索した頂点を赤く塗っている



これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

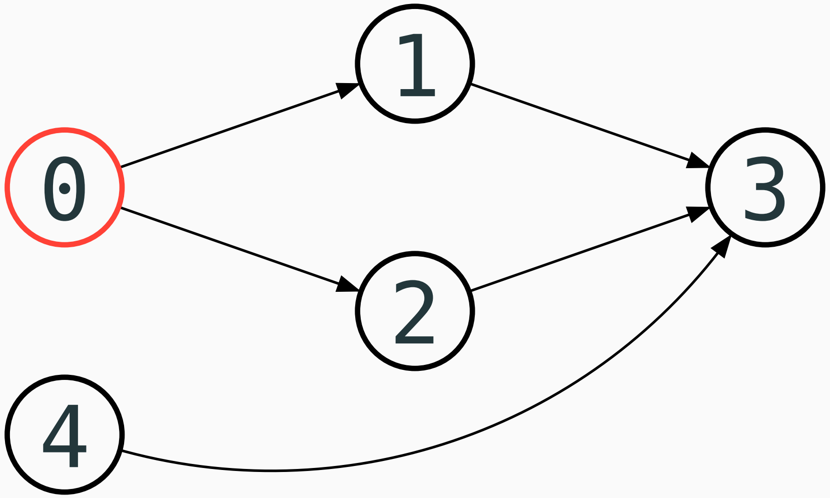


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

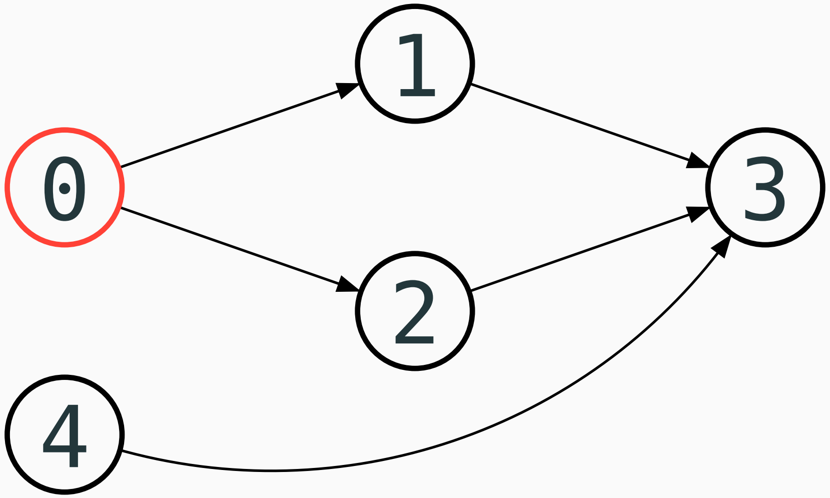


グラフの例。探索した頂点を赤く塗っている

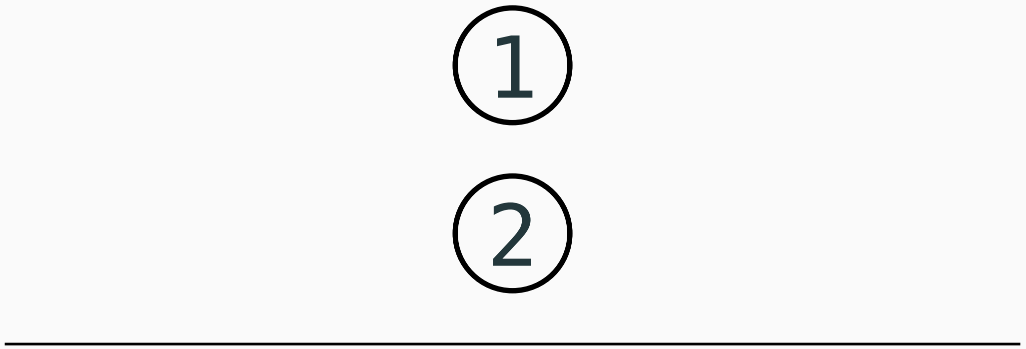
これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存



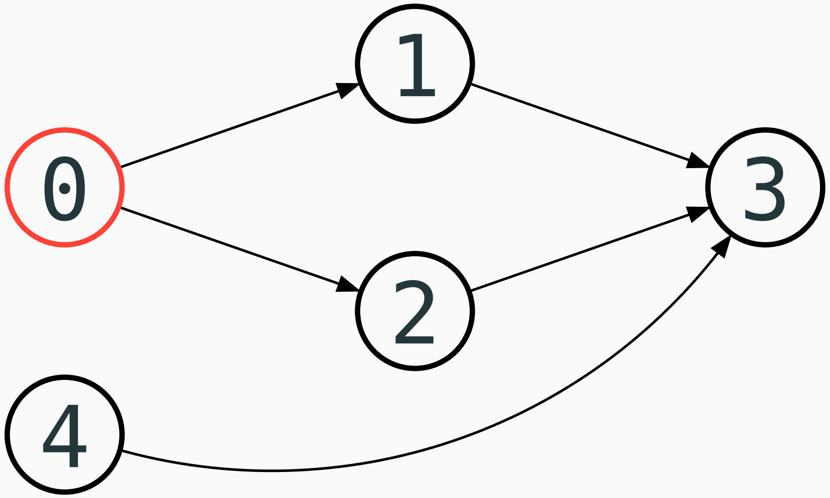
グラフの例。探索した頂点を赤く塗っている



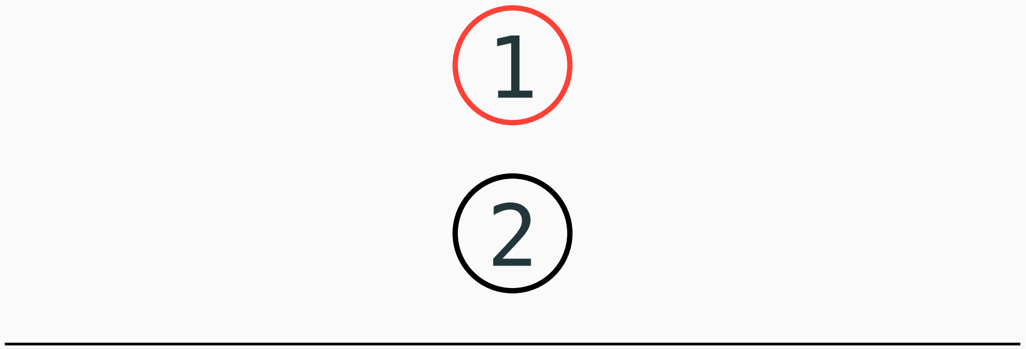
これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存



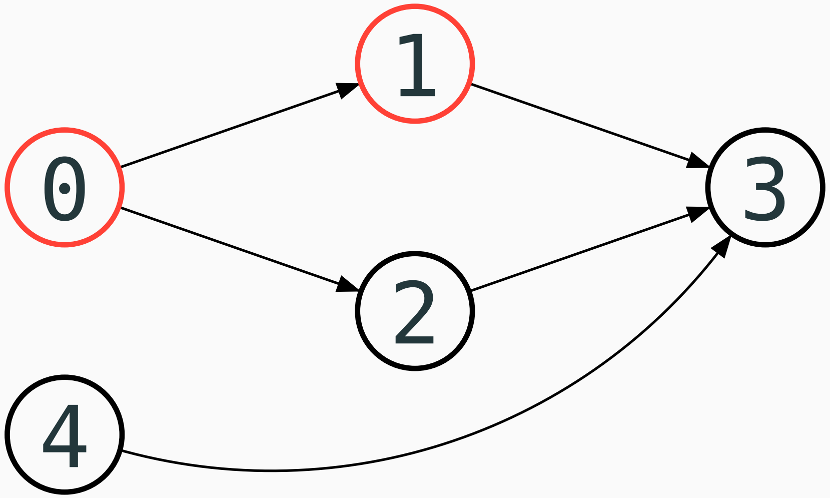
グラフの例。探索した頂点を赤く塗っている



これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

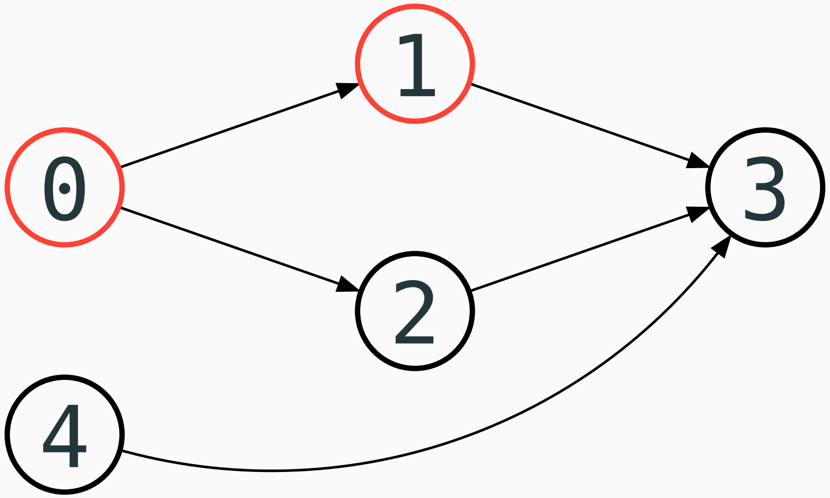


グラフの例。探索した頂点を赤く塗っている

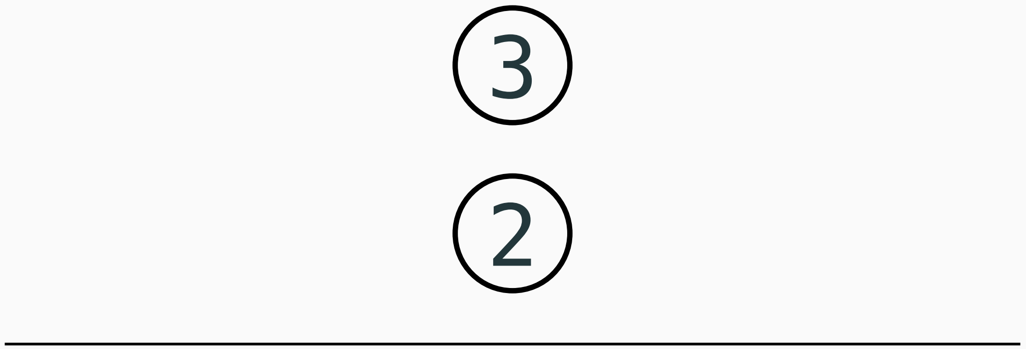
これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存



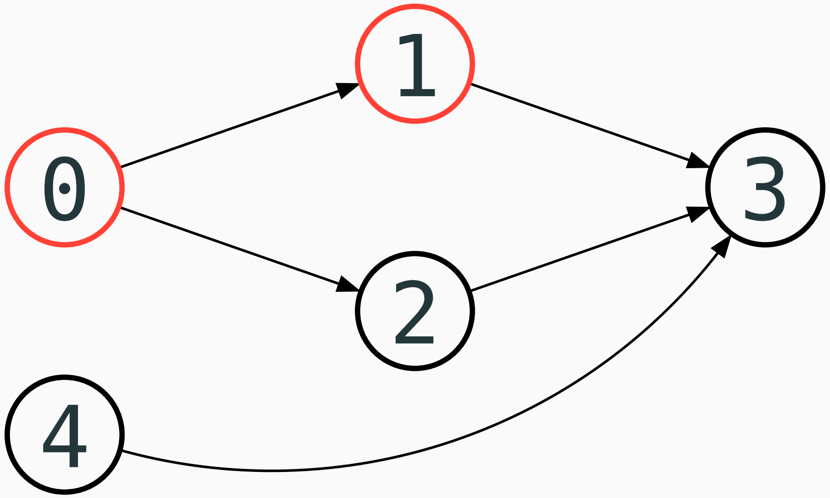
グラフの例。探索した頂点を赤く塗っている



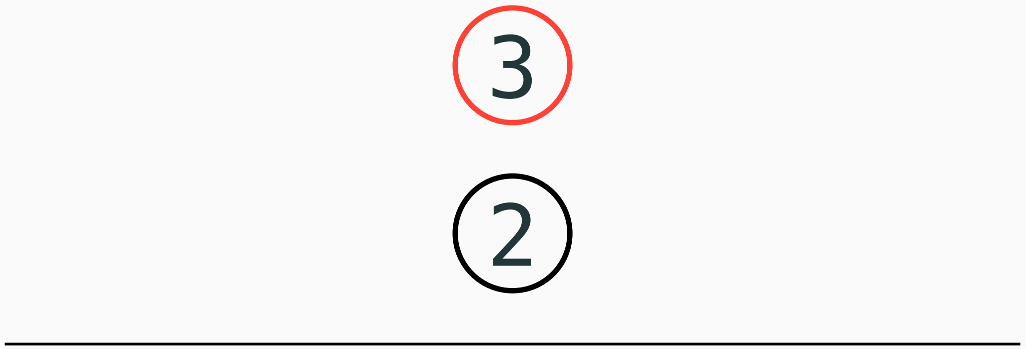
これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存



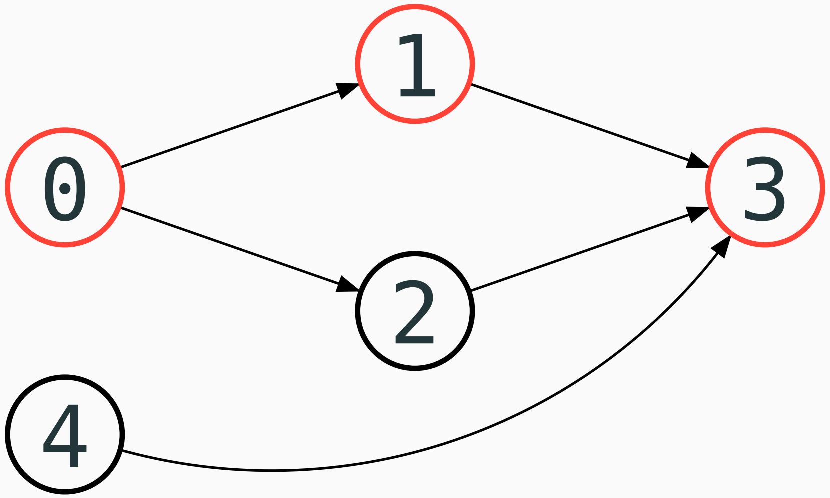
グラフの例。探索した頂点を赤く塗っている



これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

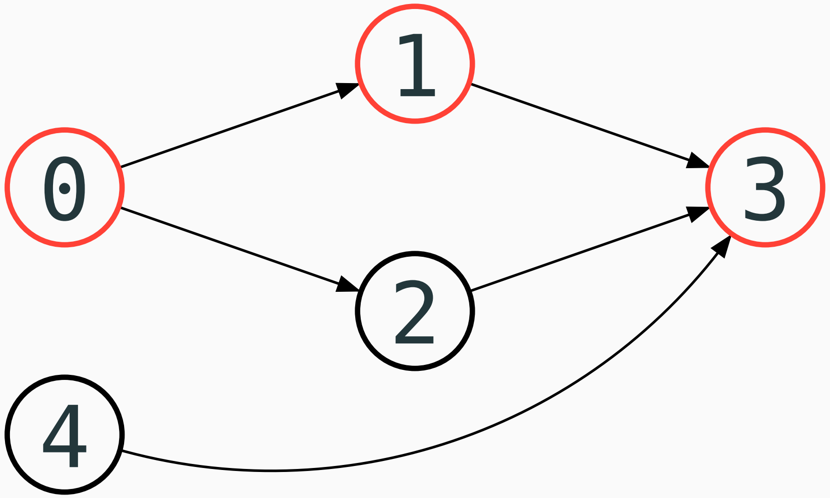


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

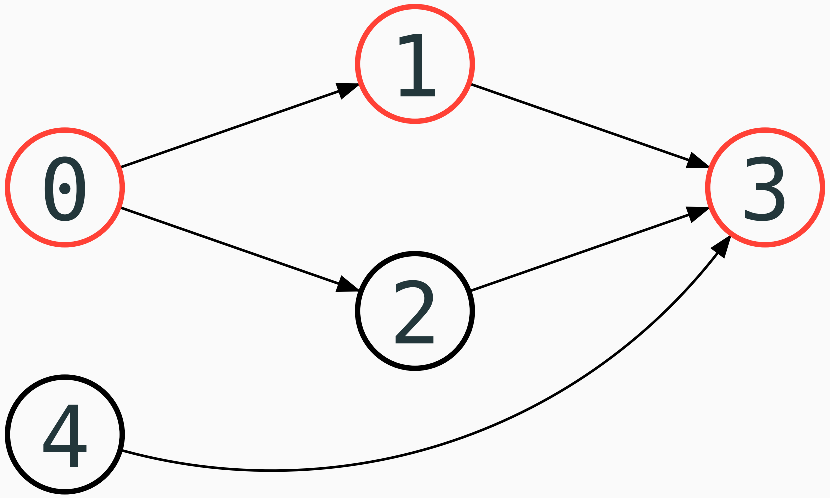


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

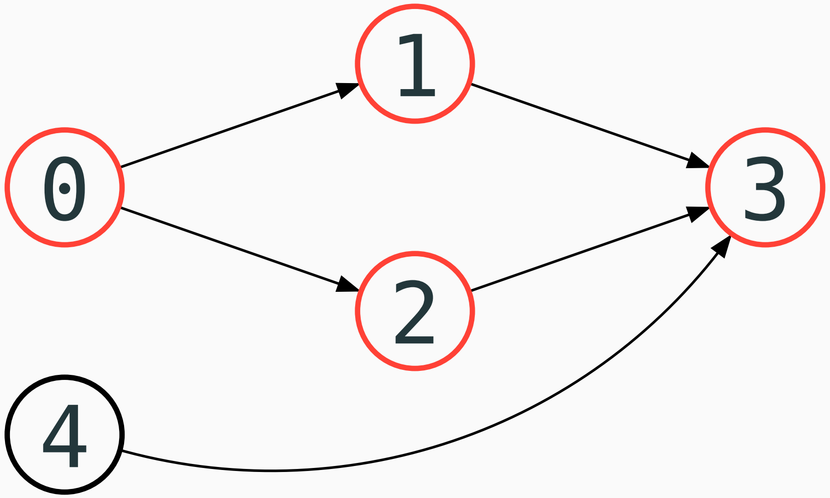


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

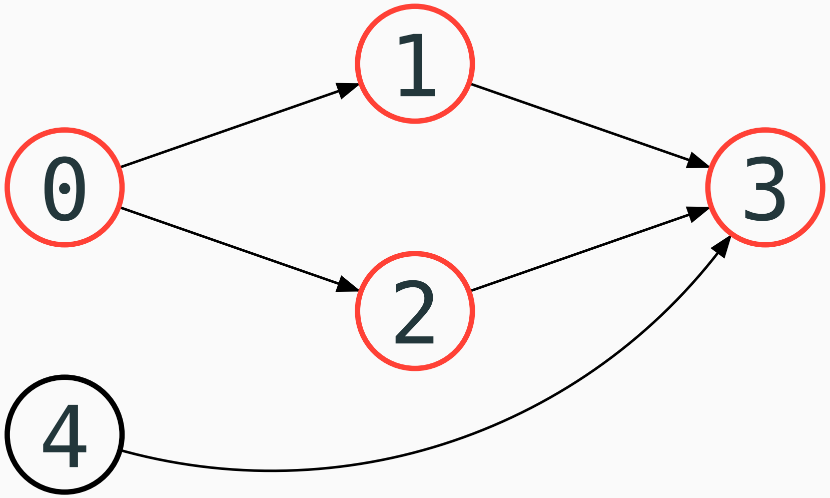


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

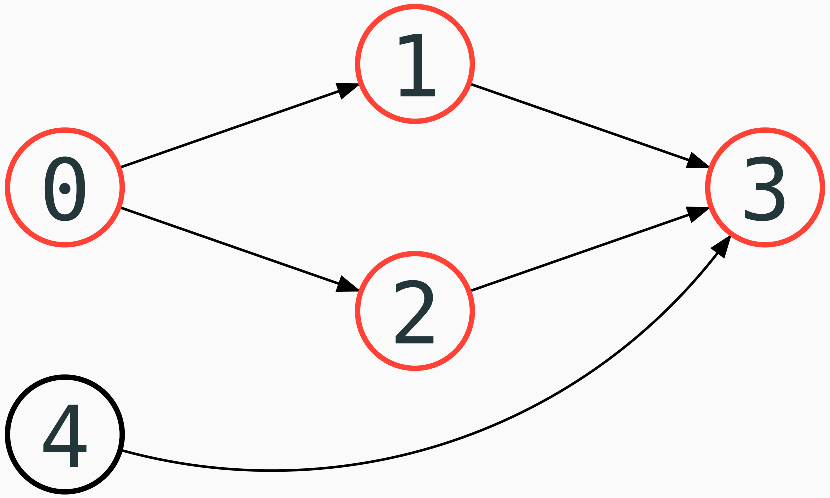


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件の保存

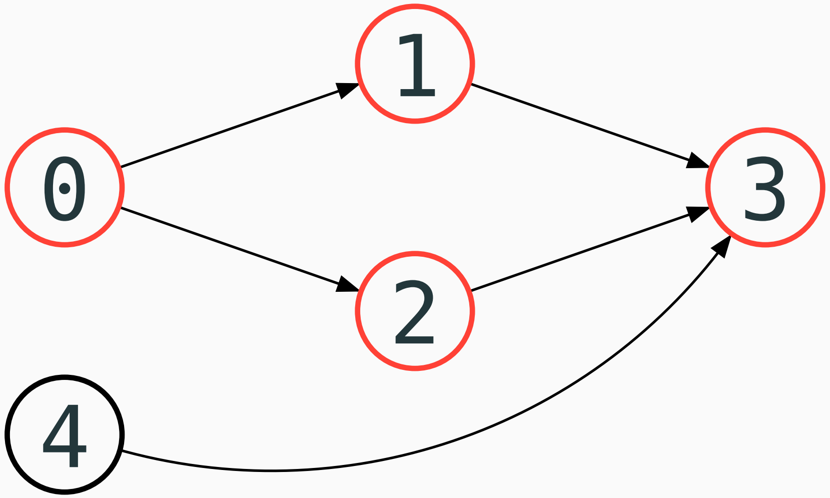


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上
  - DFS の各ステップが不変条件を**保存**する

不変条件が到達可能性を導く

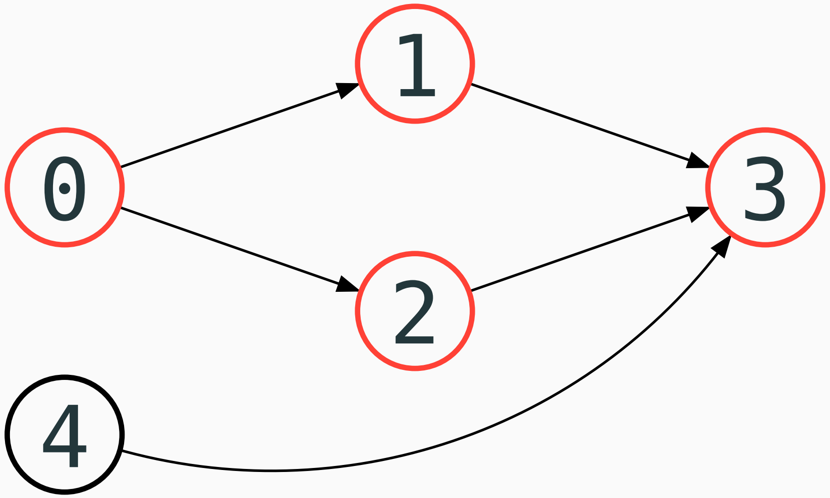


グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- スタックが空になったら計算が完了
- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上

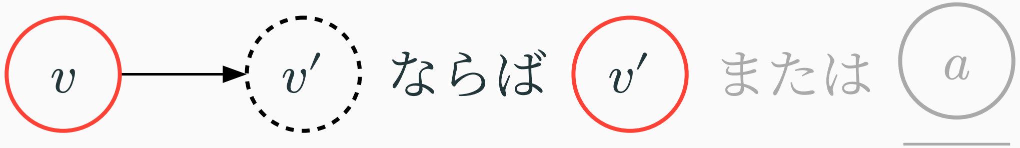
不変条件が到達可能性を導く



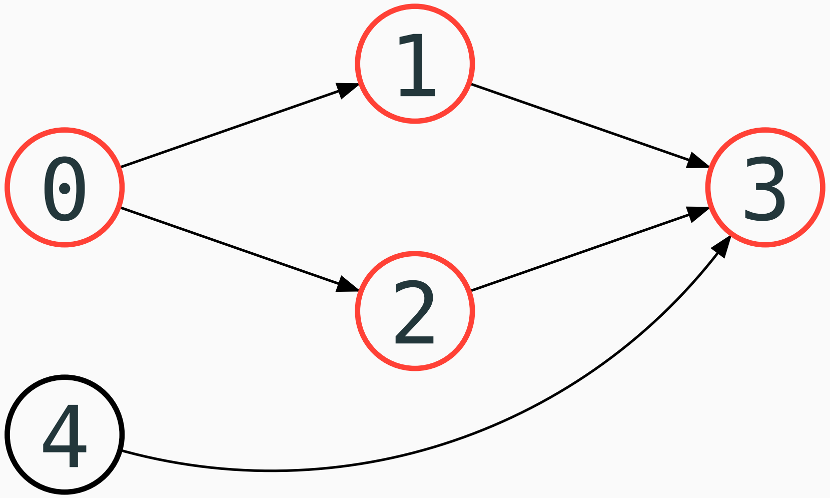
グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- スタックが空になったら計算が完了
- **不変条件**: 探索済みの頂点から 1 ステップ先の頂点は既に探索済み or スタック上



不変条件が到達可能性を導く



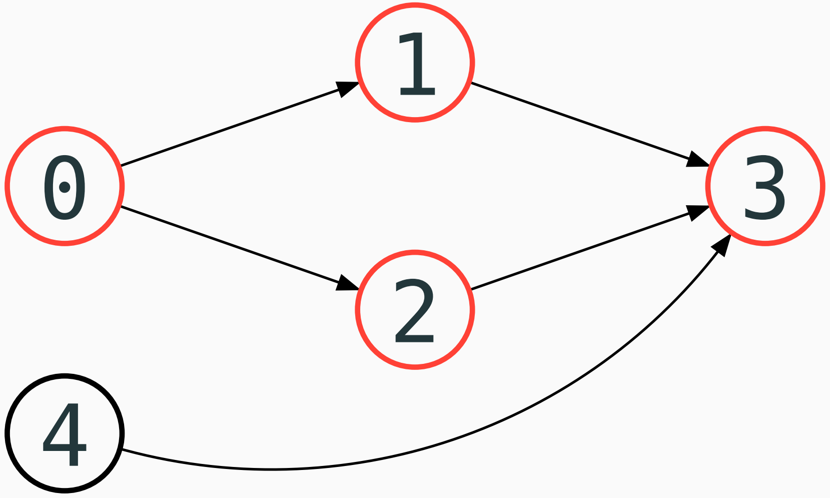
グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- スタックが空になったら計算が完了
- **不変条件**: 探索済みの頂点から **N** ステップ先の頂点は既に探索済み or スタック上

$v \xrightarrow{\mathbf{N}} v'$ ならば v' または a

不変条件が到達可能性を導く



グラフの例。探索した頂点を赤く塗っている

これから探索する頂点のスタック

- **不変条件**: 探索済みの頂点から **N** ステップ先の頂点は既に探索済み
  - つまり、始点から到達可能な頂点は探索済み
- 逆に、別の不変条件を使うと到達可能な頂点だけ探索することが分かる
- したがって、到達可能性を計算するには DFS でグラフを探索すればよい 🎉
  - **よい不変条件が DFS の正しさを導いた**

なぜ定理証明するのか？のまとめ

- 定理証明はたいへん苦しい
- 定理証明は定理の**深い理解**をもたらす
  - プログラムの深い理解 = **よい不変条件**を見つけること
- 定理証明はとても**やりがいがある**
  - 全てが繋がった瞬間の気持ちよさはとんでもない

証明をエンジニアリングする

証明をエンジニアリングする

- 定理証明の苦しみを軽減する
  - 問題を分割することで一度に扱う複雑さを低減する
  - コードを再利用してボイラープレートを減らす
    - 定理証明パターンがありそう
    - 例: ProofData で中間的な定義や証明を整理する
- ソフトウェア開発と同じ!!
- Lean 特有の苦しみ
  - `do` 構文は糖衣構文
    - 証明時は脱糖後の式について証明を書くことになる
  - **依存型**は用法用量にお気をつけて

依存型は諸刃の剣

- 依存型: 型の中に値を含められる
  - `let i : Fin 5` のとき、`i` は 5 未満の自然数
- メリット:
  - 型の表現力が上がる
  - パフォーマンス向上
    - `def Array.get : (xs : Array α) → Fin xs.size : α` は境界チェックしない
- デメリット:
  - 型チェックが複雑になる（値の等しさもチェックしないといけないため）
    - 明示的なキャストが必要な場合はコードが冗長になる
    - `((x : Fin (n + 1)).cast (...n + 1 = 1 + nの証明...)) : Fin (1 + n)`
  - 実質的に「等しい」値でも型システム的に等しくならないことがある
    - 例: `(3 : Fin 5) ≠ (3 : Fin 10)`

依存型の利用戦略

- 使うのは控えめに
  - 🙆 型に登場する値が変わらないとき（キャストが必要無いとき）
  - 🙆 パフォーマンスが重要なとき
  - 🙅 型に登場する値が変わるとき（キャストが必要なとき）
- 値と一緒に命題を渡すほうが問題が起きないがち
  - ❓ `def Array.get : (xs : Array α) → (i : Fin xs.size) : α`
  - 👍 `def Array.get : (xs : Array α) → (i : Nat) → (lt : i < xs.size) : α`
  - `i` が単なる自然数なのでキャストの問題が起きない

まとめ

- Lean は定理証明支援系であり純粋関数プログラミング言語でもある
  - Lean で記述したプログラムの性質を Lean 内で証明できる
- Lean で正規表現ライブラリ `lean-regex` を作っている
  - しかも、`lean-regex` の正しさを Lean で検証した
- 定理証明には対象のプログラムの深い理解が必要
  - 定理証明は苦しいが、とってもやりがいがある
- **みんなも定理証明、やろう！**

定理証明がやりたくなったら

- Natural Numbers Game: Lean の楽しいチュートリアル
- Functional Programming in Lean: Lean でのプログラムの書き方と検証
- Mathematics in Lean: Lean で数学を表現する方法 (Mathlib の紹介)

おわり

みんなも定理証明、やろう！

lean-regex はいつでもコントリビュータ募集中！