

되추적 기법

주요 내용

- 제약 충족 문제
- 되추적 기법
- n -여왕말 문제
- 그래프 색칠하기

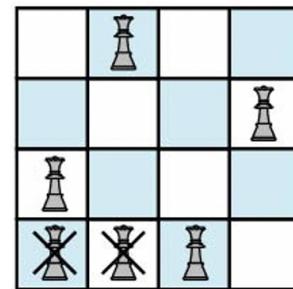
제약 충족 문제

- 여러 개의 대상 각각에 할당할 값을 특정 도메인(영역)에서 정해진 제약 조건에 따라 선택하는 문제
- 예제: n-여왕말 문제

4-여왕말 문제

- 여왕말: 체스판에서 상하좌우, 대각선 등 임의로 움직일 수 있음
- 4-여왕말 문제
 - (대상) 변수: 1번부터 4번까지 이름이 붙은 네 개의 여왕말
 - 도메인: 4×4 모양의 체스판에 포함된 1번 열부터 4번 열
 - 제약 조건: 서로 다른 두 개의 여왕말이 하나의 행, 열, 또는 대각선 상에 위치하지 않음

제약 조건 충족 여부



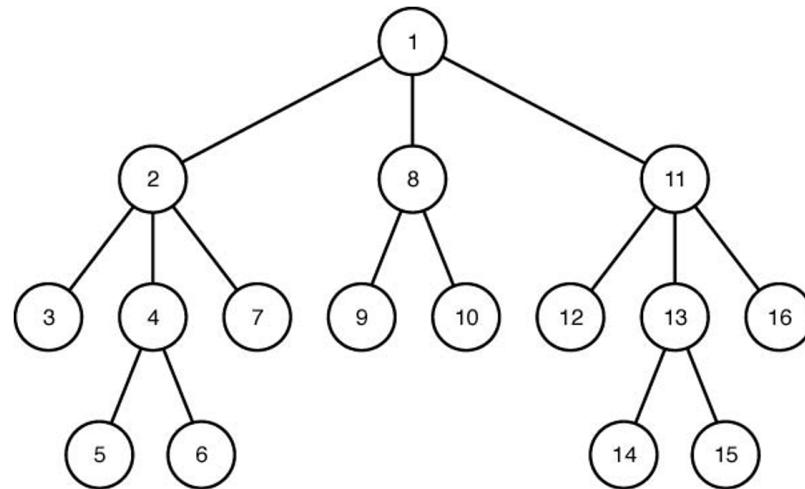
되추적 기법

- 제약 충족 문제를 해결하는 기법
- 백트래킹 backtracking으로 불리기도 함
- 많은 제약 충족 문제가 제약 조건만 서로 다를 뿐 동일한 되추적 알고리즘으로 해결 가능

되추적 기법 관련 개념

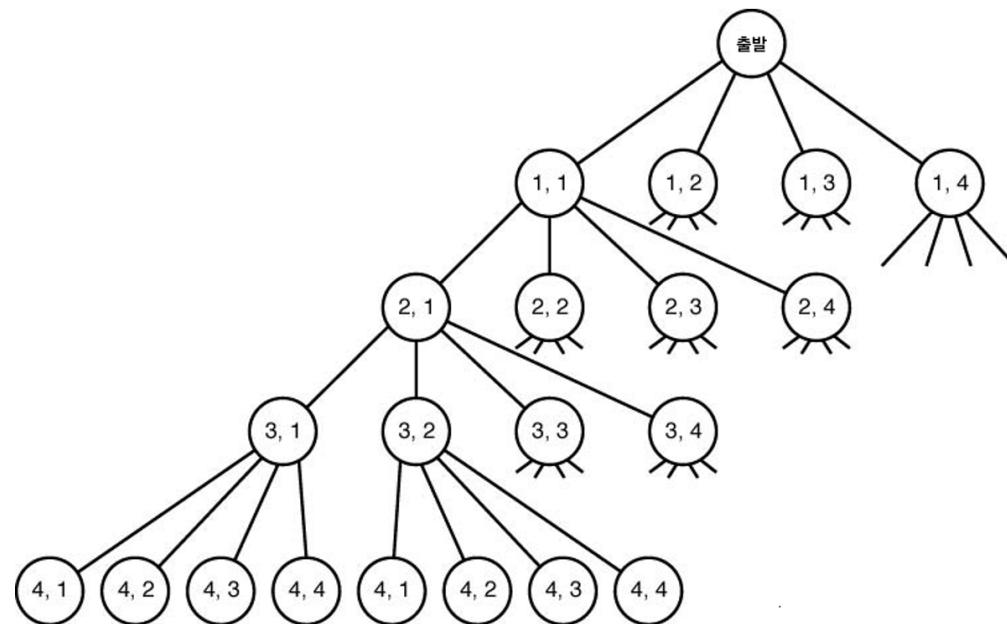
- 깊이 우선 탐색
- 상태 공간 트리
- 노드의 유망성
- 가지치기

깊이 우선 탐색



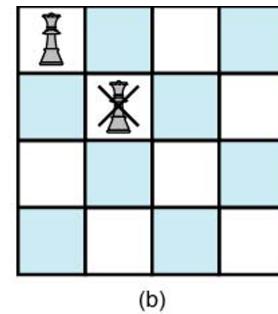
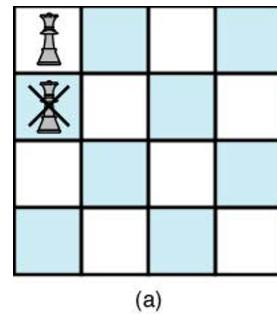
상태 공간 트리

- (대상) 변수가 가질 수 있는 모든 값으로 구성된 트리
- 예제: 4×4 로 이루어진 체스판에 네 개의 체스 여왕말을 놓을 수 있는 위치를 노드로 표현한 상태 공간 트리
 - 루트: 출발 노드이며 여왕말의 위치와는 무관함.
 - 깊이 k 의 노드: k 번째 여왕말이 놓일 수 있는 열 위치



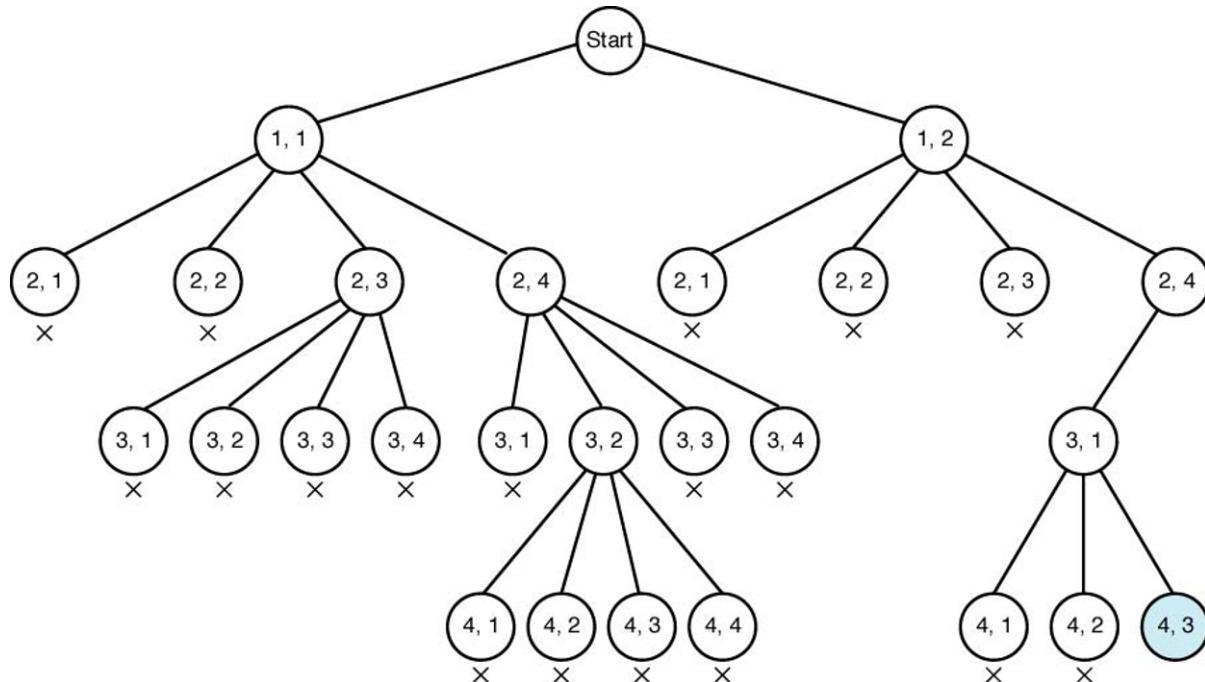
노드의 유망성

- 지정된 제약 조건을 만족시키는 노드
- 예제: 아래 그림 참고.
 - 첫째 여왕말의 위치에 따라 둘째 여왕말이 놓일 수 있는 위치에 해당하는 노드의 유망성이 결정됨
 - 둘째 여왕말에 대해 1번, 2번 칸에 해당하는 노드는 유망하지 않음

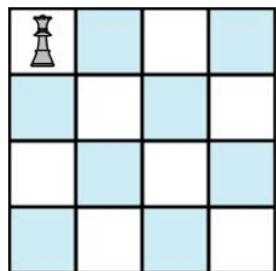


가지치기

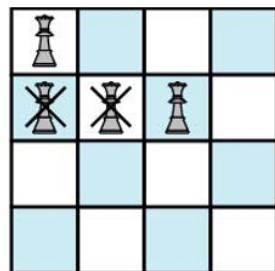
- 특정 노드에서 시작되는 가지를 제거하기
- 예제: 4×4 로 이루어진 체스판의 상태 공간 트리에서 유망하지 않은 노드를 가지치기한 결과



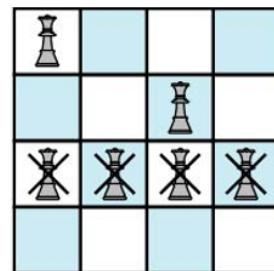
4-여왕말 문제 되추적 알고리즘



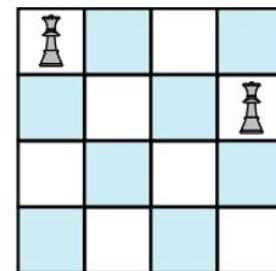
(a)



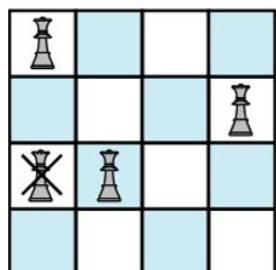
(b)



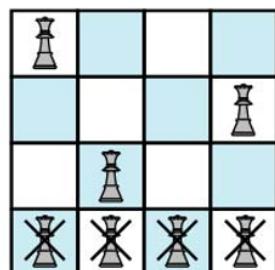
(c)



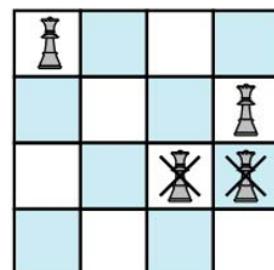
(d)



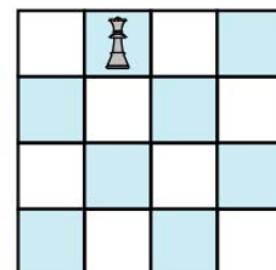
(e)



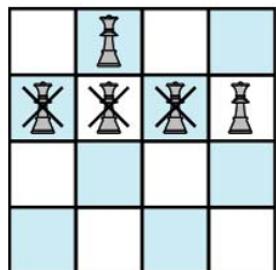
(f)



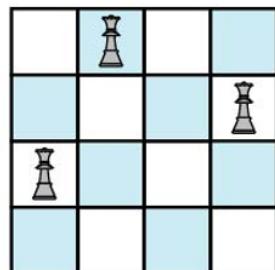
(g)



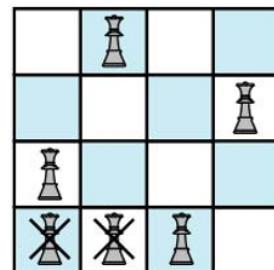
(h)



(i)



(j)



(k)

되추적 알고리즘 vs. 깊이 우선 탐색

- 4-여왕말 문제 해결 알고리즘: 되추적 알고리즘 대 깊이 우선 탐색
- 깊이 우선 탐색 알고리즘: 155 개의 노드 탐색
- 되추적 알고리즘: 27개의 노드 탐색

n -여왕말 문제

- (대상) 변수: 1번부터 n 번까지 이름이 붙은 n 개의 여왕말
- 도메인: $n \times n$ 모양의 체스판에 포함된 1번부터 n 번 열
- 제약 조건: 서로 다른 두 개의 여왕말이 하나의 행, 열, 또는 대각선 상에 위치하지 않음

예제: 8-여왕말 문제

```
In [1]: from collections import defaultdict

variables = range(1, 9)

domains = defaultdict(list)
columns = range(1, 9)
for var in variables:
    domains[var] = columns
```

```
In [2]: domains
```

```
Out[2]: defaultdict(list,
                    {1: range(1, 9),
                     2: range(1, 9),
                     3: range(1, 9),
                     4: range(1, 9),
                     5: range(1, 9),
                     6: range(1, 9),
                     7: range(1, 9),
                     8: range(1, 9)})
```

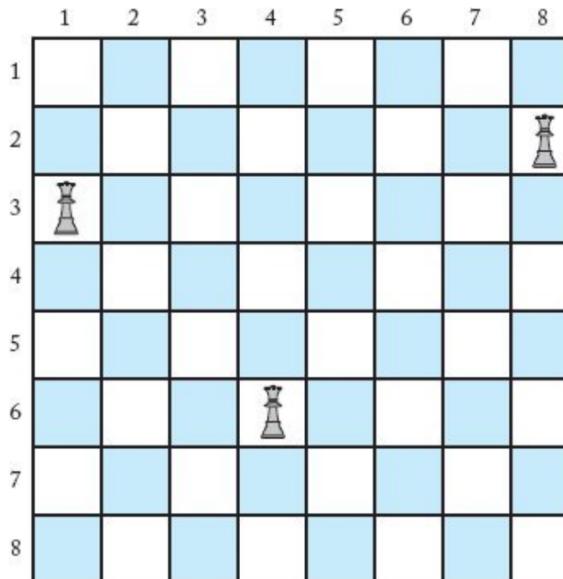
유망성 판단

위치가 이미 정해진 여왕말들을 대상으로 다음 세 개의 제약 조건이 성립함을 확인한다.

- 동일한 행에 위치하지 않기: 각각의 여왕말이 다른 행에 위치되도록 하기에 알고리즘이 작동하기에 자연스럽게 처리됨.
- 동일한 열에 위치하지 않기: 서로 다른 키에 대해 동일한 값이 사용되지 않도록 해야 함.

- 동일한 대각선 상에 위치하지 않기: 두 개의 여왕말 q_1, q_2 가 동일한 대각선 상에 위치하려면 행과 열의 차이의 절댓값이 같아야 함(아래 그림 참고. 아래 식에서 예를 들어 q_{1r} 과 q_{1c} 는 각각 q_1 이 위치한 행과 열의 좌표를 가리킴.

$$\text{abs}(q_{1r} - q_{2r}) == \text{abs}(q_{1c} - q_{2c})$$



```
In [3]: def promising_queens(assignment=defaultdict(int)):

    # q1: 모든 여왕말 대상으로 유망성 확인
    for q1r, q1c in assignment.items(): # q1의 행과 열

        # q2 = 아랫쪽에 자리한 여왕말들과 함께 제약 조건 성립 여부 확인
        # q1r과 q2r은 자연스럽게 다름
        for q2r in range(q1r + 1, len(assignment) + 1):
            q2c = assignment[q2r]
            if q1c == q2c:                      # 동일 열에 위치?
                return False
            if abs(q1r - q2r) == abs(q1c - q2c): # 동일 대각선상에 위치?
                return False

    # 모든 변수에 대해 제약조건 만족됨
    return True
```

되추적 함수 구현

- 지금까지 설명한 내용을 재귀 함수로 구현
- 재귀는 새로운 여왕말을 위치시킬 때마다 호출

```
In [4]: def backtracking_search_queens(num_queens,
                                         assignment=defaultdict(int)):

    variables = range(1, num_queens+1)
    domains = defaultdict(list)

    columns = range(1, num_queens+1)
    for var in variables:
        domains[var] = columns

    if len(assignment) == len(variables):
        return assignment

    unassigned = [v for v in variables if v not in assignment]
    first = unassigned[0]

    for value in domains[first]:
        local_assignment = assignment.copy()
        local_assignment[first] = value

        if promising_queens(local_assignment):
            result = backtracking_search_queens(num_queens,
                                                 local_assignment)

            if result is not None:
                return result

    return None
```

n -여왕말 문제 되추적 알고리즘의 시간 복잡도

- n 개의 여왕말이 주어졌을 때 상태 공간 트리에 포함된 탐색 대상 노드의 수

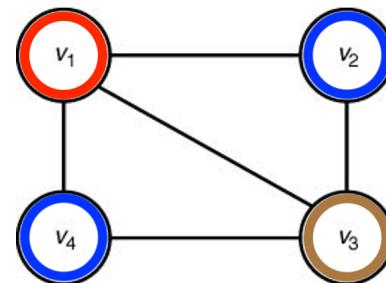
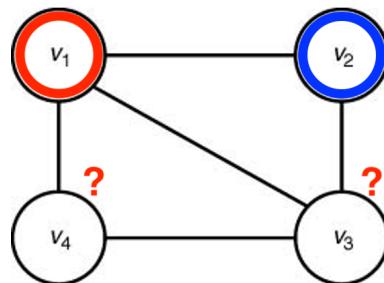
$$1 + n + n^2 + n^3 + \cdots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

- 이보다 더 효율적인 알고리즘은 아직 알려지지 않았음.

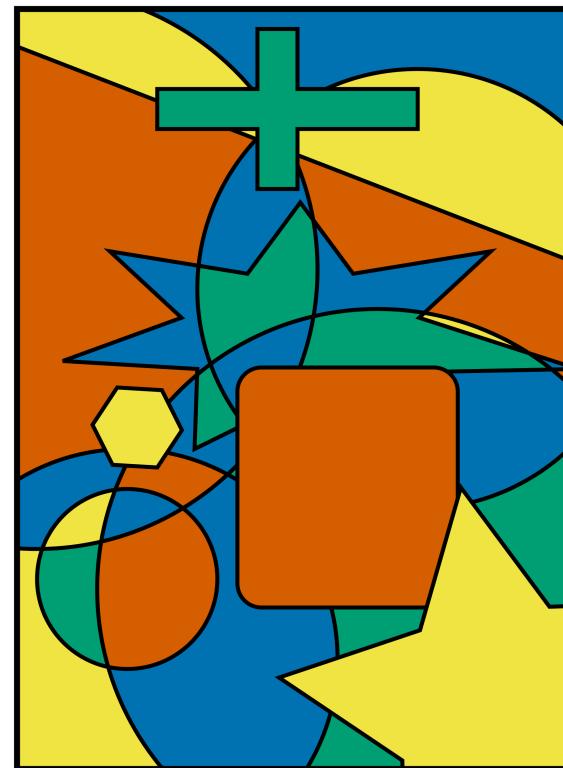
m -색칠 문제와 4색 정리

m -색칠 문제

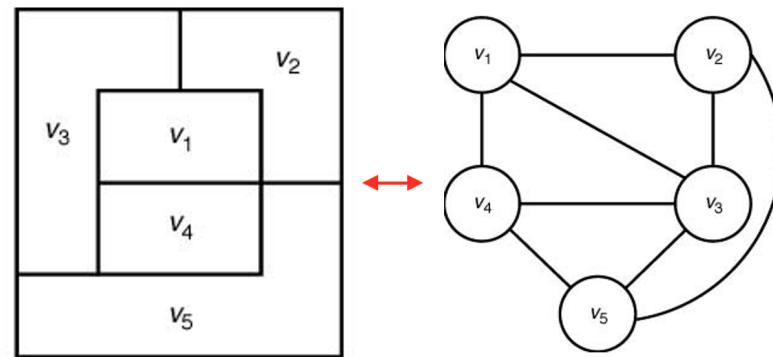
- 최대 m 개의 색을 이용하여 **무방향 그래프의 서로 인접한 노드가 서로 다른 색을 갖는다**는 색칠 조건이 만족되도록 색칠하는 문제



지도 색칠

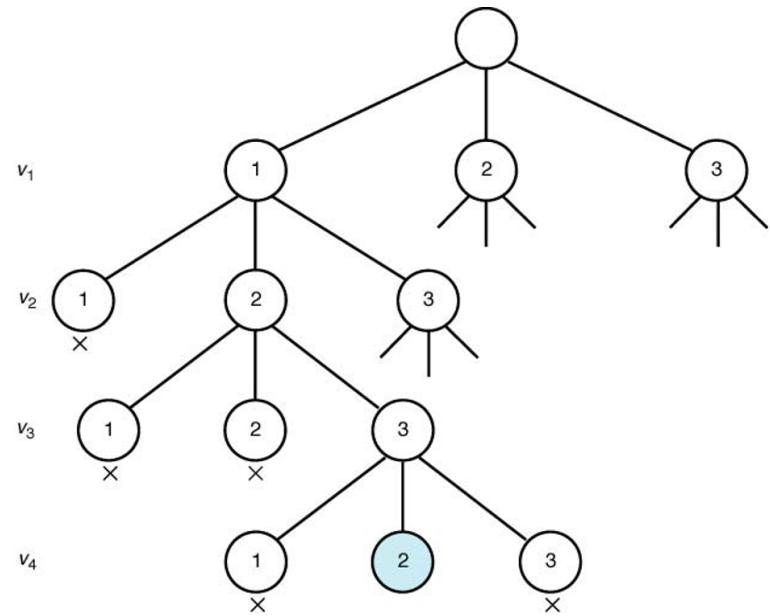
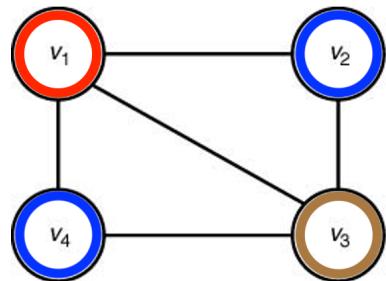


- 지도와 지도를 평면 그래프로 서로 일대일 대응
- 평면 그래프: 서로 교차하는 간선이 없는 무방향 그래프
 - 노드: 지도에 사용된 국가, 지역 등의 영역
 - 간선: 서로 인접한 두 영역 연결



m -색칠 문제와 되추적 기법

- (대상) 변수: 1번부터 n 번까지 이름이 붙은 n 개의 노드
- 도메인: 1번부터 m 번까지 이름이 붙은 m 종류의 색
- 제약 조건: 간선으로 연결된 이웃 노드는 서로 다른 색을 가져야 함.



3-색칠 문제 예제

```
In [5]: from collections import defaultdict
```

```
# 변수: 네 노드의 번호, 즉, 1, 2, 3, 4
variables = [1, 2, 3, 4]
```

```
# 도메인: 각각의 노드에 칠할 수 있는 가능한 모든 색상
# 3-색칠하기: 1(빨강), 2(파랑), 3(갈색)
domains = defaultdict(list)
colors = [1, 2, 3]
```

```
for var in variables:
    domains[var] = colors
```

```
In [6]: domains
```

```
Out[6]: defaultdict(list, {1: [1, 2, 3], 2: [1, 2, 3], 3: [1, 2, 3], 4: [1, 2, 3]})
```

유망성 판단

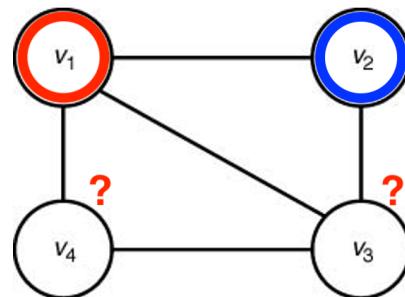
- `variable`: 새롭게 색이 할당된 노드
- `constraints`: 평면 그래프의 간선 정보를 담은 사전
- `assignment`: 되추적 기법 적용 기간동안 색이 할당된 노드들의 정보를 담은 사전

```
In [7]: def promising_colors(variable: int,
                           constraints=defaultdict(list),
                           assignment=defaultdict(int)):

    for var in constraints[variable]: # 이웃 노드에 대해 조건 확인
        if (var in assignment) and (assignment[var] == assignment[variable]):
            return False

    return True
```

간선 정보 예제



```
In [8]: # 각 노드에 대한 이웃 노드의 리스트
constraints = {
    1 : [2, 3, 4],
    2 : [1, 3],
    3 : [1, 2, 4],
    4 : [1, 3]
}
```

되추적 함수 구현

- 재귀는 새로운 노드에 색을 지정할 때마다 호출

```
In [9]: def backtracking_search_colors(num_nodes,
                                         num_colors,
                                         constraints=defaultdict(list),
                                         assignment=defaultdict(int)):

    variables = range(1, num_nodes+1)
    domains = defaultdict(list)
    colors = range(1, num_colors+1)
    for var in variables:
        domains[var] = colors

    if len(assignment) == len(variables):
        return assignment

    unassigned = [v for v in variables if v not in assignment]
    first = unassigned[0]
    for value in domains[first]:
        local_assignment = assignment.copy()
        local_assignment[first] = value

        if promising_colors(first, constraints, local_assignment):
            result = backtracking_search_colors(num_nodes, num_colors,
                                                 constraints, local_assignment)
            if result is not None:
                return result

    return None
```

```
In [10]: backtracking_search_colors(4, 3, constraints)
```

```
Out[10]: defaultdict(int, {1: 1, 2: 2, 3: 3, 4: 2})
```

```
In [11]: if not backtracking_search_colors(4, 2, constraints):
           print("두 가지 색으로 색칠 제약조건 충족 불가능")
```

두 가지 색으로 색칠 제약조건 충족 불가능

m -색칠 문제 되추적 알고리즘의 시간 복잡도

- n 개의 노드를 m 개의 색으로 칠해야 하는 문제의 상태 공간 트리의 노드의 수

$$1 + m + m^2 + m^3 + \cdots + m^n = \frac{m^{n+1} - 1}{m - 1}$$

- 이보다 더 효율적인 알고리즘은 아직 알려지지 않았음.

4색 정리

- 1852년: 영국인 프랜시스 구쓰리 Francis Guthrie가 세운 가설. 영국의 인접한 두 개의 주를 서로 다른 색으로 칠하기 위해 필요한 색이 최소 4개
- 1976년: K. Appel과 W. Haken이 최초의 증명 제시. 500 여쪽 논문. 일부 내용은 컴퓨터 프로그램 사용
- 2005년: 조지 곤티에 George Gonthier가 두 사람의 증명 검증

