인공지능

programming #2 (CSP Programming)

C184036 이소연

제출일 : 11월 25일 (토)

- Problem #0 : 프로그램에서 주어진 데이터를 가지고, 각 프로그램을 수행해 본 결과를 비교

- Problem #1 : (1) 암호 풀이 문제) (2) 호주 지도 Map Coloring

- Problem #2 : 대한민국의 지도 색칠하기 문제

< Problem #0>

1. heuristic만 사용

```
for x, y in CONSTRAINTS:
    problem.addConstraint(lambda x, y: x != y, (x, y))

# Solve problem
for solution in problem.getSolutions():
    print(solution)
```

```
{'E': 'Wednesday', 'B': 'Tuesday', 'C': 'Monday', 'F': 'Tuesday', 'A': 'Wednesday', 'D': 'Monday', 'G': 'Monday'}
{'E': 'Wednesday', 'B': 'Monday', 'C': 'Tuesday', 'F': 'Monday', 'A': 'Wednesday', 'D': 'Tuesday', 'G': 'Tuesday'}
{'E': 'Tuesday', 'B': 'Wednesday', 'C': 'Monday', 'F': 'Wednesday', 'A': 'Tuesday', 'D': 'Monday', 'G': 'Monday'}
{'E': 'Tuesday', 'B': 'Monday', 'C': 'Wednesday', 'F': 'Monday', 'A': 'Tuesday', 'D': 'Wednesday', 'G': 'Wednesday'}
{'E': 'Monday', 'B': 'Tuesday', 'C': 'Wednesday', 'F': 'Tuesday', 'A': 'Monday', 'D': 'Wednesday', 'G': 'Wednesday'}
{'E': 'Monday', 'B': 'Wednesday', 'C': 'Tuesday', 'F': 'Wednesday', 'A': 'Monday', 'D': 'Tuesday', 'G': 'Tuesday'}
```

2. heuristic + backtracking

```
# 문제 해결
for solution in problem.getSolutions():
    print(solution)
```

```
{'E': 'Wednesday', 'B': 'Tuesday', 'C': 'Monday', 'F': 'Tuesday', 'A': 'Wednesday', 'D': 'Monday', 'G': 'Monday'}
{'E': 'Wednesday', 'B': 'Monday', 'C': 'Tuesday', 'F': 'Monday', 'A': 'Wednesday', 'D': 'Tuesday', 'G': 'Tuesday'}
{'E': 'Tuesday', 'B': 'Wednesday', 'C': 'Monday', 'F': 'Wednesday', 'A': 'Tuesday', 'D': 'Monday', 'G': 'Monday'}
{'E': 'Tuesday', 'B': 'Monday', 'C': 'Wednesday', 'F': 'Monday', 'A': 'Tuesday', 'D': 'Wednesday', 'G': 'Wednesday'}
{'E': 'Monday', 'B': 'Tuesday', 'C': 'Wednesday', 'F': 'Tuesday', 'A': 'Monday', 'D': 'Wednesday', 'G': 'Wednesday'}
{'E': 'Monday', 'B': 'Wednesday', 'C': 'Tuesday', 'F': 'Wednesday', 'A': 'Monday', 'D': 'Tuesday', 'G': 'Tuesday'}
```

```
# 백트래킹 횟수 출력
print(problem.backtrack_counter)
```

```
{'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0, 'G': 0}
```

3. 그냥 backtracking만 사용

```
solution = backtrack(dict())
print("Solution:", solution)
print("Backtracking Counts:", backtrack_counter)
```

```
Solution: {'A': 'Monday', 'B': 'Tuesday', 'C': 'Wednesday', 'D': 'Wednesday', 'E': 'Monday', 'F': 'Tuesday', 'G': 'Wednesday'}
Backtracking Counts: {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0, 'G': 0}
```

- 결과

heuristic을 사용한 것이 단순 backtracking 기법을 사용한 것보다 성능이 더 좋다. 그리고 단순한 문제라 그런지 backtracking은 일어나지 않았다.

<Problem #1_(1)>

1. heuristic만 사용

```
# Output the solutions
for solution in solutions:
    print(f"T={solution['T']} W={solution['W']} O={solution['O']} F={solution['F']} U={solution['U']} R={solution['R']}")
```

```
T=9 W=3 O=8 F=1 U=7 R=6
T=9 W=2 O=8 F=1 U=5 R=6
T=8 W=6 O=7 F=1 U=3 R=4
T=8 W=4 O=6 F=1 U=9 R=2
T=8 W=3 O=6 F=1 U=7 R=2
T=7 W=6 O=5 F=1 U=3 R=0
T=7 W=3 O=4 F=1 U=6 R=8
```

2. heuristic + backtracking

```
# 제약 조건 체크 횟수 출력
print("제약 조건 체크 횟수 (대략적인 백트래킹 횟수):", constraint_check_counter)
```

```
T=9 W=3 O=8 F=1 U=7 R=6
T=9 W=2 O=8 F=1 U=5 R=6
T=8 W=6 O=7 F=1 U=3 R=4
T=8 W=4 O=6 F=1 U=9 R=2
T=8 W=3 O=6 F=1 U=7 R=2
T=7 W=6 O=5 F=1 U=3 R=0
T=7 W=3 O=4 F=1 U=6 R=8
제약 조건 체크 횟수 (대략적인 백트래킹 횟수): 967687
```

3. 그냥 backtracking만 사용

```
# Start the backtracking process with an empty assignment and a counter
solution, backtracking_steps = backtrack({}, 0)
print("Solution:", solution)
print("Backtracking steps:", backtracking_steps)
```

```
Solution: {'F': 0, 'T': 1, 'U': 4, 'W': 7, 'R': 6, 'O': 3}
Backtracking steps: 1336
```

- 결과

heuristic을 사용한 backtracking이 더 많이 일어났다. heuristic 기법의 성능이 더 좋다.

<Problem #1_(2)>

1. heuristic만 사용

```
# 제약 조건을 문제에 추가합니다.
for x, y in constraints:
    problem.addConstraint(lambda x, y: x != y, (x, y))

# 가능한 해결책들을 찾습니다.
solutions = problem.getSolutions()

# 찾은 첫 번째 해결책을 표시합니다.
first_solution = solutions[0] if solutions else None
print(first_solution)
```

```
{'SA': 'blue', 'NSW': 'green', 'Q': 'red', 'NT': 'green', 'V': 'red', 'WA': 'red', 'T': 'blue'}
```

2. heuristic + backtracking

```
# 솔루션을 반복하면서 각 변수의 값이 변경될 때마다 카운트를 증가시킵니다.
# 이 예시에서는 단순히 모든 솔루션을 출력합니다.
for solution in solutions:
    print(solution)
    # 값 변경 횟수를 업데이트하는 로직은 CSP 솔버의 내부 구현에 따라 달라집니다.
    # 여기에서는 이를 시뮬레이션하기 위한 구체적인 로직이 없습니다.

# 값 변경 횟수를 출력합니다.
print("Value Change Count:", change_count)
```

```
{'SA': 'blue', 'NSW': 'green', 'Q': 'red', 'NT': 'green', 'V': 'red', 'WA': 'red', 'T': 'blue'}
{'SA': 'blue', 'NSW': 'green', 'Q': 'red', 'NT': 'green', 'V': 'red', 'WA': 'red', 'T': 'green'}
{'SA': 'blue', 'NSW': 'green', 'Q': 'red', 'NT': 'green', 'V': 'red', 'WA': 'red', 'T': 'red'}
{'SA': 'blue', 'NSW': 'red', 'Q': 'green', 'NT': 'red', 'V': 'green', 'WA': 'green', 'T': 'blue'}
{'SA': 'blue', 'NSW': 'red', 'Q': 'green', 'NT': 'red', 'V': 'green', 'WA': 'green', 'T': 'green'}
{'SA': 'blue', 'NSW': 'red', 'Q': 'green', 'NT': 'red', 'V': 'green', 'WA': 'green', 'T': 'red'}
{'SA': 'green', 'NSW': 'blue', 'Q': 'red', 'NT': 'blue', 'V': 'red', 'WA': 'red', 'T': 'blue'}
{'SA': 'green', 'NSW': 'blue', 'Q': 'red', 'NT': 'blue', 'V': 'red', 'WA': 'red', 'T': 'green'}
{'SA': 'green', 'NSW': 'blue', 'Q': 'red', 'NT': 'blue', 'V': 'red', 'WA': 'red', 'T': 'red'}
{'SA': 'green', 'NSW': 'red', 'Q': 'blue', 'NT': 'red', 'V': 'blue', 'WA': 'blue', 'T': 'blue'}
{'SA': 'green', 'NSW': 'red', 'Q': 'blue', 'NT': 'red', 'V': 'blue', 'WA': 'blue', 'T': 'green'}
{'SA': 'green', 'NSW': 'red', 'Q': 'blue', 'NT': 'red', 'V': 'blue', 'WA': 'blue', 'T': 'red'}
{'SA': 'red', 'NSW': 'green', 'Q': 'blue', 'NT': 'green', 'V': 'blue', 'WA': 'blue', 'T': 'blue'}
{'SA': 'red', 'NSW': 'green', 'Q': 'blue', 'NT': 'green', 'V': 'blue', 'WA': 'blue', 'T': 'green'}
{'SA': 'red', 'NSW': 'green', 'Q': 'blue', 'NT': 'green', 'V': 'blue', 'WA': 'blue', 'T': 'red'}
{'SA': 'red', 'NSW': 'blue', 'Q': 'green', 'NT': 'blue', 'V': 'green', 'WA': 'green', 'T': 'blue'}
{'SA': 'red', 'NSW': 'blue', 'Q': 'green', 'NT': 'blue', 'V': 'green', 'WA': 'green', 'T': 'green'}
{'SA': 'red', 'NSW': 'blue', 'Q': 'green', 'NT': 'blue', 'V': 'green', 'WA': 'green', 'T': 'red'}
Value Change Count: {'WA': 0, 'NT': 0, 'Q': 0, 'NSW': 0, 'V': 0, 'SA': 0, 'T': 0}
```

3. 그냥 backtracking만 사용

```
solution = backtrack(dict())
print(solution)
print(backtracking_count)
```

```
{'WA': 'red', 'NT': 'green', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'SA': 'blue', 'T': 'red'}
{'WA': 0, 'NT': 1, 'Q': 0, 'NSW': 1, 'V': 0, 'SA': 2, 'T': 0}
```

- 결과

heuristic + backtracking 이 성능이 제일 좋았다. 단순 backtracking 기법만 사용했을 때 backtracking 이 사용되었다.

<Problem #2>

- 코드

```
from constraint import *
```

```
# 대한민국의 광역시와 특별자치시도
regions = ["서울특별시", "부산광역시", "대구광역시", "인천광역시", "광주광역시", "대전광역시", "울산광역시", "세종특별자치시",
           "경기도", "강원도", "충청북도", "충청남도", "전라북도", "전라남도", "경상북도", "경상남도", "제주특별자치도"]

# 인접한 지역 정보
adjacencies = {
    "서울특별시": ["경기도"],
    "부산광역시": ["경상남도"],
    "대구광역시": ["경상북도"],
    "인천광역시": ["경기도"],
    "광주광역시": ["전라남도"],
    "대전광역시": ["충청남도", "충청북도"],
    "울산광역시": ["경상남도"],
    "세종특별자치시": ["충청남도"],
    "경기도": ["서울특별시", "인천광역시", "충청북도", "강원도"],
    "강원도": ["경기도", "충청북도"],
    "충청북도": ["서울특별시", "대전광역시", "충청남도", "강원도", "경기도"],
    "충청남도": ["세종특별자치시", "대전광역시", "충청북도", "경상남도"],
    "전라북도": ["전라남도", "경상북도"],
    "전라남도": ["광주광역시", "경상남도", "전라북도"],
    "경상북도": ["대구광역시", "울산광역시", "경상남도", "전라북도"],
    "경상남도": ["부산광역시", "울산광역시", "대구광역시", "경상북도", "전라남도", "충청남도"],
    "제주특별자치도": []  # 제주는 다른 지역과 인접하지 않음
}

# 문제 초기화
problem = Problem()
```

```
# 각 지역에 대해 색상 변수를 추가합니다.
problem.addVariables(regions, [1, 2, 3])
```

```
# 인접한 지역은 서로 다른 색을 가져야 합니다.
for region, neighbors in adjacencies.items():
    for neighbor in neighbors:
        problem.addConstraint(lambda x, y: x != y, (region, neighbor))
```

```
# 해결책 찾기
solutions = problem.getSolutions()
```

- 결과

```
# 결과 출력
print(f"총 해결책 수: {len(solutions)}")
if solutions:
    print("한 예시 해결책:", solutions[0])
```

```
총 해결책 수: 6912
한 예시 해결책: {'경상남도': 3, '충청북도': 3, '경기도': 2, '충청남도': 2, '경상북도': 2, '전라남도': 2, '강원도': 1, '대전광역시': 1, '전라북도': 3, '대구광역시': 1, '서울특별시': 1, '울산광역시': 1, '광주광역시': 3, '부산광역시': 2, '세종특별자치시': 3, '인천광역시': 3, '제주특별자치도': 3}
```

총 해결책 수는 6912개이다.

<전체적 결과>

전체적으로 성능은 모두 heuristic이 성능이 좋았다.