계산복잡도
Computational Complexity

크리스토스 H. 파파디미트리우
Chistos H. Papadimitiriou
캘리포니아 대학교 샌디에이고

번역 김태원
2023년 8월 12일

차 례

제 I 편

알고리즘

알고리즘 책은 계산복잡도를 다루는 장으로 끝나기 마련이니, 알고리즘에 관한 기본 사실 몇 가지 돌이키며 본고를 시작하는 편이 적절하겠다. 이어지는 세 장에서 우리의 목표는 간단하지만 중요한 요점을 조금 지적하는 것이다. 바로 계산*computational* 문제란 해결되어야 하는 것일 뿐만 아니라, 탐구할 가치가 있는 객체이기도 하다는 점이다. 문제와 알고리즘은 수학적으로 형식화되고 분석될 수 있다. 차례로 이를테면 언어*languages*나 튜링장치*Turing machines*가 그렇다. 그리고 정확한 형식주의는 별 상관 없다. 다항 시간 계산가능성*Polynomial-time computability*은 계산 문제에서 바라는 성질로, 실용적인 해결가능성*practical solvability*의 직관적인 개념과 동질이다. 여러 상이한 계산 모델*models*은 효율성의 다항 손실*polynomial loss*로 또 다른 모델을 시뮬레이션할 수 있다—비결정론*nondeterminism*이라는 예외, 즉 제 시뮬레이션에 지수시간*exponential time*을 요구하는 것으로 보이는 예외를 제외하면 말이다. 그리고 알고리즘을 아예 지니지 않는 문제가 존재하는데, 아무리 비효율적인 것조차 지니지 않는다.

제 1 장

문제와 알고리즘

알고리즘은 문제를 풀기 위한 자세한 스텝별*step-by-step* 방법론이다. 다만 문제*problem*란 뭔가*?* 우리는 이 장에서 중요한 예시 세 개를 소개한다

1.1 그래프 도달가능성

그래프 $G = (V, E)$는 노드nodes V와 선edges E, 즉 노드 쌍의 집합이다 (이를테면 그림 1.1을 보라. 우리의 그래프는 모두 유한하고 유향directed이겠다). 여러 계산 문제는 그래프를 다룬다. 그래프에 관해 가장 기본적인 문제는 이것이다. 그래프 G와 두 노드 $1, n \in V$가 주어질 때 1에서 n까지 경로path는 존재하는가? 우리는 이 문제를 도달가능성REACHABILITY[†]이라고 부른다. 가령, 그림 1.1에는 분명 노드 1에서 $n = 5$까지 경로, $(1, 4, 3, 5)$가 존재한다. 대신 만약 선 $(4, 3)$의 방향을 역전하면, 그런 경로는 존재하지 않는다.

흥미로운 문제가 대다수 그렇듯, 도달가능성은 가능한 일례*instances*의 무한 집합을 지닌다. 각 일례는 수학적인 객체 (우리의 경우, 그래프와 그 두 노드로), 곧 우리가 질문을 묻고 답을 기대하는 대상이다. 이때 질문이 속한 특정 종류가 문제를 특징짓는다. 도달가능성은 "예" 혹은 "아니오" 가운데 하나의 답안을 요구하는 질문이라는 점에 유의하라. 이런 문제는 결정문제*decision problems*라고 부른다. 복잡도 이론에서 보통 우리는 온갖 상이한 답안을 요구하는 문제보단 결정문제만 다루는 편이 편리하게 통합적이고 단순하다고 본다. 그러니 결정문제는 본고에서 중요한 역할을 맡겠다.

우리는 우리의 문제를 푸는 알고리즘에 관심이 있다. 다음 장에서 우리는 튜링장치*Turing machine*를 다루겠는데, 이는 임의의 알고리즘을 표현하기 위한 형식 모델formal

[†]복잡도 이론에서, 계산 문제는 단지 풀어야 하는 것일 뿐 아니라, 다만 그 자체로 흥미로운 수학적 객체이기도 하다. 문제가 수학적 객체로 다뤄질 때, 그 이름을 대문자로 표기하겠다. [역자: 대문자 표기를 드러냄표(`circemph`)로 대체한다.]

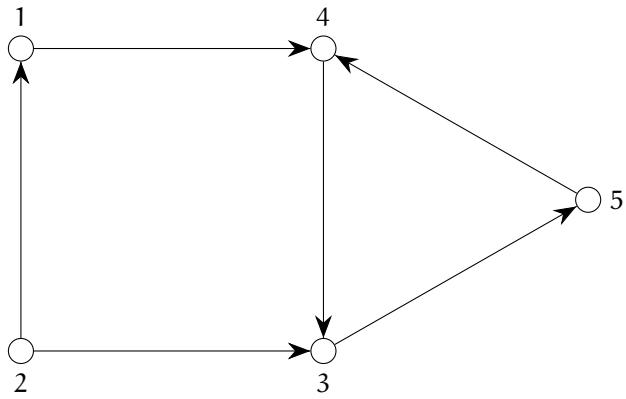

그림 1.1: 그래프.

model이다. 지금으로서는 우리의 알고리즘을 비형식적으로 기술하겠다. 가령 도달가능성은 이른바 **탐색 알고리즘***search algorithm*으로 풀 수 있다. 이 알고리즘은 이렇게 작동한다. 알고리즘에 걸쳐 우리는 노드의 집합을 지니는데, 이를 S라고 표기한다. 처음에는, $S = \{1\}$이다. 각 노드는 **표시되거나***marked* **표시되지 않는다***unmarked*. 노드 i가 표시된다는 말은 i가 과거 어느 지점 S에 있었다는 (혹은, 지금 S에 있다는) 뜻이다. 처음에는, 오직 1만 표시된다. 알고리즘의 각 반복iteration에서, 우리는 노드 $i \in S$를 선택해 S에서 제거한다. 우리는 그런 다음 i로부터 하나씩 모든 선 (i, j)를 처리한다. 노드 j가 표시되지 않았다면, 표시하고, S에 추가한다. 이 과정은 S가 빌 때까지 계속된다. 이 지점에서, 우리는 노드 n이 표시되었다면 "예", 표시되지 않았다면 "아니오"라고 답한다.

이 친숙한 알고리즘이 도달가능성을 해결한다는 사실은 분명할 것이다. 증명은 1에서 노드까지 경로가 존재하는 경우 그리고 오직 그 경우if and only if 노드가 표시된다고 확증할 것이다. 또 분명한 것은, 다만, 알고리즘에 대한 우리의 기술에는 중요한 디테일이 생략되었다는 사실이다. 가령, 그래프는 어떻게 알고리즘에 입력으로 표현되는가? 이런 적절한 표현은 우리가 사용하는 알고리즘의 특정 모델에 의존하기에, 이는 우리가 특정 모델을 지니기 전까지는 미룰 사안이다. 이 논의의 요점은 (2.2절을 보라) 바로 **정확한 표현이 별로 상관 없다는 것이다.** 그동안 그래프가 인접행렬adjacency matrix로 (그림 1.2로) 주어진다고 가정할 수 있는데, 이는 모든 요소가 랜덤 액세스random access 느낌으로 알고리즘에 의해 접근될 수 있는 행렬이다.[†]

알고리즘 자체에도 불분명한 지점이 있다. 원소 $i \in S$는 어떻게 S의 모든 원소 가운데 선택될 수 있는가? 이때 선택은 탐색 스타일에 막대한 영향을 미칠 것이다. 가령, 우리가 S에 가장 오래 머무른 노드를 항상 선택한다면, (달리 말해, 우리가 S를 큐queue

[†]실은, 7장에서 복잡도 이론에 대한 도달가능성의 중요한 쓰임새를 살필 예정인데, 거기서는 그래프가 암시적으로*implicitly* 주어진다. 즉, 입력데이터로 그 인접행렬의 각 요소를 계산할 수 있다.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

그림 1.2: 인접 행렬.

로 규현한다면) 탐색으로는 너비 우선*breadth-first*이 이어지고, 최소 경로가 확인될 것이다. (가장 최근에 추가된 노드를 선택하여) S가 스택stack으로 정비되었다면, 우리는 일종의 깊이 우선*depth-first* 탐색을 지니게 된다. S를 간수하는 다른 방법들은 완전히 상이한 종류의 탐색으로 이어질 것이다. 그러나 알고리즘은 이들 선택 모두에 대해 올바로 작동한다.

뿐만 아니라, 효율적으로 작동한다. 그렇다고 확인하기 위해, 인접행렬의 각 요소는 단 한 번, 행에 대응하는 꼭지점이 선택될 때 방문된다는 점에 유의하라. 그리하여, 우리는 선택된 노드로부터 선을 처리하며 대략 n^2 연산을 사용한다 (결국, 그래프에는 최대 n^2개의 선이 있을 따름이다). 요구되는 다른 간단한 (집합 S에서 원소 선택하기, 꼭지점 표시하기, 꼭지점이 표시됐는지 전하기 같은) 연산은 각각 어떻게든 상수시간constant time에 끝날 수 있으니, 우리는 탐색알고리즘이 n개의 노드를 지닌 그래프 상의 두 노드가 연결되느냐 마느냐 하는 결정을 최대 n^2에 비례하는 시간 안에, 혹은 $\mathcal{O}(n^2)$ 안에 내린다고 결론지을 수 있다.

막 사용한 $\mathcal{O}$ 표기와 그 동족들은 복잡도 이론에서 아주 유용하니, 간략한 삽입구를 열어 이들을 형식적으로 정의한다.

정의 1.1. 우리는 $\mathbb{N}$으로 모든 음이 아닌 정수의 집합을 표기한다. 복잡도 이론에서 우리는 $\mathbb{N}$에서 $\mathbb{N}$으로 가는 함수를 다루는데, 이를테면 $n^2, 2^n, n^3 - 2n + 5$가 그렇다. 우리는 글자 n을 그런 함수의 표준 인수arguemnt로 사용하겠다. 정수가 아닌 것들이나 음의 값—가령 $\sqrt{n}, \log n, \sqrt{n} - 4\log^2 n$—을 용인하는 방식으로 우리의 함수를 표기할 때라도, 우리는 언제나 그 값을 음이 아닌 정수로 생각할 것이다. 즉, 이들 예제처럼, $f(n)$으로 표기되는 모든 함수가 우리에게 정말 뜻하는 바는 $\max\{\lceil f(n) \rceil, 0\}$이다.

그리하여, f와 g는 $\mathbb{N}$에서 $\mathbb{N}$으로 가는 함수라고 하자. 우리는 양의 정수 c와 n_0이 존재하여 모든 $n \geq n_0$에 대해 $f(n) \leq c \cdot g(n)$인 경우 $f(n) = \mathcal{O}(g(n))$이라고 쓴다 ("$f(n)$은 $g(n)$의 빅오big oh다", 혹은 "f는 g의 차수order다"라고 발음한다). $f(n) = \mathcal{O}(g(n))$은 f가 g만큼 자라거나 더 느리다는 것을 비형식적으로 뜻한다. 그 반대가 일어나면, 즉, $g(n) = \mathcal{O}(f(n))$이라면, 우리는 $f(n) = \Omega(g(n))$이라고 쓴다. 끝으로, $f(n) = \theta(g(n))$은 $f(n) = \mathcal{O}(g(n))$이고 $f(n) = \Omega(g(n))$임을 뜻한다. 후자는 f와 g가 정확히 똑같은 증가율*rate of growth*를 지닌다는 것을 뜻한다.

보이기 쉬운 예를 들면,수학 $p(n)$이 계수degree d의 다항식인 경우, $p(n) = \theta(n^d)$다. 곧, 다항식의 증가율은 다항식의 첫 0이 아닌 항non-zero term으로 포착된다. 이어지는 바는 아마 복잡도 이론에 대한 함수의 증가에 관해 가장 중요하고 유용한 사실이다. $c > 1$가 정수이고 $p(n)$이 모든 다항식이라면, $p(n) = \mathcal{O}(c^n)$이지만, 이는 $p(n) = \Omega(c^n)$인 경우가 아니다. 곧, 모든 다항식은 모든 지수식보다 엄격하게*strictly* 느리게 자란다 (증명으로는 문제 1.4.9를 보라). 지수가 하나 낮아지면, 같은 성질은 $\log n = \mathcal{O}(n)$이라는 것을 암시한다 —실은, 모든 k승에 대해 $\log^k n = \mathcal{O}(n)$이라는 것을 암시한다. □

1.1.1 다항시간 알고리즘

도달가능성에 대한 예측 $\mathcal{O}(n^2)$으로 돌아오자. 이 간단한 문제가 이 간단한 알고리즘으로 만족스럽게 풀릴 수 있다는 사실은 놀랍지 않을 테다. 사실 $\mathcal{O}(n^2)$라는 예측에는 여기서 중요하게 고려하지 않는 방식이더라도 비관적인 면이 있다. 문제 1.4.3을 보라. 다만 이렇게 만족스럽다고 느끼는 원천의 정체를 밝히는 일은 중요하다. 정체은 바로 증가의 비율 $\mathcal{O}(n^2)$이다. 본고는 이런 다항*polynomial* 증가율을 적절한 시간 제한으로 받아들일 계획이다. 이에 대조되는 2^n 같은 지수 증가율이나 심지어는 $n!$이 우려의 원인일 것이다. 이런 우려가 이어져, 알고리즘 넘어 알고리즘이 필요할 수도 있다. 이 경우 우리는 다항시간 안에 문제를 풀지 못했다고 보며, 이런 판단을 수중의 문제가 어렵다는*intractable* 증거로 삼는다. 다시 말해, 실상 효율적인 해로 처리할 수 없는 문제로 본다. 바로 그때 본고의 방법론이 행차한다.

다항시간유계bound와 비다항시간유계 간의 이분법, 그리고 "실질적으로 실현 가능한 계산"이라는 직관적인 개념을 통한 다항알고리즘의 식별에는 논란의 여지가 없지 않다. 다항이 아니지만 효율적인 계산은 존재하며, 실무에는 효율적이지 않은 다항계산도 존재[†]한다. 이를테면, n^{80} 알고리즘의 실용적인 가치는 아마 제한될 것이다. 그리고 $2^{\frac{n}{100}}$처럼 지수적인 (혹은, 더 흥미로운 $n^{\log n}$처럼 하위지수적인subexponential) 증가율을 보이는 알고리즘이 훨씬 더 쓸 만할 것이다.

다만 다항 패러다임을 옹호하는 강력한 논변들도 존재한다. 우선 모든 다항 비율은 결국 모든 지수 비율이 능가할 것이고, 후자는 문제의 일례들로 구성된 유한 집합이 아니라면 선호될 것이다. 물론 이 유한 집합은 실무에서 나타나기 쉬운 모든 일례나 우리 우주의 테두리에서 존재할 수 있는 것들을 포함할 수도 있다... 더 중요한 점은 우리의

[†]실은 각 예외에 예제를 제공하는 중요한 문제가 있다. 바로 선형계획*Linear Programming*이다. (논의와 레퍼런스를 위해서는 9.5.34절을 보라) 이 기본적인 문제를 위해 널리 쓰이는 고전적인 알고리즘은 심플렉스 방법*simplex method*인데, 최악의 경우 지수시간이라고 알려져 있지만 실무에서는 일관되게 최상의 성능을 보이고, 사실 기대 성능은 다항시간이라고 증명할 수 있다. 이와 대조적으로 이 문제를 풀기 위해 발견된 최초의 다항알고리즘은 타원체*ellipsoid* 알고리즘인데, 쓸 수 없을 정도로 느리다. 다만 선형계획론의 이야기는 실상 계산복잡도 이론의 방법론에 대한 반박이 아니라 절묘한 논변이 되어줄 수 있다. 물론 문제가 실용적인 알고리즘을 지니면 다항시간에 풀 수 있다는 것은 분명하다. 그런데도 다항시간알고리즘과 경험적으로 좋은 알고리즘이 필연적으로 일치하지는 않는다.

알고리즘 경험에 따르면 n^{80}이나 $2^{\frac{n}{100}}$처럼 극단적인 증가가 실무에서 거의 나타나지 않는다는 관찰이다. 다항알고리즘은 보통 작은 지수를 지니며 합리적인 곱셈 상수를 지니고, 지수알고리즘은 대개 정말로 비실용적이다.

우리의 관점에 또 다른 비판이 있을 수 있는데, 바로 계산복잡도 이론은 가장 기피하고 싶은 상황 속 알고리즘 성능만 측정할 뿐이라는 것이다. 알고리즘의 지수적인 최악 성능은 통계적으로 사소한 수준의 입력에서 비롯한 것일 수 있으나, 평균적으로는 만족할 만큼의 성능을 보일 수 있다. 물론 최악의 경우에 반해 예상된 알고리즘 행태에 대한 분석이 더 유익할 수도 있다. 불행히도, 실무에서는 문제의 **입력 분포***input distribution*를 거의 알 수 없다. 이는 있을 수 있는 각각의 실례 가운데 무엇이 알고리즘의 입력으로 나타나냐는 확률이다. 따라서 실로 유익한 평균의 경우에 대한 분석은 불가능하다. 더군다나 단지 특정 일례 하나만 풀 생각이고 알고리즘은 지독한 성능을 보일 때면, 우리가 통계적으로 사소한 예외와 마주친 적 있다는 기억은 작은 도움이나 위로가 되리다.