5장 되추적

주요 내용

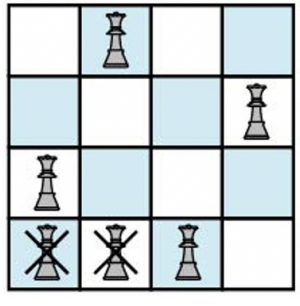
- 1절 되추적 기법
- 2절 n-퀸 문제
- 5절 그래프 색칠하기
- 부록: 제네릭 프로그래밍

1절 제약충족 문제와 되추적 기법

제약충족 문제(CSP, constraint-satisfaction problems)

- 특정 변수에 할당할 값을 지정된 **도메인**(영역, 집합)에서 정해진 조건에 따라 선택하는 문제

- 예제: 4-퀸 문제(체스 퀸(queen) 네 개의 위치 선정하기)
    - 변수: 네 개의 퀸
        - 즉, 1번 퀸부터 4번 퀸.
    - 도메인: {1, 2, 3, 4}
        - 즉, 1번 열부터 4번 열.
    - 조건: 두 개의 퀸이 하나의 행, 열, 또는 대각선 상에 위치하지 않음.



되추적 기법(백트래킹, backtracking)

- 제약충족 문제를 해결하는 일반적인 기법
- 문제에 따라 다른 제약충족 조건만 다를 뿐 문제해결을 위한 알고리즘은 동일함.
- 여기서는 두 개의 문제를 이용하여 되추적 기법의 활용법을 설명함.

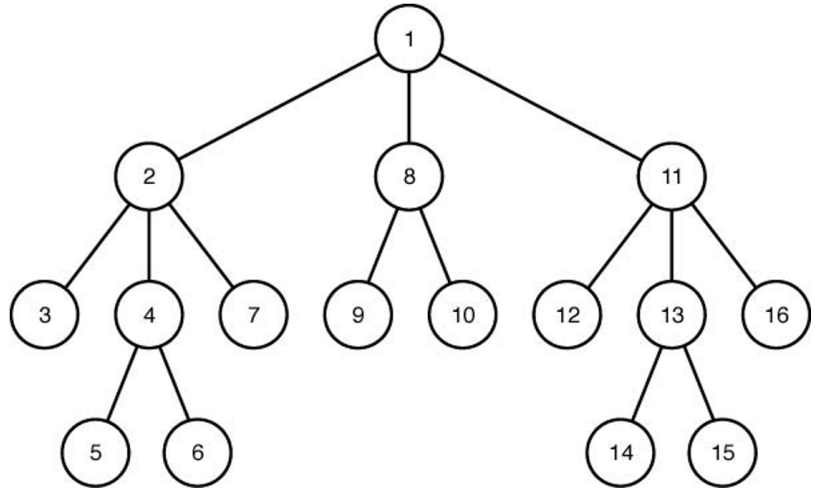
주요 기초개념

- 깊이우선 탐색
- 상태 공간 나무
- 마디의 유망성
- 가지치기

깊이우선 탐색

- DFS(depth-first-search): 뿌리 지정 나무(rooted tree)를 대상으로 하는 탐색기법.
- 왼편으로 끝(잎마디)까지 탐색한 후에 오른편 형제자매 마디로 이동

- 예제:
    - 아래 뿌리 지정 나무의 뿌리에서 출발하여 왼편 아랫쪽 방향으로 진행.
    - 더 이상 아래 방향으로 진행할 수 없으면 부모 마디로 돌아간 후 다른 형제자매 마디 중 가장 왼편에 위치한 마디로 이동 후 왼편 아랫쪽 방향으로의 이동 반복

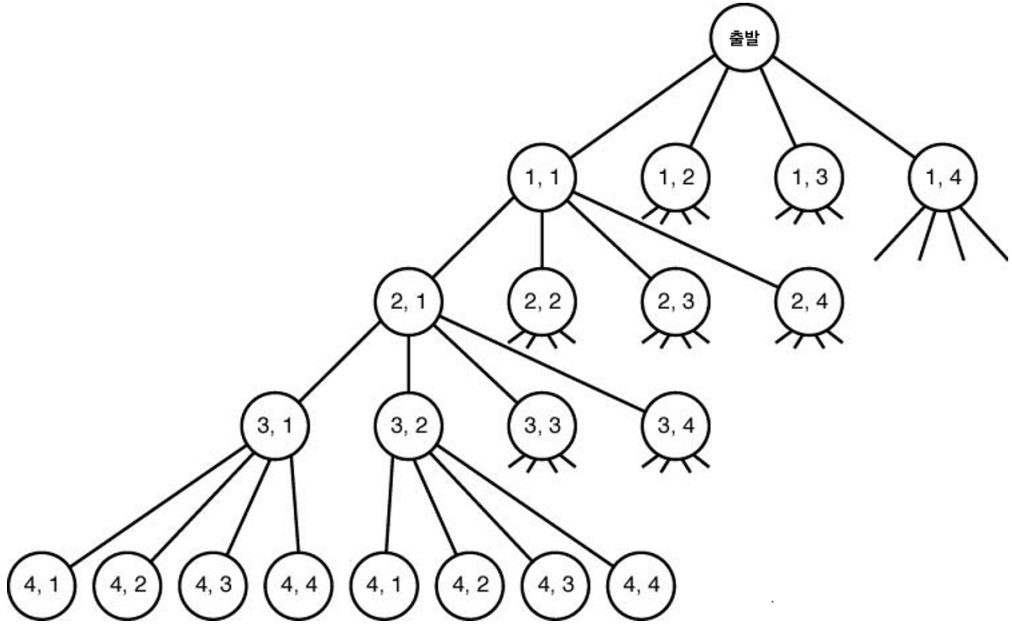


상태 공간 나무(state space tree)

- 변수가 가질 수 있는 모든 값을 마디(node)로 갖는 뿌리 지정 나무

- **깊이**: 깊이가 0인 뿌리에서 출발하여 아래로 내려갈 수록 깊이가 1씩 증가.

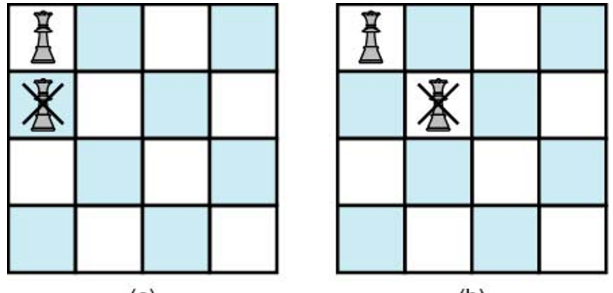
- 예제: `4x4`로 이루어진 체스판에 네 개의 체스 퀸을 놓을 수 있는 위치를 마디로 표현한 상태 공간 나무
    - 뿌리는 출발 마디로 표현하며, 체스 퀸의 위치와 상관 없음.
    - 깊이 k의 마디: k 째 퀸이 놓일 수 있는 위치



마디의 유망성

- 지정된 특정 조건에 해당하는 마디를 **유망하다**라고 부름.

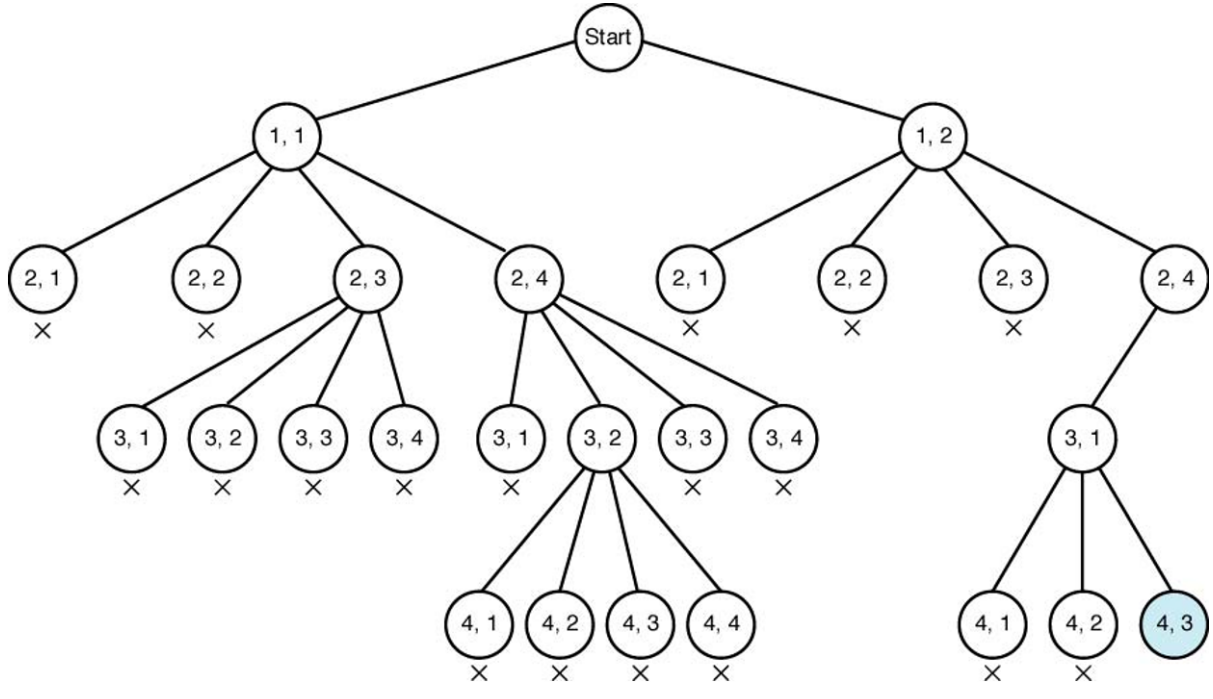
- 예제: 네 개의 퀸을 위치시켜야 할 경우 첫째 퀸의 위치에 따라 둘째 퀸이 놓일 수 있는 위치의 유망성이 결정됨.
    - 아래 그림에서 2번 행의 1, 2번 칸은 유망하지 않음.



가지치기(pruning)

- 특정 마디에서 시작되는 가지 제거하기

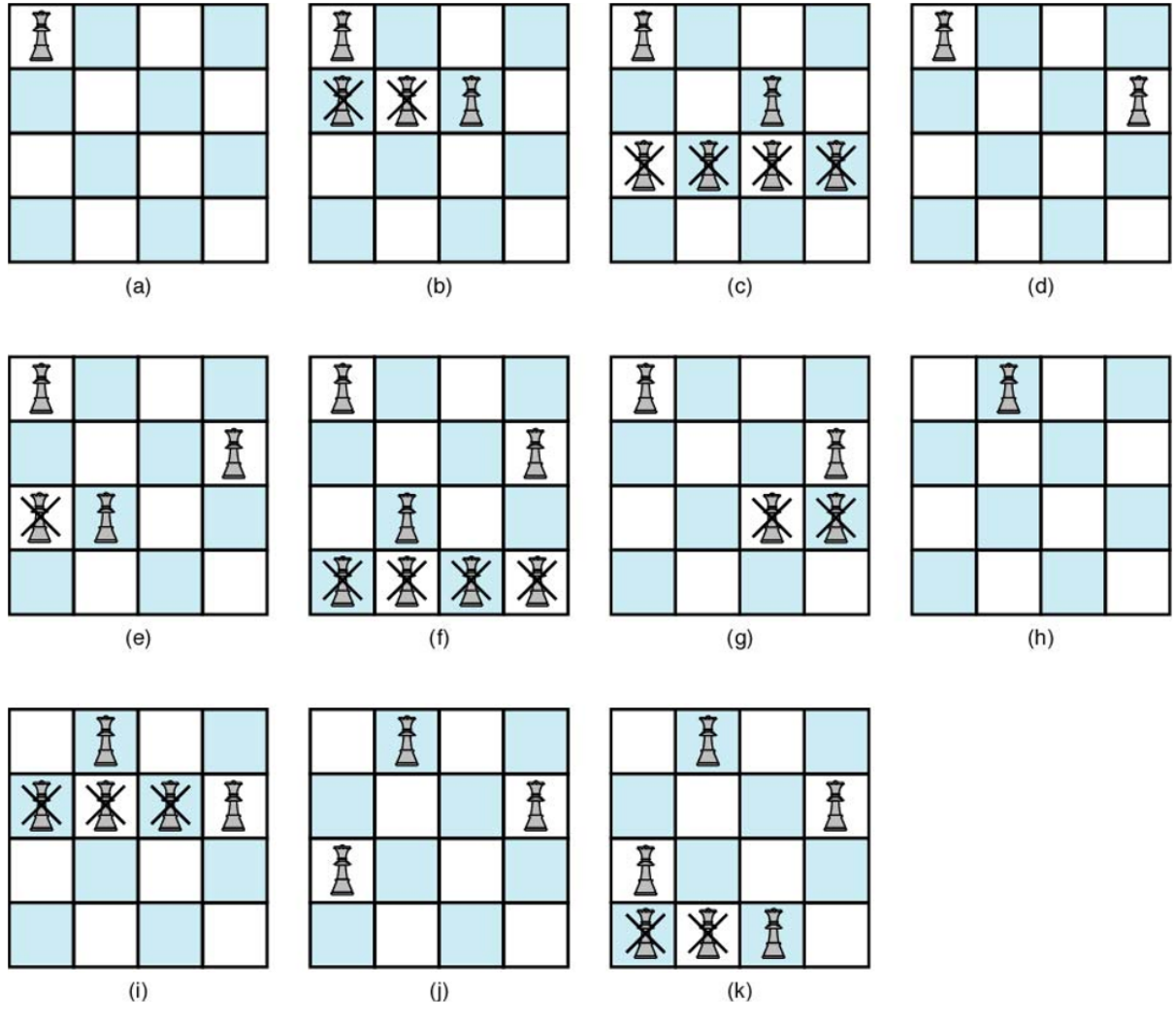
- 예제: 4 x 4로 이루어진 체스판에 네 개의 체스 퀸을 놓을 수 있는 위치를 마디로 표현한 상태 공간 나무에서 유망하지 않은 마디에서 가지치기를 실행하면 아래 그림이 생성됨.



되추적 알고리즘

1. 상태 공간 나무의 뿌리로부터 깊이우선 탐색(DFS) 실행.
2. 탐색 과정에서 유망하지 않은 마디를 만나면 가지치기 실행 후 부모 마디로 되돌아감(되추적, backtracking).
3. 이후 다른 형제자매 마디를 대상으로 깊이우선 탐색 반복. 더 이상의 형제자매 마디가 없으면 형제자매가 있는 조상까지 되추적 실행.
4. 탐색이 더 이상 진행할 수 없는 경우 알고리즘 종료

예제: 되추적 알고리즘을 활용한 4-퀸 문제 해결



깊이우선 탐색 대 되추적 알고리즘 비교

- 4-퀸 문제를 순수한 깊이우선 탐색으로 해결하고자 할 경우: 155 마디 검색
- 4-퀸 문제를 되추적 알고리즘으로 해결하고자 하는 경우: 27 마디 검색

2절 n-퀸 문제

- 4-퀸 문제를 일반화시킨 n-문제를 해결하는 되추적 알고리즘 구현하기

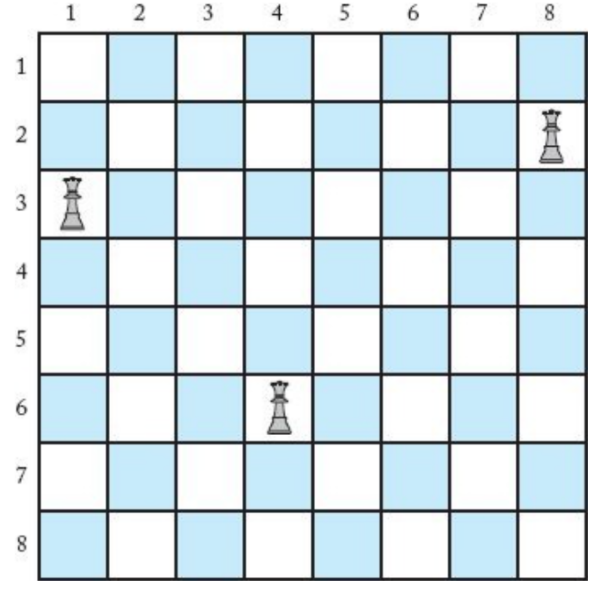
- 문제: n 개의 퀸(queen)을 서로 상대방을 위협하지 않도록 n x n 체스판에 위치시키기
    - 변수: n 개의 퀸
        - 즉, 1번 퀸부터 n번 퀸.
    - 도메인: {1, 2, ..., n}
        - 즉, 1번 열부터 n번 열.
    - 조건: 두 개의 퀸이 하나의 행, 열, 또는 대각선 상에 위치하지 않음.

유망성 판단

- 두 개의 퀸 q_1 , q_2 가 같은 대각선 상에 위치하려면 행과 열의 차이의 절댓값이 동일해야 함. (아래 그림 참조)

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

단, $(q_{1,r}, q_{1,c})$ 와 $(q_{2,r}, q_{2,c})$ 는 각각 q_1 과 q_2 가 위치한 행과 열의 좌표를 가리킴.



예제: 4-퀸 문제 해결 되추적 알고리즘

In [1]:
```
from typing import List, Dict

# 변수: 네 개의 퀸의 번호, 즉, 1, 2, 3, 4
variables = [1, 2, 3, 4]

# 도메인: 각각의 퀸이 자리잡을 수 있는 가능한 모든 열의 위치.
domains: Dict[int, List[int]] = {}
columns = [1, 2, 3, 4]
for var in variables:
    domains[var] = columns
```

- 4-퀸 문제의 경우 각각의 퀸 모두 동일하게 1열부터 4열 어딘가에 위치할 수 있음. 단, 그 중에서 조건을 만족시키는 열을 찾아야 함.

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

되추적 함수 구현

- 아래 되추적 함수 `backtracking_search_queens()`는 일반적인 n-퀸 문제를 해결함.
    - `assignment` 인자: 되추적 과정에서 일부의 변수에 대해 할당된 도메인 값의 정보를 담은 사전을 가리킴.
    - 인자가 들어오면 아직 값을 할당받지 못한 변수를 대상으로 유망성을 확인한 후 되추적 알고리즘 진행.
    - 되추적 알고리즘이 진행되면서 `assignment`가 확장되며 모든 변수에 대해 도메인 값이 지정될 때가지 재귀적으로 알고리즘이 진행됨.

In [3]:

```python
def backtracking_search_queens(assignment: Dict[int, int] = {}):
    """assignment: 각각의 변수를 키로 사용하고 키값은 해당 변수에 할당될 값"""

    # 모든 변수에 대한 값이 지정된 경우 조건을 만족시키는 해가 완성된 것임
    if len(assignment) == len(variables):
        return assignment
    # 아직 값을 갖지 않은 변수들이 존재하면 되추적 알고리즘을 아직 할당되지 않은 값을 대상으로 이어서 진행
    unassigned = [v for v in variables if v not in assignment]
    first = unassigned[0]

    for value in domains[first]:
        # 주의: 기존의 assignment를 보호하기 위해 복사본 활용
        # 되추적이 발생할 때 이전 할당값을 기억해 두기 위해서임.
        local_assignment = assignment.copy()
        local_assignment[first] = value
        # local_assignment 값이 유망하면 재귀 호출을 사용하여 변수 할당 이어감.
        if promissing_queens(first, local_assignment):
            result = backtracking_search_queens(local_assignment)
            # 유망성을 이어가지 못하면 되추적 실행
            if result is not None:
                return result

    return None
```

유망성 확인 함수

In [4]:
```python
def promissing_queens(variable: int, assignment: Dict[int, int]):
    """새로운 변수 variable에 값을 할당 하면서 해당 변수와 연관된 변수들 사이의 제약조건이
    assignment에 대해 만족되는지 여부 확인

    n-퀸 문제의 경우: 제약조건이 모든 변수에 대해 일정함.
                    즉, 새로 위치시켜야 하는 퀸이 기존에 이미 자리잡은 퀸들 중 하나와
                    동일 행, 열, 대각산 상에 위치하는지 여부를 확인함"""

    # q1r, q1c: 첫째 퀸이 놓인 마디의 열과 행
    for q1r, q1c in assignment.items():
        # q2r = 첫째 퀸 아래에 위치한 다른 모든 퀸들을 대상으로 조건만족여부 확인
        for q2r in range(q1r + 1, len(assignment) + 1):
            q2c = assignment[q2r] # 둘째 퀸의 열
            if q1c == q2c: # 동일 열에 위치?
                return False
            if abs(q1r - q2r) == abs(q1c - q2c): # 대각선상에 위치?
                return False

    # 모든 변수에 대해 제약조건 만족됨
    return True
```

In [5]:
```
backtracking_search_queens()
```

n-퀸 문제 되추적 알고리즘의 시간 복잡도

- n 개의 퀸이 주어졌을 때 상태공간트리의 마디의 수는 다음과 같음.

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

- 따라서 되추적 알고리즘이 최대 n의 지수승 만큼 많은 수의 마디를 검색해야 할 수도 있음.
- 하지만 검색해야 하는 마디 수는 경우마다 다름.
- 효율적인 알고리즘이 아직 알려지지 않음.

부록: 얕은(shallow) 복사 vs 깊은(deep) 복사

- 리스트의 `copy()` 메서드는 얕은 복사 용도로 사용된.
  - 1차원 리스트일 경우 새로운 리스트를 복사해서 만들어 냄.
  - 하지만 2차원 이상의 리스트 일 경우 모든 것을 복사하지는 않음. 아래 코드 참조.

In [6]:
```
# 얕은 복사
aList = [1, 2, 3, 4]
bList = aList
cList = aList.copy()
aList[0] = 10
print("얕은 복사:", aList[0] == cList[0])

dList = [[5, 6], [7, 8]]
eList = dList.copy()
dList[0][1] = 60
print("얕은 복사:", dList[0] == eList[0])
```

```
얕은 복사: False
얕은 복사: True
```

Python 3.6
(known limitations)

```
1  aList = [1, 2, 3, 4]
2  bList = aList
3  cList = aList.copy()
4
5  aList[0] = 10
6
7  print("얕은 복사:", aList[0] == cList[0])
8
9  dList = [[5, 6], [7, 8]]
10 eList = dList.copy()
11
12 dList[0][1] = 60
13 print("얕은 복사:", dList[0] == eList[0])
```

Edit this code

line that just executed
next line to execute

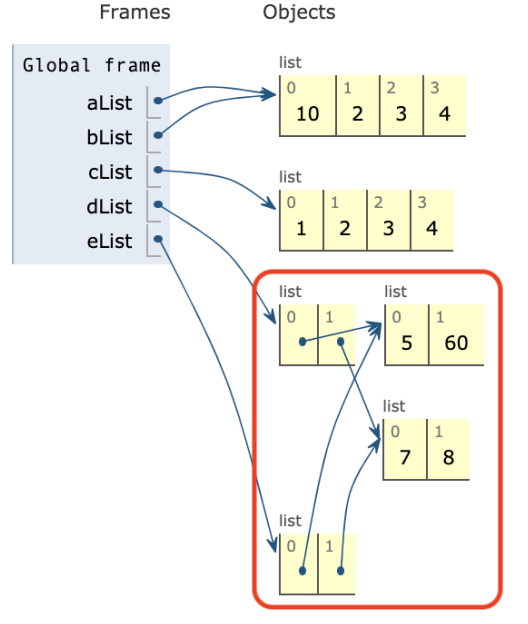
<< First | < Prev | Next > | Last >>

Done running (9 steps)

Customize visualization (NEW!)

Print output (drag lower right corner to resize)

```
얕은 복사: False
얕은 복사: True
```



- 깊은 차원까지 복사를 하려면 깊은 복사(deep copy)를 사용해야 함.
    - 방식1: 새로 정의
    - 방식2: copy 모듈의 deepcopy( ) 함수 활용
        - copy( ) 함수: 얕은 복사. 리스트의 copy( ) 메서드와 동일하게 작동.
        - deepcopy( ) 함수: 깊은 복사.

In [7]:
```
# 얕은 복사 vs. 얕은 복사
from copy import copy, deepcopy

aList = [1, 2, 3, 4]
bList = aList
cList = copy(aList)
aList[0] = 10
print("얕은 복사:", aList[0] == cList[0])

dList = [[5, 6], [7, 8]]
eList = deepcopy(dList)
dList[0][1] = 60
print("깊은 복사:", dList[0] == eList[0])
```

```
얕은 복사: False
깊은 복사: False
```

Python 3.6
(known limitations)

```
1  from copy import copy, deepcopy
2
3  aList = [1, 2, 3, 4]
4  bList = aList
5  cList = copy(aList)
6
7  aList[0] = 10
8
9  print("얕은 복사:", aList[0] == cList[0])
10
11 dList = [[5, 6], [7, 8]]
12 eList = deepcopy(dList)
13
14 dList[0][1] = 60
15 print("깊은 복사:", dList[0] == eList[0])
```

Edit this code

line that just executed
next line to execute

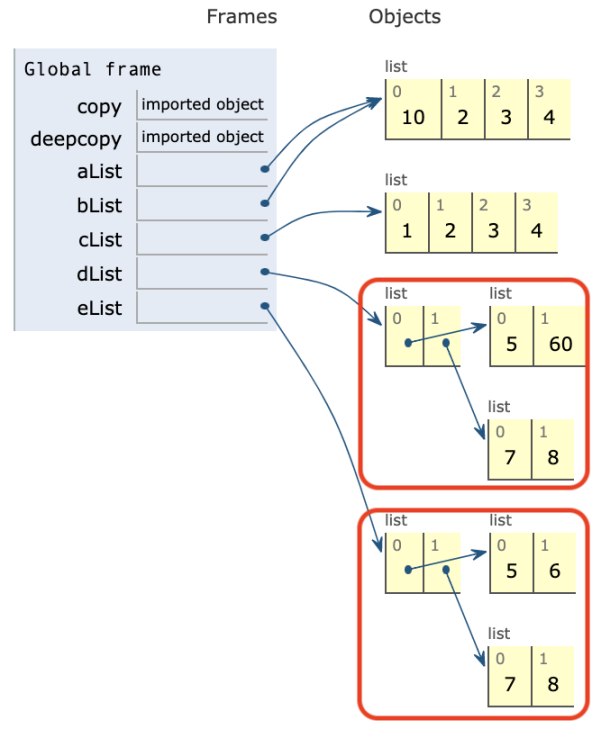
<< First | < Prev | Next > | Last >>

Done running (10 steps)

Customize visualization (NEW!)

Print output (drag lower right corner to resize)

```
얕은 복사: False
깊은 복사: False
```



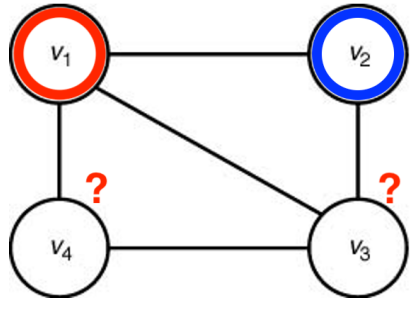
5절 그래프 색칠하기

m-색칠하기

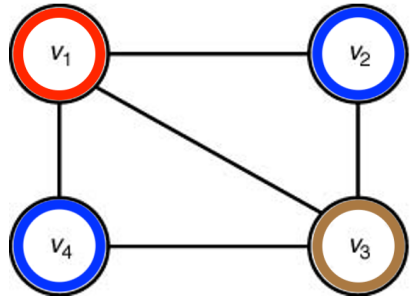
- 주어진 비방향그래프에서 서로 인접한 마디를 최대 m 개의 색상을 이용하여 서로 다른 색을 갖도록 색칠하는 문제

예제

- 아래 그래프에 대한 2-색칠하기 문제의 해답은 없음.



- 3-색칠하기 문제에 대해서는 해답 존재.



주요 응용분야

- 지도 색칠하기

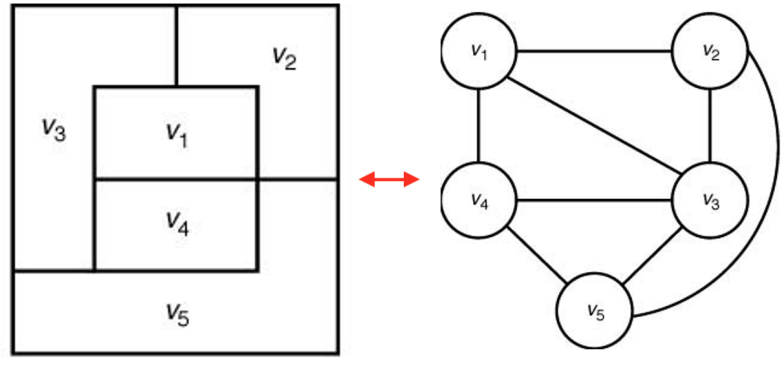
평면그래프

- 서로 교차하는 이음선이 없는 그래프

- 지도를 평면그래프로 변환 가능
    - 마디: 지도의 한 지역
    - 이음선: 서로 인접한 두 지역 연결

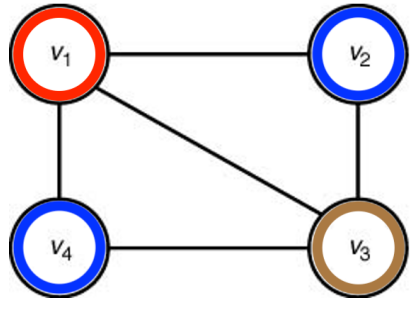
예제

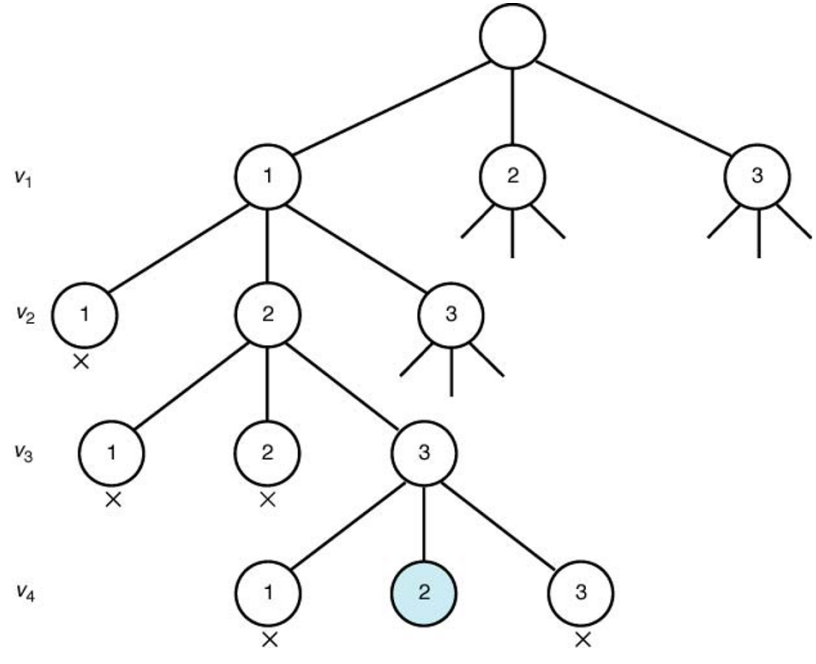
- 왼편의 지도를 오른편의 평면그래프로 변환 가능함.



예제: 3-색칠하기 문제 해결 되추적 알고리즘

```
colors = [빨강, 파랑, 갈색]
       = [1   , 2   , 3   ]
```




v_1
v_2
v_3
v_4
1
2
3
1
2
3
1
2
3
1
2
3

In [8]:
```
from typing import List, Dict

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

# 도메인: 각각의 마디에 칠할 수 있는 가능한 모든 색상
# 3-색칠하기: 1(빨강), 2(파랑), 3(갈색)
domains: Dict[int, List[int]] = {}
columns = [1, 2, 3]
for var in variables:
    domains[var] = columns
```

- 3-색칠하기 문제의 경우 각각의 마디에 동일하게 빨강, 파랑, 갈색 어느 색도 취할 수 있음. 단, 그 중에서 조건을 만족시키는 색상을 찾아야 함.

In [9]:
```
domains
```

Out[9]:
```
{1: [1, 2, 3], 2: [1, 2, 3], 3: [1, 2, 3], 4: [1, 2, 3]}
```

되추적 함수 구현

- 아래 되추적 함수 `backtracking_search_colors()`는 일반적인 m-색칠하기 문제를 해결함.
    - `assignment` 인자: 되추적 과정에서 일부의 변수에 대해 할당된 도메인 값의 정보를 담은 사전을 가리킴.
    - 인자가 들어오면 아직 값을 할당받지 못한 변수를 대상으로 유망성을 확인한 후 되추적 알고리즘 진행.
    - 되추적 알고리즘이 진행되면서 `assignment`가 확장되며 모든 변수에 대해 도메인 값이 지정될 때가지 재귀적으로 알고리즘이 진행됨.

In [10]:

```python
def backtracking_search_colors(assignment: Dict[int, int] = {}):
    """assignment: 각각의 변수를 키로 사용하고 키값은 해당 변수에 할당될 값"""

    # 모든 변수에 대한 값이 지정된 경우 조건을 만족시키는 해가 완성된 것임
    if len(assignment) == len(variables):
        return assignment
    # 아직 값을 갖지 않은 변수들이 존재하면 되추적 알고리즘을 아직 할당되지 않은 값을 대상으로 이어서 진행
    unassigned = [v for v in variables if v not in assignment]
    first = unassigned[0]

    for value in domains[first]:
        # 주의: 기존의 assignment를 보호하기 위해 복사본 활용
        # 되추적이 발생할 때 이전 할당값을 기억해 두기 위해서임.
        local_assignment = assignment.copy()
        local_assignment[first] = value
        # local_assignment 값이 유망하면 재귀 호출을 사용하여 변수 할당 이어감.
        if promissing_colors(first, local_assignment):
            result = backtracking_search_colors(local_assignment)
            # 유망성을 이어가지 못하면 되추적 실행
            if result is not None:
                return result

    return None
```

유망성 확인 함수

In [11]:
```
def promissing_colors(variable: int, assignment: Dict[int, int]):
    """새로운 변수 variable에 값을 할당 하면서 해당 변수와 연관된 변수들 사이의 제약조건이
    assignment에 대해 만족되는지 여부 확인

    m-색칠하기 문제의 경우: 이웃마디의 상태에 따라 제약조건이 달라짐.
                          즉, 마디 variable에 할당된 색이 이웃마디의 색과 달라야 함.
                          이를 위해 각각의 마디가 갖는 이웃마디들의 리스트를 먼저 확인해야 함."""
    # 각 마디에 대한 이웃마디의 리스트
    constraints = {
        1 : [2, 3, 4],
        2 : [1, 3],
        3 : [1, 2, 4],
        4 : [1, 3]
    }

    for var in constraints[variable]:
        if (var in assignment) and (assignment[var] == assignment[variable]):
            return False

    return True
```

In [12]:
```
backtracking_search_colors()
```

Out[12]:
```
{1: 1, 2: 2, 3: 3, 4: 2}
```

m-색칠하기 문제 되추적 알고리즘의 시간 복잡도

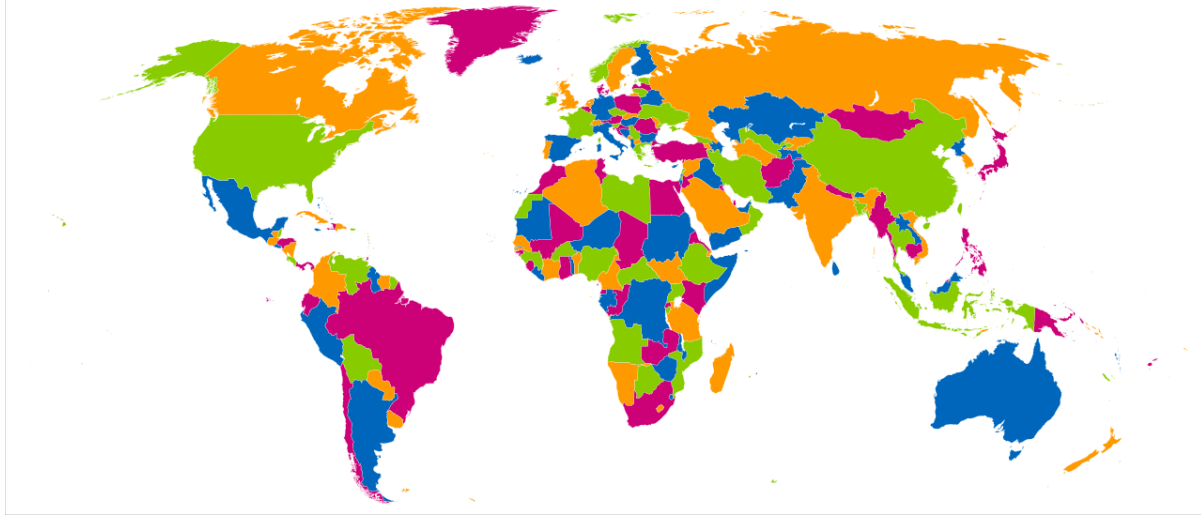
- n 개의 마디를 m 개의 색으로 칠해야 하는 문제의 상태공간트리의 마디의 수는 다음과 같음.

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

- 따라서 되추적 알고리즘이 최대 m과 n의 지승 만큼 많은 수의 마디를 검색해야 할 수도 있음.
- 하지만 검색해야 하는 마디 수는 경우마다 다름.
- 효율적인 알고리즘이 아직 알려지지 않음.

참고: 4색정리

- 4-색칠하기 문제는 언제나 해결가능함.



<그림 출처: 위키피디아: 4색정리 (https://ko.wikipedia.org/wiki/4색정리)>

- 1852년에 영국인 Francis Guthrie가 영국 지도를 작성할 때 인접한 각 주를 다른 색으로 칠하기 위해 필요한 최소한의 색상의 수에 대한 질문에서 유래한 문제임.

- 해결
    - 1976년에 K. Appel과 W. Haken 이 해결
    - 500페이지 이상의 증명으로 이루어졌으며 일부 증명은 컴퓨터 프로그램을 사용하였음.
    - 증명에 사용된 컴퓨터 프로그램에 대한 신뢰성 때문에 100% 인정받지 못하였음. 하지만 사용된 컴퓨터 프로그램의 문제가 발견된 것은 아님.
    - 2005년에 G. Gonthier에 의해 두 사람의 증명이 옳았음이 검증됨.

m-색칠하기 문제 해결가능성 판단 알고리즘

- m 이 1 또는 2인 경우: 쉽게 판단됨.
- $m = 3$ 인 경우: 효율적인 알고리즘 아직 찾지 못함.
    - 즉, 임의의 평면 지도에 대해 서로 인접한 지역은 다른 색상을 갖도록 3 가지 색상만을 이용하여 색칠할 수 있는지 여부를 판단하는 일이 매우 어려움.

부록: 제네릭 프로그래밍

- 특정 데이터 형식에 의존하지 않으면서 보다 일반적인 유형(type)에 대해 작동하는 알고리즘 작성 기법

- 동일한 알고리즘을 보다 일반적인 상황에 대해서 작동하도록 하는 프로그래밍 기법

- 제네릭 프로그래밍에 사용되는 기법은 다양함
    - 모듈 활용
        - 파이썬처럼 고차함수를 지원하는 프로그래밍언어의 경우 모듈화를 활용할 수 있음.
    - 다형성(polymorphism) 활용: 다양한 유형(자료형)을 마치 하나의 자료형 처럼 다루는 기법
        - 자바, 파이썬 등에서 제공하는 제네릭 프로그래밍 기법에 사용됨

모듈 활용

- n-퀸 문제의 되추적 함수 `backtracking_search_queens()`와 m-색칠하기 문제의 되추적 함수 `backtracking_search_colors()`는 사실상 동일한 알고리즘을 사용함.

- 차이점: 유망성을 확인하는 함수 `promissing_queens()`와 `promissing_colors()`가 서로 다름.

- 두 함수가 사용되는 위치를 `backtracking_search()` 함수의 매개변수(파라미터)로 추상화하여 사용 가능.
    - 대신에 두 개의 유망성 함수는 각각 독립된 모듈(파이썬 파일)로 저장. 서로 다른 모듈에 저장하기에 동일한 이름, 예를 들어 `promissing()` 등으로 지정 가능.

되추적 알고리즘: 고차 함수 활용

- `promissing` 매개변수 추가: 유망성 확인 함수를 인자로 받음.

- `promissing` 매개변수가 기대하는 함수 인자의 유형(type)
    - 매개변수 자료형: `int`와 `Dict[int, int]`
    - 반환값 자료형: `bool`
    - 따라서 `promissing` 함수의 유형: `Callable[[int, Dict[int, int]], bool]`

- 참조: 파이썬 유형 힌트 사용법 (https://python.flowdas.com/library/typing.html)

- 이어지는 코드는 두 개의 promissing 함수가 각각 `constraint_queens.py`와 `constraint_colors.py` 저장되어 있음을 가정함.

In [13]:

```python
from typing import List, Dict, Callable

def backtracking_search(promissing: Callable[[int, Dict[int, int]], bool],
                        assignment: Dict[int, int] = {}):
    """assignment: 각각의 변수를 키로 사용하고 키값은 해당 변수에 할당될 값"""

    # 모든 변수에 대한 값이 지정된 경우 조건을 만족시키는 해가 완성된 것임
    if len(assignment) == len(variables):
        return assignment

    # 아직 값을 갖지 않은 변수들이 존재하면 되추적 알고리즘을 아직 할당되지 않은 값을 대상으로 이어서 진행
    unassigned = [v for v in variables if v not in assignment]
    first = unassigned[0]
    for value in domains[first]:
        # 주의: 기존의 assignment를 보호하기 위해 복사본 활용
        # 되추적이 발생할 때 이전 할당값을 기억해 두기 위해서임.
        local_assignment = assignment.copy()
        local_assignment[first] = value
        # local_assignment 값이 유망하면 재귀 호출을 사용하여 변수 할당 이어감.
        if promissing(first, local_assignment):
            result = backtracking_search(promissing, local_assignment)
            # 유망성을 이어가지 못하면 되추적 실행
            if result is not None:
                return result

    return None
```

예제: 4-퀸과 4-색칠하기 문제

- 아래 코드는 4-퀸과 4-색칠하기 문제를 준비하는 세팅과정임.
    - 4-색칠하기 문제의 경우 3-색칠하기가 가능하면 3 가지 색상만 사용함.
    - 되추적 알고리즘의 특성상 가능하면 적은 열/색상을 사용함.

In [14]:
```
# 변수: 네 개의 퀸 또는 네 마디의 번호, 즉, 1, 2, 3, 4
variables : List[int] = [1, 2, 3, 4]

# 도메인: 네 개의 퀸이 위치할 수 있는 가능한 모든 열 또는 각각의 마디에 사용될 수 있는 가능한 모든 색상
domains: Dict[int, List[int]] = {}
columns = [1, 2, 3, 4]  # 네 개의 열 또는 색상
for var in variables:
    domains[var] = columns
```

4-퀸 문제 해결

In [15]:
```
from constraint_queens import promissing

print(backtracking_search(promissing))
```

```
{1: 2, 2: 4, 3: 1, 4: 3}
```

4-색칠하기 문제 해결

- 3 가직 색상만 사용되는 것에 주의할 것.

In [16]:
```
from constraint_colors import promissing

print(backtracking_search(promissing))
```

```
{1: 1, 2: 2, 3: 3, 4: 2}
```

다형성 활용

- n-퀸과 m-색칠하기 문제에서 변수 또는 도메인 값들이 숫자가 아닌 문자열 등 다른 자료형이 사용될 수 있음.
    - 체스판의 열은 원래 1에서 8까의 숫자 대신에 a 부터 h 사이의 알파벳으로 표시됨.
    - 지도를 색칠할 때 영역은 번호 보다는 이름으로 표현함.

- 하지만 자료형이 달라진다 하더라도 되추적 알고리즘 자체가 변하지는 않음. 다만, 유망성을 확인하는 함수가 사용되는 자료형에 따라 조금 수정될 필요는 있음.

- 예를 들어, n-퀸 문제의 유망성에서 대각선상에 위치하는지 여부를 판정할 때 숫자를 다루는 경우와 문자열을 다루는 경우는 조금 다를 수밖에 없음.

클래스 활용

- 서로 연관된 속성과 함수들을 하나의 클래스로 묶어서 활용할 수도 있음.
    - 되추적 알고리즘과 유망성 확인 함수를 하나의 클래스로 묶어서 사용
    - 문제에 따라 클래스의 인스턴스를 다르게 생성

- 주의사항
    - 되추적 알고리즘(`backtracking_search()`)는 변하지 않음.
    - 유망성 확인 메서드(`promissing()`)는 문제(인스턴스)별로 달라짐.

제네릭 클래스

- 문제마다 사용되는 자료형이 다를 수 있음.
    - 열 또는 색상: 1, 2, 3, 4 대신에 a, b, c, d 등 문자열 사용할 수도 있음.

- 되추적 알고리즘은 사용되는 자료형에 상관없이 작동함.

- 따라서 일반적인 유형에 대해 작동하는 제네릭 클래스의 메서드로 구현 가능

- 제네릭 클래스의 사용법은 다음과 같음.

```
from typing import Generic, TypeVar
  # 제네릭 변수. 여기서는 두 개의 제네릭 변수 지정
  # 해당 클래스의 인스턴스를 선언하면 적절한 자료형으로 자동 대체됨.
  V = TypeVar('V')
  D = TypeVar('D')

  # 아래 제네릭클래스에서 사용되는 변수 또는 값의 일부가 V 또는 D 자료형과 연관됨
  class 제네릭클래스(Generic[V, D]):
      클래스본문
```

추상클래스와 추상메서드

- 유망성 확인 함수인 `promissing()` 메서드는 추상메서드(abstract method)로 지정한 후 문제에 따라 다르게 정의해야 함.

- 추상메서드: 클래스 내에서 함수로 선언만 되고 정의되지 않은 상태로 남겨진 메서드

- 추상클래스: 추상메서드를 한 개 이상 포함한 클래스

- 추상클래스와 추상메서드 사용법은 다음과 같음.
    - 전제조건: abc 모듈에서 ABC 클래스와 abstractmethod 장식자 불러오기
    - 추상클래스는 ABC 클래스를 상속해야 함.

```
from abc import ABC, abstractmethod

class 추상클래스명(ABC):
  @abstractmethod
  def 추상메서드(self, 매개변수, ..., 매개변수):
      ...

  # 기타 메서드 및 속성
```

되추적 알고리즘 구현: 클래스 활용

- 이후에 사용되는 코드는 고전 컴퓨터 알고리즘 인 파이썬 (https://github.com/davecom/ClassicComputerScienceProblemsInPython) 의 3장 코드를 단순화하였음.

제네릭 클래스 선언

- 제네릭 클래스 선언에 필요한 제네릭 변수 지정
    - V: 주어진 문제에 사용되는 변수들의 자료형.
        - 예를 들어 1, 2, 3 등의 int 또는 서울, 경기, 인천 등의 str
        - 활용: `variables: List[V]`
    - D: 주어진 문제에 사용되는 도메인에 포함된 값들의 자료형.
        - 예를 들어 1, 2, 3 등의 int 또는 a, b, c, 빨강, 녹색, 파랑 등의 str
        - 활용: `domains: List[D]`

In [17]:
```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod

V = TypeVar('V')
D = TypeVar('D')
```

- `Constraint` 클래스 선언
    - 유망성 확인함수를 추상메서드로 가짐.
    - `Generic` 클래스와 `ABC` 클래스 모두 상속

In [18]:
```
class Constraint(Generic[V, D], ABC):
    @abstractmethod
    def promissing(self, variable: V, assignment: Dict[V, D]) -> bool:
        ...
```

- `Backtracking` 클래스 선언
    - 되추적 알고리즘을 메서드로 가짐.
    - `Generic` 클래스 상속
- 주의: `Optional` : 경우에 따라 `None` 이 값으로 사용될 수 있음을 암시함.
    - `Optional[T]`의 의미: 기본적으로 자료형 T의 값을 사용하지만 `None` 이 사용될 수도 있음.

In [19]:

```
class Backtracking(Generic[V,D]):
    # 제약조건을 지정하면서 인스턴스 생성
    def __init__(self,
                 variables: List[V],
                 domains: Dict[V, List[D]],
                 constraint: Constraint[V,D]) -> None:
        self.variables: List[V] = variables
        self.domains: Dict[V,List[D]] = domains
        self.constraint = constraint

    # promissing 메서드가 Constraint 클래스에서 선언되었기에 따로 인자로 받지 않음.
    def backtracking_search(self,
                            assignment: Dict[V,D] = {}) -> Optional[Dict[V,D]]:
        if len(assignment) == len(self.variables):
            return assignment
        unassigned: List[V] = [v for v in self.variables if v not in assignment]
        first: V = unassigned[0]
        for value in self.domains[first]:
            local_assignment = assignment.copy()
            local_assignment[first] = value
            if constraint.promissing(first, local_assignment):
                result = self.backtracking_search(local_assignment)
                if result is not None:
                    return result

        return None
```

8-퀸 문제 해결하기

- 8개의 퀸과 8개의 열 사용하기
    - 열을 a, b, ..., h 로 사용

- 퀸이 위치하는 열을 번호가 아닌 문자열로 처리하기에 `promissing()` 메서드 정의가 이전과 조금 달라짐.
    - 동일한 열 또는 동일한 대각선 상에 있는지 여부는 순서가 중요함.
    - `enumerate()` 함수를 활용하여 항목의 인덱스를 활용할 수 있음.

In [20]:
```
# 변수: 여덟 개의 퀸 1, 2, ..., 8
variables : List[int] = [1, 2, 3, 4, 5, 6, 7, 8]

# 도메인: 여덟 개의 퀸이 위치할 수 있는 가능한 모든 열. 문자열 사용
domains: Dict[int, List[str]] = {}
columns = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

for var in variables:
    domains[var] = columns

# columns의 순서 지정하기
col2ind = {column : index+1 for (index, column) in enumerate(columns)}
```

- 문제에 맞는 `Constraint` 클래스의 인스턴스 생성하려면 먼저 `Constraint` 클래스를 상속하는 구상클래스를 선언해야 함.
    - 이유: 추상 클래스는 바로 인스턴스를 만들 수 없음.
    - `promissing()` 메서드를 구현해야 함.
        - 이전에 사용한 constraint_queens 모듈의 `promissing()` 함수와 동일.
    - 주의: 열의 순서가 중요하기에 `col2ind` 사전을 활용함

In [21]:

```python
class ConstraintQueens(Constraint[int, str]):

    # promissing 메서드 정의해야 함.
    def promissing(self, variable: int, assignment: Dict[int, str]) -> bool:
        # q1r, q1c: 첫째 퀸이 놓인 마디의 열과 행
        # 키값이 문자열이기에 해당 문자열의 인덱스를 대신 사용함. 즉, col2ind 활용
        for q1r, q1c in assignment.items():
            q1c = col2ind[assignment[q1r]] # 첫째 퀸의 열
            # q2r = 첫째 퀸 아래에 위치한 다른 모든 퀸들을 대상으로 조건만족여부 확인
            for q2r in range(q1r + 1, len(assignment) + 1):
                q2c = col2ind[assignment[q2r]] # 둘째 퀸의 열
                if q1c == q2c: # 동일 열에 위치?
                    return False
                if abs(q1r - q2r) == abs(q1c - q2c): # 대각선상에 위치?
                    return False

        # 모든 변수에 대해 제약조건 만족됨
        return True
```

In [22]:
```
constraint = ConstraintQueens()
backtracking = Backtracking(variables, domains, constraint)

backtracking.backtracking_search()
```

Out[22]:
```
{1: 'a', 2: 'e', 3: 'h', 4: 'f', 5: 'c', 6: 'g', 7: 'b', 8: 'd'}
```

4-퀸 문제

In [23]:
```python
# 변수: 네 개의 퀸 1, 2, 3, 4
variables : List[int] = [1, 2, 3, 4]

# 도메인: 네 개의 퀸이 위치할 수 있는 가능한 모든 열. 문자열 사용
domains: Dict[int, List[str]] = {}
columns = ['a', 'b', 'c', 'd']

for var in variables:
    domains[var] = columns

# columns의 순서 지정하기
col2ind = {column : index+1 for (index, column) in enumerate(columns)}
```

In [24]:
```python
constraint = ConstraintQueens()
backtracking = Backtracking(variables, domains, constraint)

backtracking.backtracking_search()
```

Out[24]:
```
{1: 'b', 2: 'd', 3: 'a', 4: 'c'}
```

4-색칠하기

- 색상을 번호가 아닌 RGB(빨강, 녹색, 파랑)으로 표현
    - 이전에 사용한 constraint_colors 모듈의 `promissing()` 함수와 동일.
    - n-퀸 문제와는 달리 도메인 값의 순서가 중요하지 않음.

In [25]:
```
# 변수: 네 개의 퀸 또는 네 마디의 번호, 즉, 1, 2, 3, 4
variables : List[int] = [1, 2, 3, 4]

# 도메인: 네 개의 퀸이 위치할 수 있는 가능한 모든 열 또는 각각의 마디에 사용될 수 있는 가능한 모든 색상
domains: Dict[int, List[str]] = {}
columns = ['R', 'G', 'B']
for var in variables:
    domains[var] = columns
```

- 문제에 맞는 Constraint 클래스를 구상클래스로 상속하기

In [26]:
```python
class ConstraintColors(Constraint[int, str]):

    # promissing 메서드 정의해야 함.
    def promissing(self, variable: int, assignment: Dict[int, str]) -> bool:
        # 각 마디에 대한 이웃마디의 리스트
        constraints = {
            1 : [2, 3, 4],
            2 : [1, 3],
            3 : [1, 2, 4],
            4 : [1, 3]
        }

        for var in constraints[variable]:
            if (var in assignment) and (assignment[var] == assignment[variable]):
                return False

        return True
```

In [27]:
```
constraint = ConstraintColors()
backtracking = Backtracking(variables, domains, constraint)

backtracking.backtracking_search()
```

Out[27]:
```
{1: 'R', 2: 'G', 3: 'B', 4: 'G'}
```