**REPORT**

로고이(가) 표시된 사진

자동 생성된 설명

**과목명 : 자료구조 심화학습**

**과제내용 : 1. Greedy algorithm의 개념**

**2. Dijkstra 최단경로 알고리즘과 비교분석**

**3. Greedy algorithm 알고리즘 예시**

**교수명 : 심종익 교수님**

**학 년 : 3학년**

**학 번 : 201901366**

**성 명 : 서희준**

**제출일 : 23.08.07.(일)**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**<목차>**

1. **Greedy algorithm 개념**
2. **비교분석 (Dijkstra 최단 경로, Greedy)**
3. **Greedy algorithm 예제 프로그램**
4. **Greedy algorithm 의 개념**

탐욕 알고리즘은 현재 상황에서 가장 좋은 것만 선택하는 방법으로 문제를 해결하는 알고리즘이다. 하지만, 현재의 선택들이 나중에 어떤 결과를 미칠지 고려하지 않는 특징이 있다. 즉, 인과 관계를 따지며 전체적 흐름으로 최적의 선택을 도출하는 방법과 다르다. 여러가지 경우의 수 중 하나를 택해야 할 때 지역적으로 최적인 선택을 한다. 그 선택들로, 전체 문제에 대한 전역적 해답을 구한다. 이때 주의해야 할 것은 지역적으로 선택한 최적의 답이 전역적으로 최적인 답이 아닐 수 있다는 것이다. 따라서, greedy 알고리즘이 적용되는 조건은 한정적이다. 지역적으로 최적인 해들이 전역적인 해답으로 이어질 수 있는 때 탐욕 알고리즘을 적용할 수 있다. 이러한 이유로 탐욕 알고리즘은 최적해를 구하는데 사용되는 근사적인 방법으로 분류된다.

탐욕 알고리즘은 문제를 더 작은 하위 문제들로 분할하고, 각 하위 문제의 해를 구해 전체 문제를 해결하는 경우에 유용하다. 하지만, 각 단계에서의 선택에서 미래의 단계의 결과를 고려하지 않는다는 것에서 동적 계획법과 차이점을 보인다. 이러한 특징은 탐욕 알고리즘이 적용되는 범위를 한정적으로 만들지만, 현재만을 고려한다는 점에서 시간적 이점을 보인다. 각 단계에서의 최적의 해가 결과적으로 최적의 해가 된다는 보장은 없다. 그럼에도 불구하고, 근사 알고리즘으로 사용할 수 있고, 대부분의 경우 계산속도가 빠르기 때문에 문제 해결에 실용적이다. 이러한 이유로, greedy 알고리즘을 적용할 수 있는 문제의 구조를 분석하는 것이 중요하다. 그 구조를 매트로이드라고 한다.

예시를 들어 탐욕 알고리즘이 적용되는 구조와 아닌 구조를 설명한다. 이 알고리즘을 사용하여 해결할 수 있는 대표적인 예시로는 "동전 거스름돈" 문제가 있다. 문제는 주어진 금액을 가장 적은 동전 개수로 할당하는 것이다. 할당해야 할 금액이 25센트이고 사용 가능한 동전이 1센트, 5센트, 10센트가 있다. 여기서 탐욕 알고리즘을 적용하면, 각 단계에서 가장 큰 동전을 선택한다. 10센트 동전을 두 번 선택하고 마지막으로 5센트 동전을 선택하여 총 세 개의 동전을 사용한다. 이러한 경우 각 단계의 최적의 선택(가장 큰 동전)이 전체의 최적해로 이어질 수 있다. 하지만, 항상 최적의 해가 될 수는 없다. 사용 가능한 동전이 1센트, 3센트, 4센트이고, 할당해야 할 금액이 6센트인 경우, 탐욕 알고리즘을 적용하면, 4센트 동전 하나와 1센트 동전 두 개를 선택한다. 하지만 전역적 최적해는 3센트 동전 두 개를 사용하는 것이다. 이처럼 특정한 경우에만 이 알고리즘이 유효하다는 것을 알 수 있다.

이 알고리즘은 2가지 조건을 만족해야 문제에 적용할 수 있다. 첫번째로 탐욕 선택 속성이다. 이전 단계의 선택이 이후 단계에 영향을 주지 않는 문제를 말한다. 두번째는 최적 부분 구조이다. 앞에서 설명한 것과 같이 하위 부분 문제의 최적해가 전체에서도 최적해가 될 수 있는 문제를 말한다. 하위 부분 문제의 해로 최종 해를 구한다는 것이 동적 프로그래밍과 유사하지만, 동적 계획법의 경우 현재 단계에서 선택할 때 이전 단계에 영향을 받기 때문에 첫번째 조건을 포함하지 않는다.

|  |  |  |
| --- | --- | --- |
| **특징** | **Greedy Algorithm (탐욕 알고리즘)** | **Dynamic Programming (동적 계획법)** |
| **실행 가능성 (Feasibility)** | **순간적으로 가장 최선의 선택을 하며, 전역 최적해로 이끌어낼 것을 기대함** | **각 단계에서 현재 문제와 이전에 해결된 부분 문제의 최적해를 고려하여 최적해를 계산함** |
| **최적성 (Optimality)** | **최적해를 보장하지 않을 수 있음** | **최적해를 보장함** |
| **재귀 (Recursion)** | **각 단계에서 지역적으로 최선의 선택을 따름** | **재귀적인 공식을 사용하여 이전에 계산된 상태들을 이용함** |
| **메모이제이션 (Memoization)** | **이전 선택을 다시 검토하거나 수정하지 않으므로 메모리 측면에서 효율적임** | **메모이제이션을 위해 동적 계획법 테이블을 필요로 하며 메모리 사용량이 증가함** |
| **시간 복잡도 (Time Complexity)** | **일반적으로 Greedy 메소드가 더 빠름, 예를 들어 Dijkstra의 최단 경로 알고리즘은 O(ELogV + VLogV) 시간이 걸림** | **일반적으로 동적 계획법이 더 느림, 예를 들어 Bellman Ford 알고리즘은 O(VE) 시간이 걸림** |
| **방식 (Fashion)** | **순차적으로 앞으로 진행하며 선택을 하며 이전 선택을 검토하거나 수정하지 않음** | **바텀 업 또는 탑 다운 방식으로 작동하여 더 작은 최적 부분해들로부터 해결해 나감** |

1. **비교분석 (dijkstra 알고리즘 : priority queue | greedy**

다익스트라 알고리즘은 모두 최단 경로를 찾는 알고리즘이다. 다익스트라 알고리즘은 하나의 시작 정점에서 다른 모든 정점까지의 최단 경로를 찾는 알고리즘이다. 방문하지 않은 정점들 중에서 가장 가까운 정점을 선택하여 해당 정점을 방문하고, 해당 정점으로부터 인접한 정점들의 거리를 갱신하는 방식으로 해결한다. 가장 일반적인 방식은 배열과 반복문을 이용하여 최단 경로를 구한다. 다익스트라 알고리즘은 음의 가중치가 없는 경우에만 동작한다. 음의 가중치를 고려하는 경우 추가 알고리즘이 필요하다.

그리디 다익스트라 알고리즘은 다익스트라 알고리즘의 일종으로, 최소 힙(min heap)을 사용하여 구현한다. 가장 작은 가중치가 계속 루트에 위치하게 만들면서, 그리디 알고리즘을 적용할 수 있다. 각 단계에서의 최적의 선택이 곧 전역적 최적의 해가 된다. 매 단계에서 최소 힙에서 가장 가까운 정점을 선택하고, 해당 정점을 방문한 뒤 인접한 정점들의 거리를 최신화하는 것은 동일하다.

주된 차이점은 데이터 구조에서 나타나다. 다익스트라 문제에 그리디 알고리즘을 적용할 수 있는 데이터 구조가 필요하다. 그리디 알고리즘을 적용한다고 적용하더라도 전역적 최적해를 얻지 못할 수 있기 때문에 적용할 수 있는 조건을 만드는 것이다. 최소힙을 사용하여 가중치가 제일 작은 vertex가 항상 root에 위치하게 한다. 이로써, 항상 루트를 선택하더라도, 항상 작은 가중치를 가진 vertex가 선택될 것이다. 또 다른 차이점은 효율성에 있다. 어떤 알고리즘이 더 좋은지는 상황에 따라 다를 수 있다.

우선순위 큐를 사용하지 않는 기본적인 다익스트라 알고리즘은 시간 복잡도가 O(V^2)이다. V는 정점의 수이다. 정점의 개수가 적을 때(특히 V^2 보다 작을 때), 그리디 다익스트라 알고리즘보다 일반적이 다익스트라 알고리즘이 더 효율적이다. 최소 힙 자료구조로 구현한 그리디 다익스트라 알고리즘은 시간 복잡도가 O(E log V)이다. 여기서 E는 간선의 수이다. 간선의 개수가 많고, 그래프가 밀집된 경우, 그리디 다익스트라 알고리즘은 일반적으로 다익스트라 알고리즘보다 효율적이다. 그 차이는 밀집된 그래프일수록 더 커진다.

따라서 선택할 알고리즘은 그래프의 크기와 구조, 그리고 입력 데이터의 특성에 따라 다르게 알고리즘을 다르게 설정할 수 있다. 일반적으로 그리디 다익스트라 알고리즘이 다익스트라 알고리즘보다 더 선호되며, 특히 간선의 수가 많을 때 더 효율적이다. 그러나 실제 문제에서는 알고리즘의 효율성 외에도 다른 요소들도 고려되어야 한다.

다음은 dijkstra 알고리즘을 그리디 방식으로 적용한 프로그램이다.

1. **Dijkstra \_ greedy**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

먼저 최소 힙 구조를 만들기 위해 구조체를 사용자 정의형으로 선언한다. GraphType은 그래프의 정보를 담는 구조체이다. n은 그래프의 정점 수를 의미하고, weight는 각 정점들 간의 가중치를 저장하는 인접 행렬이다. HeapNode는 우선순위 큐를 위한 요소로 사용되는 구조체로, 정점과 해당 정점까지의 가중치를 저장한다. PriorityQueue는 최소 힙을 구현한 우선순위 큐로, HeapNode 요소들을 저장하고 관리한다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

이 프로그램은 최소힙을 관리하기 위한 함수가 요구된다. 따라서, 위의 그림과 같이 우선순위 큐를 초기화하고, 힙에 삽입 및 제거 (최소 힙 구조화 포함), 힙이 비어있는지 확인하는 함수가 있다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

init\_priority\_queue는 우선순위 큐를 초기화하는 함수로, 우선순위 큐의 크기를 0으로 설정한다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

is\_empty함수는 우선순위 큐가 비어있는지 확인한다. 비어 있을 때 참을 반환한다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

enqueue: 우선순위 큐에 요소를 삽입하는 함수로, 새 요소를 적절한 위치에 삽입하여 최소 힙의 구조를 유지한다. 따라서 삽입될 때마다 최소 힙 형태가 유지되도록 아래에서 위로 올라가며, 들어갈 수 있는 위치에 노드를 삽입한다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

dequeue: 우선순위 큐에서 최소 가중치를 가진 노드를 제거하여 반환한다. 이때 루트가 되는 노드가 최소 힙에서 항상 최소 가중치이기 때문에 매번 루트를 제거한다. 루트를 제거할 때 힙의 크기를 하나 줄이며, 가장 마지막 리프 노드를 루트로써, 비교하며 버블 다운 방식으로 들어갈 위치를 재조정한다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

dijkstra\_shortest\_path 함수는 주어진 그래프와 시작 정점을 입력으로 받아서, 해당 시작 정점으로부터 모든 정점까지의 최단 경로를 계산한다. 이 함수는 앞에서 정의한 우선순위 큐를 사용하여 최단 경로를 구한다. 따라서, 기본 큐를 초기화하는 부분과 가중치를 weight[o][o]을 기준으로 weight[o][i]로 초기화한다. Line 150과 같이 출발점은 이미 방문한 것으로 표시하고, 자기 자신을 가리키는 루프간선의 가중치는 0으로 초기화한다. (0)을 기준으로 초기화한 간선의 가중치로 최소힙을 생성한다. (우선 순위 큐 생성- line 156)

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

다음은 다익스트라에 핵심이 되는 부분이다. While()의 조건문에 힙이 비었는지 확인하는 코드를 호출해 힙이 빌때까지 (더 이상의 최신화할 수 있는 최단 경로가 없을 때) 최단 경로를 최신화한다. Line 102에서 가중치가 가장 작은 정점을 힙으로부터 꺼낸다. 이때 u는 최소힙의 정점이다. 예를 들어 아래와 같은 그래프가 있다고 가정하자.

텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

여기서 \*는 연결되지 않는 간선이다. 그래프를 인접 행렬로 표현한 것이다. 예를 들어 정점 (0,1)에서의 간선 E(7)이다. 각 위치에 있는 값들은 간선의 가중치이다. 현재 이 프로그램에서는 E( (0,1), (0,2), (0,3), (0,4), (0,5), (0,6) )의 최소 경로를 찾으려고 하는 것이다. E(0,2)은 연결되어 있지 않은 간선이다. 따라서 정점 2로 가기 위해서 다른 정점을 거쳐 가는 방법을 고려할 수 있다. 연결된 간선들도 마찬가지이다. 만약 E(0,1) = 7의 가중치보다 작은 경로가 있다면, <0,4> <4,1>의 경로를 택할 수 있는 것이다. 이랬을 때 경로의 가중치는 E<0,4> = 3 ,<4,1> = 2로 5라는 가중치이다. 이는 <0,1> =7보다 짧은 경로이기 때문에 다른 정점을 거쳐가는 것이 더 단거리인 경우이다. 이러한 과정은 정점 0 에 대한 가중치를 최소 힙으로 구조화했을 때 더 간단해진다. 처음 구조화 했을 때 저장되는 가중치는 E( (0,1), (0,2), (0,3), (0,4), (0,5), (0,6) )로 우회가 아닌 직접 연결에 대한 초기값이다. 따라서 힙에는 0, 7, 100000, 100000, 3, 10, 100000 이 저장되고(\*은 연결되지 않은 간선으로 100000라는 값을 줘 무조건 우회하는 선택하는 하도록 한다.), 정점 0이 루트를 차지한다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

알고리즘의 과정을 파악하기 위해 출력문을 추가하였다. 앞서, 설명한 것과 같이 루트에는 정점 0이 저장되었다가, line 102에서 u에 저장된다. 접근한 정점은 방문한 것으로 표시하고, 이후 line 112에서 최단 경로를 탐색할 때, 방문한 노드로 구분한다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

처음 반복문에서 실행되는 부분이다. 이와 같이 원래의 가중치보다 더 작은 경로를 찾기 위해 정점을 모두 탐색한다.

**“if (!visited[w] && distance[u] + g->weight[u][w] < distance[w])”** 탐색한다. 여기서의 비교는 0과 인접한 정점에 대한 비교이다. 이전에 초기화된 값과 같으므로, 현재 가중치보다 작은 가중치가 없는 것이 당연하다. If문 안으로 들어오게 된다면, 현재 경로보다 더 짧은 경로를 찾은 것으로, 경로에 대한 가중치를 최신화해야 한다. 따라서, line 118과 같이 새로운 가중치를 최소 힙에 삽입하고, 최소 힙을 다시 형성한다. 만약 최단 경로를 찾지 못했다면, 최소 힙의 노드는 점차 줄어들 것이고, 모든 탐색이 끝난 것으로 while반복문이 종료된다.

현재 0은 원래 초기값과 비교하는 것으로, 최단 경로를 찾지 못했고, 다시 최소 힙에서 루트 값을 가져온다.

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

두번째로 작은 가중치는 정점 4이다. **텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명** 정점 4의 가중치는 3으로, <0,3> 으로 이동한 뒤 <3,?>으로 이동이 가중치 7, \* 10보다 작을 가능성이 매우 크다. 작은 가중치를 가지는 간선을 통해 우회하는 선택을 하는 것이다. 이러한 선택을 할때 이동 간선이 짧아야 원래 경로비용보다 적은 값을 가질 확률이 올라간다. 따라서 최소 힙을 가중치에 대해 구조화한 것이다. 정점 0에서 정점 4를 거쳐서 정점1로 가는 것이 5로, 원래 비용 7보다 작은 것을 알 수 있다. 그럼 정점0에서 정점1의 거리는 7이 아닌 5가 될 수 있다. 아래와 같이 최신화한다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

마찬가지로, 반복하면 정점 0에서 정점 4를 거쳐 3으로 가는 것이 지금으로써 최단 경로라는 것을 쉽게 알 수 있다. 다시 생각해보면 **텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명** 정점 0에서 정점 3으로 가는 간선은 존재하지 않았다. 따라서, 정점 4에서 정점 3으로 가는 경로가 현재 최단 경로라는 것을 쉽게 알 수 있다. 다시 최단 경로를 최신화하고, 최소 힙에 삽입한다. 이때 삽입되는 이유는 이보다 더 짧은 경로가 있는지 확인하기 위함이다. 진행되는 과정은 무조건 최소 힙의 루트에서 값을 가져와 가장 높은 가능성이 있는 정점을 거쳐서 새로운 경로를 찾는 것이다. 이때 선택은 greedy 알고리즘이 적용된 것이다. 가장 큰 가능성이 있는 것을 각 단계에서 선택하지만, 그것이 최종적으로 최적의 선택인지는 모호하다. 따라서, 한번 더 최소힙에 넣고, 확인하는 것이다. 이런식으로 반복하면 아래와 같은 결과가 출력된다.

**<결과>**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

**정점 0에서 각 정점으로의 최단 경로거리를 나타낸 것이다.**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

main함수에서는 그래프를 인접 행렬로 표현한 뒤, 그래프를 출력한다. 그리고 시작 정점을 0으로 하여 다익스트라 알고리즘을 실행하여 각 정점까지의 최단 경로를 출력한다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

연결되지 않은 간선은 INF로 저장했기 때문에 가중치가 아닌 \*로 출력되도록 하였다.

결론적으로 출력된 값을 보면, 최단 경로는 다음과 같다.

Node 0에서 다른 노드까지의 최단 경로: 노드 0:

최단 거리 = 0 -> 노드에서 자기 자신까지의 거리는 항상 0이기 때문이다.

노드 1: 최단 거리 = 5 -> 0에서 Node 1까지의 최단 경로는 0 -> 4 -> 1이며, 총 가중치는 5이다.

노드 2: 최단 거리 = 9 -> Node 0에서 Node 2까지의 최단 경로는 0 -> 4 -> 1 -> 2이며, 총 가중치는 9이다.

노드 3: 최단 거리 = 11 -> Node 0에서 Node 3까지의 최단 경로는 0 -> 4 -> 1 -> 2 -> 3이며, 총 가중치는 11이다.

노드 4: 최단 거리 = 3 -> Node 0에서 Node 4까지의 최단 경로는 0 -> 4이며, 총 가중치는 3이다.

노드 5: 최단 거리 = 10 -> Node 0에서 Node 5까지의 최단 경로는 0 -> 4 -> 1 -> 5이며, 총 가중치는 10이다.

노드 6: 최단 거리 = 8 -> Node 0에서 Node 6까지의 최단 경로는 0 -> 4 -> 6이며, 총 가중치는 8이다.

1. **Dijkstra \_ normal**

일반적은 dijkstra 알고리즘은 배열과 반복문을 이용해 처리한다. 밀집된 그래프가 아닌 경우 일반적인 방법으로 처리하는 것이 효율적이다.

**텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

함수와 데이터 구조가 훨씬 간단해진 것을 확인 할 수 있다. 여기서 간선의 최소 가중치를 찾는 함수는 findMin()함수이다. 호출될 때마다 정점의 수만큼 탐색을 실행한다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

findMin함수를 살펴보면, 가장 작은 가중치를 찾기 위해 n만큼 반복하는 것을 확인할 수 있다. 여기서 n은 정점의 총 개수로, **dijkstra\_shortest\_path()** 함수에서 실인자로 전달된다. 각 정점에 대한 최소 가중치를 탐색할 때 노드 총 개수만큼 반복하며 탐색하기 때문에 시간 복잡도가 O(V^2)인 것이다. 아래 함수의 반복문에서 반복 횟수를 확인할 수 있다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

정점 0을 제외하고, 각 정점에서의 최소 가중치를 찾아 정점 0에서 각 정점으로 이어질 수 있는 최소 경로를 찾는다.

마찬가지로 preset과정이 필요하다. 처음 정점 0에서의 인접 정점들로 distance를 초기화하고, 모든 정점을 unvisited로 표시한다. 정점 0의 가중치를 0으로 초기화하고, 방문한 것으로 표시해 비교대상에서 제외한다. Line101 이하의 부분이 이 함수의 핵심 부분으로, 기본적인 알고리즘은 그리디 알고리즘을 적용한 dijkstra와 동일하다. 처음 정점 0에서 시작하여 최소 가중치로 이어지는 정점을 찾는다. 그 정점을 거쳐서 이동하는 경로가 원래의 직접적인 경로보다 작을 때(line 110), 최소경로로 경로값을 최신화(line 113)한다. 최신화된 값은 line 104에서 다시 최소 가중치 간선을 찾을 때 반영된다. 여전히 경로비용이 작다면, 그것을 거쳐서 이동하는 또 다른 경로가 발생할 가능성이 있다. 이러한 반복은 모든 정점에서의 최소 경로를 찾은 후 종료된다.

**<출력결과>**

**텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명**

**출력결과는 동일한 것을 확인할 수 있다.**

앞서 설명한 것과 같이 주된 차이점은 데이터 구조에서 나타난다. 우선순위 큐를 사용하지 않는 기본적인 다익스트라 알고리즘은 반복문을 사용하기 때문에 시간 복잡도가 O(V^2)이다. 정점의 개수가 적을 때(특히 V^2 보다 작을 때), 그리디 다익스트라 알고리즘보다 일반적이 다익스트라 알고리즘이 더 효율적일 수 있으나, 간선의 개수가 많아질수록 그리디 알고리즘을 적용한 다익스트라 알고리즘이 유리하다. 최소 힙 자료구조로 구현한 그리디 다익스트라 알고리즘은 시간 복잡도가 O(E log V)이기 때문이다.

1. **Greedy algorithm 예제 프로그램 : 작업 순서 스케줄링 ( Job sequencing )**

Greedy 알고리즘만으로 문제를 해결하기에는 고려해야 할 조건들이 까다롭다. 최적의 해를 찾기 위해서는 추가적인 정렬, 탐색, 조건문이 필요할 수 있다. 각 단계에서의 최고의 선택이 전체에서도 최고의 선택이 될 보장이 없기 때문에 문제를 분석하는 것이 중요하다. 특히 단계별 최고의 선택을 할 때 어떤 기준으로 최고를 결정할 것인지 고민해 볼 필요가 있다.

아래 프로그램은 탐욕 알고리즘을 적용한 예제 프로그램 중 작업 순서를 스케줄링 하는 문제이다. 주어진 작업들을 가장 효율적으로 처리하여 최대 이익을 얻는 것이 문제의 해이다. 각 작업은 ID, 마감 기한(dead line), 그리고 마감기간 내 처리 시 얻을 수 있는 이익이 있다. 작업소요 시간은 마감기한으로 판단한다. Deadline이 1이라고 한다면, 그 작업을 처리하는데 1이라는 시간이 소요된다. 각 작업의 마감기한을 비교해서, 가장 늦은 마감기한을 찾는다. 결론적으로 이 프로그램은 가장 늦은 마감기한을 기준으로 몇 개의 작업을 처리하는 것이 가장 큰 이익을 만들 수 있을지 찾는 문제이다.

알고리즘은 먼저 주어진 작업들의 마감 기한과 이익으로부터 최대 마감 기한 값을 찾는다. 이후 작업들을 이익에 따라 내림차순으로 정렬한다. 여기서 선택할 때 탐욕 알고리즘을 적용하여, 가장 높은 이익을 가진 작업들부터 선택하되, 최대 마감 기한을 초과하지 않도록 한다. 이러한 알고리즘은 최대 이익을 만드는 결과를 만든다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

위의 표와 같은 작업이 주어졌을 때, 최대 마감 기한은 4이다. 이후 이것을 이익에 따라 내림차순으로 정렬하면, 아래의 표와 같이 정렬된다.

**텍스트, 번호, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

최대 마감 기한이 4이기 때문에 모든 작업들은 4 이전에 종료되어야 한다. 가장 높은 이익을 가진 작업 J4를 선택해도 이는 최대 마감 기한을 초과하지 않기 때문에 작업 스케줄에 추가한다. 이때 작업 소요시간이 3이기 때문에 마감전까지 1의 작업을 추가로 할 수 있다. 따라서, 이후 J5, J2를 작업 스케줄에 포함하지 못한다. 따라서, 작업 소요시간이 1인 J3 작업을 선택해 스케줄에 추가한다. 이때, 총 이익은 100 + 40으로 140의 최대 이익을 얻을 수 있다. 아래는 위의 작업의 과정을 프로그램으로 만든 것이다.

**텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

구조체를 사용해 작업에 대한 자료들을 하나의 자료형으로 선언하였다. 이를 배열로 할당 받아, 각 작업에 대한 ID, deadline, 이익으로 초기화한다. 이후 이익을 기준으로 내림차순으로 정렬할 때 퀵 정렬을 이용한다. 따라서, SWAP이라는 함수를 매크로로 선언하였다. 동적 할당에 사용하는 MALLOC 매크로는 마감기간에 내 처리할 수 있는 작업들을 저장하는데 사용한다. 작업의 수만큼 동적으로 배열을 할당한다.

**텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

작업에 대한 id, 마간 기한, 이익을 초기화한다. 이후 퀵 정렬 함수를 호출해 작업들을 이익을 기준으로 내림차순으로 정렬한다. 이후 작업 스케줄링 함수를 호출하여, 최대 마감 기한을 찾고, 기한 내 처리할 수 있는 작업들을 선택한다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

위의 함수는 가장 늦은 마감 기한을 찾는 함수이다. 마감 기한을 찾고, 그 값을 함수로 반환한다.

**텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

앞서, 설명한 바와 같이, 가장 늦은 마감기한을 찾고, 그만큼의 공간을 동적으로 할당 받는다. result배열에는 작업 스케줄에 포함할 수 있는 작업들을 저장하기 위해 인덱스 값을 저장한다. Slot은 마감기한이 중복된 작업들을 비교에서 제외하기 위해서 사용한다. 예를 들어 최대 마감기한이 5이고, 작업 소요기간이 2인 작업이 2개가 있다고 가정하자. 이때, 작업 소요기간이 같은 작업들은 둘 중 더 큰 이익인 것만 선택한 뒤 다른 것은 고려하지 않는다. 따라서, 처음에 slot을 거짓으로 모두 초기화한 한 뒤, 작업 스케줄에 추가할 때 true로 변경한다.

반복문을 보면, 탐욕법을 사용해 가장 큰 이익이 있는 작업부터 선택하는 것을 확인할 수 있다. 이때, 마감기한이 작은 것부터 순서대로 저장하기 위해 “**int j = arr[i].dead - 1; “** 마감기한을 이용한다. 한 작업의 마감기한이 2라고 한다면, 2-1인 slot[1]에 그 값이 저장될 것이다. 이후 0부터 max까지의 배열에서 출력할 때, 저장된 값을 마감기한이 작은 순서부터 출력할 수 있다. 출력하는 함수는 아래와 같다. Result는 작업 스케줄에 추가한 작업의 인덱스 값을 저장하기 때문에 slot이 참이 되는 인덱스를 이용해 전체 작업들 중 스케줄에 추가된 작업들만 출력한다. 마지막에는 스케줄에 있는 작업의 이익을 모두 더해 출력한다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

마감기한이 넘지 않는 범위에서 작업들을 스케줄에 추가하기 위해, maxDeadline변수을 0으로 초기화하고, 이후 반복문에서 작업 소요시간이 넘지 않는 작업들만 스케줄에 추가한다. 스케줄에 추가할 때는 스케줄에 작업이 있다는 것을 추가하기 위해 “ maxDealine += arr[i].dead “ 처럼 처리한 작업만큼의 시간을 더한다.

**<결과>**

위 프로그램의 결과값은 다음과 같다.

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

위의 경우와 달리, 탐욕 알고리즘을 적용할 수 있는 문제들은 한정되어 있다. 지역적으로 최적인 선택이 전역적으로 매번 최적인 결과로 이어지지 않을 수 있다. 하지만, 최적에 근사값을 항상 도출해낸다. 미래의 결과를 고려하지 않는 측면에서 시간, 공간 복잡도가 좋지만, 적용할 수 있는 조건이 까다롭다는 단점이 있다. 따라서, 탐욕 알고리즘을 적용하기 위해 문제를 분석하고 적용 가능한지 판단하는 것이 중요하다.

**<참고자료(출처)>**

1. Introduction to Greedy Algorithm – Data Structures and Algorithm Tutorials . (08 May, 2023). Retrieved from https://www.geeksforgeeks.org/introduction-to-greedy-algorithm-data-structures-and-algorithm-tutorials/.
2. 알고리즘 - 그리디 알고리즘(Greedy Algorithm) . (2021). https://hongjw1938.tistory.com/172.
3. [알고리즘] 탐욕 알고리즘(Greedy Algorithm) . (2021년).

https://hanamon.kr/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%ED%83%90%EC%9A%95%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-greedy-algorithm/.

1. 다익스트라 알고리즘(C언어) [출처] 다익스트라 알고리즘(C언어)|작성자 예빈 . (2020).

https://blog.naver.com/PostView.nhn?blogId=haesung2121&logNo=221853607639&parentCategoryNo=&categoryNo=11&viewDate=&isShowPopularPosts=true&from=search.