제네릭 프로그래밍

주요 내용

- 제네릭 프로그래밍 소개
- 제네릭 되추적 알고리즘

제네릭 프로그래밍이란?

- 하나의 자료형에만 의존하지 않고 공통 속성을 갖는 여러 자료형daya type에 대해 동일하게 작동하는 알고리즘을 작성하는 기법
- 제네릭 프로그래밍 기법으로 작성된 알고리즘은 하나의 문제만이 아닌 공통 속성을 갖는 여러 문제를 대상으로 적용 가능
- 제네릭 프로그래밍 구현 기법
    - 모듈 활용: 파이썬처럼 고계 함수higher-order function를 지원하는 프로그래밍언어의 경우 활용
    - 다형성polymorphism 활용: 다양한 자료형을 마치 하나의 자료형 처럼 다루는 기법. 자바, 파이썬 등에서 제공하는 제네릭 클래스 활용.

모듈 활용

다음 두 알고리즘은 사실상 동일한 되추적 알고리즘을 사용함

- n-여왕말 문제: `backtracking_search_queens(num_queens, assignment)` 함수
- m-색칠 문제: `backtracking_search_colors(num_nodes, num_colors, constraints, assignment)` 함수

두 알고리즘의 근본적인 차이점: 유망성 함수

- n-여왕말 문제: `promissing_queens()` 함수
- m-색칠 문제: `promissing_colors()` 함수

`backtracking_search()` 함수: 제네릭 되추적 알고리즘

In [1]:

```
from typing import List, Dict, Callable

def backtracking_search(variables: List[int],
                        domains: Dict[int, List[int]],
                        promissing: Callable[[int, Dict[int, int]], bool],
                        assignment: Dict[int, int] = {}):

    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 promissing(first, local_assignment):
            result = backtracking_search(variables, domains,
                                         promissing, local_assignment)
            if result is not None:
                return result

    return None
```

`promissing` 매개변수의 자료형으로 지정된 `Callable[[int, Dict[int, int]], bool]`의 의미

- 함수 인자 사용: `Callable`
- 두 개의 인자 사용: `[int, Dict[int, int]]`
    - 첫째 인자: 정수(`int`)
    - 둘째 인자: 키와 값 모두 정수를 사용하는 사전 객체(`Dict[int, int]`)
- 반환값 자료형: `bool`

자료형 힌트

- 파이썬 최신 버전: 함수의 매개변수의자료형을 명시 가능
- 참고: 자료형 힌트 지원

4-여왕말 문제 해결

- 변수: 네 개의 여왕말

In [2]:
```
variables : List[int] = [1, 2, 3, 4]
```

- 도메인: 하나의 여왕말이 위치할 수 있는 가능한 모든 칸. 항상 네 종류의 값 사용.

In [3]:
```
domains: Dict[int, List[int]] = {}

columns = [1, 2, 3, 4]  # 네 개의 열 또는 색
for var in variables:
    domains[var] = columns
```

4-여왕말 문제의 유망성 함수가 `constraint_queens` 모듈에 선언되어 있다고 가정

```
from typing import List, Dict

def promissing(variable: int, assignment: Dict[int, int]):

    for q1r, q1c in assignment.items():

        for q2r in range(q1r + 1, len(assignment) + 1):
            q2c = assignment[q2r]
            if q1c == q2c:
                return False
            if abs(q1r - q2r) == abs(q1c - q2c):
                return False

    return True
```

In [4]:
```
from constraint_queens import promissing

backtracking_search(variables, domains, promissing=promissing)
```

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

4-색칠 문제 해결

- 변수: 다섯 개의 노드

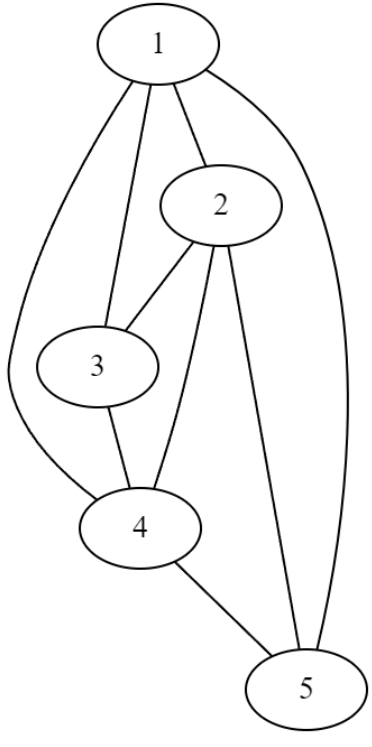
In [5]:
```
variables : List[int] = [1, 2, 3, 4, 5]
```

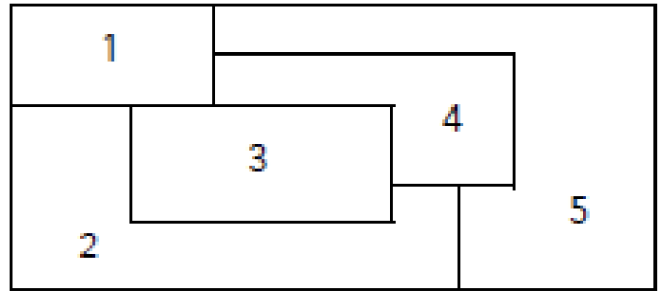
- 도메인: 하나의 노드가 가질 수 있는 가능한 모든 색. 항상 네 종류의 색 사용.

In [6]:
```
domains: Dict[int, List[int]] = {}

columns = [1, 2, 3, 4]  # 네 개의 열 또는 색
for var in variables:
    domains[var] = columns
```

아래 그래프와 같은 4-색칠 문제 해결에 사용되는 유망성 판단 함수는 다음과 같으며 `constraint_colors` 모듈에 저장되어 있다고 가정





```python
from typing import List, Dict

def promissing(variable: int, assignment: Dict[int, int]):

    constraints = {
        1 : [2, 3, 4, 5],
        2 : [1, 3, 4, 5],
        3 : [1, 2, 4],
        4 : [1, 2, 3, 5]
    }

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

    return True
```

In [7]:
```
from constraint_colors import promissing

backtracking_search(variables, domains, promissing=promissing)
```

Out[7]:
```
{1: 1, 2: 2, 3: 3, 4: 4, 5: 3}
```