

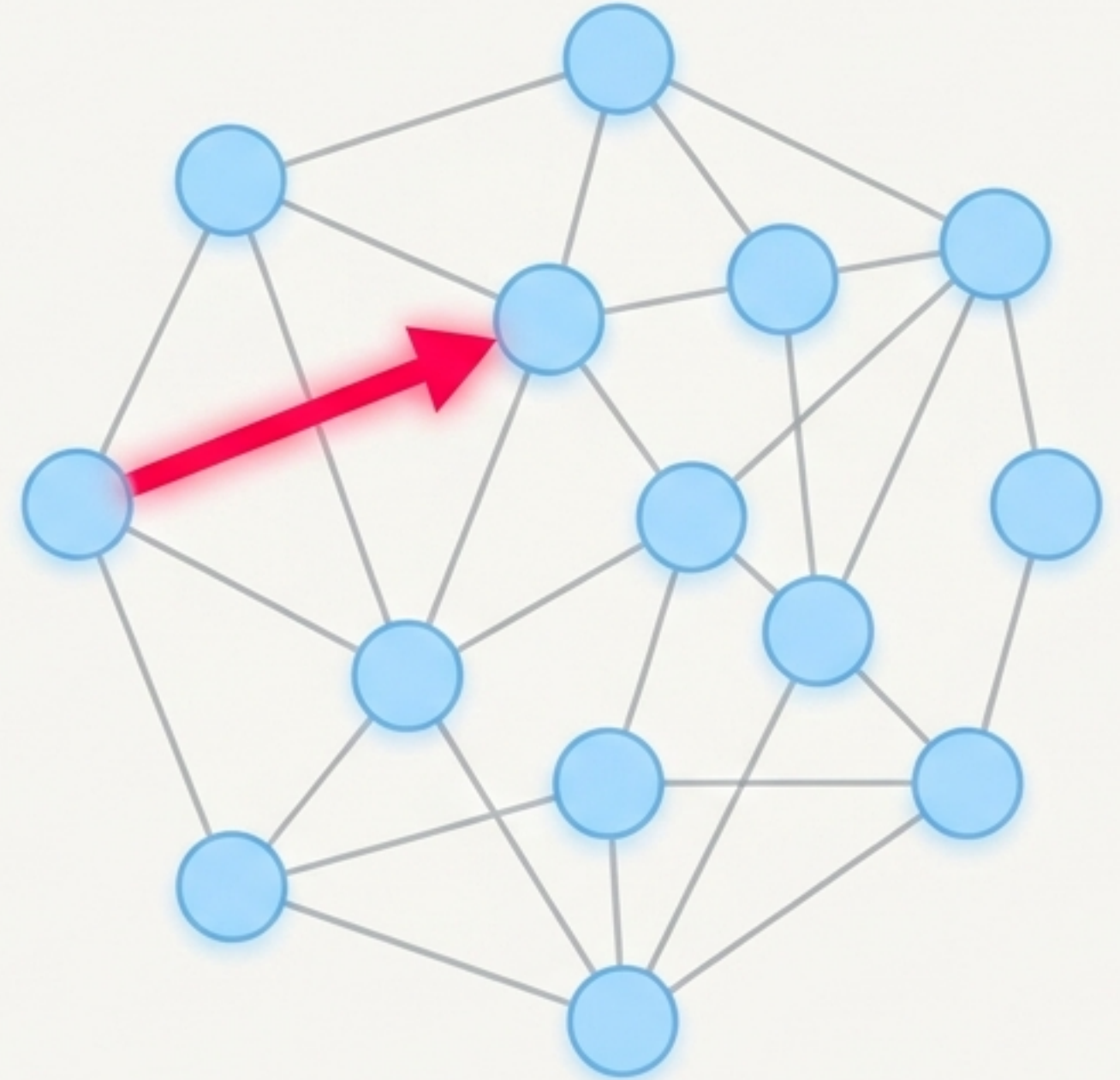
```
def find_path(graph, start, visited=None):  
    if visited is None:  
        visited = set()  
        visited.add(start)  
        for next in graph[start] - visited:  
            find_path(graph, next, visited)  
    return visited
```

Pythonで読み解くー 筆書きアルゴリズム

令和3年秋期 応用情報技術者試験
午後問3 完全解説

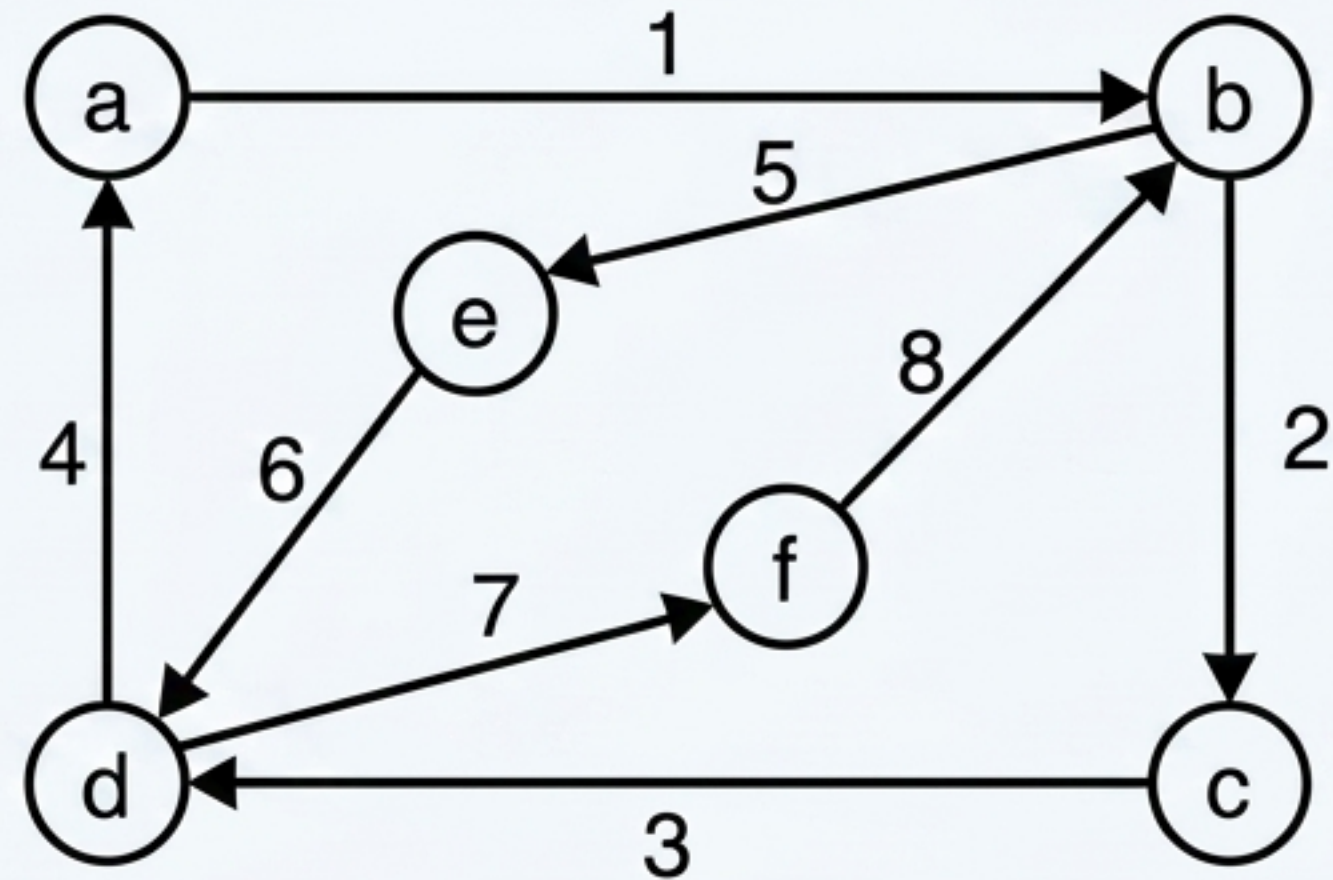
データ構造とグラフ理論が交差する
「バックトラック」の視覚的理解

```
def find_path(graph, start, visited=None):  
    if visited is None:  
        visited = set()  
    visited.add(start)  
    for next in graph[start] - visited:  
        find_path(graph, next, visited)  
    return visited
```



すべての辺を一度だけ通り、始点へ戻る

ゴールは「有向オイラー回路（一筆書き）」の発見。点aを出発し、すべての辺（1～8）を正確に1ずつ通過して点aに戻るアルゴリズムを構築する。



> 一筆書きの経路（辺の順序）：
[1, 2, 3, 7, 8, 5, 6, 4]

なぜこの順序になるのか？
内部の動きを解剖する。

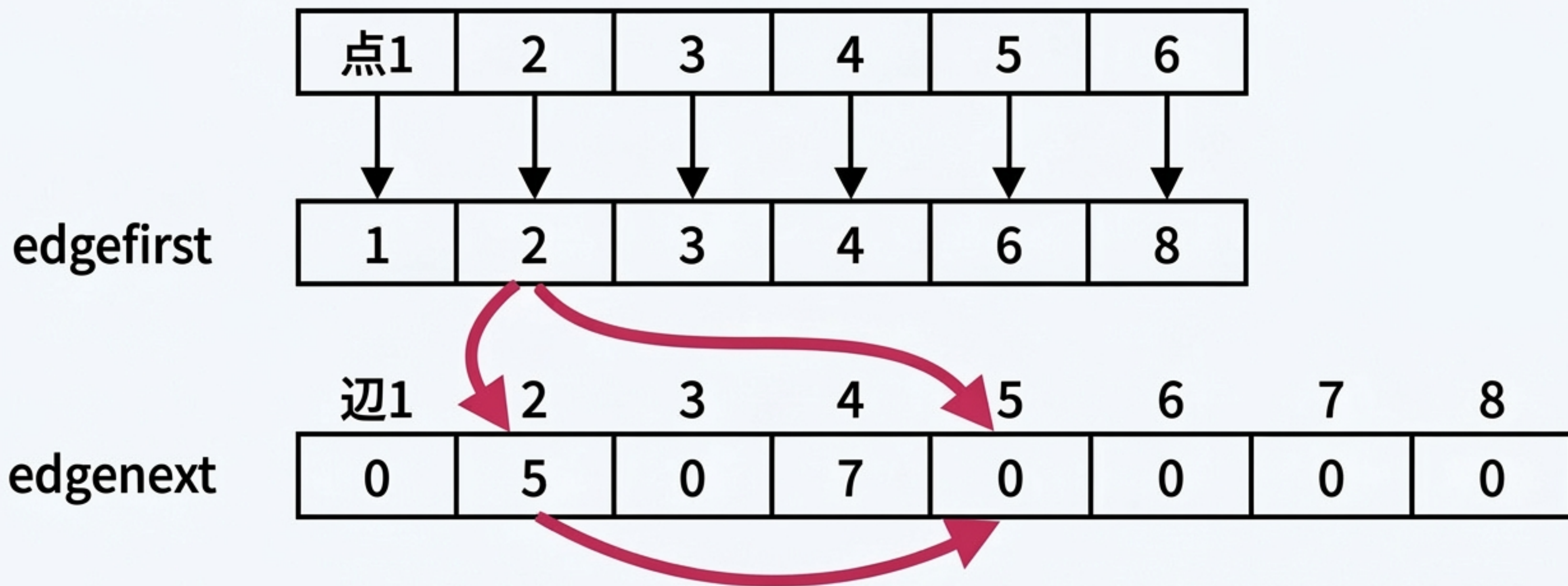
数学的条件：そもそも一筆書きは可能か？

	無向グラフ	有向グラフ
オイラーグラフ (始点に戻る・一筆書き可能)	すべての頂点の次数が「偶数」	すべての頂点で「入次数 = 出次数」
半オイラーグラフ (始点に戻らない・一筆書き可能)	次数が奇数の点（奇点）が「ちょうど2つ」	1点が「入次数 = 出次数 + 1」、もう1点が「出次数 = 入次数 + 1」（他は同数）

今回のグラフはすべての点で「入次数 = 出次数」を満たす強連結な有向グラフ（オイラーグラフ）である。

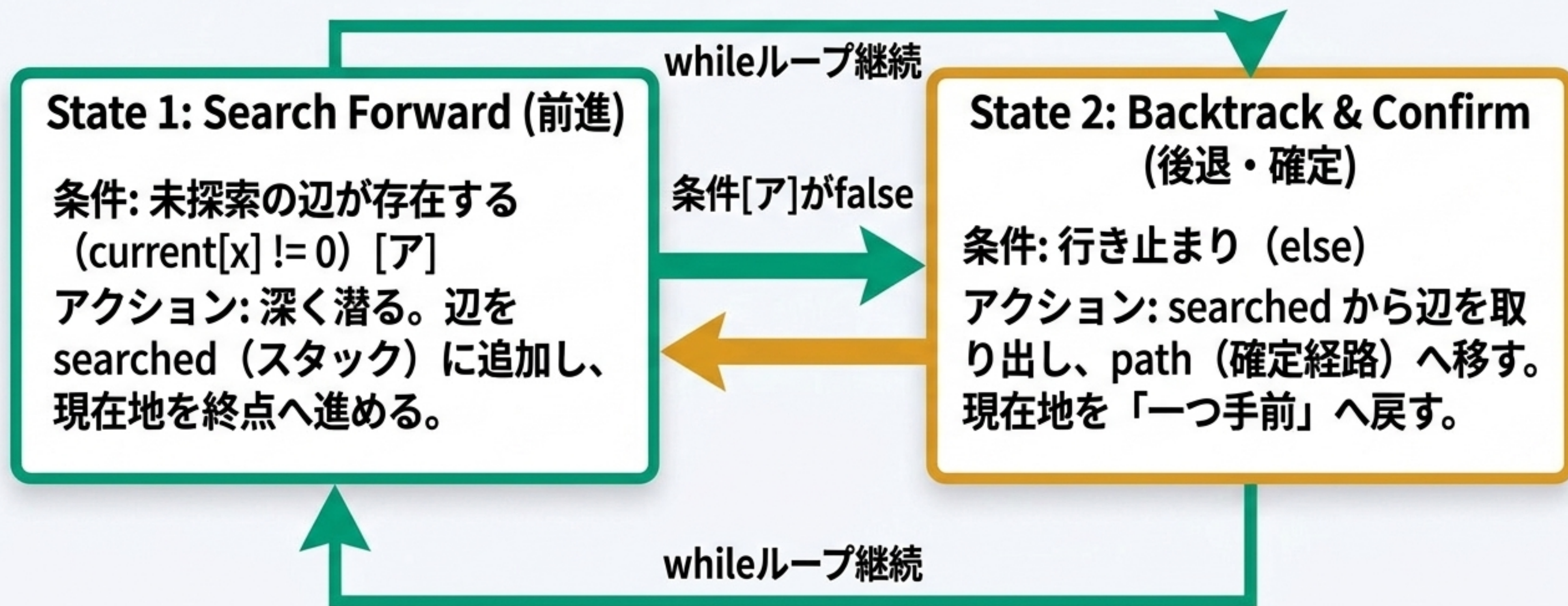
トポロジーをデータに変換する（配列の設計）

複雑なオブジェクトを使わず、2つの配列（edgefirst, edgenext）の連携のみでグラフの接続関係を「連結リスト」としてメモリ上に表現している。



アルゴリズムのアーキテクチャ (directedE)

再帰処理(Recursion)を使わず、スタック（配列）を用いた while ループの反復処理でエレガントに実装されている。



Step Step 1~3: 深さ優先での前進 (The Forward Dive)

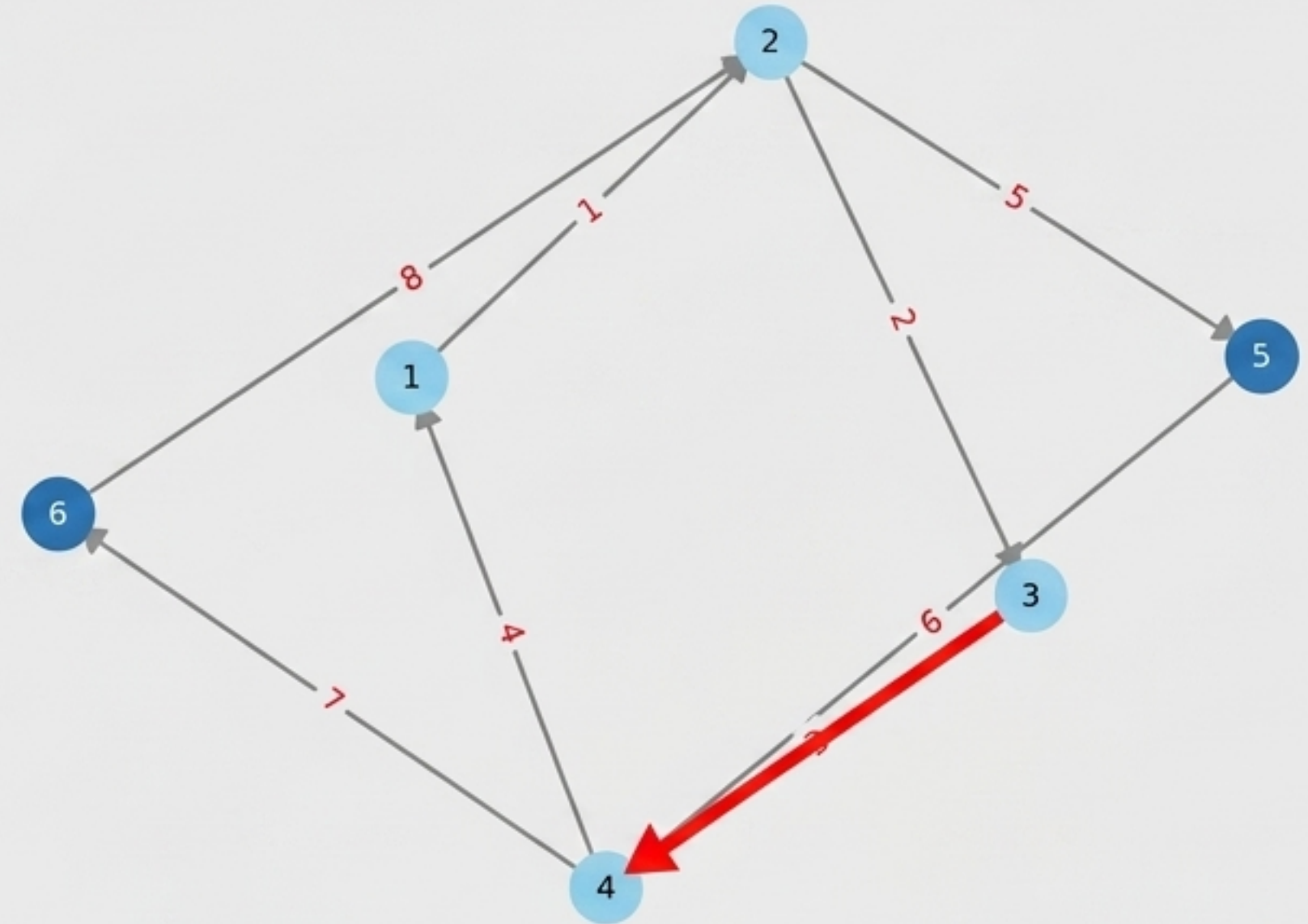
Using Split-Screen Debugger

Code State

```
if (current[x] != 0):  
    temp ← current[x]  
    searched[top] ← temp  
    current[x] ← edgenext[temp] // [イ]  
    x ← end[temp]  
    top ← top + 1
```

辺を通るたび、その始点の「次の未探索辺」を edgenext から引いて current を更新する。

Physical Graph Topology



経路は 1 → 2 → 3 と進み、点d（ノード4）へ到達。

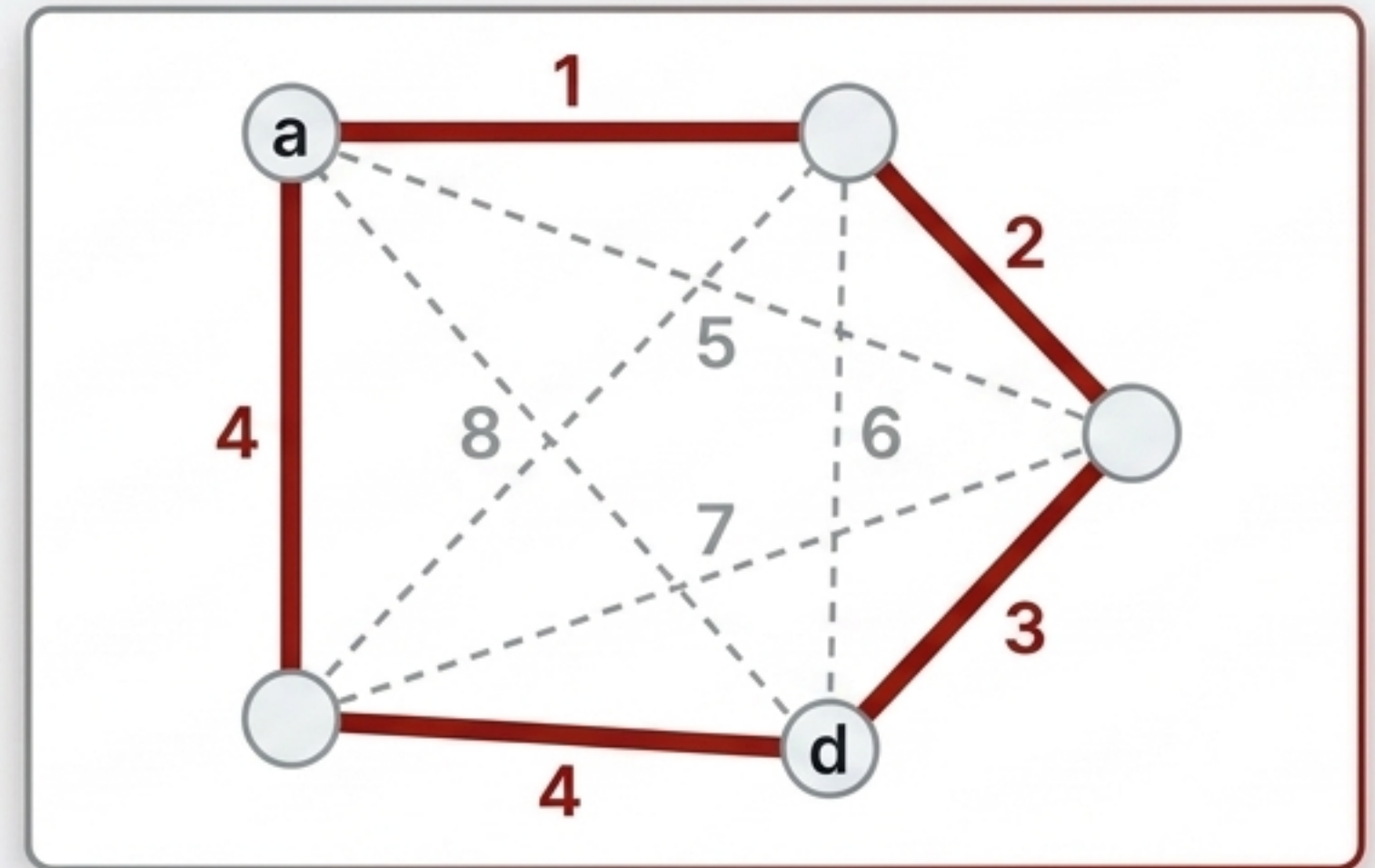
Step 4: 行き止まりの到達 (The Dead End)

Code State

```
while (last が 1 以上):  
    if (current[x] != 0): // [ア]  
        ...  
    else:  
        // バックトラック処理
```

辺4を通して点a（ノード1）へ戻った。
しかし、点aから出発する未探索の辺はもう存在しない（`current[1] == 0`）。

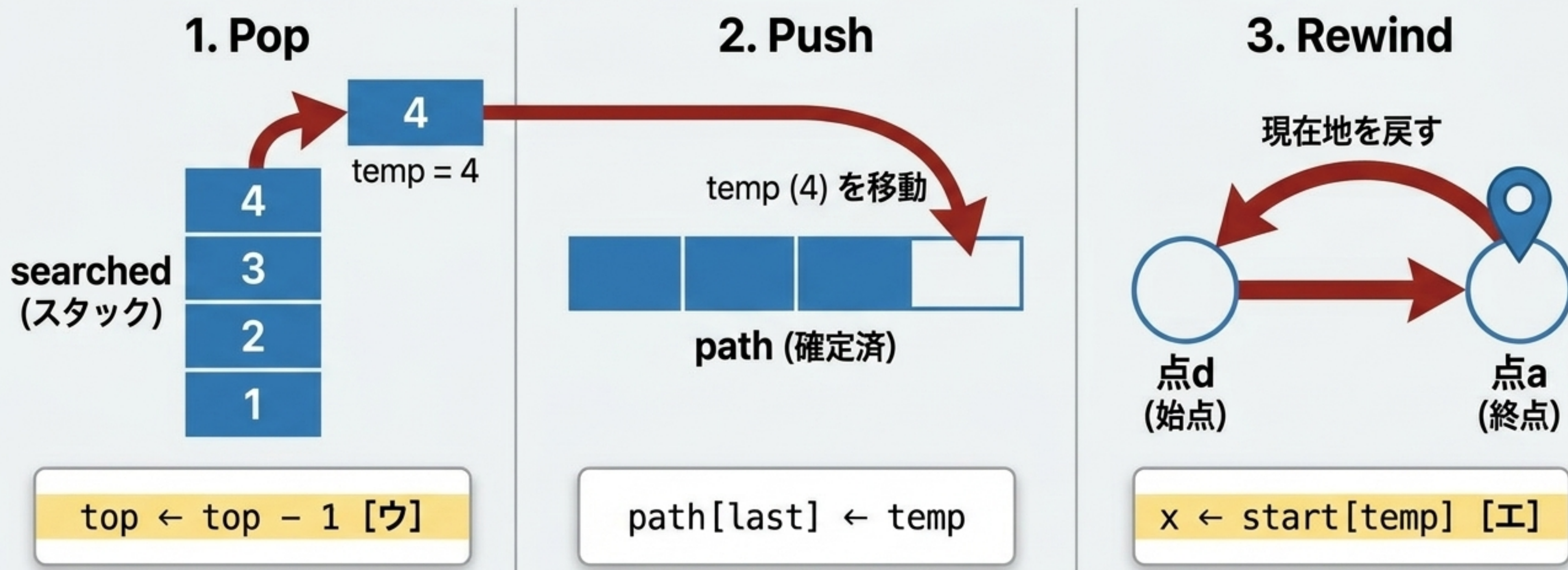
Physical Graph Topology



前進フェーズ (if) が終了し、アルゴリズムは壁にぶつかる。「次はどうする？」
→ バックトラック (else) の起動。

バックトラックのメカニズム (Stack Operations)

行き止まりに達したら、直前に通った辺を searched から取り出し path (確定) へ移す。
そして、現在地をその辺の「始点」に戻して新たな活路を探す。



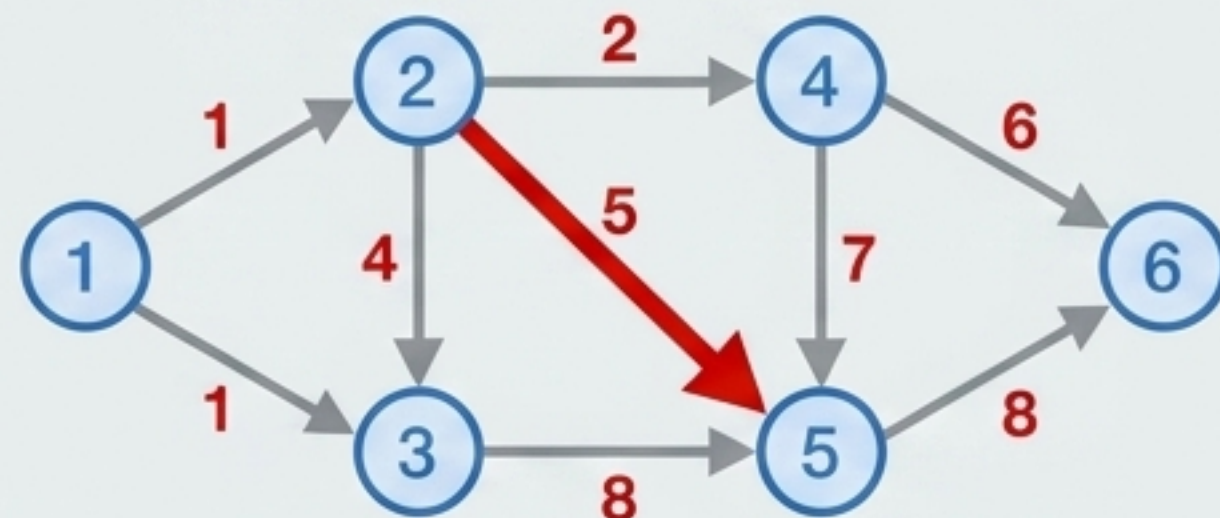
Step 5~8: 探索の再開と残余グラフの走破

Code State

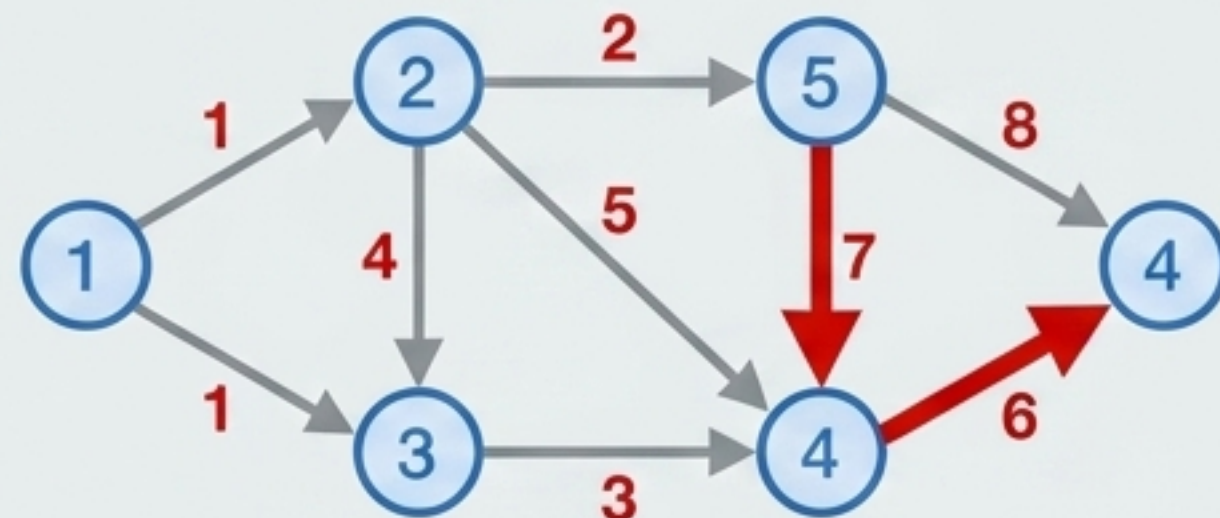
```
void cont_list {  
  int (current[x] = 1);  
  
  if (current[x] != 0): // [ア]  
    if current[x] != 0  
      current[x] != 0);  
    current[x] ;  
}  
...  
}
```

点d (ノード4) に戻ったことで、current[4] に格納されていた未探索辺「7」を発見。再び if ブロック (前進フェーズ) が稼働する。

Physical Graph Topology



Graph A (Step 6)

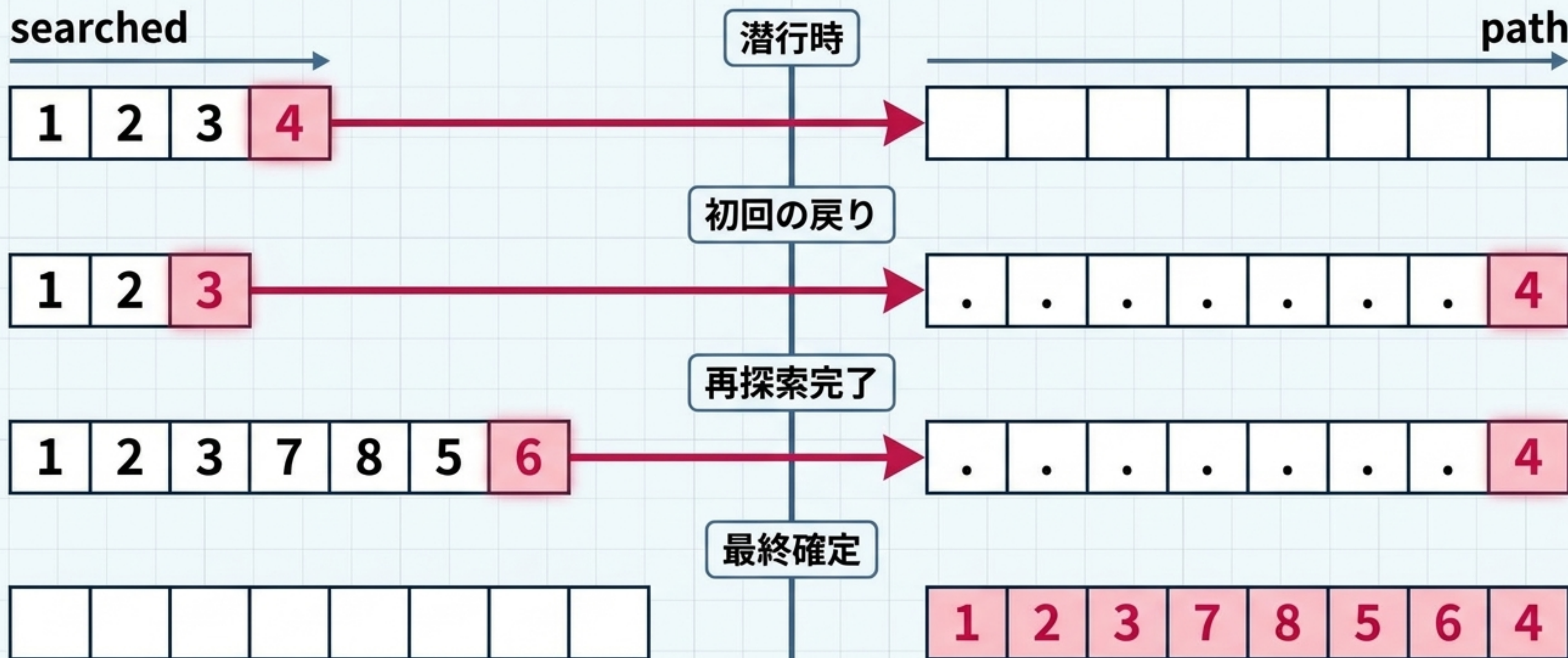


Graph B (Step 7)

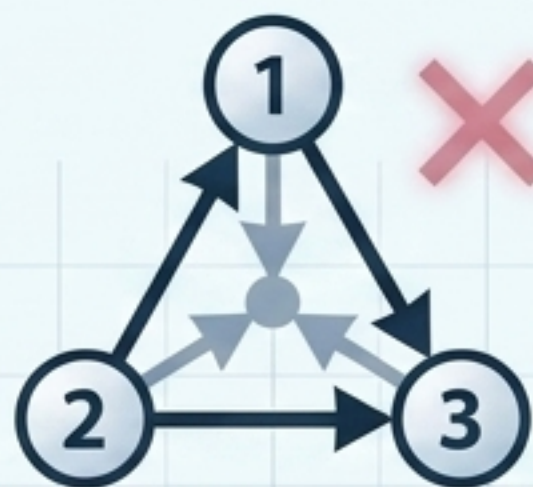
辺 7 → 8 → 5 → 6 と進み、未探索の辺をすべて刈り取る。

配列の進化プロセス（全体像の可視化）

スタック（searched）が膨張と収縮を繰り返し、すべての辺が path に逆順で確定していく。
whileループ内のif文（前進判定）はすべての辺に対して進む・戻るの「辺の数 × 2 = 16回」実行される。



例外処理：一筆書き不可能なグラフを与えたら？



Q: 無限ループに陥るか？

A: 否 (No).

- 理由: ループの終了条件は last が 1 以上であること。else ブロックを通過するたびに last は確実に減少するため、全辺数 (M 回) のループで必ず処理は「完了」する。

Q: 結果はどうなるか？

A: **破綻した経路**が出力される。

得られた path は一筆書きの条件を満たさず、出発点に戻らない、あるいは経路が途切れた**無効なデータ**となる。

コードの最適化：メモリ空間の統合 (Exam Q4)

searched を path 配列に置き換え、メモリ領域を半減させる。

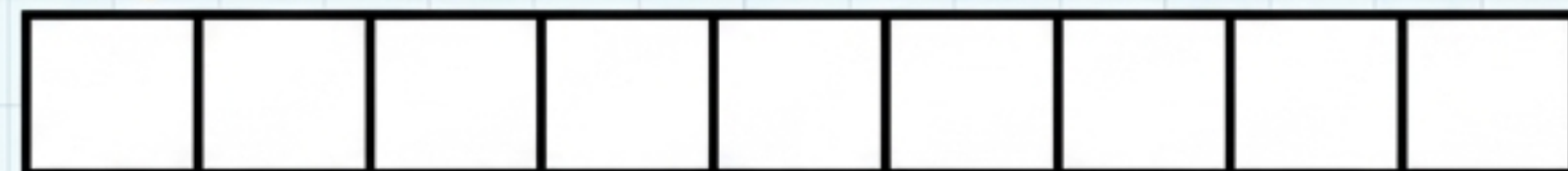
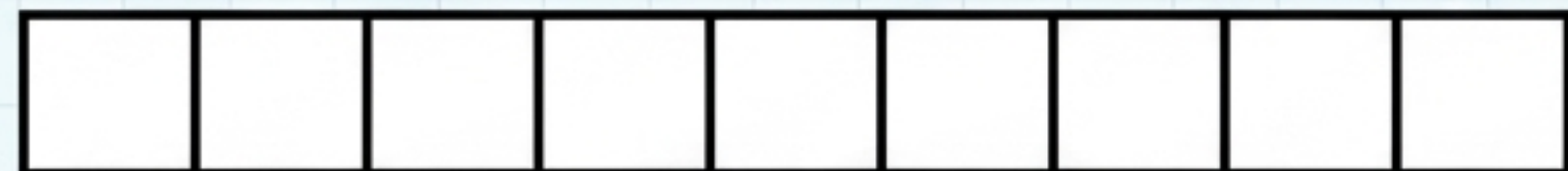
Before

searched

top

last

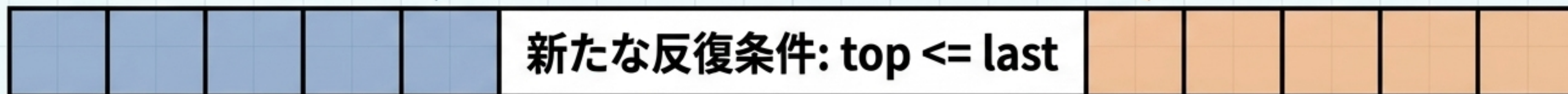
path



After

探索済 (top)

確定済 (last)



前からのポインタ(top)と後ろからのポインタ(last)がぶつかった瞬間、すべての辺の評価が完了する。

Summary & Takeaways



1. Linked-List Efficiency

edgefirst と edgenext の配列連携により、複雑なオブジェクト指向設計なしで軽量にグラフ構造を走破。



2. Elegant Backtracking

再帰呼び出し(Recursion)の代わりにスタック構造(searched)を活用し、行き止まりからの復帰処理をシンプルに実装。



2. Elegant Backtracking

再帰呼び出し(Recursion)の代わりにスタック構造(searched)を活用し、行き止まりからの復帰処理をシンプルに実装。



3. Two-Pointer Memory Optimization

top と last のポインタを一つの配列内で対向させることで、無駄なメモリ消費を抑える完璧な終了判定を実現。

グラフアルゴリズムの挙動は、データ構造（配列）とトポロジー（空間）の美しい交差点にある。