

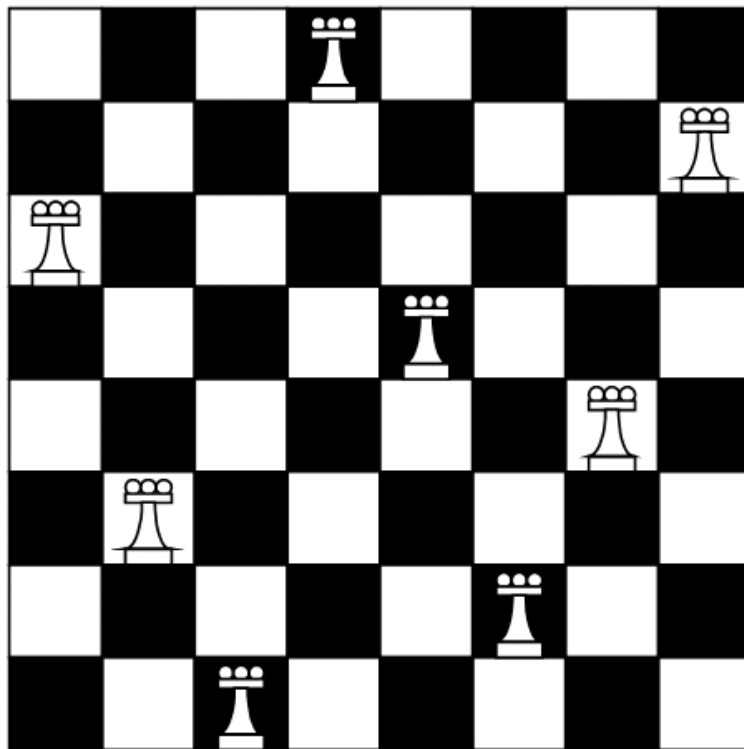
## 개요

백트래킹이란 모든 경우의 수를 전부 고려하는 알고리즘이다. 백트래킹은 구하고자 하는 해를 튜플로 나타내고 튜플에 기준 함수를 적용했을 때의 결과가 최대치, 최소치 혹은 일정 조건을 만족하게끔 만들어주는 퇴각 검색 기법으로 정의된다. 백트래킹은 해답이 될 수 있는 튜플을 완성해 나아가며 그 과정에서 미완성된 튜플에 한정 함수를 적용하여 해답의 가능성이 없는 튜플들은 더이상 진행시키지 않는 방법을 사용한다. 이때 튜플을 만드는 과정에서 스택을 사용하여 한정 함수를 만족하면 Push 만족하지 않으면 Pop 을 하는 방법을 사용한다. 이런 과정이 퇴각과 전진하는 것과 비슷하다 하여 백트래킹이라 불린다.//그래서 구조적으로 깊이 우선 탐색을 기반으로 하며 DFS의 구조를 적용할 수 있는 문제에 활용할 수 있다. 그러나 백트래킹의 경우 가지치기가 일어나므로 원시적인 방법으로 모든 경우의 수를 확인하는 알고리즘은 아니다. 반대로 분기 한정법의 경우 백트래킹과는 다르게 분기와 한정을 정해서 해를 구하는 기법이다. 백트래킹을 구현할 때 경우의 수가 너무 많아 모든 경우를 다 해보기 어려울 때, 사용하면 좋다. 가지치기 하는 기준이 되는 값이 바운드 값이다. 또한 분기 한정법의 경우에는 너비 우선 탐색을 기반으로 한다. 이번 실험에서는 이러한 백트래킹의 대표적인 문제인 NQueen 문제와 부분 집합의 합 문제를 통해서 알아보고자 한다.

### 1. 8-Queen + Monte Carlo

이번 문제에서 주어진 조건은  $n=8$ 이며 몬테카를로 알고리즘을 이용하여 20 번을 시행해보고 알고리즘이 만들어내는 가지 친 상태 공간 트리에서의 마디의 개수 추정치의 평균을 구해 볼 것이다.

먼저 N-Queen 문제부터 살펴보도록 하자. N-Queen 문제는  $N \times N$  크기의 체스판에서 N 개의 퀸을 서로 공격할 수 없도록 올려놓는 문제이다. 이때 퀸은 체스에서 가장 강력한 기물로 자신의 위치에서 상하좌우 그리고 대각선 방향으로 이어진 직선상의 어떤 기물도 공격할 수 있다.



[그림 1] 8x8 체스판에 8 개의 퀸을 서로 공격할 수 없도록 올려놓은 예

우선 이 문제를 풀기 위해 생각할 수 있는 가장 간단한 방법은 전수 조사를 하는 것이다. 퀸이 올 수 있는 모든 경우의 수를 두고, 그 중에서 답을 찾는 방법이다. N이 4 라고 가정한다면, 각 퀸은 한 열에 하나씩만 올 수 있기 때문에 이때 점검해야 할 모든 경우의 수는  $4 \times 4 \times 4 \times 4 = 256$  가지가 된다. 즉 이것은 재귀 알고리즘과 같게 된다. 그러나 이와 같은 방법은 굳이 탐색할 필요도 없는 경우까지 모두 탐색하기 때문에 비효율적이다.

그래서 이와 같은 문제를 조금 더 효율적으로 풀기 위한 방법이 백트래킹 알고리즘이다. 백트래킹 알고리즘은 전수조사 방법중 하나인 깊이우선탐색으로 시작한다. 그리고 각 노드에서 유망하지 않은 노드들은 탐색하지 않고 유망한 노드에 대해서만 탐색을 한다. 백트래킹 알고리즘은 다음과 같은 순서로 진행된다.

1. 처음 깊이 우선 탐색을 시작한다.
2. 각 노드가 유망한지 검사한다.
3. 만약 유망하다면 탐색을 계속한다.
4. 만약 유망하지 않다면 그의 부모 노드로 돌아가서 탐색을 계속한다(백트래킹).

[그림 2] 백트래킹 알고리즘

이러한 백트래킹 알고리즘을 이용하면 깊이 우선 탐색보다 검색하는 경우의 수를 줄일 수 있다. 그렇다면 이제 실제 문제로 적용해보자. 위처럼 백트래킹 알고리즘을 사용하기 위해 유망성 검사하는 규칙을 정하면 다음과 같다.

1. 퀸은 같은 행/열에 있으면 안된다.
2. 퀸은 다른 퀸의 대각선의 연장선 상에 있으면 안된다.

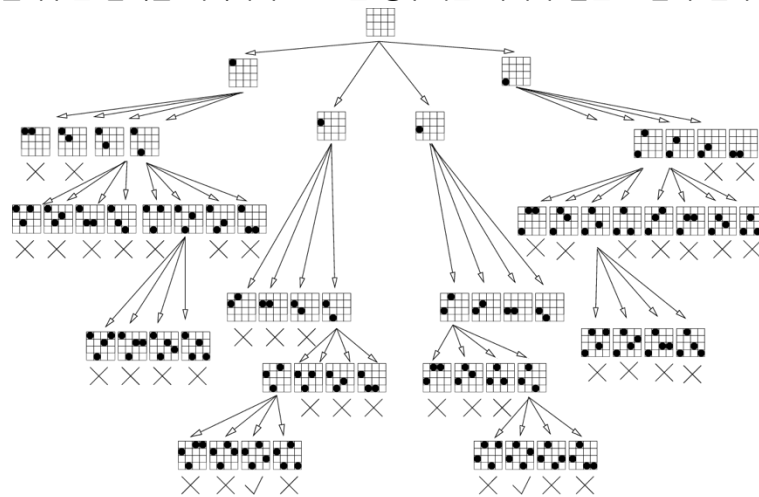
[그림 3] N-Queen 문제 규칙

이 규칙을 코드로 구현하면 다음과 같다.

```
int promising(int i) {
    int k = 1, Switch = true;
    while(k < i && Switch) {
        if(col[i] == col[k] || abs(col[i] - col[k]) == i - k)
            Switch = false;
        k++;
    }
    return Switch;
}
```

[코드 1] N-Queen promising()

이 규칙을 가지고 깊이우선 탐색을 시작하여 N=4 인 경우에는 아래와 같은 그림이 된다.



[그림 4] N=4 일때 상태 공간 트리

위의 그림처럼 유망한 노드라면 자식 노드로 들어가 다시 탐색을 시작하고, 유망하지 않다면 탐색을 그만두고 다시 부모노드로 올라가 다음 노드를 탐색한다. 이러한 방식으로 끝까지 탐색하는 것이 백트래킹 알고리즘이다. 이것을 코드로 구현하면 다음과 같다.

몬테카를로 알고리즘은 폴란드계 미국인 수학자 스타니스와프 울람이 제안한 알고리즘이다. 이 알고리즘은 원하는 결과값을 정확한 값을 얻는 방법이 아닌 난수를 이용하여 어떤 함수의 답을 확률적으로 근접하게 계산하는

방식이다. 이제 위에서 작성한 규칙과 백트래킹 알고리즘을 사용하고, 몬테카를로 알고리즘으로  $N = 8$  일 때 추정 평균치를 구하는 코드를 작성하면 다음과 같다.

```
int estimate_n_queens(int n) {
    int i = 0, j, m = 1;
    int numnodes = 1, mprod = 1;
    int random;

    while(m != 0 && i != n) {
        mprod = mprod * m;
        numnodes = numnodes + mprod * n;
        i++;
        m = 0;
        int prom_children[MAX + 1] = {0};
        for(j = 1; j <= n; j++) {
            col[i] = j;
            if(promising(i)) {
                m++;
                prom_children[j] = true;
            }
        }
        if(m != 0) {
            while(1) {
                random = (rand() % MAX) + 1;
                if(prom_children[random] == 1) {
                    j = random;
                    break;
                }
            }
            col[i] = j;
        }
    }
    return numnodes;
}
```

[코드 2] estimate\_n\_queens()

주어진 문제를 이 알고리즘을 통해 20 회를 측정하고 평균치를 구하면 다음과 같은 결과가 나온다.

```
seq[0]: numnodes = 8137
seq[1]: numnodes = 14281
seq[2]: numnodes = 11913
seq[3]: numnodes = 7113
seq[4]: numnodes = 4233
seq[5]: numnodes = 11913
seq[6]: numnodes = 9673
seq[7]: numnodes = 17033
seq[8]: numnodes = 33033
seq[9]: numnodes = 9033
seq[10]: numnodes = 14281
seq[11]: numnodes = 4233
seq[12]: numnodes = 17033
seq[13]: numnodes = 12873
seq[14]: numnodes = 21513
seq[15]: numnodes = 13193
seq[16]: numnodes = 13193
seq[17]: numnodes = 25801
seq[18]: numnodes = 13129
seq[19]: numnodes = 10633
tot = 272244    avg = 13612
```

[그림 5] estimate\_n\_queens() 결과

즉 평균적으로 약 13600 개의 마디가 추정되는 것을 확인할 수 있다.

## 2. 부분집합의 합 + Monte Carlo

부분집합의 합은  $N$  개의 서로 다른 양수들이 오름차순으로 주어지고 이들의 부분집합의 합이  $m$  이 되는 모든 부분집합을 찾는 문제로 아래와 같은 집합이 존재할 때 합이  $m$  이 되는  $S$  의 부분 집합을 모두 찾는 문제 상황으로 표현할 수 있다.

$$S = \{a_i \mid 1 \leq i \leq n, a_i \in N(\text{자연수})\}$$

예를 들어  $n=4$ ,  $S=\{5, 9, 11, 20\}$ ,  $m=25$  라 가정하면 이때의 해답은  $\{5, 9, 11\}$ ,  $\{5, 20\}$ 이 된다.

이러한 문제를 백트래킹을 사용하여 문제를 풀때 상태 공간 트리를 DFS 로 탐색하여 해결하는 상태에 도달하는 데에 있어 불필요한 탐색을 하지 않는 것을 목표로 한다. 만약 백트래킹을 사용하지 않고, 부분집합의 합 문제에서 나올 수 있는 조합의 모든 경우의 수를 전체 탐색을 통해 확인한다면 위의 집합  $S$  에 대해  $|S| = N$  일 때,  $2^n$  의 경우의 수를 탐색해야 한다. 즉 시간복잡도가  $O(2^n)$ 이 되어 매우 비효율적인 탐색이 된다.

먼저 종료 조건에 부합하는 유망성 검사를 코드로 나타내면 다음과 같다.

```
int promising(int i, int weight, int total) {
    return (weight + total >= W) && (weight == W || weight + w[i] <= W);
}
```

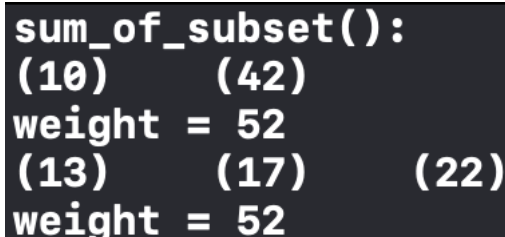
[코드 3] sum\_of\_subsets promising()

이러한 백트래킹 알고리즘을 사용한 부분집합의 합을 구하는 과정을 코드로 나타내면 다음과 같다.

```
void sum_of_subsets(int i, int weight, int total, int include[MAX])
{
    if(promising(i, weight, total)) {
        back_count++;
        if(weight == W) {
            printSubset(include);
        } else {
            include[i] = true; // 들어갔을 때
            sum_of_subsets(i + 1, weight + w[i], total - w[i],
            include);
            include[i] = false; // 안들어갔을 때
            sum_of_subsets(i + 1, weight, total - w[i], include);
        }
    }
}
```

[코드 4] sum\_of\_subsets()

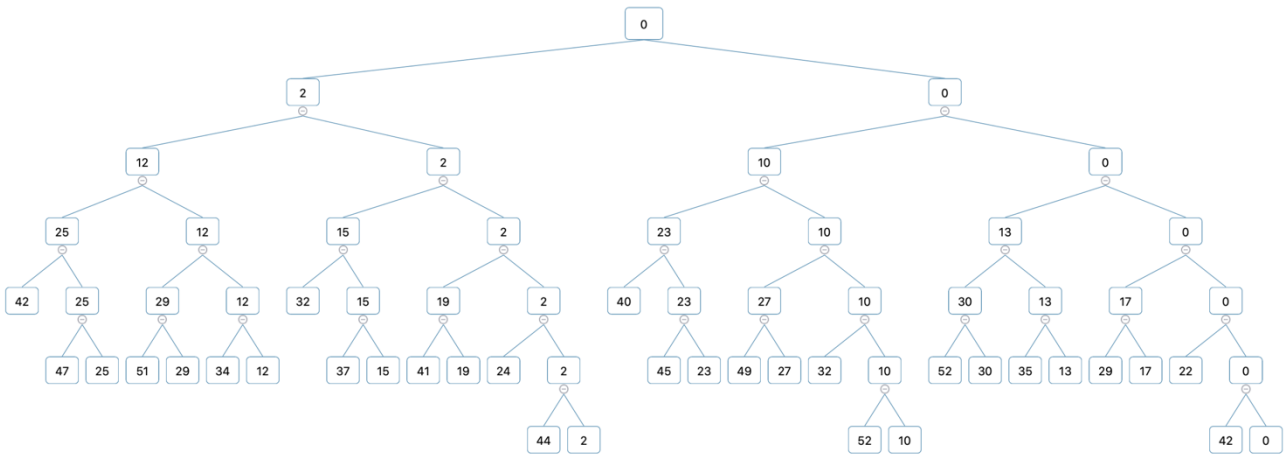
주어진 문제를 대입하여 실행하면 조합이  $\{w_2, w_6\}$ 일때,  $\{w_3, w_4, w_5\}$  일 때 주어진 조건을 만족하는 것을 알 수 있다.



```
sum_of_subset():
(10)      (42)
weight = 52
(13)      (17)      (22)
weight = 52
```

[그림 6] sum\_of\_subset()의 결과

또한 이것을 상태 공간 트리로 나타내면 다음과 같다.



[그림 7] 상태 공간 트리

그렇다면, 백트래킹을 사용하지 않고 모든 경우를 탐색했을 때와 비교했을 때 수행 성능은 어떻게 되는지 확인해보자. 기존의 코드가 호출되는 수를 파악하기 위해 back\_count 변수와 유망성 검사를 하지 않고 모든 경우를 탐색하는 함수의 호출 수를 파악하기 위해 all\_count 변수를 주었다. 또한 후자의 경우 최초 조건에 부합하는 조합이 발견되었을 때 바로 종료하는 조건을 주었다. 이것을 코드로 나타내면 다음과 같다.

```
void find_all_sum_of_subset(int i, int weight, int total, int
include[MAX]) {
    if(find_check) return;
    all_count++;
    if(weight == W) {
        printSubset(include);
        find_check = true;
    } else {
        if(i < MAX) {
            include[i] = true; // 들어갔을 때
            find_all_sum_of_subset(i + 1, weight + w[i], total - w[i],
include);
            include[i] = false; // 안들어갔을 때
            find_all_sum_of_subset(i + 1, weight, total - w[i],
include);
        }
    }
}
```

[코드 5] find\_all\_sum\_of\_subsets()

그리고 두 함수를 실행 후 비교하면 다음과 같은 결과를 얻게 된다.

```
sum_of_subset():
(10) (42)
weight = 52
(13) (17) (22)
weight = 52
back_count = 31

find_all_sum_of_subset():
(10) (42)
weight = 52
all_count = 95
```

[그림 8] 실행 결과

즉 호출의 횟수만 따져봐도 백트래킹 알고리즘을 사용한 것이 훨씬 더 효율적임을 알 수 있다.

그렇다면 마지막으로 이 백트래킹 알고리즘을 사용한 함수를 몬테카를로 알고리즘을 사용하여 효율을 추정해보도록 하자. 몬테카를로 알고리즘을 적용한 코드는 다음과 같다.

```
int estimate_sum_of_subsets(int i, int weight, int total, int
include[MAX]) {
    int
    m = 1, // 각 수준에서 유망한 마디의 개수
    mprod = 1,
    numnodes = 1; // 알고리즘이 만들어 내는 가지친 상태공간트리에서 마디의 개수
    추정치

    while(m != 0 && i < MAX) {
        mprod = mprod * m;
        if(promising(i, weight, total)) {
            include[i] = rand() % 2;
            if(m != 0 && include[i] == true) {
                weight += w[i];
            }
            numnodes += mprod * 2;
            m = 2; // 유망한 자식 마디개수를 더함
        } else m = 0;
        total -= w[i];
        i++;
    }
    tot += numnodes;
    return numnodes;
}
```

[코드 6] estimate\_sum\_of\_subsets()

이 코드를 실행하면 다음과 같은 추정치 결과를 얻을 수 있다.

```
===== estimate_sum_of_subset():
seq[ 1] numnodes = 63
seq[ 2] numnodes = 63
seq[ 3] numnodes = 63
seq[ 4] numnodes = 63
seq[ 5] numnodes = 63
seq[ 6] numnodes = 31
seq[ 7] numnodes = 63
seq[ 8] numnodes = 31
seq[ 9] numnodes = 63
seq[10] numnodes = 31
seq[11] numnodes = 31
seq[12] numnodes = 63
seq[13] numnodes = 63
seq[14] numnodes = 31
seq[15] numnodes = 63
seq[16] numnodes = 127
seq[17] numnodes = 31
seq[18] numnodes = 63
seq[19] numnodes = 63
seq[20] numnodes = 63
avg = 56
```

[그림 9] estimate\_sum\_of\_subsets() 실행 결과

즉 상태에 따라서 개수의 차이가 존재하지만 평균적으로 약 56 개의 조건 탐색이 이루어진 것을 알 수 있다.

### 3. N-Queen

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define MAX 8
#define true 1
#define false 0

int col[MAX + 1] = {0};

int promising(int i);
int estimate_n_queens(int n);

int main(void) {
    int
    ret = 0,
    tot = 0;
    srand(time(NULL));
    for(int i = 0; i < 20; i++) {
        ret = estimate_n_queens(MAX);
        tot += ret;
        printf("\tseq[%d]: numnodes = %d\n", i, ret);
    }
    printf("\ttot = %d\tavg = %d\n", tot, tot/20);
    return 0;
}

int estimate_n_queens(int n) {
    int i = 0, j, m = 1;
    int numnodes = 1, mprod = 1;
    int random;

    while(m != 0 && i != n) {
        mprod = mprod * m;
        numnodes = numnodes + mprod * n;
        i++;
        m = 0;
        int prom_children[MAX + 1] = {0};
        for(j = 1; j <= n; j++) {
            col[i] = j;
            if(promising(i)) {
                m++;
                prom_children[j] = true;
            }
        }
        if(m != 0) {
            while(1) {
                random = (rand() % MAX) + 1;
                if(prom_children[random] == 1) {
                    j = random;
                    break;
                }
            }
            col[i] = j;
        }
    }
    return numnodes;
}

int promising(int i) {
    int k = 1, Switch = true;
    while(k < i && Switch) {
        if(col[i] == col[k] || abs(col[i] - col[k]) == i - k)
            Switch = false;
        k++;
    }
    return Switch;
}
```



#### 4. SumOfSubset

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define W 52
#define MAX 6
#define true 1
#define false 0

void sum_of_subsets(int i, int weight, int total, int include[MAX]);
void find_all_sum_of_subset(int i, int weight, int total, int include[MAX]);
int estimate_sum_of_subsets(int i, int weight, int total, int include[MAX]);
int promising(int i, int weight, int total);
void printSubset(int include[MAX]);

const int w[MAX] = {2, 10, 13, 17, 22, 42};
int back_count = 0;
int all_count = 0;
int tot = 0;
int find_check = false;

int main(void) {
    int include[MAX] = { false };
    srand(time(NULL));
    printf("\tsum_of_subsets():\n");
    sum_of_subsets(0, 0, 106, include);
    printf("\tback_count = %d\n", back_count);
    printf("\tfind_all_sum_of_subset():\n");
    find_all_sum_of_subset(0, 0, 106, include);
    printf("\tall_count = %d\n", all_count);
    printf("\t===== estimate_sum_of_subset():\n");
    for(int i = 0; i < 20; i++) {
        printf("\tseq[%2d] numnodes = %d\n", i + 1, estimate_sum_of_subsets(0, 0, 106, include));
    }
    printf("\tavg = %d\n", tot/20);

    return 0;
}

void sum_of_subsets(int i, int weight, int total, int include[MAX]) {
    if(promising(i, weight, total)) {
        back_count++;
        if(weight == W) {
            printSubset(include);
        } else {
            // 들어갔을 때
            include[i] = true;
            sum_of_subsets(i + 1, weight + w[i], total - w[i], include);
            // 안들어갔을 때
            include[i] = false;
            sum_of_subsets(i + 1, weight, total - w[i], include);
        }
    }
}

void find_all_sum_of_subset(int i, int weight, int total, int include[MAX]) {
    if(find_check) return;
    all_count++;
    if(weight == W) {
        printSubset(include);
        find_check = true;
    } else {
        if(i < MAX) {
            // 들어갔을 때
            include[i] = true;
            find_all_sum_of_subset(i + 1, weight + w[i], total - w[i], include);
            // 안들어갔을 때
            include[i] = false;
            find_all_sum_of_subset(i + 1, weight, total - w[i], include);
        }
    }
}
```

```

    }
}

int estimate_sum_of_subsets(int i, int weight, int total, int include[MAX]) {
    int
    m = 1, // 각 수준에서 유망한 마디의 개수
    mprod = 1,
    numnodes = 1; // 알고리즘이 만들어 내는 가지친 상태공간트리에서 마디의 개수 추정치

    while(m != 0 && i < MAX) {
        mprod = mprod * m;
        if(promising(i, weight, total)) {
            include[i] = rand() % 2;
            if(m != 0 && include[i] == true) {
                weight += w[i];
            }
            numnodes += mprod * 2;
            m = 2; // 유망한 자식 마디개수를 더함
        } else m = 0;
        total -= w[i];
        i++;
    }
    tot += numnodes;
    return numnodes;
}

int promising(int i, int weight, int total) {
    return (weight + total >= W) && (weight == W || weight + w[i] <= W);
}

void printSubset(int include[MAX]) {
    int tot = 0;
    for(int j = 0; j < MAX; j++) {
        if(include[j] == true) {
            tot += w[j];
            printf("\t(%d)", w[j]);
        }
    }
    printf("\n\tweight = %d\n", tot);
}

```