□ 기술해설 □

알고리즘 설계 기법

한국과학기술원 **좌 경 룡***

● 목 차 ●

1. 서 론
2. 분할 해결법
3. 동적 계획법
4. Greedy 방법
5. 퇴각 검색법
6. 분기와 한정법
7. 무작위 알고리즘

1. 서 론

알고리즘이란 한 마디로 말해 "특정한 일을 수행하기 위한 절차"를 말한다. 만일 컴퓨터로 하여금 특정한 일을 수행하게 하고 싶다면 제일 먼저 해야할 일이 기계가 정확히 무엇을 어떠한 순서로 해야 하는지를 설계하고 이를 기계에게 알려주는 일－즉, 알고리즘을 설계하고 이를 알려주는 일이 핵심이 된다. 컴퓨터가 수행해야 할 알고리즘만으로 알고리즘을 제한 하는 경우 "언젠가는 종료되는 명확한 유한개의 작업수서"만을 알고리즘이라 한다. 그렇지 않은 경우는 실제 실용가치가 없기 때문이다. 여기에서 명확하다는 것은 각 작업 단계에서 다음에 수행하여야 할 작업이 무엇인지 그 상황에서 유일하게 결정되어야 한다는 것을 의미한다. 앞으로 컴퓨터가 수행하는 알고리즘으로만 알고리즘을 제한하기로 한다.

어떤 특정한 문제를 해결하기 위한 알고리즘을 설계 했다고 하자. 그러면 다음 단계는 설계한 알고리즘대로 기계를 작동시키기 위해 기계가 이해할 수 있는 언어를 사용하여 그 알고리즘을 구체적으로 프로그램의 형태로 기술하는 일이다. 이때 기계의 환경, 프로그래밍 언어, 프로그래머의 스타일 등 여러 요인에 의해 동일한 알고리즘이 수많은 다른 프로그램으로 구현될 수 있다.

컴퓨터 프로그램의 효율을 평가할 때 프로그램의 처리 속도가 중요한 척도 중의 하나인데, 그 프로그램의 알고리즘 및 위에서 언급한 여러 요인들에 의해 처리속도가 좌우된다. 최근들어 컴퓨터의 사용이 급격히 늘어나면서 컴퓨터를 이용하는 분야가 크게 확대되고 있다. 그 결과 해결해야 하는 문제가 점점 복잡해 지고 특히 다루어야 하는 자료의 분량이 폭발적으로 많아지게 되면서 효율적인 알고리즘을 사용하는 지 여부가 전체 프로그램의 동작효율에 중요한 영향을 미치게 되었다. 본 고에서는 이러한 실제 문제를 해결할 때 체계적으로 알고리즘을 설계하는데 도움이 될 수 있도록 하기 위해 일반적으로 사용되는 정형화된 알고리즘 설계 기법들을 소개한다. 특히 예를 들어 설명하는데 중점을 둔다.

2. 분할 해결법

어떤 문제 P를 해결하는 알고리즘이 다음과

* 종신회원

같은 골격을 가지고 있으면 그 알고리즘을 분할 해결법(divide-and-conquer) 알고리즘이라 할 수 있다. 분할 해결법 알고리즘은 주어진 입력의 숫자가 P를 쉽게 해결할 수 있을 정도로 충분히 작으면 있는 그대로 간단히 해결한다. 만일 그렇지 않으면 주어진 입력을 보다 작은 갯수를 가지는 여러 부분으로 나누되 나누어진 각 부분은 P에 대한 독립된 하나의 입력이 되도록 한다. 다음에 나누어진 각 입력에 대해 순환(recursive) 호출을 한다. 순환 수행이 끝나면 그 결과를 합하여 원래 문제에 대한 해를 구한다.

합병정렬(merge sort)은 잘 알려진 정렬 알고리즘으로서 전형적인 분할 해결법 알고리즘이다. 다음의 합병정렬 알고리즘에서는 원소의 갯수가 n개인 배열 A의 원소를 정렬하는데 전체를 정렬하기 위해서는 Merge_Sort(1, n)을 수행하면 된다. A(j..j)는 배열 A의 i번째 원소로부터 j번째 원소까지의 연속된 부분 배열을 의미한다. A는 전역변수(global variable)이다.

```
Algorithm Merge_Sort(first, last)
입력 : A(first..last)
출력 : A(first..last)가 정렬되어 재배치된 것
1. if first<last then
2.     mid:=⌈(first+last)/2⌉
3.     Merge_Sort(first, mid)
4.     Merge_Sort(mid+1, last)
5.     A(first..mid)와 A(mid+1, last)를 합병하여
       A(first..last)의 정렬된 결과를 구하고 이를
       A(first..last)에 저장한다.
6. endif
end Merge_Sort
```

Merge_Sort에서 분할 해결법의 골격을 살펴보자. 먼저 알고리즘에는 명시되어 있지 않지만 입력자료의 갯수(A(first..last)의 원소 갯수)가 1 이하이면 이미 정렬되어 있는 것이므로 더이상 작업을 하지 않는다. 즉, 정렬을 간단히 해결한 것이라고 볼 수 있다. 만일 갯수가 2 이상이라면 입력자료를 A(first..mid)와 A(mid+1..last)로 나누어 이에 대해 순환 수행한 수에 (단계 2, 3, 4) 정렬된 두 부분을 하나의 정렬된 부분으로 합병한다(단계 5). 이 예에서는 입력을 두 부분으로 나누었으나 경우에 따라서는 세개 이상의 부분으로 나눌 수도 있다.

분할 해결법 알고리즘을 설계할 때 가장 중요하게 고려하여야 할 부분은 나누어진 각 부분의 입력의 갯수를 효율적으로 분배해서 전체 알고리즘이 효율적으로 동작하도록 해야 한다는 것이다. Merge_Sort의 시간 복잡도를 분석하면서 이를 검토해 보자

Merge_Sort에 의해 소요되는 최악의 수행시간을 T(n)이라 하자. 이때 n은 입력배열의 원소의 갯수이다. 먼저 $n=2^k$일 경우(k는 양의 정수)에 대해 분석해 보자. 단계 3은 T(n/2) 시간이 소요되고 단계 4에서도 T(n/2) 시간이 소요된다. 단계 5는 O(n) 시간이 소요되도록 구현할 수 있으므로 다음이 성립한다.

$$T(n)=\begin{cases} O(1) & \text{if } n=1 \\ 2T(n/2)+O(n) & \text{if } n>1 \end{cases}$$

위의 경우 $T(n)=O(n \log n)$ 임을 쉽게 알 수 있다. 만일 $2^k<n\leq 2^{k+1}$일 경우(k는 양의 정수) $T(n)\leq T(2^{k+1})$ 이므로 역시 $T(n)=O(n \log n)$이다. 따라서 Merge_Sort의 시간 복잡도는 $O(n \log n)$ 이다.

이제 입력을 1개와 n－1개로 나누면－즉, 단계 2를 mid:=first로 수정하면－Merge_Sort의 시간 복잡도가 어떻게 변하는지 살펴보자. 이 경우 다음이 성립한다.

$$T(n)=\begin{cases} O(1) & \text{if } n=1 \\ T(1)+T(n-1)+O(n) & \text{if } n>1 \end{cases}$$

이 경우 $T(n)=O(n^2)$ 임을 쉽게 알 수 있다. 이 예에서 볼 수 있듯이 분할 해결법 알고리즘에서 주어진 입력을 여러 부분으로 나눌 때 나누어진 각 부분의 입력의 갯수를 효율적으로 분배하여 전체 알고리즘의 효율이 높아지도록 하는 것이 중요하다. 주어진 입력을 어떻게 나누는가에 따라 일반적으로 다음이 성립한다면

$$T(n)=\begin{cases} O(1) & \text{if } n=1 \\ aT(n/b)+O(n) & \text{if } n<1 \end{cases}$$

T(n)은 다음과 같이 된다.

$$T(n)=\begin{cases} O(n) & \text{if } a<b \\ O(n \log n) & \text{if } a=b \\ O(n^{\log_b a}) & \text{if } a<b \end{cases}$$

또 다른 분할 해결법 알고리즘으로서 다음 문제에 대한 알고리즘을 예로 든다.

문제 (최 근접 쌍) 평면상의 n개의 점에 대한 좌표가 주어졌을 때 그 중에서 가장 가까운 쌍(pair)을 찾아라.

이 문제에 대한가장 간단한 알고리즘은 n(n−1)/2개의 모든 쌍에 대해 거리를 계산해 보는 것이다. 이렇게 하면 시간 복잡도가 $O(n^2)$ 이다. 다음의 알고리즘은 O(n log n) 시간 복잡도를 가지는 분할 해결법 알고리즘이다.

Algorithm Closest_Pair(s, p, q)
입력 : 평면상의 n개의 점의 집합 S
출력 : S의 점들 중 거리가 가장 가까운 쌍(p, q)

1. 만일 |S|≤2이면 간단히 해결된다. 그렇지 않으면 다음을 수행한다.
2. 가상을 수직선 l을 사용하여 S를 두 부분 S_1, S_2로 나누되 S_1을 l의 왼쪽에, S_2를 l의 오른쪽에 있도록 하고 $|S_1|=\lfloor n/2 \rfloor$, $|S_2|=\lceil n/2 \rceil$이 되도록 한다(그림 1).
3. Closet_Pair(S_1, p_1,q_1)
4. Closest_Pair(S_2, p_2, q_2)
5. p_1과 q_1 사이의 거리와 p_2과 q_2 사이의 거리 중에서 작은 값을 δ라 하자.
6. S_1의 점들 중에서 l로부터 거리가 δ 이하인 것을 모아 P_1을 구성하고 S_2의 점들 중에서 l로부터 거리가 δ 이하인 것을 모아 P_2를 구성한다. P_1과 P_2를 y좌표값에 따라 정렬한다. 이를 각각 P_1^*, P_2^*라 하자.
7. P_1^*의 한 점 s의 y좌표값을 s_y라 할 때 y좌표값이 $s_y-\delta$와 $s_y+\delta$사이에 있고 x좌표값이 l_x와 $l_x+\delta$ 사이에 있을 영역을 W_s안에 있는 모든 점 t와 s와의 거리를 계산한다. P_1^*를 따라가면서 P_1^*의 각 점 s에 대해 위의 과정을 반복한다. 거리가 계산된 (s, t)쌍 중에서 가장 가까운 것을 선택하여 그 두점을 각각 p_3, q_3로 한다.
8. (p_1, q_1), (p_2, q_2), (p_3, q_3) 중에서 가장 가까운 것을 (p, q)로 한다.

end Closest_Pair

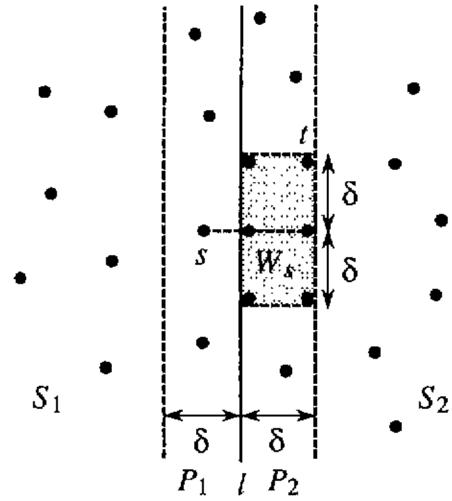

그림 1 Closest_Pair의 실행 예

위 알고리즘의 단계 2에서는 입력으로 주어진 점의 집합 S를 같은 크기의 두 집합 S_1과 S_2로 나눈다. 전체 알고리즘의 전처리 과정으로서 S의 점들을 x 좌표값에 따라 처음에 한번만 정렬해 놓고 이를 분할하면 S_1과 S_2를 O(n) 시간에 구할 수 있다. 더욱이 S_1과 S_2의 점들이 x좌표값에 따라 정렬되어 있으므로 이들은 각각 단계 3과 4의 순환수행 단계에서 입력을 분할하는데 같은 방법으로 이용될 수 있다.

이제 S_1에서 가장 가까운 쌍 (p_1, q_1), S_2에서 가장 가까운 쌍 (p_2, q_2), S_1의 한점과 S_2의 한점으로 이루어진 쌍 중에서 가장 가까운 쌍 (p_3, q_3) 중에서 두 점 p_i과 q_i 사이의 거리가 가장 가까운 쌍이 우리가 원하는 쌍이다(1≤i≤3). 처음의 두 경우에 대해서는 단계 3과 4에서 순환수행으로 구하고 마지막 경우에 대해서는 단계 5, 6, 7에서 구한다. 이때 (p_3, q_3를 구할 때 l로부터 거리가 δ이내에 있는 점만 고려해도 된다. 단계 6에서는 이러한 점들을 찾는다. 전처리 과정으로서 S의 점을 y좌표값에 따라 처음에 한번만 정렬해 놓으면 이로부터 P_1^*와 P_2^*를 O(n) 시간에 구할 수 있다. 단계 3과 4에서 순환수행될 때도 각각 P_1^*과 P_2^*를 이용하면 같은 방법으로 구할 수 있다.

단계 7에서 s를 고려할 때 W_s에 있지 않은 P_2의

점은 고려할 필요가 없다. 그러한 점들은 s로부터의 거리가 δ 보다 크기 때문이다. 또한 W_s안에 있는 점들도 서로 거리가 적어도 δ 만큼은 되어야 하므로 W_s에는 그림 1에서와 같이 6개 이하의 점들만 있을 수 있다. 따라서 단계 7은 O(n) 시간에 수행된다.

결국 Closest_Pair의 최악의 수행시간을 T(n)이라 한다면 다음이 성립한다.

$$T(n)=\begin{cases} O(1) & \text{if } n\le 2 \\ 2T(n/2)+O(n) & \text{if } n>2 \end{cases}$$

따라서 Closest_Pair의 계산 복잡도는 O(n log n)이다.

3. 동적 계획법

동적 계획법(dynamic programming)은 개념적으로 볼 때 주어진 문제를 여러 부분문제 (sub-problem)로 분할하여 순환수행 한다는 점에서 분할 해결법과 같으나 분할 해결법의 경우 한 번 순환수행된 부분문제는 다시 수행되지 않는 경우에 적당하고 동적 계획법은 다시 수행될 수 있을 때 사용되는 방법이다.

반복해서 여러번 호출되는 부분문제가 있을 경우 해당되는 부분문제가 첫번째 수행될 때 그 결과를 기록해 두었다가 이후에 다시 수행이 요구될 때 또 수행하는 것이 아니라 이전에 기록된 결과를 이용하는 것이 동적 계획법에서 핵심이 되는 부분이다. 참고로 "dynamic programming"에서 "programming"이라는 말은 컴퓨터 프로그램과는 무관하고 표를 이용하는 방법이라는 뜻이다.

동적 계획법을 사용한 알고리즘의 일반적인 구조를 설명하기 위해 다음의 문제에 대한 알고리즘을 예로 든다.

$A=(a_{ij})$를 $p\times q$행렬이라 하고 $B=(b_{ij})$를 $q\times r$ 행렬이라 하자. 두 행렬을 곱하는 알고리즘은 여러가지가 있을 수 있으나 정의에 따른 알고리즘으로 제한해 보자. 즉, $C=(c_{ij})=AB$라 할때 $c_{ij}=\Sigma_{1\le k\le q}a_{ik}b_{kj}$에 따라 C를 구하면($1\le i\le p$, $1\le j\le r$) 행렬의 원소끼리 곱셈이 일어나는 횟수는 pqr번이다. 만일 여러개의 행렬을 곱한다면 전체 알고리즘의 수행 속도가 원소끼리의 곱셈 횟수에 크게 좌우되므로 이를 계산해 보자. 예를 들어 A_1, A_2, A_3를 각각 2×3, $3\times 4+2\times 4\times 5$행렬이라고 하자. A_1, A_2, A_3를 구하기 위해서 두가지 방법이 있다. 즉, $(A_1A_2)A_3$의 순서와 $A_1(A_2A_3)$의 순서이다. 그런데 앞의 경우 전체 원소 곱셈의 횟수는 $2\times 3\times 4+2\times 4\times 5=64$번이고 뒤의 경우에는 $3\times 4\times 5+2\times 3\times 5=90$번이다. 따라서 앞의 방법이 더욱 효율적이다. 다음 문제는 일반적인 경우에 가장 효율적인 방법을 찾는 것이다.

문제 (행렬의 곱셈순서) n개의 행렬 A_1, A_2,$\cdots$, A_n이 주어져 있다. A_i가 $d_{i-1}\times d_i$ 행렬이라할때($1\le i\le n$) 전체 행렬곱 $A_1A_2\cdots A_n$을 구하려 한다. 원소 곱셈의 전체 횟수를 최소로 하기 위한 행렬 곱셈의 순서를 찾아라.

최적의 행렬곱셈 순서를 계산하기 전에 먼저 최소의 원소곱셈의 횟수를 계산해 보자. $A_i\cdots A_j$를 계산하기 위해 사용되는 최소의 원소 곱셈의 횟수를 M_{ij}라 하자($1\le i\le j\le n$). $A_i\cdots A_j$를 계산하는 순서에게 제일 마지막에 계산되는 행렬곱이$(A_i\cdots A_k)(A_{k+1}\cdots A_j)$라 하면($i\le k\le j$) 앞부분은 $d_{i-1}\times d_k$ 행렬이고 뒷 부분은 $d_k\times d_j$ 행렬이므로 다음의 순환식이 성립한다.

$$M_{ij}=\begin{cases} 0 & \text{if } i=j \\ \min_{i\le k<j}\{M_{ik}+M_{k+1,j}+d_{i-1}d_kd_j\} & \text{if } 1\le i<j\le n \end{cases}$$

위의 식은 이미 그대로 순환 알고리즘으로 구현될 수 있다. 그런데 이렇게 하면 시간 복잡도가 $\Omega(2^n)$이 되어 매우 비효율적이다. 그 이유는 동일한 순환수행이 여러번 반복하여 호출되기 때문이다. 예를 들어 n=10일 경우 $M_{3,7}$은 $M_{1,8}$에서 호출 되고 $M_{2,10}$에서도 호출되는 등 모두 37번 호출되는 데 호출될 때 마다 그 자신이 순환수행을 계속 반복한다. 만일 $M_{3,7}$이 처음 호출 될 때에만 그 자신이 순환수행을 한 후 그 결과를 기록해 두고 이후에 호출될 때는 단순히 기록된 값을 참조하기만 하면 효율적인 것이다.

이와 같이 여러번 반복되어 호출되는 부분문제를 처음에 한번만 실제 계산하여 이를 기록해

두고 이후에 호출될 때는 이를 단순히 참조하여 알고리즘의 효율을 높이는 것이 동적 계획법의 핵심이다.

이 문제의 경우 M_{ij} 값을 2차원 배열에 기록하는 것이 적당하다. 기록되는 배열 이름도 역시 M이라 하고 M_{ij}값을 M(i, j)에 기록한다고 하자. 앞으로 편의상 M_{ij}와 M(i, j)를 같은 의미로 사용하기로 하겠다. 그런데 M_{ij}를 계산하기 위해서는 $i \le k < j$인 모든 k에 대해 M_{ik}와 $M_{k+1,j}$가 계산되어 있어야 하는데 이는 그림 2에서 빗금친 부분에 해당된다. 따라서 이를 만족하는 순서를 찾아 이를 따라 단순히 배열 값만을 계산하면 순환수행 없이 M_{1n} 값을 계산할 수 있다.

```
Algorithm Min_Mult(D)
입력 : D=(d0, d1,…,dn)
출력 : M1n 값
1. for i:=1 to n do Mii:=0
2. for d:=1 to n-1 do
3.   for i:=1 to n-d do
4.     j:=i+d
5.       Mij:= min {Mik+Mk+1,j+di-1dkdj}
             i≤k<j
6.   endfor
7. endfor
8. return M1n
end Min_Mult
```

이 알고리즘의 계산 복잡도는 $O(n^3)$ 이므로 단순한 순환 알고리즘 보다 매우 효율적이다. 만일 M_{1n} 값 뿐만 아니라 최적의 행렬 곱셈순서까지 계산하려면 단계 5에서 M(i, j)를 구하는데 사용된 k값이 무엇인지를 M(i, j)에 추가로 기록하면 된다. 나중에 이 정보를 역으로 추적하면 최적의 행렬 곱셈순서를 구할 수 있다. 이렇게 해도 시간 복잡도는 변하지 않는다.

일반적으로 동적 계획법 알고리즘을 설계하는 순서를 정리하면 다음과 같다. 먼저 주어진 문제의 최적해를 구하기 위한 순환구조를 설계한 후에 이로부터 각 부분 문제가 해결되어야 할 순서를 결정한다. 각 부분 문제의 결과를 저장하기 위한 기억장소를 할당한 후에 앞에서 결정한 순서에 따라 각 부분 문제를 해결 하도록 한다. 이 때 각 부분 문제가 해결되어야 할 순서를

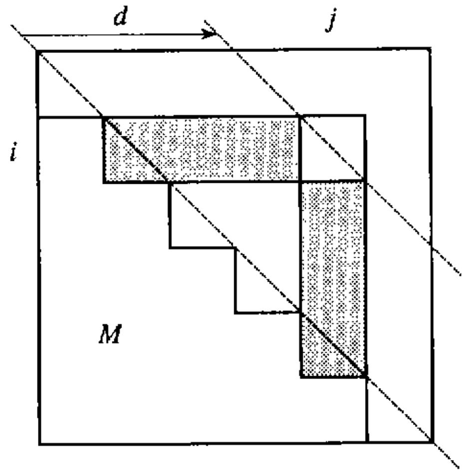

그림 2 M의 계산순서

따르기 때문에 각 부분 문제를 해결하기 위해서 순환수행 없이 단순히 이전에 계산된 값을 참조하기만 하면 된다.

또 다른 예로서 다음 문제를 효율적으로 해결할 수 있는 동적 계획법 알고리즘을 예로 든다.

<u>문제</u> **(Bitonic TSP)** 평면상의 n개의 점의 좌표가 주어져 있다. 이때 모든 점의 x좌표값은 서로 다르다. 모든 점을 통과하여 제자리로 다시 돌아오는 가장 짧은 경로를 구하라. 단, 임의의 수직선과 경로가 교차하는 점은 항상 두개 이하가 되도록 해야 한다.

경로의 모양에 제한이 없는 일반적인 문제는 NP-complete 문제이나 이 문제는 동적 계획법을 사용하여 $O(n^2)$ 시간에 해결할 수 있다. 앞으로 최적 경로의 길이를 구하는 알고리즘만을 설명한다. 이 알고리즘의 자료구조를 조금만 수정하면 최적 경로까지 구하는 알고리즘으로 쉽게 수정할 수 있다.

주어진 점들이 x좌표값에 의해 정렬된 것을 $p_1, p_2, \cdots, p_n$이라 하고 점 $p_1, p_2, \cdots, p_i$ 만을 사용하는 최적 경로를 T_i라하며 T_i의 길이를 $|T_i|$로 표시하기로 하자. 그리고 두 점 p와 q를 연결하는 선분을 pq로 표시하고 그 길이를 $|pq|$로 표시하기로 하자. 이제 $|T_i|$의 값을 구하는 방법을 살펴보자. 먼저 i가 3 이하일 경우에는 $|T_i|$를 쉽게 계산할 수 있다. 그렇지 않은 경우

$p_{i-1}p_i$는 항상 T_i에 포함되어야 한다는 것을 쉽게 알 수 있다. 이제 T_i에서 p_i에 연결된 또 다른 점이 될 가능성이 있는 것은 p_1, p_2, …, p_{i-2} 중의 하나이다. T_i에서 p_{i-1} 외에 p_i에 연결된 또 다른 점이 p_j로 고정될 경우의 ($1 \leq j \leq i-2$) 최소길이 경로를 $T_i^{(j)}$라고 하고 그 길이를 $|T_i^{(j)}|$로 표시하기로 하자. 그러면 다음이 성립한다.

$$|T_i| = \min_{1 \leq j \leq i-2} \{|T_i^{(j)}|\}$$

$T_i^{(j)}$에 있어서 p_{j+1}부터 p_{i-1}까지는 그림 3에서와 같이 항상 x좌표 값이 증가하는 순서대로 연결되어 있어야 한다. 만일 그렇지 않다면 경로의 방향에 대한 규칙을 만족할 수 없다. 또한 p_j에서 p_1을 거쳐 p_{j+1}에 이르는 나머니 경로는 T_{j-1}에서 p_jp_{j+1}를 제외한 부분과 같다. 만일 그렇지 않다면 T_{j+1}이 $p_1,\cdots,p_{j+1}$을 거치는 최소길이의 경로라는 데 모순된다. p_a부터 p_b까지 x좌표 값이 증가하는 순서대로 연결한 경로의 길이를 m_{ab}라 하자($1 \leq a < b \leq n$). 이제 다음이 성립함을 알 수 있다.

$$|T_j^{(j)}| = |p_jp_i| + m_{j+1,i} + |T_{j+1}| - |p_jp_{j+1}|$$

따라서 다음이 성립한다.

$$|T_i| = \min_{1 \leq j \leq i-2} \{|T_i^{(j)}|\}$$

$$= \min_{1 \leq j \leq i-2} \{|p_jp_i| + m_{j+1,i} + |T_{j+1}| - |p_jp_{j+1}|\}$$

$|T_i|$를 구하기 위해서는 $|T_1|$, …, $|T_{i-1}|$이 미리 계산되어 있으면 되므로 다음의 동적 계획법 알고리즘을 설계할 수 있다. 이때 $|T_i|$ 값을 저장하기 위해 일차원 배열을 사용하면 된다. 사용되는 배열을 L이라 할때 L(i)에는 $|T_i|$ 값을 기록한다. 앞으로 편의상 $|T_i|$와 L(i)을 같은 의미로 사용하기로 하겠다.

```
Algorithm Bitonic_TSP(S)
입력 : S=(p1, p2,…,pn), n≥3
출력 : |Tn|
1. |T1|, |T2|, |T3| 값을 계산한다.
2. for i:=4 to n do
3.   |Ti|:=   min     {|pjpi|+mj-1,i
           1≤j≤i-2
                        +|Tj+1|-|pjpj+1|}
4. endfor
5. return |Tn|
end BitonicTSP
```

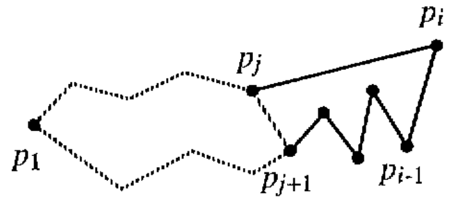

그림 3 $T_i^{(j)}$의 예

전처리 과정으로서 모든 m_{ab} 값을 $O(n^2)$ 시간에 구할 수 있고 이를 2차원 배열에 저장 해 두면 위 알고리즘이 수행되는 도중에는 $m_{j+1,i}$ 값을 O(1) 시간에 알 수 있다. 이를 이용하면 이 알고리즘의 시간 복잡도는 $O(n^2)$이다.

4. Greedy 방법

Greedy 알고리즘은 주어진 조건을 만족하는 최적해를 찾는데 주로 사용되는 기법이다. 이는 단계별로 진행되는데 각 단계에서는 현재 상태에서 가장 최적이라고 판단되는 결정을 내린다. 이 때 전체적으로 최적인지 여부는 검사하지 않고 현재 상태의 입장에서만 보았을 때 가장 최선이라고 판단되는 결정을 내린다. 따라서 Greedy 알고리즘에서는 어떻게 단계의 순서를 결정해야만 얻어진 해가 전체적으로 최적해가 될 수 있는가가 핵심이다.

Greedy 알고리즘의 일반적인 구조를 설명하기 위해 다음 문제에 대한 알고리즘을 예로 들어 설명한다.

문제 (회의실 배정) 한개의 회의실이 있는데 이에 대한 사용신청에 따라 회의실 사용표를 만들려고 한다. 사용신청은 각 회의의 시작 시간과 끝나는 시간의 집합 $S = \{(s_i, f_i) | 1 \leq i \leq n\}$으로 나타내어지는데 s_i는 회의 i의 시작 시간을, f_i는 끝나는 시간을 의미한다. 각 회의가 서로 시간이 겹치지 않게 하면서 회의실을 사용할 수 있는 최대 숫자의 회의를 찾아라. 단, 회의는 한번 시작하면 중간에 중단될 수 없으며 한 회의가 끝나는 것과

동시에 다음 회의가 시작될 수 있다.

다음의 알고리즘은 이 문제를 해결하기 위한 전형적인 Greedy 알고리즘이다.

```
Algorithm Greedy_Schedule(S, Solution)
입력 : S={(s_i,f_i)|1≤i≤n}
출력 : Solution⊆S
1. 각 회의의 종료시간을 증가하는 순서로 정렬
   한다. 일반성을 잃지 않고 이를 f_1, f_2,…,f_n이라
   하자.
2. Solution:={(s_1, f_1)}; last:=1
3. for i:=2 to n do
4.   if s_i≥f_last then
5.     Solution:=Solution∪{(s_i,f_i)}
6.     last:=i
7.   endif
8. endfor
end Greedy_Schedule
```

이제 Greedy 알고리즘의 일반적인 구조를 위의 예를 가지고 살펴보자. Greedy 알고리즘은 주어진 n개의 입력 중에서 특정한 제약을 만족하는 부분 집합을 고르는 문제에 이용된다. 위의 예에서는 입력은 S이고 제약조건은 회의들이 서로 겹치지 않아야 한다는 것이다. 입력의 부분집합 중에서 제약조건을 만족하는 것을 가능해(feasible solution)라 한다. 가능해 중에서 목적함수(objective function)를 최적으로 하는 해를 최적해(optimal solution)라 한다. 위의 예에서는 선택된 회의의 갯수가 목적함수이고 회의의 갯수를 최대로 하는 것이 목적함수를 최적으로 하는 것이다.

최적해를 찾기 위해 Greedy 알고리즘은 단계별로 현재 남아 있는 입력집합을 검사해서 그 중에서 최적해에 들어갈 수 있다고 판단되는 것을 골라낸다. 이 때 전체적으로 보았을 때 최적이 될지 여부는 무시하고 현재 단계에서 보았을 때 가장 최적이라고 판단되는 것을 무조건 고른다. 나중에 어떻게 될지는 고려하지 않고 현재 상태에서 가장 최적이라고 생각되는 것을 무조건 고르기 때문에 Greedy 알고리즘이라 한다. 현재 상태에서 보았을 때 최적해에 들어 갈 수 있다고 판단되는 것을 골라내는 규칙을 선택규칙이라 부르기로 하자. 물론 선택규칙은 문제에 따라 다르다. 선택규칙에 따라 골라낸 것을 이제까지 찾은 해에 포함시켰을 때 그것이 가능해가 된다면 그것을 현재 단계에서 새로 결정된 해로 하여 다음 단계로 계속한다.

다시 위의 알고리즘을 보면 단계 3에서 8사이가 단계별로 진행되는 부분이고 선택규칙은 현재까지 찾은 해인 Solution에 포함된 회의를 모두 마치고 났을 때 남아 있는 회의 중에서 가장 빨리 끝낼 수 있는 것을 골라내는 것이다. 전체적으로 고려하지 않고 현재 상태에서 가장 최적이 될 수 있는 것을 골라내므로 Greedy_Shedule은 전형적인 Greedy 알고리즘이다. 현재 단계에서 골라낸 것이 가능해가 될 수 있는지 판단하는 단계는 단계 4이다.

위의 예에서 보았듯이 Greedy 알고리즘에서 가장 중요한 것은 전체적으로 최적이 되도록 하는 선택규칙을 찾는 것이다. 그러나 적용된 선택규칙을 이용하면 항상 최적해를 찾을 수 있다는 것을 증명하는 것은 대부분의 경우 쉽지 않은 작업이다.

다음은 잘 알려진 Knapsack 문제인데 이에 대한 Greedy 알고리즘을 예로 제시하면서 이에 사용된 선택규칙이 올바르다는 것을 증명한다. 특이 이 증명은 Greedy 알고리즘에서 자주 사용되고 있는 증명방법을 보여주고 있기 때문에 주목할만 하다.

<u>문제</u> **(Knapsack)** 최대 MKg까지 물건을 집어 넣을 수 있는 배낭이 있고 이 배낭에 집어 넣고 싶은 n개의 물건이 있다. 물건 i를 배낭에 넣으면 p_i만큼 이득이 있고($p_i>0$) i의 무게는 w_iKg인데($w_i>0$)이를 잘라 일부를 넣을 수도 있다. 만일 물건 i를 x_i만큼 배낭에 넣는다면($0\leq x_i\leq 1$) p_ix_i만큼 이득이 있다. 전체 무게가 MKg 이하가 되면서 최대 이득을 얻을 수 있도록 하려면 각 물건을 얼마만큼 넣어야 하는가?

이 문제를 해결 할 수 있는 Greedy 알고리즘에 적용될 수 있는 선택규칙을 고려해 보자. 배낭에 들어갈 수 있는 전체 무게가 제한되어 있으므로 단위무게 당 이득이 높은 것부터 배낭에 집어 넣는 선택규칙을 생각할 수 있다. 이를 적용하면

다음과 같은 Greedy 알고리즘이 된다.

Algorithm Greedy_Knapsack(P, W, M, X)
입력 : $P=(p_1,\cdots,p_n)$, $W=(w_1,\cdots,w_n)$, M
출력 : $X=(x_1,\cdots,x_n)$
1. 각 물건을 p_i/w_i값이 감소하는 순서로 정렬한다. 일반성을 잃지 않고 $p_i/w_i \leq p_{i+1}/w_{i+1}$이라 하자 $(1 \leq i < n)$.
2. $X:=(0,\cdots,0)$
3. remained: = M
4. for i: = 1 to n do
5. if w_i > remained then goto 9
6. x_i: = 1
7. remained: = remained − w_i
8. endfor
9. if $i \leq n$ then x_i: = remained/w_i

end Greedy_Knapsack

위 알고리즘의 시간 복잡도(time complexity)는 $O(n \log n)$ 임을 쉽게 알 수 있다. 이제 이 알고리즘에서 사용된 선택규칙이 항상 최적해를 보장함을 증명한다.

먼저 Greedy_Knapsack의 결과를 $X=(x_1,\cdots,x_n)$이라 하자. 만일 모든 x_i 값이 1이라면 X는 최적해 이다. 그렇지 않은 경우 x_j를 $x_j \neq 1$이면서 가장 작은 j값을 가지는 것이라 하자. 알고리즘의 동작에 따르면 $1 \leq i \leq j$일 경우 $x_i=1$이고 $0 \leq x_j < 1$ 이며 $j < i \leq n$일 경우 $x_i=0$ 이다. 만일 X가 최적해가 아니라면 $\Sigma p_i y_i > \Sigma p_i x_i$ 인 최적해 $Y=(y_1,\cdots,y_n)$가 존재한다. Y가 최적해이므로 $\Sigma w_i y_i = M$ 이다. 또한 $\Sigma p_i y_i > \Sigma p_{i1x1}$이기 때문에 $y_i \neq x_i$인 i가 존재하는데 그러한 i 값중에서 가장 적은 것을 k라 하자. 이제 k와 j값의 관계에 따라 세가지 경우가 있을 수 있다. 만일 $k<j$이라면 $x_k=1$이므로 $y_k<x_k$이다. 만일 $k=j$이라면 $\Sigma w_i x_i = \Sigma w_i y_i = M$이기 때문에 역시 $y_k<x_k$이다. 마지막으로 $k>j$ 인 경우는 있을 수 없다. 만일 그렇다면 $\Sigma w_i y_i > M$이기 때문이다. 따라서 k와 j의 값에 관계없이 항상 $y_k<x_k$이다. 이제 y_k 값으로 증가시키고 그대신 $(y_{k+1},\cdots,y_n)$ 값들을 적당히 감소시켜 새로운 해 $Z=(z_1,\cdots,z_n)$를 만들되 $1 \leq i \leq k$인 i에 대해서는 $z_i=x_i$이고 $\Sigma_{k<i\leq n} w_i(y_i-z_i)=w_k(z_k-y_k)$이 되도록 할 수 있다. 이로부터 다음이 성립함을 알 수 있다.

$$\begin{aligned}\sum_{1\leq i\leq n} p_i z_i &= \sum_{1\leq i\leq n} p_i y_i + (z_k-y_k)w_k p_k/w_k \\ &\quad - \sum_{k<i\leq n}(y_i-z_i)w_i p_i/w_i \\ &\geq \sum_{1\leq i\leq n} p_i y_i + [(z_k-y_k)w_k \\ &\quad - \sum_{k<i\leq n}(y_i-z_i)w_i]p_k/w_k \\ &= \sum_{1\leq i\leq n} p_i y_i\end{aligned}$$

그런데 Y가 최적해이므로 $\Sigma p_i z_i > \Sigma p_i y_i$ 일 수 없다. 결국 $\Sigma p_i z_i = \Sigma p_i y_i$ 인데 X=Z 이면 X도 역시 최적해이다. X≠Z라면 위의 과정을 반복 하면 결국 총 이득을 바꾸지 않고 Y를 X로 바꿀 수 있다. 결국 X는 최적해이다.

위 증명은 Greedy 알고리즘에서 자주 사용되고 있는 증명방법을 보여주고 있다. 즉, Greedy 알고리즘이 출력하는 해 X가 최적해가 아니라고 가정한 후에 다른 최적해로부터 목적함수 값을 변화시키지 않고 X를 만들 수 있기 때문에 X가 최적해가 아니라는 것은 모순이라는 것을 보이는 방법이다.

5. 퇴각 검색법

알고리즘을 설계 할 때 종종 가능한 모든 경우를 검사 해야만 할 때가 있다. 그런데 가능한 경우의 갯수가 많을 때 문제가 된다. 예를 들어 NP-complete 문제를 해결하기 위해서는 현재까지 알려진 바에 따르면 입력의 갯수에 대한 지수함수 개의 경우를 모두 검사 할 수 밖에 없다. 이와 같이 모든 가능한 경우를 검사 해야만 할 때, 본 절과 다음 절에서 다루는 퇴각 검색법(backtracking)과 분기와 한정법(branch-and-bound)은 어떻게 하면 효율적으로 검사할 수 있는가에 관한 방법론이다.

퇴각 검색법이나 분기와 한정법은 해(solution)가 $(x_1, x_2, \cdots, x_n)$형태의 n-tuple로 표현될 수 있는 문제에 적용가능하다. 이때 x_i는 특정한 유한집합

S_i의 원소이다($1 \le i \le n$). 또한 가능한 모든 경우의 tuple의 집합을 해공간(solution space)이라 한다. 즉, 해는 해공간에 속하게 된다. 동일한 문제에 대해 여러가지 해공간이 있을 수 있고 문제에 따라서는 길이가 서로 다른 tuple로 해공간이 이루어져 있을 수도 있다. 이에 대한 예는 뒤에서 제시하겠다.

해공간과 함께 또다른 중요한 요소는 $P(x_1,\cdots,x_n)$로 표현되는 기준함수(criterion function)이다. 여기서 $(x_1,\cdots,x_n)$은 해공간의 tuple이다. 해공간의 tuple 중에서 주어진 기준함수를 만족하거나 혹은 최적화 시키는 tuple이 해로 정의되며 해를 한개 혹은 모두 찾는 문제에 퇴각 검색법과 분기와 한정법이 이용된다.

해를 찾기 위해 제일 간단한 방법은 $|S_1||S_2|\cdots|S_n|$개의 모든 tuple을 검사해 보는 것이다. 그러나 이것은 대부분의 경우 매우 효율이 낮기 마련이다. 퇴각 검색법이나 분기와 한정법은 해공간을 이루는 모든 tuple을 검사하는 순서를 적절히 조정함으로써 해공간의 모든 tuple을 다 검사하지 않아도 되도록 하는 방법이다. 이를 위해 퇴각 검색법에서는 한계함수(bounding function)를 사용한다. 한계함수 $B_k(x_1,\cdots,x_k)$는 완전한 tuple이 아니고 부분 tuple인 $(x_1,\cdots,x_k)$만 검사함으로써 해공간의 모든 tuple 중 $(x_1,\cdots,x_k)$를 부분 tuple로 가지는 모든 tuple을 더이상 검사해야할지 여부를 결정할 수 있게 해준다.

퇴각 검색 알고리즘를 설계할 때 해공간을 구성하는 모든 tuple과 부분 tuple을 상태공간트리(state space tree)로 개념적으로 구성하면 편리하다. 여기에서 개념적이라는 말은 실제 알고리즘에서 상태공간트리를 구성하는 것이 아니고 알고리즘 설계시 개념적인 도구가 되는 트리라는 말이다. 상태공간트리의 루트 노드는 null tuple을 나타내고 깊이가 i인 노드와 $i+1$인 노드 사이의 에지는 x_i값의 가능한 경우를 나타낸다.

다음 문제를 예로 들어 상태공간트리와 이에 따른 퇴각 검색법을 설명한다.

문제 (n-queen) 가로, 세로 각각 n 칸으로 구성된 서양 장기판이 있다. 여왕은 자기와 동일한 행, 열, 대각선 방향으로 공격할 수 있다. 서로 공격할 수 없도록 n개의 여왕을 배치할 수 있는 위치를 찾아라.

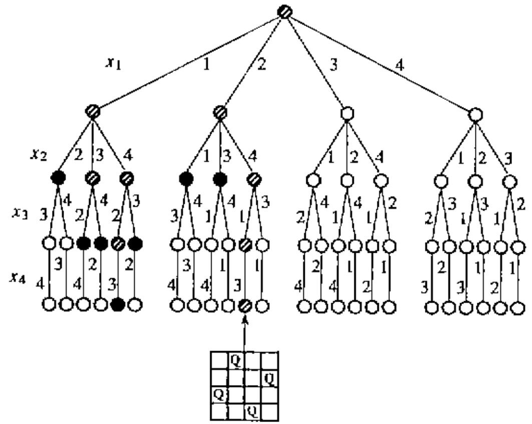

그림 4 4-queen 문제에 대한 상태공간트리

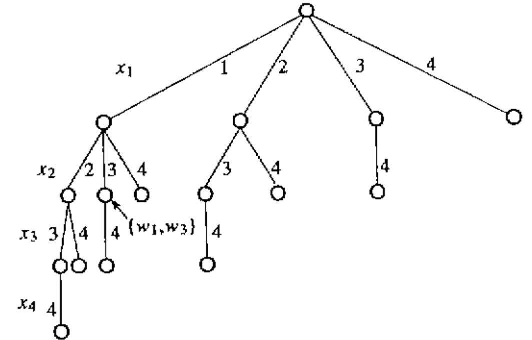

그림 5 부분집합의 합 문제에 대한 상태공간트리, Tuple의 길이가 일정하지 않는 경우

이 문제에 대해 n=4일 경우 고려해 보자. 동일한 행에는 한개의 여왕만 놓을 수 있으므로 원하는 해를 (x_1, x_2, x_3, x_4)로 표시할 수 있다. 여기서 x_i는 i번째 행에 있는 여왕이 x_i번째 열에 있다는 뜻이다($1 \le i \le 4$). 또한 동일한 열에도 여왕이 한개만 있어야 하므로 해공간을 $4!=24$개의 tuple로 구성 할 수 있고 이를 나타내는 상태공간트리는 그림 4에서 처럼 된다. 여기서 모든 잎노드(leaf node)가 해공간에 해당된다.

해공간의 tuple은 길이가 서로 다를 수도 있는데 예를 들면 다음 문제에 대한 해공간이 그러하다.

문제 (부분집합의 합(sum of subset)) n개의 양의

정수의 집합 $W=\{w_1, w_2, \cdots, w_n\}$과 양의 정수 M이 주어졌을 때 W의 부분집합 중에서 그 원소의 합이 M인 것을 찾아라.

이 문제의 경우 해는 W의 부분집합이므로 $(x_1, \cdots, x_k)$의 형태로 나타낼 수 있다($1 \le k \le n$). 이때 $x_i=j$라면 W를 정렬했을 때 j번째 원소인 w_j가 부분집합에 포함된다는 뜻이다. 이를 나타내는 상태공간트리는 그림 5에서 처럼 된다. 이 경우는 트리의 모든 노드가 해공간에 해당되고 tuple의 길이가 일정하지 않은 경우를 표현한다.

위 문제에 대해 고정된 길이의 tuple인 $(x_1, \cdots, x_n)$으로 해공간을 설계할 수도 있다. 이 경우 $x_i=1$이면 w_i가 부분집합에 포함되는 것을 나타내고 $x_i=0$이며 포함되지 않는 것을 나타낸다. 이를 나타내는 상태공간트리는 그림 6에서 처럼 된다. 이 경우에는 트리의 잎노드만이 해공간에 해당된다.

상태공간트리의 입장에서 보면 퇴각 검색법은 상태공간트리를 루트로부터 깊이 우선 탐색을 하면서 그 순서대로 해공간에 속하는 tuple을 검사해 나간다. 탐색도중의 현재 노드를 Z라 하자. Z가 해공간에 해당되는 노드가 아니더라도 현재 상태의 부분 tuple만 가지고 Z의 부트리(subtree)를 더 이상 탐색하지 않아도 된다고 판단할 수 있는 경우가 있다. 이 때 사용되는 것이 한계함수인데 이는 문제에서 주어진 기준함수로부터 설계된다.

요약해서 말하면 퇴각 검색법은 한계함수를 사용하면서 상태공간트리를 깊이 우선 탐색하며 해를 찾는 방법이라 할 수 있다.

n-queen 문제에서 상태공간트리를 그림 4 처럼 설계했을 경우 한 노드 Z에 해당되는 부분 tuple을 $(x_1, \cdots, x_k)$라 할 때 이는 첫번째 여왕부터 k번째 여왕까지의 위치가 잠정적으로 결정되었다는 것이고 이들이 서로 대각선으로 공격하는지 여부를 검사하는 것이 한계 함수가 된다. 그림 4에서 빗금친 노드와 검게 표시된 노드가 탐색되는 노드를 나타내며 특히 검게 표시된 노드는 한계함수에 의해 더이상 탐색되지 않는다고 판단되는 노드이다.

이 예에서 볼 수 있듯이 한계 함수를 적절히

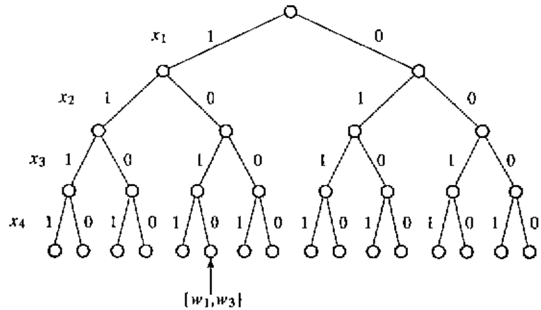

그림 6 부분집합의 합 문제에 대한 상태공간트리, Tuple의 길이가 일정한 경우

사용하면 방문되는 노드의 수를 크게 줄여 전체 효율을 높일 수 있다. 이때 실제로 전체 트리가 만들어 지는 것이 아니고 트리는 알고리즘의 동작 순서를 개념적으로 보여주고 있을 뿐이다. 즉, 그림 4에서 빗금친 노드와 검게 표시된 노드만이 실제 고려되는 부분이다.

다음 알고리즘은 퇴각 검색법의 일반적인 골격을 보여주고 있다. 이 때 $(x_1, \cdots, x_k)$는 상태공간트리의 루트로부터 현재 노드 Z까지의 경로에 해당되는 부분 tuple 이고, Z와 자식노드를 연결하는 에지에 해당되는 모든 x_{k+1} 값을 $T(x_1, \cdots, x_k)$로 표현하며 $B_k(x_1, \cdots, x_k)$는 Z에서 한계 함수 값이다. $B_k(x_1, \cdots, x_k)$ 값이 false일 경우 더 이상 부트리를 검사하지 않아도 된다는 뜻이다. 또한 X는 전역변수로서 현재 상태 부분 tuple인 $(x_1, \cdots, x_k)$를 $X(1), \cdots, X(k)$에 저장한다. 그리고 처음에는 Backtrack(1)을 호출함으로서 시작한다.

```
Algorithm Backtrack(k)
입력 : 상태공간트리에서 현재 노드의 깊이 k
출력 : 현재 노드가 해에 해당되는 tuple이면 이
       를 출력한다.
1. for each X(k)∈T(X(1),…,X(k−1)) do
2.   if Bk(X(1),…,X(k))=true then
3.     (X(1),…,X(k))가 해이면 이를 출력한다.
4.     Backtrack(k+1)
5.   endif
6. endfor
end Backtrack
```

위 알고리즘은 기준함수를 만족하는 모든 해를 출력하는 알고리즘인데 만일 기준함수를 만족하는 최적 해를 찾아야 한다면 단계 3을 수정하면 된다.

퇴각 검색법의 효율은 일반적으로 X(k)를 생성하는데 소요되는 시간과 이의 가능한 갯수, 한계함수를 계산하는데 소요되는 시간, 한계함수를 만족하는 노드의 숫자 등의 요소에 좌우된다. 또한 대부분의 경우 퇴각 검색 알고리즘의 시간 복잡도를 예측하는 것은 매우 어려운 일이며 알고리즘의 조그만 수정이 전체 수행시간에 큰 영향을 미치는 것으로 알려져 있다.

앞으로 한계함수를 설계하는 두가지 예를 제시하겠다. 먼저 앞에서 제시했던 부분집합의 합 문제에서 상태공간트리가 그림 6처럼 될 경우 가장 간단한 한계함수 $B_k(x_1,\cdots,x_k)$로서 다음의 경우에 true인 함수를 설계할 수 있다.

$$\sum_{1\le i\le k} w_i x_i + \sum_{k+1\le i\le n} w_i \le M$$

W의 원소가 크기가 증가하는 순서로 정렬되어 있다고 하자. $\Sigma_{1\le i\le k} w_i x_i \neq M$이면 다음의 경우에 true인 함수를 함께 사용하여 한계함수의 효율을 높일 수 있다.

$$\sum_{1\le i\le k} w_i x_i + w_{k+1} \le M$$

마지막으로 기준함수를 만족하는 최적해를 찾는 문제의 경우 최적해의 상한 값(upper bound) 혹은 하한 값(lower bound)을 사용하여 한계함수를 설계하는 예를 제시한다. 이를 위해 다음 문제를 고려한다.

문제 (0/1-Knapsak) 앞에서 제시한 Knapsack 문제와 동일하나 $x_i \in \{0, 1\}$이어야 한다.

이 문제는 잘 알려진 NP-complete 문제로서 상태공간트리를 부분집합의 합 문제와 유사하게 그림 6에서와 같이 구성할 수 있다. 이 경우 한계함수로서 현재 상태에서 부트리를 계속 탐색했을 경우 얻을 수 있는 최대이득에 대한 상한값을 구하고, 현재까지 구한 가능해(feasible solution) 중에서 가장 이득을 많이 내는 해보다 앞에서 구한 상한값이 적게 되는 경우 한계함수의 값을 false로 하도록 설계할 수 있다. 각 물건이 Greedy_Knapsack에서처럼 p_i/w_i 값이 감소하는 순서로 정렬되어 있다고 가정하자. 이제 상태공간트리에서 현재 노드가 부분 tuple인 $(x_1,\cdots,x_k)$에 해당된다고 하자. 이 경우 $x_1,\cdots,x_k$ 값을 현재 상태로 그대로 고정시키고 $k+1\le i\le n$인 x_i는 $0\le x_i\le 1$로 조건을 완화하여 Greedy_Knapsack으로 이득 값을 계산한다. 그러면 그 값이 현재 상태에서 계속 탐색했을 경우 얻을 수 있는 최대이득에 대한 상한값이 된다.

6. 분기와 한정법

분기와 한정법(branch-and-bound)은 퇴각 검색법과 마찬가지로 상태공간트리의 노드를 생성하며 탐색하는 방법으로서 인공지능이나 OR(Operations Research) 분야에서 활발하게 연구되고 있는 분야이다.

분기와 한정법에서 어떻게 상태공간트리의 노드를 생성하고 탐색하는지 설명하기 위해 먼저 다음을 정의한다. 상태공간트리의 한 노드가 이미 생성되어 있으나 모든 자식 노드가 아직까지 생성되어 있지 않고 있을 때 그 노드를 유효(live) 노드라 하자. 만일 어떤 노드 X의 자식 노드를 모두 다 생성하지는 않았지만 X의 부트리를 더 이상 생성하지 않아도 된다고 이미 알았다면 X는 유효노드가 아니라고 정의한다. 유효노드 중에서 현재 자식노드를 생성하고 있는 노드를 E-노드라 한다.

상태공간트리의 노드를 생성하고 탐색하는 방법 중에서 E-노드의 모든 자식 노드가 생성된 이후에만 다른 유효노드가 E-노드가 될 수 있는 방법을 총칭해서 분기와 한정법이라 한다. 노드를 생성하고 탐색할 때 현재 유효노드 중에서 E-노드를 골라내는 방법에 따라 FIFO(First-In-First-Out)-탐색법, LIFO(Last-In-First-Out)-탐색법, LC(Least-Cost)-탐색법 등으로 분기와 한정법이 분류된다.

FIFO-탐색법은 유효노드를 큐(queue)에 유지하고 LIFO-탐색법은 스택(stack)에 유지하면서 다음 E-노드를 선택할 때 큐나 스택에서 가져온다. 따라서 전자의 경우 상태공간트리를 너비

우선 탐색(breadth first search)하는 것이고 후자의 경우 깊이우선 탐색과 유사하게 된다. 그런데 이 두가지 방법에서는 단순히 상태공간트리의 모양에 따라 다음 E-노드를 결정할 뿐 주어진 입력에는 무관하게 동작하므로 효율적인 방법이라고 할 수 없다. 이러한 점을 고려한 것이 LC-탐색이다.

LC-탐색에서는 주어진 입력에 따라 현재 상태에서의 유효노드에 우선 순위를 부여한다. 즉, 부트리에 해가 있을 가능성이 많다고 생각되는 노드에 우선 순위를 높게 부여하여 그 노드를 제일 먼저 E-노드로 선택한다. 이를 위하여 대부분의 경우 힙(heap)을 사용하여 현재 상태의 유효노드를 유지한다. 만일 유효노드의 깊이가 작을 수록 우선순위를 높게 하면 FIFO-탐색이 되고 부모노드보다 자식노드의 우선순위를 높게 하면 LIFO-탐색이 되므로 FIFO-탐색과 LIFO-탐색은 LC-탐색의 특별한 경우라 할 수 있다. 보통 분기와 한정법이라 하면 LC-탐색 분기와 한정법을 의미한다. 앞으로 LC-탐색에 대해서만 고려하기로 한다. 또한 상태공간트리의 해공간은 항상 잎노드에만 해당되는 경우에 대해서만 설명한다. 그렇지 않은 경우에는 앞으로 설명할 방법을 조금만 수정하면 된다.

퇴각 검색법과 마찬가지로 분기와 한정법은 해가 tuple의 형태로 표시될 수 있을 경우에 사용된다. 특히 목적함수(objective function) 값을 최소 혹은 최대로 하는 최적해를 찾는 문제에 주로 사용된다. 목적함수의 값을 최대로 하는 문제는 함수의 부호만 바꾸면 최소화 하는 문제가 되므로 앞으로 최소화 문제만 고려한다.

분기와 한정법에서 현재 유효노드 중에서 다음 E-노드를 선택하기 위한 우선순위를 결정하기 위해 다음과 같은 비용함수를 정의한다. X를 상태공간트리의 노드라 하자. X가 가능해(feasible solution)에 해당되는 잎노드이면 이에 대한 목적함수 값이 c(X) 값이 된다. 만일 X가 잎노드이면서 가능해에 해당되지 않으면 $c(X)=\infty$이다. X가 잎노드가 아니면 c(X)는 X를 루트로 하는 부트리에 속하는 노드의 비용함수 값 중에서 가장 작은 값이다.

LC-탐색에서 현재 유효노드 중에서 비용함수 값이 가장 작은 노드를 E-노드로 선택하면 항상 최소 갯수의 노드만을 생성하면서 최적해에 해당되는 노드에 다다를 수 있다. 그런데 문제를 해결하기 전에 미리 비용함수 값을 알아낼 수 없으므로 c(X)의 근사 추정 값인 $\hat{c}(X)$를 사용한다. 만일 $\hat{c}(X)$ 값이 c(X) 값과 큰 차이가 없다면 매우 효율적으로 노드를 생성하면서 상태공간트리를 탐색할 수 있다. 다음은 분기와 한정법의 골격을 보여주고 있다.

Algorithm BB(T, $\hat{c}$)

입력 : 상태공간트리의 루트 노드 T
　　　비용함수의 근사 추정함수 $\hat{c}$

출력 : 최적해에 해당되는 노드

1. E:=T
2. 유효노드를 위한 리스트를 초기화 한다.
3. 만일 E가 가능해에 해당되면 루트로부터 E까지의 경로를 최적해로 출력하고 종료한다.
4. E의 모든 자식노드를 현재의 유효노드의 리스트에 집어 넣는다.
5. 만일 현재 유효노드가 한개도 없으면 해가 없다는 것을 출력하고 종료한다.
6. 현재의 유효노드 리스트에서 $\hat{c}(\ )$ 값이 가장 작은 노드를 선택하여 이를 E로 설정한다.
7. goto 3

end BB

위 알고리즘에서 유효노드 리스트는 보통 힙(heap)을 사용한다. 또한 모든 노드 X에 대해 X가 가능해지면 =c(X)이고 그렇지 않을 경우 $\hat{c}(X) \le c(X)$이면 이 알고리즘은 항상 최적해를 찾는다는 것이 보장된다. 만일 E가 단계 3에서 찾는 가능해에 해당되는 노드라면 현재의 다른 유효노드 X'에 대해 $c(E)=\hat{c}(E)\le\hat{c}(X')\le c(X')$이기 때문이다.

이제 $\hat{c}(X)$를 설계하는 방법에 대해 고려해 보자. X가 부분 tuple $(x_1,\cdots,x_k)$를 잠정적으로 결정했기 때문에 c(X)에서 이에 해당되는 값은 쉽게 알 수 있다. 이를 g(X)라 하자. 이를 이용하면 $c(X)=g(X)+h(X)$의 형태로 나타낼 수 있다. 즉, X를 루트로 하는 부트리에서 가능해에 해당되는 노드 중 목적함수 값이 가장 작은 노드를 Z라 할때 h(X)는 X에서 Z까지 가기 위한 추가 비용이다.

따라서 $\hat{c}(X)$를 구하기 위해서는 $g(X)$ 값은 이미 알고 있으므로 $h(X)$ 값만 예측하면 된다. 이를 $\hat{h}(X)$라 하면 $\hat{c}(X)$를 다음과 같이 나타낼 수 있다.

$$\hat{c}(X) = g(X) + \hat{h}(X)$$

만일 $\hat{h}(X) \leq h(X)$이면 $\hat{c}(X) \leq c(X)$이므로 알고리즘 BB는 항상 최적해를 찾게 된다.

이제까지 살펴본 바와 같이 분기와 한정 알고리즘을 설계할 때 가장 핵심이 되는 것은 $\hat{h}(X) \leq h(X)$인 예측값 $\hat{h}(X)$를 구하는 것이다. 또한 $c(X) - \hat{c}(X)$ 값이 작을수록 알고리즘이 효율적으로 동작한다는 것이 알려져 있다. 따라서 분기와 한정 알고리즘의 효율을 높이기 위해서는 $\hat{h}(X) \leq h(X)$이면서 $h(X)$ 값에 가장 근접한 $\hat{h}(X)$를 찾는 것이 중요하다. 다음 문제에 대해 $\hat{c}(X)$ 값을 구하는 예를 보인다.

문제 (15-puzzle) 15개의 타일과 한개의 빈 공간이 그림 7처럼 4×4 구역에 배치되어 있다. 한번에 한개의 타일만을 움직여서 그림 7에서처럼 정렬하고 싶다. 정렬이 가능한 임의의 배치가 주어졌을 때 타일을 움직이는 횟수를 최소로 하면서 이를 정렬하는 순서를 찾아라

이 문제에서 타일을 움직이는 것 대신 빈 공간을 움직인다고 생각해도 되므로 현재 상태에서 빈 공간이 움직이는 네 방향에 따라 최대 네개의 자식노드를 할당하는 상태공간트리를 구성할 수 있다. 상태공간트리의 한 노드 X에 대해 $g(X)$를 타일이 이제까지 움직인 회수로 하면 된다. 그리고 제자리에 있지 않은 타일(빈공간 제외)을 제자리로 보낼려면 적어도 한번은 움직여야 한다. 따라서 $\hat{h}(X)$를 현재 상태에서 제자리에 있지 않은 타일의 갯수로 할 수 있다. 이 경우 $\hat{h}(X) \leq h(X)$라는 것은 자명하다. 이와 같이 $\hat{c}(X)$를 정의하였을 경우 그림 8은 알고리즘 BB가 생성하는 노드와 수행 순서를 보여주고 있다. 이때 각 노드의 $\hat{c}(X)$ 값은 $g(X) + \hat{h}(X)$의 형태로 표시되어 있고 E-노드로 선택되는 순서가 괄호 안의 번호로 표시되어 있다.

분기와 한정법에서도 퇴각 검색법의 경우에서처럼 한계함수를 사용할 수 있다. 보통 주어진 문제의 최적해에 대한 상한값(upper bound)을

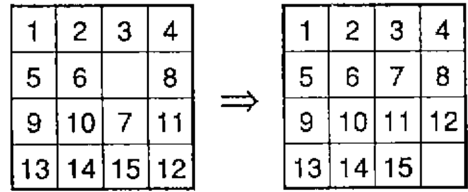

그림 7 15-puzzle 문제

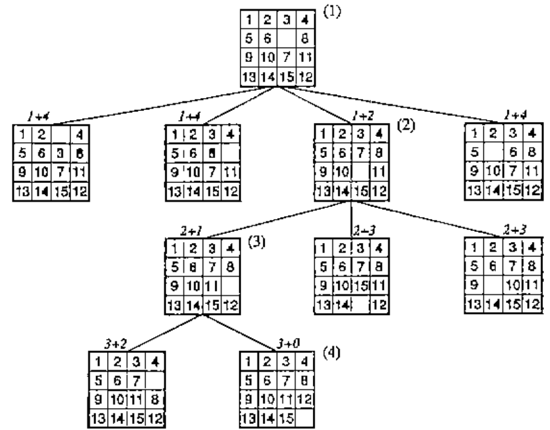

그림 8 15-puzzle 문제에 대한 수행 예

한계함수로 사용한다. 이를 U라 하자. 처음에는 U=∞로 하고 탐색을 하면서 U값을 보정한다. 예를 들어 X를 현재 E-노드라 할때 X의 부트리에 있는 가능해를 예측하여 이에 대한 목적함수 값을 얻거나 혹은 다른 방법으로 최적해에 대한 상한값을 얻는다. 이를 $u(X)$하자. 탐색도중에 만일 $u(X) < U$이면 U를 $u(X)$로 새롭게 보정한다. 알고리즘 BB의 단계 4에서 현재 E-노드의 자식노드 Y를 생성할 때 $\hat{c}(Y) > U$이면 Y를 유효노드 리스트에 집어 넣지 않아도 된다. 왜냐하면 (최적해의 목적함수 값) $\leq U < \hat{c}(Y) \leq c(Y)$이기 때문이다. 다음 문제에 대해 $\hat{c}(X)$와 $u(X)$를 설계하는 예를 보인다.

문제 (기한부 스케줄링) 한개의 CPU를 사용하는 n개의 작업(job)이 있다. 각 작업 i에 대해 세가지 조건 (p_i, d_i, t_i)가 주어지는데 이는 다음을 의미한다. 작업 i는 t_i 시간이 소요되는데 작업이 완료되어야 하는 시각 d_i까지 완료되지 못하면 p_i만큼의 벌칙이 부과된다. 모두 완료시간 안에 끝낼

수 있는 작업을 선택하되 선택되지 않은 작업들에 의해 부과되는 벌칙의 합이 최소가 되도록 하라.

이 문제에 대해 상태공간트리를 그림 6처럼 구성할 수 있다. 현재 노드를 X라 하고 X에 대한 부분 tuple을 $(x_1,\cdots,x_k)$라 하면 $\hat{c}(X)$를 다음과 같이 간단히 설계할 수 있다.

$$g(X)=\sum_{1\le i\le k} p_i(1-x_i)$$

$$\hat{h}(X)=0$$

이 경우 $\hat{c}(X)\le c(X)$인 것은 명백하다. 물론 $\hat{h}(X)$ 값을 좀 더 효율적으로 설정하는 것은 어려운 일이 아니다. 이제 u(X)를 다음과 같이 설계할 수 있다.

$$u(X)=\sum_{1\le i\le k} p_i(1-x_i)+\sum_{k\le i\le n} p_i$$

이를 이용하면 상한값 U를 유지하면서 탐색할 수 있다.

7. 무작위 알고리즘

알고리즘의 진행 도중에 현재 상태에서 앞으로 어떤 방향으로 계속진행해야 할지 판단을 내려야 할때가 있다. 가능성이 있는 여러 선택방향 중에서 한개를 선택할때 난수(random number)를 사용하여 무작위로 선택하여 수행을 계속하는 알고리즘을 총칭하여 무작위 알고리즘(randomized algorithm)이라 한다.

다음 경우를 고려해 보자 선택을 해야하는 시점에서 선택의 가능성이 매우 많이 있어 그 중에서 최선의 선택을 하기 위해서는 많은 비용 즉, 많은 수행시간이 사용된다고 하자. 이때 무작위로 선택하면 수행시간은 빨라지게 되나 나쁜 선택을 할 가능성이 있다. 만일 전자에 경우에 의해 수행 시간에서 이득을 얻을 가능성이 후자에 의해서 손해를 볼 가능성 보다 크다면 무작위 알고리즘은 매우 유용하다.

또 다른 면을 고려해 보자. 일반적으로 평균 시간복잡도(average-case time complexity)를 분석 할때 입력의 확률분포를 가정한다. 그런데 실제 입력이 항상 그 가정을 따르지 않는 경우가 많다. 이 경우 무작위 알고리즘을 설계할 수 있으므로 유용하다. 예를 들어 퀵정렬(quick sort) 알고리즘과 같이 입력이 무작위로 섞여 있을 때 효율적으로 동작하는 알고리즘이 있다고 하자. 이 경우 어떤 입력이 주어지더라도 일단 그 입력의 순서를 무작위로 섞어 놓고 정렬을 하도록 무작위 알고리즘을 설계하면 입력의 확률 분포에 무관하게 효율적인 시간 복잡도가 보장될 것이다.

무작위 알고리즘의 특징은 입력자료 뿐만이 아니라 난수발생기의 동작에 의해서 알고리즘의 수행이 좌우 된다는 것이다. 따라서 똑같은 입력이 주어진다고 하더라고 난수발생기의 동작에 의해 매번 서로 다르게 동작한다. 그러나 출력은 항상 동일하다. 이러한 이유로 인해 최악의 경우의 입력이라는 것은 있을 수 없고, 난수발생기가 최악의 난수를 만들어 낼 때가 최악의 경우이다.

다음의 알고리즘은 퀵정렬을 무작위 알고리즘으로 수정한 것인데 난수발생기가 최악의 경우를 거의 만들어 내지 않는 예를 보이고 있다. 다음의 알고리즘에서 Random(i, j)는 i 이상 j 이하의 정수를 발생시키는 난수발생기이고 A는 초기에 n개의 입력을 가지고 있는 전역변수(global variable)이다. 처음에는 Randomized_Quicksort (1, n)을 호출함으로써 수행이 시작된다.

Algorithm Randomized_Quicksort(first, last)
입력 : A(first..last)
출력 : A(first..last)가 정렬되어 재배치된 것
1. 만일 first≥last이면 종료된다.
2. pivot: = Random(first, last)
3. A(first..last)에서 A(pivot)의 위치를 찾고 A(first..pivot − 1)의 값이 A(pivot + 1..last) 값 보다 작도록 A(first..last)를 재배열한다.
4. Randomized_Quicksort(first, pivot − 1)
5. Randomized_Quicksort(pivot + 1, last)
end Randomized_Quicksort

위 알고리즘에서 입력을 두 부분으로 나누는 위치를 선택할 때 난수발생기를 이용하고 나머지 부분은 퀵정렬과 동일하다. Randomized_Quicsort는 최악의 수행시간과 평균 수행시간이 각각

$O(n^2)$과 O(n log n)으로 무작위 알고리즘이 아닌 경우와 동일하다. 그러나 최악의 경우를 만들어낼 가능성이 매우 희박한데 좀더 자세히 살펴보면 n!개의 입력조합 중에서 수행시간이 O(n log n)이 아닌 경우 확률이 $O(1/n^k)$이다. 이때 k는 임의의 양수이다.

마지막으로 무작위 알고리즘이 효율적으로 동작하는 다른 예를 제시한다.

문제 (리스트 탐색) 길이가 n인 연결 리스트(linked list)가 두 배열 key(1..n)과 next(1..n)을 사용하여 표현되어 있다. 이때 key는 자료를 나타내는 부분이고 next는 연결을 나타내는 부분이다. 또한 1≤i≤n이고 next(i)≠nil인 모든 i에 대해 key(i)<key(next(i))이다. 임의의 입력 k가 주어졌을 때 key(i)=k인 i를 찾아라.

이 문제에 대해 다음과 같은 무작위 알고리즘을 설계할 수 있다.

```
Algorithm Randomized_Search(head, k)
입력 : 연결 리스트의 시작위치 head
       위치를 찾으려는 값 k
출력 : 연결 리스트에서 k의 위치
1. i:=head
2. while i≠nil and key(i)≤k do
3.   if key(i)=k then return i
4.   j:=Random(1, n)
5.   if key(i)<key(j)<k then i:=j
6.   i:=next(i)
7. endwhile
end Randomized_Search
```

이 알고리즘에서 만일 단계 4와 5를 제외하면 연결 리스트를 탐색하는 가장 기본적인 알고리즘이 되고 최악 수행시간과 평균 수행시간이 모두 O(n)이 된다. 이 알고리즘에서는 난수발생기를 이용하여 탐색 위치를 건너뛸 수 있도록 하는 단계를 추가하여(단계 4, 5) 무작위 알고리즘이 되었다. Randomized_Search의 평균 수행시간은 $O(\sqrt{n})$으로 무작위 알고리즘이 매우 효과적인 예를 보이고 있다.

참고문헌

다음은 알고리즘 설계 및 분석 분야에서 자주 참고되는 문헌이다. 본 고에서는 그 중에서 [Aho-Hopcroft-Ullman 83], [Baase 88], [Cormen-Leiserson-Rivest 92], [Horowitz-Sahni 78], [Manber 89], [Preparata-Shamos 85] 등을 참조하였다.

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, Massachusetts, 1983.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[3] S. Baase, *Computer Algorithms : Introduction to Design and Analysis*, Addison-Wesley, Reading, Massachusetts, second edition, 1988.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1992.

[5] M. Gondran and M. Minoux, *Graphs and Algorithms*, John Wiley & Sons, New York, 1984.

[6] D. Harel, *Algorithmics : The Sprit of Computing*, Addison-Wesley, Reading, Massachusetts, 1987.

[7] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, Maryland, 1978.

[8] T. C. Hu, *Combinatorial Algorithms*, Addison-Wesley, Reading, Massachusetts, 1982.

[9] D. E. Knuth, *Fundamental Algorithms, The Art of Computer Programming, vol. 1*, Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[10] J. V. Leeuwen, Ed., *Algorithms and Complexity, handbook of Theoretical Computer Science, vol. A*, Elsevier, Amsterdam, 1990.

[11] U. Manber, *Introduction to Algorithms : A Creative Approach*, Addison-Wesley, Reading, Massachusetts, 1989.

[12] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization : Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[13] F. P. Preparata and M. I. Shamos, *Computational Geometry : An Introduction*, Springer-Ver-

lag, New York, 1985.
[14] R. Sedgewick, *Algorithms,* Addison-Wesley, Reading, Massachusetts, 1983.
[15] M. N. S. Swamy and K. Thulasiraman, *Graphs, Networks, and Algorithms,* John Wiley & Sons, New York 1981.

좌 경 룡

1971 서울대학교 전기공학과를 졸업하고, 노스웨스턴대학교 전기 및 전산학과에서 석사(1977)와 박사(1980)학위를 받았음. 현재 한국 과학기술원 전산학과 교수로 근무 중

관심 분야 : 그래프 이론, 알고리즘 설계 및 해석, 계산기하학, 시스템 고장진단