Scheduling Unit-time tasks with Release time and Deadline

2018년 8월 10일 금요일

삼성소프트웨어멤버쉽 구재현 (koosaga.com)

Contents

1 소개 **1**

2 Forbidden Regions **2**

3 단순한 알고리즘 **3**

4 $O(n^2)$ 알고리즘 **5**

5 References **6**

1 소개

작업 스케줄링 문제는 주어진 제약 조건 하에서 여러 개의 작업을 적당한 시간과 기계에 대응시키고, 대응시킨 후의 **점수** 를 최대화/최소화 하는 문제이다.

실제 응용에서의 요구 사항이 언제나 같을 수 없기 때문에, 이 제약 조건과 점수의 정의는 이론적으로 무한하고, 명확한 정의는 각각의 문제가 요구하는 상황에 따라서 달라지기 마련이다. 또한, 이 명확한 정의가 어떠한 형태로 구성되어 있는지 여부만으로도, 문제의 해결 양상이 급격히 달라지는 경우가 잦다. Peter Brucker의 Scheduling Algorithm에는 다양한 점수 함수와 제약 조건의 예가 약 10개 - 20개 정도 제시되어 있다. [1]

스케줄링 문제 중에서 자주 소개되는 형태의 문제로는 Deadline Scheduling이 있고, 이를 아래에 정의한다.

- 각각의 작업은 (d_i, t_i) 라는 tuple으로 표기되며, 이는 i번째 작업이 d_i 시간 안에 완료되어야 하며, 작업을 수행하는 데 t_i 의 시간이 걸림을 뜻한다. d_i, t_i 는 정수이다.

- 하나의 기계는 한 순간 한 작업만 할 수 있으며, 한번 시작한 작업은 끝내야만 한다 (non-preemptive).

- 점수 함수는, 모든 작업을 처리했을 때는 1이고, 그렇지 않았을 때는 0이다.

이러한 류의 문제는 걸리는 시간을 무시하고 데드라인 순으로 정렬하였을 때 쉽게 해결할 수 있음이 잘 알려져 있다. 고로, 위 문제는 $O(n \log n)$에 해결할 수 있는 문제이다.

하지만, 일반적으로는 어떠한 작업에 대해서 Deadline만 존재하는 경우는 드물다. 과제에 대한 공지가 나와야지 과제를 시작할 수 있듯이, 대부분의 작업은 일을 시작할 수 있는 시간이라는 제약 조건이 추가적으로 존재한다. 이 개념을 Release Time이라고 정의한다. 일반화된 문제인 Release Time Scheduling을 다음과 같이 정의한다.

- 각각의 작업은 (r_i, d_i, t_i) 라는 tuple으로 표기되며, 이는 i번째 작업이 r_i 시간 뒤에 시작해야 하고, d_i 시간 안에 완료되어야 하며, 작업을 수행하는 데 t_i 의 시간이 걸림을 뜻한다. r_i, d_i, t_i 는 정수이다.

- 하나의 기계는 한 순간 한 작업만 할 수 있으며, 한번 시작한 작업은 끝내야만 한다 (non-preemptive).
- 점수 함수는, 모든 작업을 처리했을 때는 1이고, 그렇지 않았을 때는 0이다.

Release Time Scheduling 문제는 NP-hard 문제로 [2], 문제에 대한 충분한 제약 조건이 없으면 풀기 곤란하다. 일반적으로 이 문제에 걸리는 제약 조건은 $t_i = T$, 즉 모든 작업을 처리하는 데 일정한 시간이 소요된다는 가정이 있다. 특히, $T = 1$일 때 이 문제는 일반적인 Deadline Scheduling과 비슷한 전략을 사용할 수 있다. 예를 들어, 시작 시간 순서대로 우선순위 큐에 작업을 넣으면서, 현재 할 수 있는 작업 중에서 가장 Deadline이 촉박한 작업을 뽑아내는 전략이 유효하다.

고로, $T \neq 1$ 일 때 역시 비슷한 전략이 될 것이라고 추측할 수 있다. 일반성을 잃지 않고 모든 Release Time과 Deadline을 T로 나누면, 결국에는 위와 같은 문제가 되기 때문이다. 하지만 놀랍게도 $T \neq 1$일 때는 이 전략을 그대로 사용하는 것이 대단히 복잡해진다. 이는, 현재 순간에 일을 한 것과 다음 순간에 일을 한 것이 독립이 되지 않고, 일을 할 수 있는 상황이다 하더라도 일을 하지 않고 기다리는 게 이득을 줄 수 있기 때문이다. 이 문제는 2017년 ICPC World Finals에서 "Scenery" 라는 이름으로 출제되었을 때, $n \leq 10000$ 알고리즘이 통과함에도 불구하고 한 팀도 풀지 못했던 난이도를 가진 문제였다. [3]

이 문제는 다행이도 다항 시간, 그것도 $O(n \log n)$ 에 해결할 수 있는 풀이가 존재한다. Garey et. al이 1981년 발표한 "Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines" 논문이 이 풀이에 대해서 설명한다.[2] 풀이의 내용이 상당히 이해하기 어렵지만, 다항 시간 풀이를 찾는 방법이 상당히 아름답기 때문에 시도해 볼 만한 가치가 충분히 있는 내용이라고 여긴다. 이번 과제에서는 $O(n \log n)$ 은 생략하고, 문제를 해결할 수 있는 $O(n^2)$ 풀이를 다룬다.

2 Forbidden Regions

시간이 증가하는 순으로 작업을 배정하는 스케줄러를 생각해 보자. 어떠한 순간에 일을 하겠다고 마음을 먹었다면, 그 때의 선택은 할 수 있는 일 중에서 데드라인이 가장 작은 일을 수행하는 것이 무조건 이득임을 알 수 있다. 이는 그리디 알고리즘에서 흔히

사용하는 Exchange Argument를 사용하면 쉽게 보일 수 있다. 어떠한 시점에서 일을 하겠다고 했을 때 데드라인이 큰 일을 선택한다면, 이 선택을 하지 않고도 그와 같은 결과를 보일 수 있기 때문이다. 즉, 이 문제에서 곤란한 부분은 어떠한 순간에 어떠한 일을 해야 하는 지가 아니라, 어떠한 순간에 일을 해야 하는 지 말아야 하는 지이다.

이 문제는 Forbidden Regions라는 개념을 도입해서 이 부분의 곤란함을 제거한다. Forbidden Region은 끝점을 포함하지 않는 개구간들의 합집합이며, 어떠한 시간이 이 개구간 안에 들어갈 경우, 그때 할 작업이 있더라도 불구하고 스케줄링을 하면 안된다는 것을 뜻한다. 만약 우리가 구한 Forbidden Regions가 완전한 정보를 제공한다면, 어떠한 순간에 Forbidden Regions에 들어왔다면 구간이 끝날 때까지 기다리고, 그렇지 않을 경우에는 무조건 할 수 있는 일 중 가장 데드라인이 작은 일을 수행해 버리면 된다. 즉, $T = 1$일 때와 큰 차이가 없는 상황이 생긴 것이다. 이를 이벤트가 바뀔 때마다 시행해 주면, 구간의 개수가 C개 일 때 $O(N \log N + C)$ 에 문제를 해결할 수 있다.

이 문제가 놀라운 이유는, 위의 역할을 해 주는 Forbidden Region을 다항 시간 안에 구할 수 있다는 것이다. 먼저, 가장 단순한 알고리즘으로 Forbidden Region을 정의하고, 이 Forbidden Region을 고려한 그리디 알고리즘이 왜 올바른지를 증명한다.

3 단순한 알고리즘

알고리즘은 기본적으로 이미 구해놓은 Forbidden Region이 존재할 때 이를 늘릴 수 있을 때까지 늘려나가는 방식으로 진행된다. 현재 이미 구해놓은 Forbidden Region을 FR 이라고 정의하자. $r_i \le d_j$ 를 만족하는 임의의 두 구간 $[r_i, d_i]$ 와 $[r_j, d_j]$ 를 고른 후, 다음과 같은 방식으로 새로운 Forbidden Region을 찾는다.

- 1. $r_i \le r_k, d_k \le d_j$ 를 만족하는 작업 $[r_k, d_k]$들을 모두 모은 후, 이들의 시작 / 끝 시간을 모두 무시한다. (즉, 개수만 구해 주면 된다.)
- 2. 이들을 모두 구간 $[r_i, d_j]$에 매칭해 줄 것인데, 이 때 d_j 에서부터 거꾸로 Greedy하게 스케줄링을 시도한다. 현재 Forbidden Region을 고려한 상태에서, 최대한 끝점을 뒤로 밀면서 스케줄링을 하는 것이다.
- 3. 이후 3가지 케이스가 있다.

- Case 1. 모든 구간을 스케줄링하지 못했다. 이 경우에 답은 자명하게 불가능이다.
- Case 2. 마지막으로 스케줄된 구간의 시작점 r_{last} 가 $r_{last} < T + r_i$ 를 만족하는 것이다. 이 경우에는, 만약에 다른 구간이 $(r_{last} - T, r_i)$ 와 같은 열린 구간에서 시작된다면 해당 구간에 대한 스케줄링이 불가능해 질 것이다. 고로 이 구간을 Forbidden Region에 추가한다.
- Case 3. $r_{last} \geq T + r_i$ 를 만족하는 경우이다. 이 경우에는 무시한다.

이 과정을 Forbidden Region을 늘릴 수 없거나, 불가능함을 찾지 못할 때까지 반복한다. 불가능함을 찾지 못했다면, 항상 스케줄링이 가능하다 (이는 후에 증명한다.)

Lemma 1. Deadline First Scheduling을 $r_i \leq r_k, d_k \leq d_j$ 를 만족하는 작업들에 대해서만 수행했을 때, Forbidden Region에 있지 않은 한 항상 새로운 작업을 수행했고 (즉, 작업 목록이 비지 않았고) 결과적으로 스케줄링에 실패했다면, 위 Forbidden Region을 찾는 알고리즘에서 $[r_i, d_j]$ 를 입력으로 넣었을 때 불가능을 반환한다.

Proof. Forbidden Region의 구간 개수에 대한 귀납법으로 가능하며 자세한 증명은 생략한다. □

Remark. 그림으로 그려보면 매우 직관적이다.

Lemma 2. 위 방법으로 불가능함을 보이지 못하고 Forbidden Region을 찾았다면, Deadline First 그리디 알고리즘은 항상 스케줄링에 성공한다.

Proof. Deadline First Scheduling 과정에서 맨 처음으로 스케줄링에 실패한 구간 (즉, 스케줄링을 하였을 때 deadline 이후에 끝남이 확인된) 을 $[r_i, d_i]$ 라고 하자.

먼저, 실패 전의 스케줄링 과정에서, Forbidden Region 안에 있지 않았으나 할 일이 없어서 아무 일도 하지 않은 상황이 있었다. 이 경우 그 시점 전과 뒤가 개별적인 문제가 된다. 그 경우에는 시점 전의 잘 스케줄링된 구간을 지워버리고, 그 시점 뒤만 생각하도록 한다. 이런 식으로 귀납적으로 나아가면, 결국 Forbidden Region 안에 있지 않았으나 할 일이 없어서 아무 일도 안한 경우는 존재하지 않는다. 이제 두 가지 케이스를 살펴본다.

- Case 1. i 이전에 스케줄링된 모든 작업 j가 $d_j \leq d_i$ 를 만족한다. 이 경우 해당 구간은 모두 Forbidden Region에 가려져 있거나 작업이 진행되고 있다. 고로 $[min_j(r_j), d_i]$ 구간에 대해서 Lemma 1에 의해 가정에 모순이다.

- Case 2. i 이전에 스케줄링된 작업 중 $d_j > d_i$ 를 만족하는 작업이 존재한다. 이 중 가장 최근에 처리된 작업을 j라고 하자. j와 i 사이에 있던 작업들은 d_i 이하의 데드라인을 가지며, Release time이 모두 j를 처리한 이후이다. (그렇지 않다면, j를 처리할 때 사용 가능한 작업이었기 때문에, j를 뽑는 선택이 데드라인이 제일 촉박했다는 것에 대해 모순이다.) 사이에 있던 작업 중 Release time이 최소인 작업을 k 라고 하자. 그렇다면, $[r_k, d_i]$ 라는 구간에 대해서 Lemma 1에 의해 가정에 모순이다.

□

Theorem 1. 위 알고리즘은 정당하다.

Proof. 위 알고리즘은 스케줄링 가능한 작업들을 불가능하다 하지 않는다. Forbidden Region이 0일 때 불가능하다 한 구간이 있으면 자명하고, 그렇지 않을 때는 Forbidden Region의 길이에 대한 귀납법으로 보일 수 있다. 위 알고리즘은 스케줄링 불가능한 작업들을 가능하다 하지 않는다. 이는 Lemma 2에 의해 자명하다. □

4 $O(n^2)$ 알고리즘

앞서 서술한 알고리즘은 이해의 편의를 돕기 위해서 굉장히 비효율적인 형태로 적혀 있으며, 다항 시간 알고리즘도 아니다. 하지만 이 알고리즘을 $O(n^3)$의 다항시간 알고리즘으로 바꾸는 것은 어렵지 않다.

먼저, Forbidden Region이 많아야 $O(n)$ 개의 서로 다른 구간으로 이루어져 있다는 것을 관찰하자. 각각의 케이스에서 새롭게 만드는 Forbidden Region은 모두 끝점을 r_i로 가진다. 즉, Forbidden Region을 이루는 집합에서 서로 다른 끝점이 많아야 n개라는 것이다. 끝점이 같은 여러 개의 구간이 있다면 그 중에서 가장 길이가 긴 것만 알면 되니, 결국 $O(n)$ 개 이상의 구간을 관리할 필요가 없다.

또한, 어떠한 $[r_i, *]$ 형태의 구간에서 위 과정을 진행할 경우, 만들어내는 Forbidden Region은 항상 해당 구간의 왼쪽에 존재한다. 즉, $[r_i, *]$ 형태의 구간에게 있어서 중요

한 Forbidden Region은 $r_j > r_i$ 인 구간들만 만들어 낼 수 있다는 것이다. 그렇다면, 구간의 후보들을 볼 때 r의 감소순으로 보면 한번 처리했던 후보를 다시 처리할 필요가 완전히 사라진다. 이렇게 될 경우 하나의 구간을 처리하는 데 $O(n)$ 이고, 많아야 $O(n^2)$ 번만 위의 과정을 시도하면 되기 때문에, 시간 복잡도가 $O(n^3)$이 된다.

여기서 중복 계산을 포착해서 없애는 것으로 계산량을 줄일 수 있다. 편의상 모든 구간들이 r_i의 증가순으로 정렬되어 있다고 하자. $f(i,j)$ 를 $[r_i, d_j]$ 구간에서 해당 알고리즘을 돌린 후 마지막으로 처리한 구간의 시작점 위치라고 하자. $f(i,j)$ 를 그때 그때 계산하는 것이 앞서의 방법이었으나, $f(i+1,j)$ 를 알면, $f(i,j)$를 상수 시간 안에 유도해 낼 수 있다. 이 관찰을 통해서 시간 복잡도는 $O(n^2)$으로 줄어든다.

5 References

1. P.Brucker, Scheduling Algorithms http://www.math.nsc.ru/LBRT/k5/Scheduling/BruckerSchedulingAlgorithms_Full.pdf

2. "Scheduling Unit-time Tasks With Arbitrary Release Times and Deadlines" by Garey, Johnson, Simons, and Tarjan (SICOMP, 1981) https://epubs.siam.org/doi/abs/10.1137/0210018

3. ACM ICPC World Finals 2017 Solution sketches http://www.csc.kth.se/~austrin/icpc/finals2017solutions.pdf