

DP Algorithm



IMHOJEONG

2021.01.19 ImHoJeong

DP

1950년대 개발됨

항공 우주공학에서 경제학에 이르기까지 다양한 분야에서 응용



Richard Bellman

참고 : https://en.wikipedia.org/wiki/Richard_E._Bellman

what is DP?

복잡한 문제를 재귀 방식으로 더 간단한 하위 문제로 분해해 단순화하는 것

Programming => 여기서는 Coding이 아닌 테이블을 채운다는 문자적 의미

동적 계획법, 메모화기는 함께 동작

동적 계획법 VS 분할 정복

주요 차이점 : 부분 문제들이 겹침 VS 부분 문제들이 독립적

why is DP?

메모하기(이미 푼 부분 문제들의 테이블 유지하기)를 이용하는 방법

많은 문제에서 지수적 복잡도 => 다항적 복잡도

$$O(c^n), O(3^n) \Rightarrow O(n^c), O(c^n)$$

동적 계획법 = 재귀 + 메모하기

why is DP?

최적 부분 구조(Optimal Substructure) - 부분 문제들에 대한 최적의 해답을 가진 문제의 최적의 해답

겹치는 부분 문제(Overlapping Subproblems) - 여러 번 반복되는 몇 가지 부분 문제들을 포함하는 재귀적 해법

탐욕, 분할 정복 기법처럼 역시 모든 문제를 풀 수 없음

동적 계획법 VS 재귀 - 재귀적 호출의 메모화기의 활용 유무

부분 문제들이 독립적이고 반복이 없다면 메모화기를 사용하는 것이 의미 $\times \Rightarrow$ 동적 계획법은 모든 문제의 해법 \times

How solve DP Problems?

Bottom-up :

부속 문제들로 나뉘어져서 필요한 경우 이
부속 문제들이 풀리고 그 해답이 저장

각각의 계산된 값을 재귀 함수의 마지막
동작으로 저장하고

&

재귀 함수의 맨 첫 동작에서는
미리 계산된 값이 존재하는지 검사

Bottom-up vs Top-down

Top-down :

작은 입력 인자 (함수 계산 후)
---> 인자 값 증가

값들을 계산할 때 모든 계산된 값을
테이블(메모리)에 저장

더 큰 인자들이 계산될 때 =>
작은 인자들에서 미리 계산된 값들이
사용될 수 있음

DP Problems examples

가장 긴 공통 부분 수열 (LCS)

가장 긴 공통 부분 문자열 (LCS)

편집 거리 (Edit distance)

다익스트라, 벨만-포드, 플로이드-워셜

연속 행렬 곱셈, 0/1 배낭, 부분집합의 합, 외판원 순회 문제 등

·
·
·

가장 긴 공통 부분 수열 (LCS)

Longest Common SubSequence

주어진 여러 개의 수열 모두의 부분수열이 되는 수열들 중에 가장 긴 것을 찾는 문제

diff 유틸리티의 근간 & 생물 정보학에서 많이 응용

https://en.wikipedia.org/wiki/Diff#Algorithmic_derivatives

참고 :

https://ko.wikipedia.org/wiki/%EC%B5%9C%EC%9E%A5_%EA%B3%B5%ED%86%B5_%EB%B6%80%EB%B6%84_%EC%88%98%EC%97%B4

가장 긴 증가하는 부분 수열 (LIS)

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

주어진 한 개의 수열의 부분수열들 중에 오름차순으로 정렬된 가장 긴 것을 찾는 문제

Git과 같은 버전 관리 시스템에서 사용됨

동적 계획법 + 이진탐색을 통해

$$O(n^2) \Rightarrow O(n \log n)$$

<https://www.geeksforgeeks.org/longest-common-substring-dp-29/>

참고 : https://en.wikipedia.org/wiki/Longest_increasing_subsequence

가장 긴 공통 부분 문자열(LCS)

Longest Common SubString

주어진 여러 개의 문자열 모두의 부분문자열이 되는 문자열들 중에 가장 긴 것을 찾는 문제

편집 거리(Edit Distance)

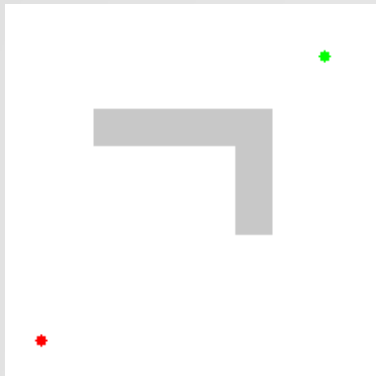
많은 문자열 알고리즘이 동적 계획법을 활용함

한 문자열을 다른 문자열로 변환하는 데 필요한 최소 작업 수를 계산해

두 문자열이 서로 얼마나 다른지를 수량화하는 방법

하나의 알고리즘도 다양한 관점으로 볼 수 있다

다익스트라 알고리즘(Dijkstra's Algorithm)



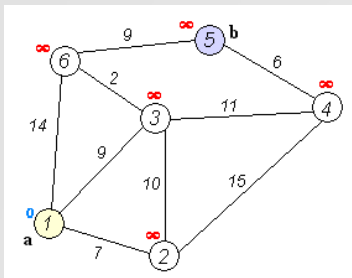
로봇 모션 계획 문제

주어진 두 개의 꼭짓점 P와 Q 사이의 최소 거리를 가지는 경로는?

R이 P에서 Q로 가는 최단 경로에 있는 꼭짓점이라면,
이 경로는 마찬가지로 R까지 가는 최단 경로라는 사실을 이용

네트워크 라우팅 프로토콜(IS-IS, OSPF)에서 주로 이용

다익스트라 알고리즘(Dijkstra's Algorithm)



동적 계획법의 관점 - 최적성의 원리를 최단 경로 문제의 맥락에서 해석한 것

주어진 두 개의 꼭짓점 P와 Q 사이의 최소 거리를 가지는 경로는?

R이 P에서 Q로 가는 최단 경로에 있는 꼭짓점이라면,

이 경로는 마찬가지로 R까지 가는 최단 경로라는 사실을 이용

네트워크 라우팅 프로토콜(IS-IS, OSPF)에서 주로 이용

다익스트라 알고리즘(Dijkstra's Algorithm)

- 각 교차로에는 현재까지의 최단 거리가 적힘

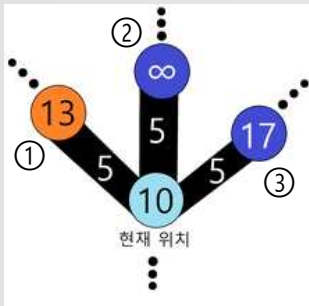
- 현재 위치에서 이웃 교차로까지 5의 거리로 갈 수 있다면?

이웃 교차로에는 15의 거리로 도착할 수 있음

① 왼쪽 교차로 - 이미 다른 경로를 통해 13의 거리로 도착 가능

② 가운데 교차로 - 방문하지 않은 교차로이므로 갱신

③ 오른쪽 교차로 - 더 짧은 거리로 도착할 수 있으므로 갱신



다익스트라 알고리즘(Dijkstra's Algorithm)

$O(|V|^2)$

edge relaxation : 초기 거리 값을 부여하고, 단계를 거듭하여 개선

```
int start = 0;
int end = V-1; ①
int[][] adjMatrix = new int[V][V];
int[] distance = new int[V];
boolean[] visited = new boolean[V];

Arrays.fill(distance, Integer.MAX_VALUE); ②
distance[start] = 0;
```

1. 모든 꼭짓점을 미방문 상태로 표시
 - 모든 미방문 꼭짓점의 집합을 만듦
2. 모든 꼭짓점에 시험적 거리 값 부여
 - 초기값을 0으로, 다른 모든 꼭짓점을 무한대로 설정
 - 초기점을 현재 위치로 설정

참고 :

다익스트라 알고리즘(Dijkstra's Algorithm)

$O(|V|^2)$

```
for(int i = 0 ; i < V ; i++){
    ③
    int min = Integer.MAX_VALUE;
    int current = 0;

    for(int j = 0 ; j < V ; j++){
        if(!visited[j] && min > distance[j]){
            min = distance[j];
            current = j;
        }
    }
    ④
    visited[current] = true;
    ⑤ if(current == end) break;

    // ....
}
```

3. 현재 꼭짓점에서 미 방문 인접 꼭짓점을 찾아 그 임시 거리를 현재 꼭짓점에서 계산

- 새로 계산한 시험적 거리를 현재 부여된 값과 비교해 더 작은 값을 넣는다

4. 만약 현재 꼭짓점에 인접한 모든 미방문 꼭짓점까지의 거리를 계산했다면,
현재 꼭짓점을 방문한 것으로 표시 & 미방문 집합에서 제거

5. 두 꼭짓점 사이의 경로를 찾는 경우

- 도착점이 방문한 상태로 표시되면 멈추고 알고리즘을 종료

참고 :

다익스트라 알고리즘(Dijkstra's Algorithm)

$O(|V|^2)$

```
// ....  
for(int j = 0 ; j < V ; j++){  
    ⑦  
    if(!visited[j] && adjMatrix[current][j] != 0 &&  
        distance[j] > min + adjMatrix[current][j]){  
  
        distance[j] = min + adjMatrix[current][j];  
  
    }  
}
```

6. 완전 순회 경로를 찾는 경우

- 미방문 집합에 있는 꼭짓점들의 시험적 거리 중 최솟값이 무한대?

=> 출발점과 미방문 집합 사이에 연결이 없는 경우

=> 멈추고 알고리즘 종료

7. 아니면 시험적 거리가 가장 작은 다음 미방문 꼭짓점을
새로운 현재 위치로 설정

- 3. 단계로 되돌아가 현재 꼭짓점과 미방문 인접 꼭짓점을
찾아

참고 :

벨만-포드 알고리즘(Bellman-Ford Algorithm)

가중 유형 그래프에서 최단 경로 문제를 푸는 알고리즘

변의 가중치는 음수일 수 있음!!

다익스트라 알고리즘 : 벨만 포드 알고리즘 보다 실행속도도 더 빠르고 동일한 작업을 수행

$$\theta(|V||E|)$$

참고 :

https://ko.wikipedia.org/wiki/%EB%B2%A8%EB%A8%BC-%ED%8F%AC%EB%93%9C_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

플로이드-워셜 알고리즘(Floyd-WarShall Algorithm)

$$\theta(|V|^3)$$

변의 가중치가 음이거나 양인 가중 그래프에서 최단 경로들을 찾는 알고리즘

why? 그래프에서는 최대 변이 있을 수 있고, 모든 변의 조합을 확인하기 때문

$$\Omega(|V|^2)$$

1에서 N까지 번호가 매겨진 V를 꼭짓점으로 갖는 그래프 G

i에서 j로 집합 $\{1, 2, \dots, k\}$ 의 꼭짓점들 만을 경유지로 거쳐 가는 최단 경로를 반환하는 함수 $\text{ShortestPath}(i, j, k)$

플로이드-워셜 알고리즘(Floyd-Warshall Algorithm)

1에서 N까지 번호가 매겨진 V를 꼭짓점으로 갖는 그래프 G

i에서 j로 집합 $\{1, 2, \dots, k\}$ 의 꼭짓점들만을 경유지로 거쳐 가는 최단 경로를 반환하는 함수 ShortestPath(i, j, k)

(목표 : $\{1, 2, \dots, N\}$ 에 있는 꼭짓점만을 이용해 모든 꼭짓점 i에서 모든 꼭짓점 j로 가는 경로를 찾는 것)

각각의 꼭짓점 상에 대해,

1) k를 통과하지 않는 경로 - 집합 $\{1, \dots, k-1\}$ 에 있는 꼭짓점만 거쳐감

2) k를 통과하는 경로 - i에서 k까지와 k에서 j까지 가는 경로 모두 $\{1, \dots, k-1\}$ 에 있는 꼭짓점만을 거쳐감

플로이드-워셜 알고리즘(Floyd-Warshall Algorithm)

i 에서 j 까지 1에서 $k-1$ 의 꼭짓점만을 거쳐가는 경로 중

최선의 경로는 $\text{ShortestPath}(i,j,k-1)$ 에 의해 정의됨

+

만약 i 에서 k 를 거쳐 j 로 가는 더 나은 경로가 있다면,

그 경로는 i 에서 k 까지 ($\{1, \dots, k-1\}$ 만을 거쳐서) 가는 경로와

k 에서 j 까지($\{1, \dots, k-1\}$ 만을 거쳐서) 가는 경로를 합친 것

플로이드-워셜 알고리즘(Floyd-WarShall Algorithm)

$$\text{shortestPath}(i,j,k) \begin{cases} w(i,j) & \text{if } k = 0 \\ \min(\text{shortestPath}(i,j,k-1), \text{shortestPath}(i,k,k-1), \text{shortestPath}(k,j,k-1)) & \text{if } k \neq 0 \end{cases}$$

$w(i,j)$: 꼭짓점 i,j 간의 변의 가중치

처음에 모든 (i,j) 쌍에 대해서 $k = 1$ 일 때 $\text{shortestPath}(i,j,k)$ 를 계산하고,

다음으로 $k = 2$ 일 때를 계산하는 방식 ...

$k = N$ 이 될 때까지 계속하면, 모든 (i,j) 쌍에 대해서 최단 경로를 가짐

플로이드-워셜 알고리즘(Floyd-Warshall Algorithm)

```
int[][] dist = new int[N][N];  
  
for(int i = 0; i < N ; i++){  
    Arrays.fill(dist[i], Integer.MAX_VALUE);  
}  
  
for(int i = 0; i < N ; i++){  
    int start = Integer.parseInt(st.nextToken());  
    int end = Integer.parseInt(st.nextToken());  
    int cost = Integer.parseInt(st.nextToken());  
    distance[start][end] = Math.min(distance[start][end], cost);  
}
```

1. distance 배열을 Int의 최댓값으로 초기화

2. 변 start와 end의 가중치 중 최솟값으로 설정

플로이드-워셜 알고리즘(Floyd-Warshall Algorithm)

```
for(int k = 0 ; k < N ; k++){  
    for(int i = 0 ; i < N ; i++){  
        for(int j = 0 ; j < N ; j++){  
            if(distance[i][j] > distance[i][k] + distance[k][j]){  
                distance[i][j] = distance[i][k] + distance[k][j];  
            }  
        }  
    }  
}
```

③

3. 경유지를 거쳐가는 것을 고려하는 것이 핵심!

- 왜 이 공식이 나오게 되었는가?

만약 i 에서 k 를 거쳐 j 로 가는 더 나은 경로가 있다면,
그 경로는 i 에서 k 까지 ($i, \dots, k-1$)만을 거쳐서 가는 경로 + k 에서 j 까지 ($\{1, \dots, k-1\}$)만을 거쳐서 가는 경로를 합친 것이라는 것은 자명하기 때문!

연속 행렬 곱셈(Chained Matrix Multiplications)

주어진 행렬 시퀀스를 곱하는 가장 효율적인 방법에 관한 최적화 문제

괄호를 어떻게 묶어도 얻은 결과는 동일하게 유지됨

$$((AB)C)D = ((AB)C)D = (AB)(CD)$$

곱에는 영향을 미치지 않지만 괄호 안의 순서는 곱을 계산하는 데 필요한 연산 수(계산 복잡도에 영향 줌)

참고 : https://en.wikipedia.org/wiki/Matrix_chain_multiplication

연속 행렬 곱셈(Chained Matrix Multiplications)

예를 들어, A가 10×30 , B가 30×5 , C가 5×60

$$(AB)C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$$

$$A(BC) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000$$

=> 1번째 방법이 효율적 => n 행렬 곱의 최적 괄호를 결정하는 방법은?

무식하게 푸는 방법에 있어선, 행렬 수 n이 크면 매우 느리고 비 실용적

더 빠른 해결책은 문제를 하위 문제 세트로 분해해 생각해 볼 수 있어

0/1 배낭(0-1 Knapsack)

조합 최적화의 문제 - 고정된 크기의 배낭에 의해 가장 귀중한 물건으로 채워야 하는 문제

i	v	w	w							
				0	1	2	3	4	5	6
1	5	4	i	0						
2	4	3	1							
3	3	2	2							
4	2	1	3							
Capacity=6			4							

예) 고정된 예산이나 시간 제약 하에서

분할할 수 없는 일련의 프로젝트

or 작업 중에서 선택해야 하는 리소스 할당

원자재를 줄이는 가장 낭비가 적은 방법 찾기, 투자 및 포트폴리오 선택

자산 담보 증권화를 위한 자산 선택 등 실제 의사 결정 프로세스에 사용

참고 : https://en.wikipedia.org/wiki/Knapsack_problem

부분집합의 합 (Subset Sum) (SSP)

knapsack problem와 multiple subset sum 문제의 특별한 경우의 문제

정수의 다중집합 S , 목표합 T 가 있고 정수의 하위 집합이 정확히 T 인 것을 구하는 문제

1. 모든 입력이 양수

2. 입력이 양수 또는 음수 그리고 $T = 0$ ex) $\{-1, -3, -2, 9000, 5, 8\}$

3. 모든 입력이 양수이고 T 가 모든 입력의 합의 절반이 되는 경우 (Partition Problem)

외판원 순회 문제(Traveling Salesman problem)

도시 목록과 각 도시 쌍 사이의 거리를 고려할 때

각 도시를 정확히 한 번 방문하고 원래 도시로 돌아가는 최단 경로는?

https://www.youtube.com/watch?v=-YKX6Njn_BQ

Thank you!

39ghwid@naver.com