A* 알고리즘과 **Johnson** 알고리즘 조사

개요

A* 알고리즘

A* 알고리즘(에이스타 알고리즘)은 그래프에서 한 시작점에서 목표 지점까지의 최단 경로를 찾는 휴리스틱 기반 탐색 기법입니다. 1968년 Peter Hart, Nils Nilsson, Bertram Raphael 등이 Shakey 라는 로봇의 경로 계획 문제를 연구하는 과정에서 개발했으며, Dijkstra 알고리즘을 확장하여 휴리스틱 함수를 도입한 형태입니다. A*는 완전성(해가 존재하면 반드시 찾음)과 최적성(올바른 휴리스틱 사용 시 최단 경로 보장)을 갖추고 있으며, 다양한 분야에서 활용됩니다. 예를 들어, 길찾기 기능이 있는 웹 지도나 게임의 경로finding, 로봇의 움직임 계획 등에서 A* 알고리즘이 널리 사용됩니다. 휴리스틱(heuristic)을 통해 탐색 방향을 똑똑하게 안내하기 때문에 "똑똑한" 알고리즘으로 불리기도 합니다.

Johnson 알고리즘

Johnson 알고리즘은 가중치 그래프에서 모든 쌍의 최단 경로(all-pairs shortest paths)를 효율적으로 찾는 알고리즘입니다. 1977년 Donald B. Johnson이 제안한 방법으로, Floyd-Warshall 알고리즘(O(V³))에 비해 희소(sparse) 그래프에서 효율적으로 동작하도록 만들어졌습니다. Johnson 알고리즘의 핵심은 그래프에 음수 가중치 간선이 있어도 사용 가능하다는 점으로, Bellman-Ford 알고리즘을 이용해 음의 가중치를 제거하는 재가중치(reweighting) 기법을 적용한 뒤 여러 번의 Dijkstra 알고리즘을 수행하여 모든 쌍 최단 거리를 계산합니다. 단, 음수 사이클(negative cycle)이 존재하면 최단 경로가 정의되지 않으므로 그 경우에는 동작을 중단합니다. Johnson 알고리즘은 컴퓨터 네트워크의 라우팅 최적화, 물류 경로계획 등 여러 경로 문제에 응용될 수 있으며, 특히 그래프가 희소 그래프인 경우 효율성과 메모리 측면에서 장점을 보입니다.

특징

A* 알고리즘의 특징

탐색 전략 및 최적성**:**

A*는 휴리스틱 함수 h(n)를 사용하여 남은 거리나 비용을 추정하고, 현재까지의 실제 비용 g(n)과 합산한 평가값 f(n) = g(n) + h(n)이 가장 낮은 노드를 우선 확장합니다. 휴리스틱 함수가 허용적(Admissible)이고 일관적(Consistent)이라면 A는 최단 경로의 발견을 보장하며 (완전성과 최적성) 탐색 효율도 높습니다. 올바른 휴리스틱을 사용하면 불필요한 경로 탐색을 줄여 Dijkstra 알고리즘보다 적은 노드를 확장할 수 있으며, A*는 허용적 휴리스틱 사용 시, 확장 노드 수 측면에서도 최적임이 증명되었습니다 (Pearl, 1984).

시간 복잡도**:**

A*의 이론적 시간 복잡도는 일반적인 그래프 탐색과 마찬가지로 최악의 경우 지수적입니다 (분기계수 b와 해까지의 깊이 d에 대해 O(b^d) 수준). 휴리스틱이 부실할 경우(예: 항상 0으로 추정하면 Dijkstra와 동일) 최악 성능은 O(E + V log V)로 Dijkstra 알고리즘 수준까지 악화됩니다. 하지만 휴리스틱이 문제의 해에 가까울수록 탐색 공간이 줄어들어 실제 성능은 휴리스틱의 품질에 크게 의존합니다. 예를 들어 휴리스틱이 실제 비용과 거의 같다면 A*는 거의 직선 경로만 따라가서 시간 복잡도가 경로 길이에 비례하게 됩니다.

공간 복잡도**:**

A*는 탐색 과정에서 오픈 리스트(Open list)와 클로즈드 리스트(Closed list)를 사용하여 모든 생성된 노드를 저장하기 때문에 공간 복잡도가 O(b^d)로 높습니다. 이로 인해 경로가 매우 긴 문제나 상태공간이 큰 문제에서는 메모리 사용량이 매우 커질 수 있습니다. 실제로 A*는 경로 길이가 깊어질수록 많은 노드를 메모리에 유지해야 하므로, 메모리 제약이 있는 환경에서는 문제가 될 수 있습니다. 이러한 단점을 보완하기 위해 메모리 제한 A* 등 변형 알고리즘이 연구되었습니다.

장점**:**

A*는 한 개의 출발-목표 쌍에 대한 최단 경로를 구하는 데 매우 효과적이며, 휴리스틱 설계를 통해 문제 유형에 특화된 최적화가 가능합니다. 휴리스틱이 잘 설계된 경우 불필요한 경로 탐색을 배제하여 탐색 속도가 빠르며, 많은 게임이나 내비게이션 시스템이 A*를 사용하여 실시간 경로 탐색을 수행하고 있습니다. 또한 해가 존재하면 반드시 찾아내는 완전성을 가집니다.

단점**:**

A*의 성능은 휴리스틱 함수의 품질에 민감합니다. 만약 휴리스틱이 실제 비용보다 크게 과대추정하면 최단 경로를 보장할 수 없고, 과소평가는 되지만 너무 부정확하면 Dijkstra 알고리즘처럼 많은 경로를 탐색하게 되어 효율이 떨어집니다. 또한 앞서 언급한 메모리 사용량 문제도 실무에서 제약이 될 수 있습니다. 마지막으로, A*는 한 번 실행으로 한 쌍의 경로만 구하므로 여러 출발-도착 쌍의 경로가 모두 필요한 경우에는 각 쌍마다 반복 실행해야 하는 단점이 있습니다. 이는 하나의 출발지에서 모든 노드까지 거리 계산이 가능한 Dijkstra의 단일 출발지 최단 경로와 비교되는 점으로, A*는 목표가 특정할 때만 효율을 발휘하도록 목표 지향적인 대가를 치르는 셈입니다.

Johnson 알고리즘의 특징

문제 범위**:**

Johnson 알고리즘은 모든 노드 쌍 사이의 최단 경로를 한 번의 알고리즘으로 계산한다는 점에서 Floyd-Warshall 알고리즘과 같은 전쌍 최단 경로 문제를 해결합니다. Floyd-Warshall이 동적 계획법으로 3중 중첩 루프(O(V³))를 사용하는 데 반해, Johnson 알고리즘은 희소 그래프에서 더 나은 성능을 보이도록 설계되었습니다. 이 알고리즘은 Bellman-Ford와 Dijkstra 알고리즘 두 가지를 조합함으로써 각 알고리즘의 장점을 활용합니다.

시간 복잡도**:**

Johnson 알고리즘의 시간 복잡도는 그래프의 간선 수(E)와 정점 수(V)에 따라 O(V·E + V² log V) (이진 힙 기반 구현 시)로 분석됩니다. 구체적으로, Bellman-Ford 단계가 O(V·E), 이후 각 정점을

시작점으로 V번 Dijkstra 알고리즘을 실행하는 단계가 O(V * (E log V))입니다. 이 복잡도를 Floyd-Warshall의 $O(V^3)$와 비교하면, 그래프가 충분히 희소하여 E가 V^2보다 작을 경우 Johnson 알고리즘이 더 효율적임을 알 수 있습니다. 예를 들어 E가 O(V) 정도인 희소 그래프에서는 Johnson 알고리즘이 대략 $O(V^2 \log V)$로 작동하여 V가 커질수록 $O(V^3)$보다 훨씬 빠릅니다. 반면, 그래프가 밀집 그래프($E \approx V^2$)인 경우에는 시간 복잡도가 $O(V^3 \log V)$에 근접하여 Floyd-Warshall과 큰 차이가 없어지는 단점이 있습니다.

공간 복잡도**:**

Johnson 알고리즘은 결과적으로 모든 쌍의 최단 경로 거리를 구해야 하므로, 출력으로 V×V 크기의 거리 행렬을 저장한다고 하면 공간 복잡도는 $O(V^2)$가 됩니다. 하지만 알고리즘 자체가 동작하는 동안에는 인접 리스트 등 그래프 저장에 O(V+E), Bellman-Ford의 거리 배열 O(V), 각 Dijkstra의 우선순위 큐 등에 O(V) 등이 사용되어 추가 공간 복잡도는 O(V + E) 수준입니다. 따라서 그래프가 매우 크고 V^2가 메모리에 담기 어려운 경우에는 Johnson 알고리즘 결과를 모두 저장하기보다는 필요한 쌍만 계산하는 등 추가 고려가 필요합니다.

장점**:**

Johnson 알고리즘의 가장 큰 장점은 음의 가중치(edge weight)를 허용한다는 것입니다. Bellman-Ford 알고리즘을 통해 음의 가중치를 가진 그래프를 적절히 재가중치하여 모든 간선 가중치가 비음수가 되도록 변환함으로써, 이후 단계에서 효율적인 Dijkstra 알고리즘을 적용할 수 있습니다. 이 과정에서 음수 사이클이 발견되면 존재하지 않는 최단 경로가 있으므로 즉시 중단할 수 있어, Johnson 알고리즘은 음수 사이클 검출 기능도 포함하고 있습니다. 또한 Floyd-Warshall 대비 메모리 사용이 적고, 그래프가 클 때도 Dijkstra의 우수한 성능을 활용할 수 있어 실용성이 높습니다. 실제 연구 결과에서도 Johnson 알고리즘이 Floyd-Warshall보다 런타임과 메모리 사용 면에서 우수한 사례들이 보고되었습니다. 특히 한 그래프에서 여러 출발-목적지 쌍의 최단 경로를 구해야 하는 경우, 각 쌍마다 A*나 Dijkstra를 반복 실행하는 것보다 Johnson 알고리즘으로 한 번에 처리하는 편이 전체적으로 유리할 수 있습니다.

단점**:**

Johnson 알고리즘은 구현 단계가 다소 복잡하며 두 가지 알고리즘을 결합해야 하므로 구현 부담이 큽니다. 또한 모든 쌍에 대한 결과를 구하는 것이므로, 필요하지 않은 쌍의 경로까지 계산하는 비효율이 있을 수 있습니다. 예를 들어, 단 하나의 출발지로부터 수십만 노드까지의 최단 경로만 필요할 경우, APSP는 과도한 연산이 됩니다. 게다가 그래프가 작거나 밀집한 경우에는 Floyd-Warshall 알고리즘이 구현 난이도가 낮고 비슷한 성능을 내기 때문에 Johnson 알고리즘의 이점이 크지 않을 수 있습니다. 마지막으로, Dijkstra 알고리즘을 여러 번 실행하므로 우선순위 큐 연산에 의한 로그 인자가 붙어 수학적 복잡도가 증가한다는 점도 고려해야 합니다 (이 부분은 Fibonacci 힙 등을 사용하면 완화할 수 있지만 구현이 더 복잡해집니다).

동작 방식

A* 알고리즘 동작 방식

A* 알고리즘은 휴리스틱을 활용하여 최단 경로를 찾는 최적 우선 탐색(best-first search) 형태로 동작합니다. 탐색을 진행할 그래프의 각 노드에는 시작 노드로부터의 누적 비용 g(n)과 목표

노드까지의 추정 비용 h(n)이 할당되며, 이 둘의 합 f(n) = g(n) + h(n)를 평가 값으로 사용합니다. 알고리즘은 다음과 같은 절차로 이루어집니다:

1. 초기화: 시작 노드를 오픈 리스트(Open list)에 넣고 g(start)=0, h(start)를 계산하여 f(start)를 설정합니다. 이때 휴리스틱 함수 h(n)는 현재 노드 n에서 목표까지의 비용을 추정하며, 문제가 되는 영역에 따라 맨해튼 거리, 유클리드 거리 등 적절한 함수를 사용합니다. 클로즈드 리스트(Closed list)는 비워둡니다.

2. 노드 선택: 오픈 리스트에서 f 값이 가장 작은 노드 n을 꺼냅니다. (만약 우선순위 큐로 구현하면 자동으로 최솟값 노드를 뽑을 수 있습니다.) 이 노드를 확장(expand)하여 이웃 노드들을 살펴봅니다. 오픈 리스트에서 꺼낼 때 해당 노드가 목표(goal) 노드이면 탐색을 종료하고 경로를 반환합니다.

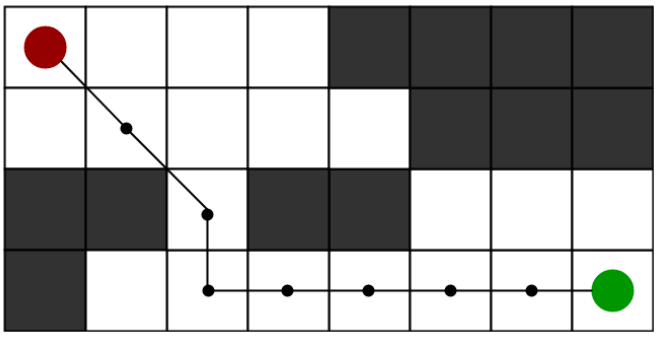
3. 이웃 탐색 및 갱신: 선택된 노드 n의 인접 노드들 각각에 대해:

    - 이미 Closed에 속한 노드는 무시합니다 (이미 최단 경로가 확정된 노드이므로 다시 볼 필요 없습니다).

    - 새로운 노드이거나 더 짧은 경로로 도달할 수 있다면 g(neighbor) 값을 g(n) + w(n,neighbor)로 새로 계산합니다. 이때 w(n,neighbor)는 n에서 이웃으로 가는 간선의 가중치입니다.

    - g(neighbor)가 기존 값보다 작으면 더 좋은 경로를 발견한 것이므로 f(neighbor) = g(neighbor) + h(neighbor)로 갱신하고 오픈 리스트에 넣습니다. 이미 오픈 리스트에 있는 노드라도 더 작은 f 값이 나오면 값을 갱신합니다.

4. 반복: 오픈 리스트가 빌 때까지 2~3 과정을 반복합니다. 오픈 리스트가 비었다는 것은 목표에 도달할 수 있는 경로가 없음을 의미하므로 이 경우 실패를 리턴합니다. 일반적으로는 목표 노드를 꺼냈을 때 해당 경로가 최단 경로가 되므로 알고리즘이 종료됩니다.



A* 탐색은 위 과정에서 휴리스틱 함수의 특성에 따라 확장 영역이 달라집니다. 예를 들어,위 그림에서 빨간색 출발점에서 녹색 목표까지 장애물(검은 칸)을 피해가는 경로를 찾을 때, A* 알고리즘은 휴리스틱으로 맨해튼 거리를 사용하여 오른쪽과 아래 방향 위주로 노드를 확장하게 됩니다. 그 결과 휴리스틱이 없는 경우(예: Dijkstra 알고리즘)보다 적은 노드를 검사하며도 최단 경로를 찾아낼 수 있습니다. Closed 리스트에는 이미 최적 경로가 확정된 노드들이 차곡차곡

쌓이게 되고, Open 리스트에는 앞으로 확장해야 할 후보 노드들이 우선순위에 따라 유지됩니다. A* 알고리즘은 이러한 구조 덕분에 항상 f 값이 가장 작은 경로를 우선으로 탐색하고, 휴리스틱이 정확할수록 불필요한 경로 탐색을 크게 줄입니다.

A의 휴리스틱은 일반적으로 허용적(admissible)으로 설계해야 하는데, 이는 어느 노드에서 목표까지의 실제 최단 비용을 결코 초과하지 않도록 추정하는 것을 의미합니다. 허용적 휴리스틱은 A*의 최적성을 보장하는데 필수적입니다. 또한 일관적(consistent) 휴리스틱일 경우 알고리즘의 효율이 더 향상되어 각 노드는 한 번만 확장되면 완료되는 성질이 있습니다. 이러한 이유로 맨해튼 거리나 유클리드 거리 같은 직관적인 휴리스틱이 자주 쓰이며, 문제에 따라 domain-specific한 휴리스틱이 고안되기도 합니다.

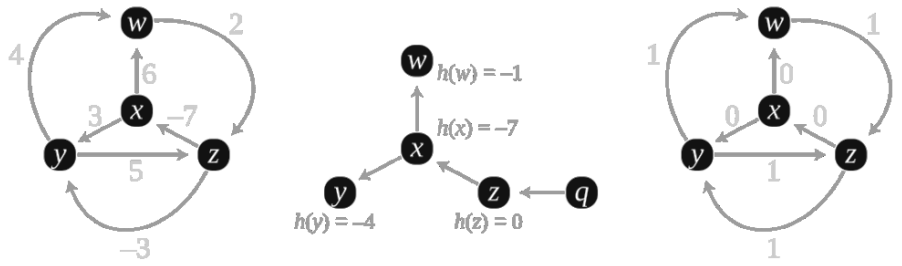
Johnson 알고리즘 동작 방식

Johnson 알고리즘은 두 단계로 구성됩니다. 첫 단계는 Bellman-Ford 알고리즘을 활용한 그래프 재가중치(reweighting) 단계이며, 두 번째 단계는 Dijkstra 알고리즘을 여러 번 수행하여 각 노드를 출발점으로 하는 최단 경로를 구하는 단계입니다. 전체 절차를 순서대로 설명하면 다음과 같습니다:

1. 그래프 보강: 주어진 그래프 G에 새로운 정점 q를 추가합니다. q에서 원래 그래프의 모든 정점으로 가중치 0인 간선을 추가하여 확장된 그래프 G'를 만듭니다. 이제 q에서 출발하는 Bellman-Ford 알고리즘을 수행할 준비가 되었습니다.

2. **Bellman-Ford** 단계: 새로운 정점 q를 시작점으로 Bellman-Ford 알고리즘을 실행하여 각 정점까지의 최소 거리 h(v)를 계산합니다. 이 값 h(v)는 원래 그래프에서 발생했던 음의 가중치 영향을 보정하는 잠재값(potential) 역할을 합니다. 만약 이 과정에서 음수 사이클이 발견되면 Johnson 알고리즘은 종료하며, "음수 사이클 존재로 실패"를 보고합니다. 음수 사이클이 없으면 h(v) 값들은 유한한 값으로 결정됩니다.

3. 간선 가중치 재계산: 원래 그래프 G의 모든 간선 (u→v)에 대해 새로운 가중치 w'(u,v) = w(u,v) + h(u) - h(v)로 재가중치합니다. 이렇게 하면 모든 간선 가중치가 0 이상인 비음수 값을 갖게 됩니다. 이는 h(v)가 경로상의 잠재값으로 작용하여, 원래 그래프에서 u에서 v로 가는 경로의 상대적 비용만 남기고 공통 부분을 상쇄시키는 효과가 있습니다. 재가중치는 경로 간 우열 순서(preserves path ordering)는 바꾸지 않으며, 거리만 보정하는 성질이 있습니다.



Johnson 알고리즘의 *3*단계*: (*좌*)* 원래 그래프와 가중치*, (*중*)* 새로운 정점 *q* 추가 후 *Bellman-Ford*로 계산된 *h* 값*, (*우*)* 재가중치된 그래프의 가중치들*.* 음수 가중치가 모두 제거되었다*.*

4. 보강 정점 제거: 추가했던 정점 q와 그 관련 간선들을 그래프에서 제거합니다. 이제 원래 정점들만 남은 그래프의 모든 간선 가중치가 비음수로 되었으므로, Dijkstra 알고리즘을 사용할 수 있는 상태가 되었습니다.

5. 모든 쌍 최단 경로 계산: 그래프의 각 정점 s에 대해 Dijkstra 알고리즘을 실행하여 s로부터 다른 모든 정점까지의 최단 거리를 계산합니다. Dijkstra 알고리즘은 재가중치된 가중치 w'를 사용하므로 음수 간선을 신경쓸 필요가 없습니다. s를 바꿔가며 V번 반복 실행하면, 결과적으로 모든 쌍 (s, t)에 대한 최단 거리 dist'(s, t)를 구할 수 있습니다.

6. 재가중치 보정: 각 쌍에 대해 원래 그래프에서의 실제 최단 거리 dist(s, t)는 dist'(s, t) + h(t) - h(s)로 계산하여 얻습니다. 이렇게 함으로써 재가중치 과정에서 보정했던 값들을 다시 복원하여 원래 그래프의 거리 값을 획득합니다. 최종적으로 모든 쌍 (s, t)에 대한 최단 거리 행렬을 얻을 수 있습니다.

이 과정 전체에서 핵심은 Bellman-Ford로 계산된 h 값을 활용해 음수 간선 문제를 제거했다는 점입니다. Bellman-Ford는 음수 사이클만 없으면 정확한 최단 거리를 계산하므로 h(v)는 q→v 최단 거리입니다. 이 값을 사용해 가중치를 조정하면 어떤 경로의 상대적인 비용 순서는 변하지 않고, 대신 모든 경로 비용에 동일한 상수 값을 더하거나 빼는 효과를 줍니다. 따라서 원래 그래프에서 최단 경로였던 경로는 재가중치된 그래프에서도 여전히 최단 경로로 남으며, 비용만 변경되었을 뿐 경로의 우열에는 영향이 없게 됩니다. 재가중치 후에는 Dijkstra 알고리즘의 제약 조건(비음수 가중치)이 만족되므로 각 노드에서의 Dijkstra 탐색을 통해 효율적으로 모든 거리를 구할 수 있게 됩니다. 이처럼 Johnson 알고리즘은 음수 간선 허용과 전체 쌍 거리 계산이라는 두 가지 어려운 문제를, 한 번의 Bellman-Ford와 여러 번의 Dijkstra로 치환하여 해결하는 기법인 것입니다.

완성 코드

A* 알고리즘 Python 구현

아래 코드는 파이썬으로 구현한 A* 알고리즘의 예시이다. 그래프는 인접 리스트(`graph`)로 표현되며, 우선순위 큐(`heapq` 모듈)를 이용하여 오픈 리스트에서 가장 작은 `f` 값을 가진 노드를 선택한다. `heuristic` 함수는 사용자 정의로 받아서 사용하며, 기본값은 0으로 설정된다 (즉, 휴리스틱을 제공하지 않을 경우 균일 비용 탐색과 동일하게 동작). 코드 실행 후에는 목표 노드까지의 최단 거리와 탐색에서 확장한 노드 수를 반환한다.

Python

```python
import heapq

def a_star(graph, start, goal, heuristic=None):
    """
    A* 최단 경로 알고리즘
```

```
    A* 알고리즘을 사용하여 그래프에서 시작 노드부터 목표 노드까지의 최단 경로를 찾습니다.
    이 함수는 휴리스틱 함수를 사용하여 탐색 효율을 높이며,
    최단 거리와 탐색 과정에서 확장된 노드의 수를 반환합니다.

    Args:
        graph (dict): 그래프를 나타내는 딕셔너리.
                      키는 노드이며, 값은 (이웃 노드, 가중치) 튜플의 리스트입니다.
                      예: {'A': [('B', 1), ('C', 4)], 'B': [('D', 2)]}
        start: 탐색을 시작할 노드.
        goal: 탐색 목표 노드.
        heuristic (function, optional): 각 노드에서 목표 노드까지의 예상 비용을 반환하는 휴리스틱 함수.
                                        인자로 노드를 받고 숫자를 반환해야 합니다.
                                        제공되지 않으면 모든 노드에 대해 0을 반환하는 기본 휴리스틱이 사용됩니다 (이는 Dijkstra 알고리즘과 동일).

    Returns:
        tuple: (최단 거리, 확장된 노드 수) 튜플을 반환합니다.
               목표 노드에 도달할 수 없는 경우 최단 거리는 float('inf')가 됩니다.
    """
    # 휴리스틱 함수가 제공되지 않으면 0으로 처리
    if heuristic is None:
        heuristic = lambda x: 0

    # 오픈 리스트 (우선순위 큐): 원소는 (f, node, g)
    open_heap = []
    heapq.heappush(open_heap, (heuristic(start), start, 0))
    # 각 노드의 현재까지 발견된 최소 g 값을 저장
    best_g = {start: 0}
    # 확장한 노드 수를 세기 위한 변수
    expansions = 0

    # 탐색 루프
    while open_heap:
        f, node, g = heapq.heappop(open_heap)
        # 꺼낸 항목이 이미 더 좋은 경로로 처리된 경우 스킵
        if g > best_g.get(node, float('inf')):
            continue
        expansions += 1
```

```
        # 목표 노드를 꺼낸 경우 성공
        if node == goal:
            return best_g[node], expansions
        # 이웃 노드 확장
        for neighbor, weight in graph.get(node, []):
            new_g = g + weight
            # 더 좋은 경로를 발견한 경우만 처리
            if new_g < best_g.get(neighbor, float('inf')):
                best_g[neighbor] = new_g
                new_f = new_g + heuristic(neighbor)
                heapq.heappush(open_heap, (new_f, neighbor, new_g))
    # 목표에 도달하지 못한 경우
    return float('inf'), expansions
```

위 구현에서 `open_heap`은 (f값, 노드, 현재 g값)을 원소로 가지며 항상 f값이 작은 순으로 노드를 팝(pop)한다. `best_g` 딕셔너리는 각 노드별로 발견된 최단 거리를 기록하여, 이미 더 좋은 경로가 있는 노드는 무시하도록 한다. 이러한 구문(`if g > best_g[node]: continue`)은 우선순위 큐에 동일 노드의 중복 항목이 존재할 수 있기 때문에 필요하다 (더 오래된, 비최적 경로 정보는 버림). 목표 노드를 확장(pop)하는 순간 알고리즘을 종료하며, 이때의 `best_g[goal]`이 최단 경로 비용이다. 또한 반환값으로 `expansions`를 함께 주어 몇 개의 노드를 확장했는지 알 수 있도록 했다. (확장한 노드 수는 오픈 리스트에서 꺼낸 노드 수와 같다.)

Johnson 알고리즘 **Python** 구현

다음 코드는 파이썬으로 구현한 Johnson 알고리즘의 예시이다. 그래프는 인접 리스트로 주어지며, Bellman-Ford 알고리즘으로 얻은 `h` 값을 사용해 간선 가중치를 재계산한 후 각 정점에 대해 Dijkstra를 수행한다. 구현의 간편함을 위해, 여기서는 Fibonacci 힙 대신 파이썬 기본 힙을 사용하였다. 또한 코드에서는 전체 모든 쌍 최단 거리 행렬(`all_pairs_dist`)을 계산하고, Bellman-Ford 단계에서 음수 사이클 감지도 수행한다.

Python

```
import math, heapq

def johnson_all_pairs(graph):
    """
    존슨(Johnson)의 모든 쌍 최단 경로 알고리즘
```

```
    음수 가중치 간선을 포함할 수 있는 희소 그래프에서 모든 노드 쌍 간의 최단 경로를 찾습니다.
    이 알고리즘은 벨만-포드(Bellman-Ford) 알고리즘을 사용하여 간선 가중치를 재조정하고,
    재조정된 양수 가중치 그래프에서 다익스트라(Dijkstra) 알고리즘을 각 노드에 대해 실행합니다.

    Args:
        graph (dict): 그래프를 나타내는 딕셔너리.
                      키는 노드(정수)이며, 값은 (이웃 노드, 가중치) 튜플의 리스트입니다.
                      노드는 0부터 V-1까지의 정수로 표현되어야 합니다.
                      예: {0: [(1, -1), (2, 4)], 1: [(2, 3), (3, 2)], 2: [(3, -5)],
3: []}

    Returns:
        dict: 모든 노드 쌍 간의 최단 거리를 포함하는 딕셔너리.
              `result[u][v]`는 노드 `u`에서 노드 `v`까지의 최단 거리를 나타냅니다.
              도달할 수 없는 경우 값은 `math.inf`입니다.

    Raises:
        ValueError: 그래프에 음수 사이클이 존재하는 경우 발생합니다.
    """
    V = len(graph)
    # 1. 새로운 노드 q 추가
    q_node = V
    graph_ext = {u: list(neighbors) for u, neighbors in graph.items()}
    graph_ext[q_node] = []
    for u in range(V):
        graph_ext[q_node].append((u, 0))
    # 2. Bellman-Ford 실행 (q_node를 시작 노드로)
    dist = {node: math.inf for node in graph_ext}
    dist[q_node] = 0
    for _ in range(len(graph_ext) - 1):
        updated = False
        for u, neighbors in graph_ext.items():
            for v, w in neighbors:
                if dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    updated = True
        if not updated:
            break
    # 음수 사이클 존재 확인
    for u, neighbors in graph_ext.items():
```

```
        for v, w in neighbors:
            if dist[u] + w < dist[v]:
                raise ValueError("음수 사이클이 존재합니다.")
    h = dist  # 각 정점의 잠재값 h 저장
    # 3. 간선 가중치 재계산 (reweight)
    new_graph = {}
    for u, neighbors in graph.items():
        new_neighbors = []
        for v, w in neighbors:
            new_w = w + h[u] - h[v]
            new_neighbors.append((v, new_w))
        new_graph[u] = new_neighbors
    # 4. 추가했던 q_node는 더 이상 사용 안 함 (new_graph에 포함되지 않음)
    # 5. 각 정점마다 Dijkstra 수행
    all_pairs_dist = {u: {} for u in graph}
    for src in graph:
        # Dijkstra from src on reweighted graph
        dist_src = {node: math.inf for node in graph}
        dist_src[src] = 0
        pq = [(0, src)]
        while pq:
            d, u = heapq.heappop(pq)
            if d > dist_src[u]:
                continue
            for v, w in new_graph[u]:
                if d + w < dist_src[v]:
                    dist_src[v] = d + w
                    heapq.heappush(pq, (dist_src[v], v))
        # 거리 보정 및 저장: dist'(u,v) + h(v) - h(u)
        for v in graph:
            if dist_src[v] < math.inf:
                all_pairs_dist[src][v] = dist_src[v] + h[v] - h[src]
            else:
                all_pairs_dist[src][v] = math.inf
    return all_pairs_dist
```

위 코드의 `johnson_all_pairs` 함수는 입력 그래프(`graph`)를 받아 모든 쌍 최단 거리 결과를 `all_pairs_dist` 딕셔너리 형태로 반환한다. 먼저 새로운 노드 `q_node`를 추가하고, 0 가중치 간선을 연결한 확장 그래프 `graph_ext`를 만든다. 그런 다음 Bellman-Ford 알고리즘을 실행하여 `dist` 사전에 `q_node`로부터 각 노드의 최단 거리 값을 계산한다. 이때 `updated` 플래그를 통해

조기에 수렴하면 루프를 탈출하도록 최적화했다. 음수 사이클 검사는 Bellman-Ford 종료 후 한 번 더 모든 간선을 검사하여 개선 여지가 있는지 확인하는 방식으로 구현했다. 음수 사이클이 발견되면 `ValueError` 예외를 발생시켜 알려준다.

그 후 원래 그래프의 간선 가중치를 `new_w = w + h[u] - h[v]` 계산식으로 변환하여 `new_graph`를 구성한다. 이렇게 계산된 `new_graph`에서 모든 간선 가중치는 0 이상의 값이 된다. 마지막으로 각 정점 `src`를 시작점으로 Dijkstra 알고리즘을 수행한다. Dijkstra 구현은 일반적인 우선순위 큐 사용 방식이며, `dist_src` 사전에 시작점으로부터의 거리를 기록한다. 우선순위 큐에서 꺼낸 값이 이미 최단거리가 아닌 경우 무시하는 패턴(`if d > dist_src[u]: continue`)을 사용하여 효율을 높였다. Dijkstra 종료 후에는 `dist_src[v]`가 재가중치 그래프에서의 거리 `dist'(src,v)`이므로, 이에 `h[v] - h[src]`를 더하여 원래 거리 `dist(src,v)`로 보정한 뒤 `all_pairs_dist`에 저장한다.

동일한 테스트 데이터 사용 및 시각화

최단경로 알고리즘 성능 비교**: A***와 **Johnson(**특정 조건 하**)**

성능 비교는 노드 수와 간선 수를 변화시키는 랜덤 유향 그래프를 기반으로 진행되었습니다. 각 노드 간 간선 가중치는 1부터 10까지의 양수 정수 값을 가집니다.

Johnson 알고리즘은 원래 음수 가중치 간선을 포함할 수 있는 그래프에서 모든 쌍 최단 경로를 찾는 데 특화되어 있지만, 이 실험에서는 양수 가중치 그래프에서 테스트되었기 때문에 그 주요 이점인 음수 가중치 처리 능력이 발휘되지 못했습니다. (음수 가중치를 포함한 그래프의 경우 결과값이 연속적으로 나타나지 않고, 다른 알고리즘들이 처리할 수 없어 직접적인 비교를 할 수 없기에 제외하였습니다.)

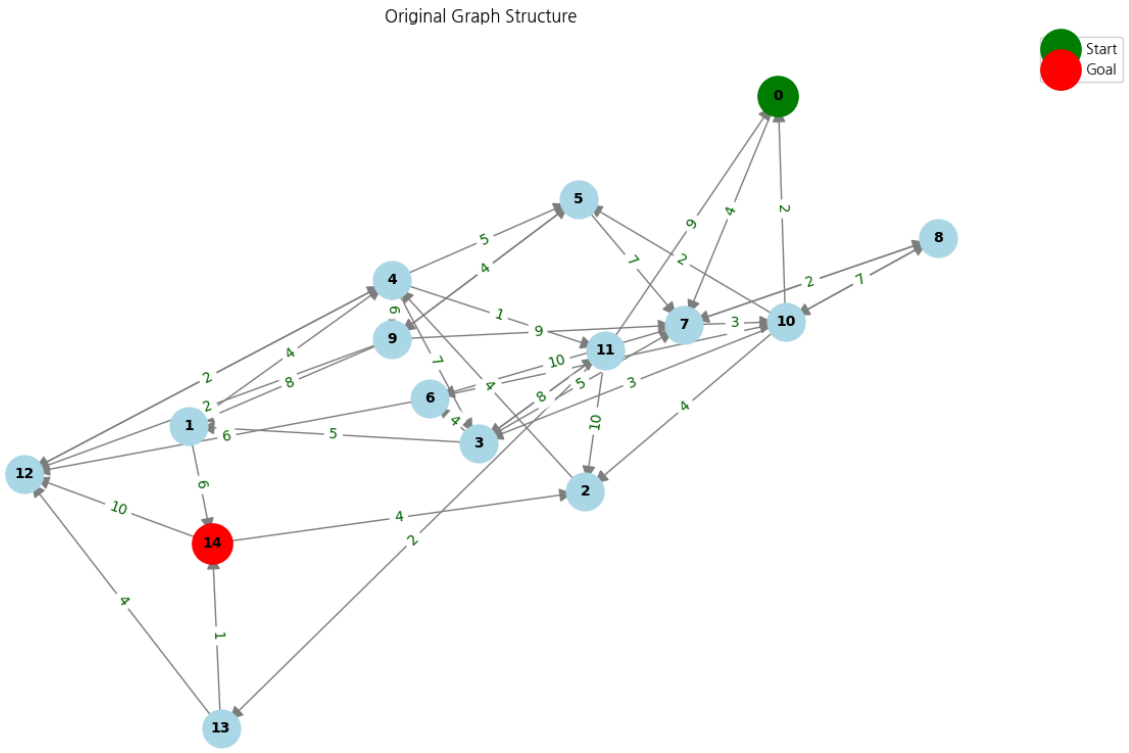
동일한 테스트 데이터 사용 및 시각화

두 알고리즘의 성능을 비교하기 위해, 동일한 조건에서 A* 알고리즘과 Johnson 알고리즘을 테스트하였습니다. 비교 실험은 격자형 그래프**(grid)**가 아닌, 임의의 연결성을 가진 랜덤 유향 그래프를 기반으로 수행되었습니다. (제공된 `generate_graph` 함수는 격자형 그래프가 아닌, 무작위 간선을 추가하는 일반적인 유향 그래프를 생성합니다. 또한 `euclid` 휴리스틱은 노드에 임의의 좌표를 부여하여 계산하므로, 격자형 그래프의 맨해튼 거리를 직접적으로 사용한 것은 아닙니다.)

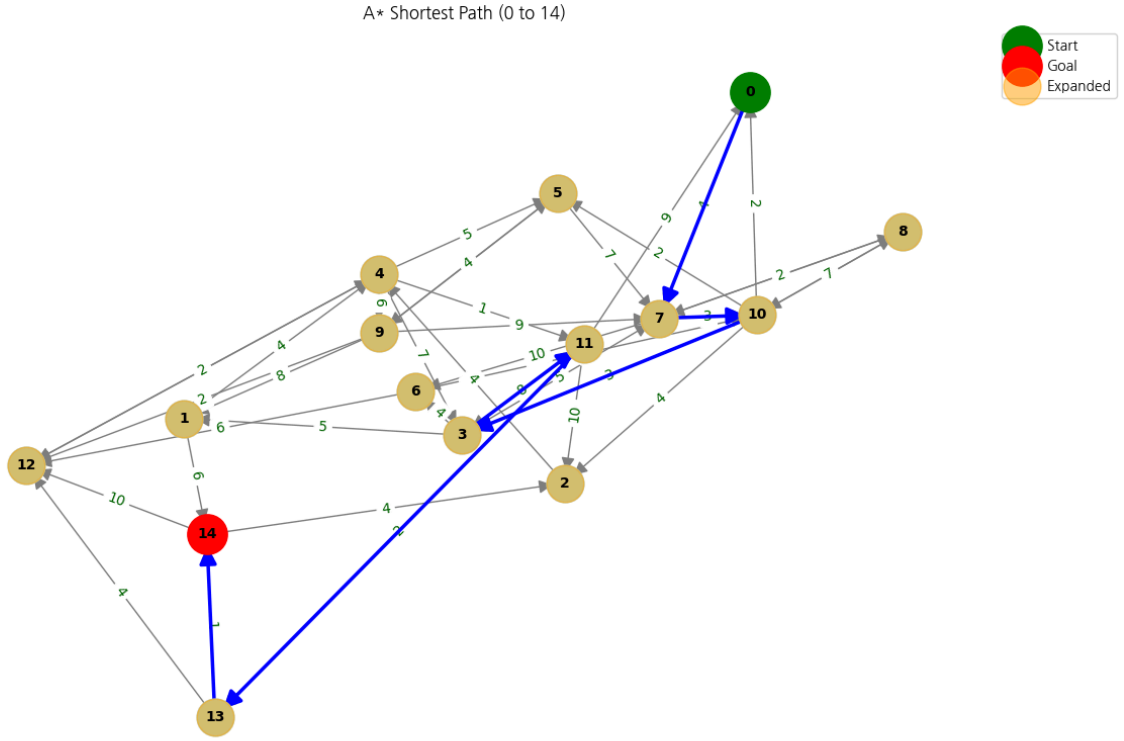
그래프의 노드 수와 간선 수를 변화시키면서 각 알고리즘으로 최단 경로를 구하고, 실행 시간과 방문한 노드 수를 측정하였습니다. 그래프에서 각 노드는 임의의 이웃 노드와 연결되어 있으며, 간선 가중치는 1부터 10까지의 정수 값을 무작위로 부여하였습니다.

A* 알고리즘은 그래프의 시작 노드(0)를 출발하여 마지막 노드(n-1)를 목표로 설정하였고, 휴리스틱 함수로 임의로 부여된 2차원 좌표 간의 유클리드 거리를 사용하였습니다. Johnson 알고리즘은 동일한 그래프에 대해 모든 쌍 최단 경로를 계산합니다. 공정한 비교를 위해 이 보고서에서는 A*가 단일 출발-목표 쌍에 대한 최단 경로를 찾는 반면, Johnson 알고리즘은 모든 쌍의 최단 경로를 계산하는 데 소요된 총 시간과 각 노드에서 **Dijkstra**를 수행하며 확장한 노드 수의 총합으로 성능 지표를 해석하였습니다. 이는 Johnson 알고리즘이 특정 쌍의 경로를 얻기까지의 총 작업량으로 간주됩니다.

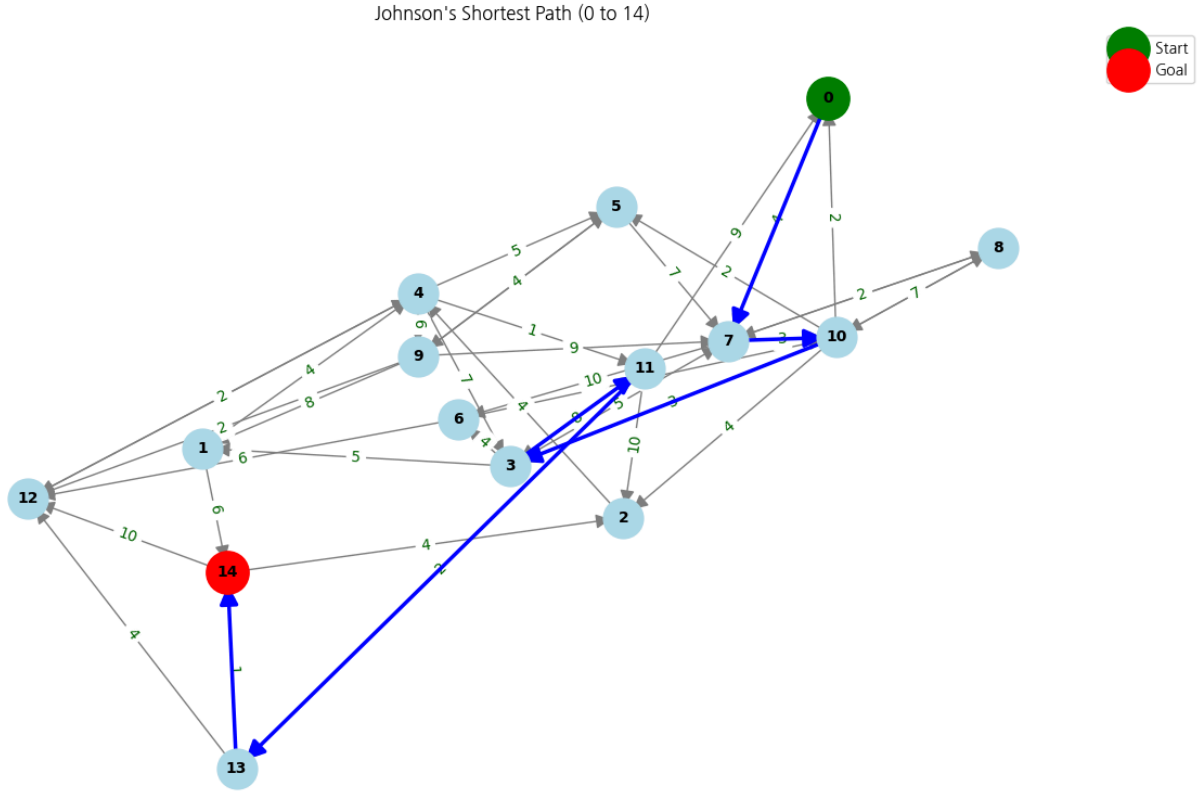
1. 그래프 경로 시각화**(1**개 세트**)**

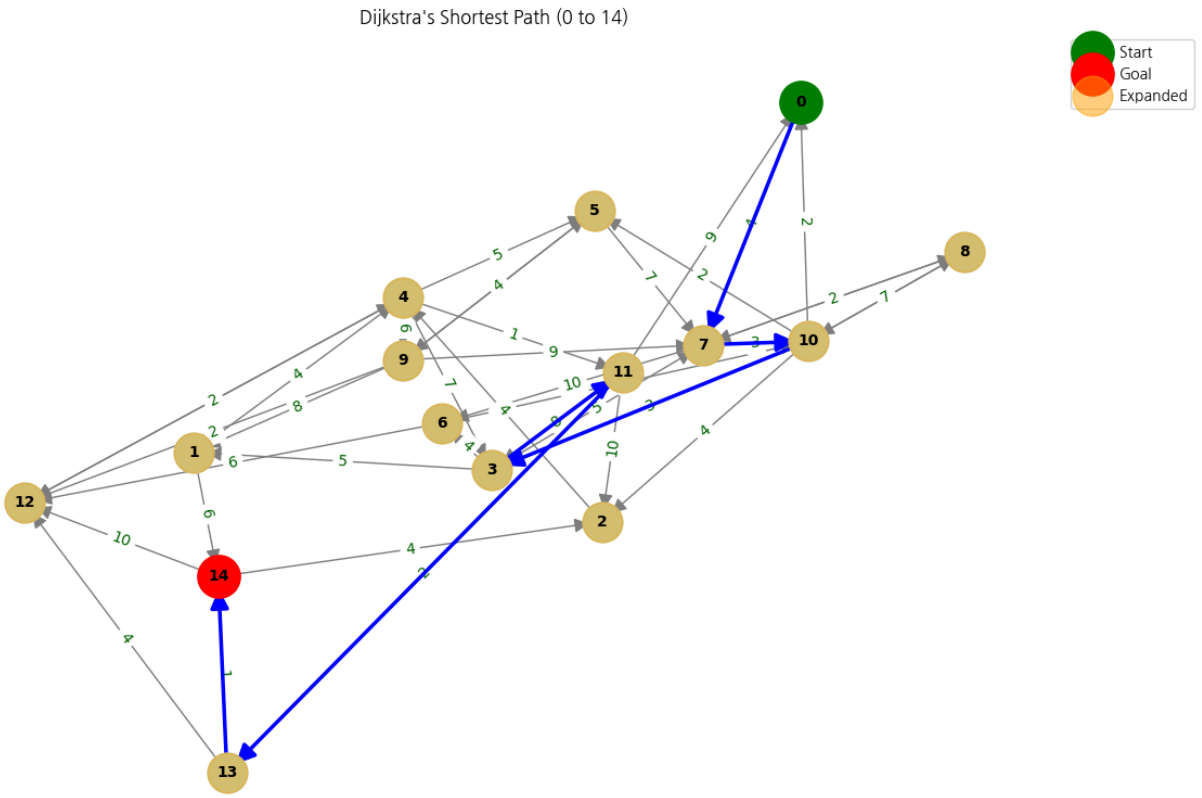


A* Shortest Path (0 to 14)



Johnson's Shortest Path (0 to 14)





A*와 **Dijkstra:** 두 알고리즘 모두 단일 시작-목표 노드 간의 최단 경로를 찾습니다.

- 찾은 경로**:** 시각화된 파란색 선으로 표시되며, 이는 해당 알고리즘이 발견한 최단 경로입니다.
- 확장된 노드**:** 주황색으로 표시된 노드들은 각 알고리즘이 최단 경로를 찾기 위해 "고려하거나 방문한" 노드들입니다. A*는 휴리스틱을 사용하기 때문에 일반적으로 Dijkstra보다 더 적은 노드를 확장하면서도 최단 경로를 찾을 수 있습니다.

Johnson 알고리즘**:** 이 알고리즘은 단일 경로를 찾는 것이 아니라, 그래프 내의 모든 노드 쌍 간의 최단 경로를 계산합니다.

- 시각화에서는 Johnson이 계산한 모든 경로 중 특정 시작**-**목표 노드 간의 경로만 추출하여 보여준 것입니다**.** Johnson은 탐색 과정에서 '확장된 노드' 개념을 직접적으로 반환하지 않아 시각화에 포함되지 않았습니다.

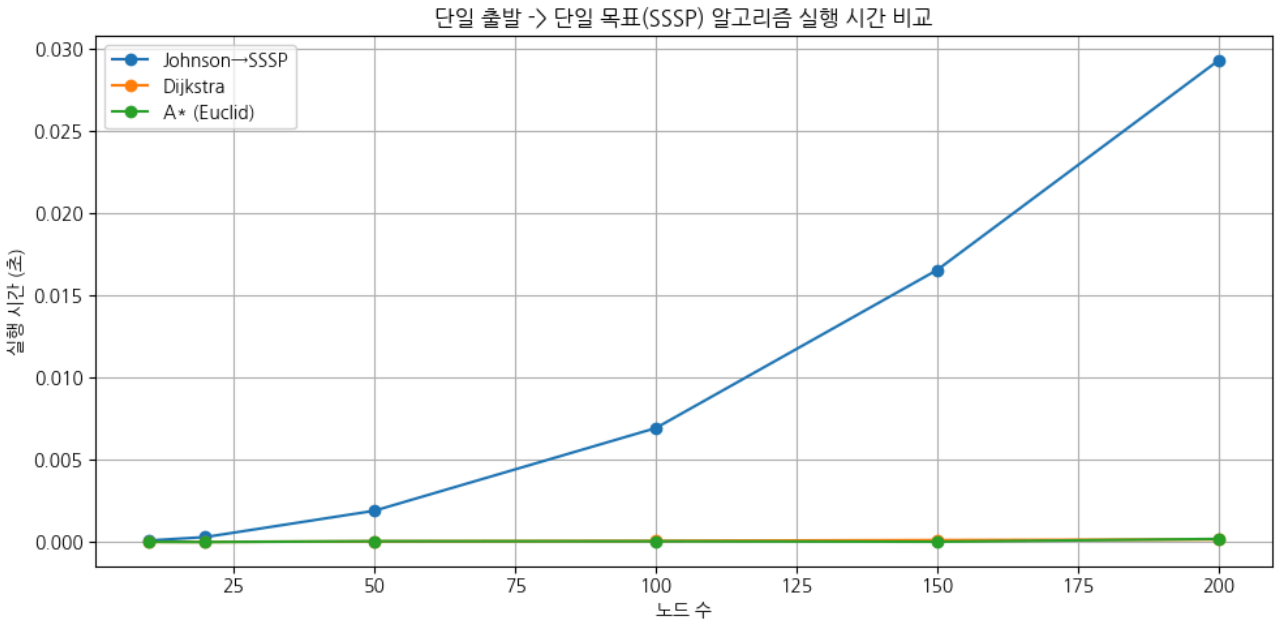
결론

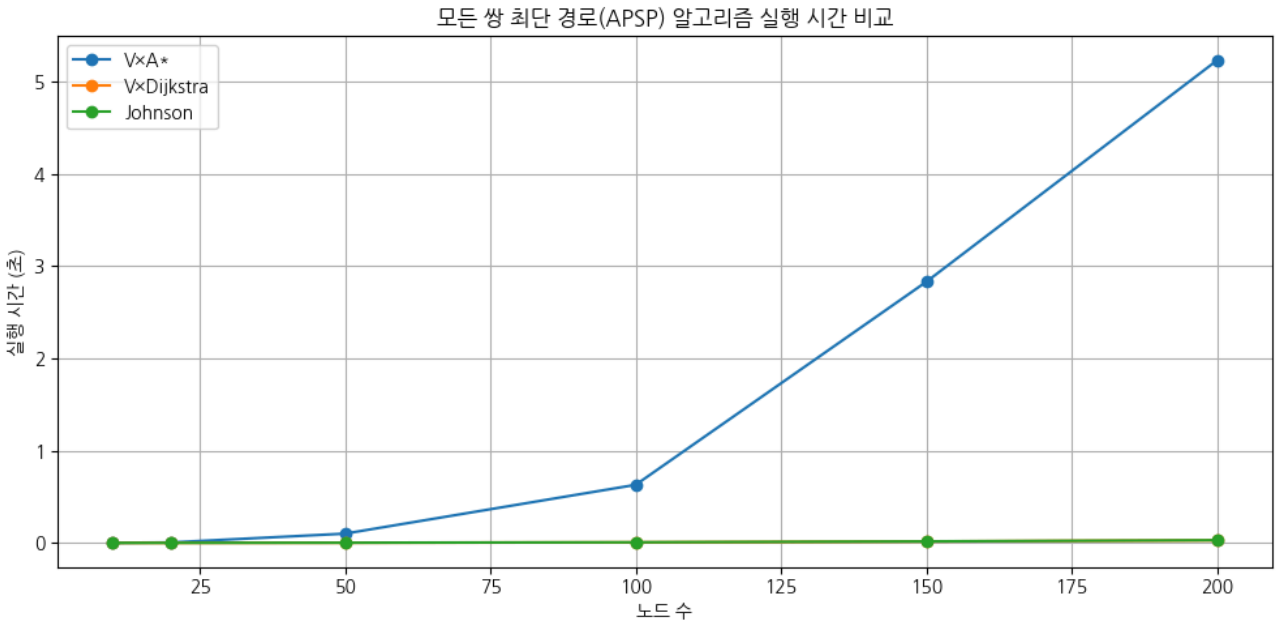
시각화된 이미지에서 A*와 Johnson이 동일한 경로를 보여줄 수도 있습니다. 이는 해당 그래프에서 특정 시작**-**목표 노드 간의 최단 경로가 유일하거나**,** 또는 여러 최단 경로 중 알고리즘들의 내부 로직이 우연히 같은 경로를 선택했기 때문입니다. 모든 최단 경로 알고리즘은 가장 짧은 경로를 찾아내지만, 동일한 길이를 가진 여러 경로가 존재할 경우 어떤 경로를

반환할지는 알고리즘의 구현 방식에 따라 달라질 수 있습니다. 즉, 최단 경로는 유일하지 않을 수 있으며, 따라서 다른 경로가 나올 가능성도 항상 있습니다.

2. 실행 시간 비교

아래 그래프는 그래프 노드 수에 따른 알고리즘의 실행 시간(초)을 비교한 것입니다.





위 그래프를 통해 노드 수가 증가함에 따라 **Johnson** 알고리즘의 실행 시간이 급격히 증가하는 것을 볼 수 있습니다. Johnson 알고리즘은 모든 쌍 최단 경로를 구하는 본연의 목적 때문에, 그래프의 모든 노드를 시작점으로 다익스트라를 수행해야 합니다. 이는 노드 수 V에 대해 대략

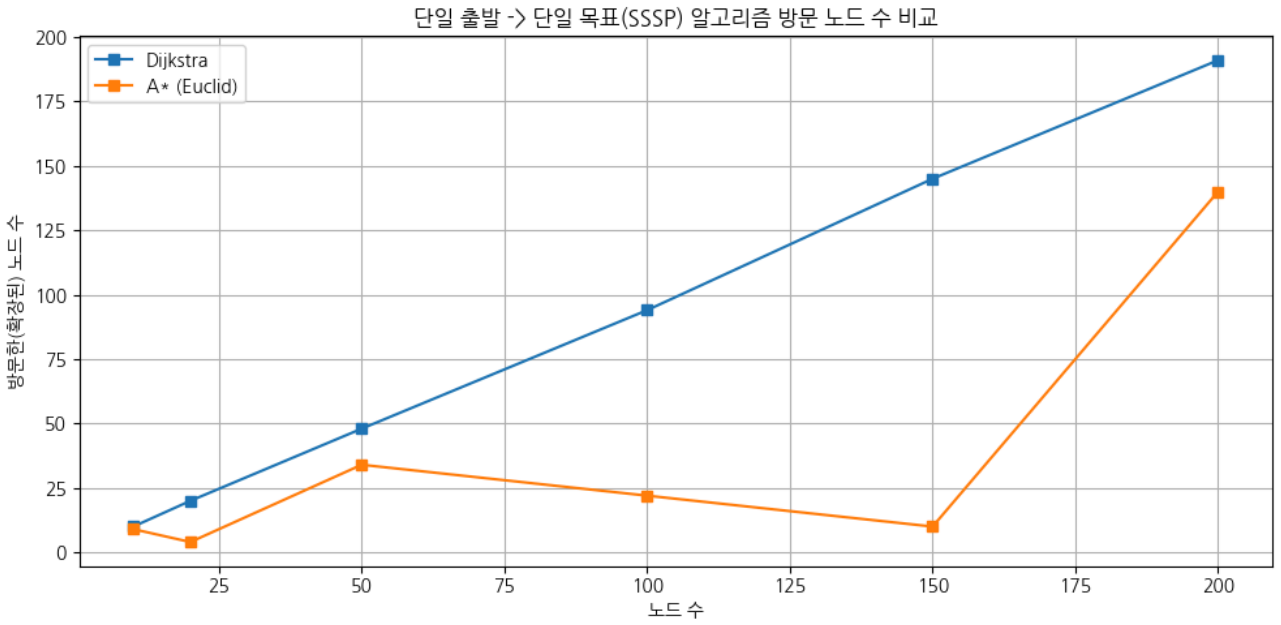
O(V·(E+VlogV))의 시간 복잡도를 가집니다. 따라서 노드 수가 증가할수록 제곱에 가까운 형태로 실행 시간이 증가합니다.

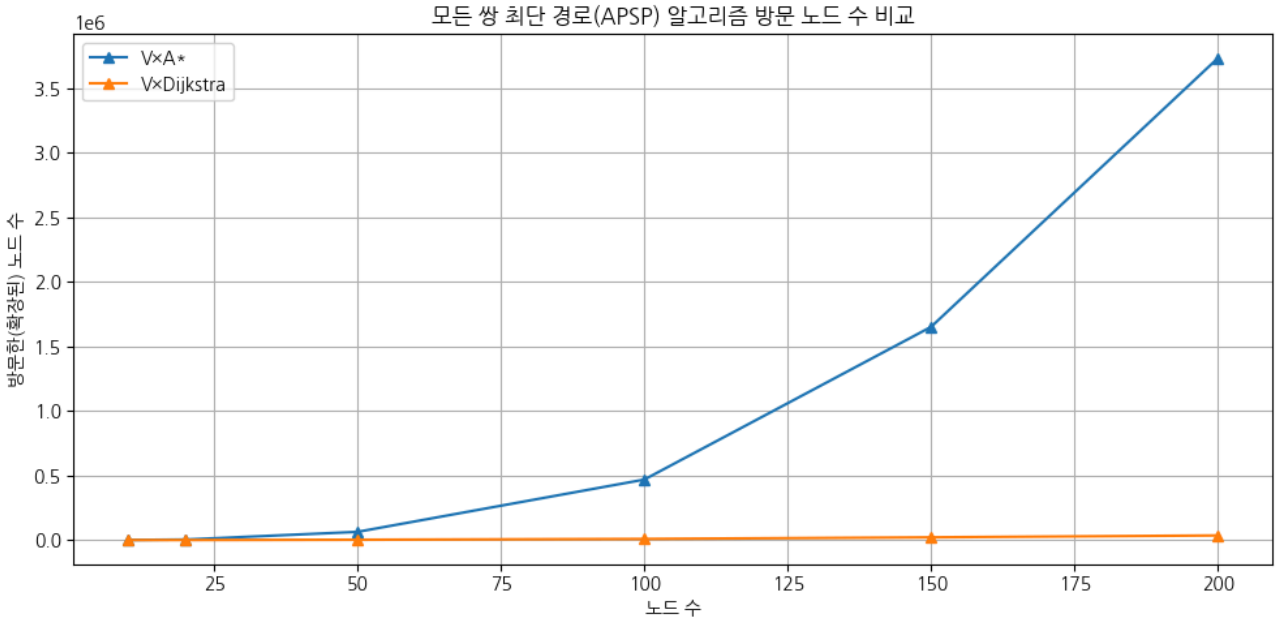
반면 **A*** 알고리즘은 특정 시작 노드(0)에서 특정 목표 노드(n-1)까지 한 쌍의 경로만 탐색하므로 실행 시간 증가가 상대적으로 완만합니다. A*는 휴리스틱(`euclid` 함수)을 사용하여 목표 지향적인 탐색을 수행하기 때문에, 불필요한 경로 탐색을 줄여 효율적입니다.

예를 들어, 실험에서 약 200개의 노드가 있는 그래프의 경우 (데이터는 200, 600 설정), A* 알고리즘의 경로 탐색은 매우 짧은 시간에 완료된 반면, Johnson 알고리즘은 A*보다 훨씬 긴 시간을 소요했습니다. 이러한 차이는 알고리즘의 근본적인 목적과 시간 복잡도에서 기인합니다. A*는 단일 SSSP 문제에 최적화되어 불필요한 연산을 줄이는 반면, Johnson은 APSP 문제 해결을 위해 모든 노드 쌍에 대한 정보를 계산해야 하므로 연산량이 훨씬 많습니다. 이는 실제로 한두 쌍의 경로만 필요하다면 Johnson 알고리즘이 비효율적일 수 있음을 보여줍니다.

2. 방문한 노드 수**(**확장량**)** 비교

다음 그래프는 같은 실험에서 알고리즘이 탐색 과정에서 방문한 노드 수(탐색한 그래프 정점 개수)를 비교한 것입니다. Johnson 알고리즘의 방문 노드 수는 각 출발점에서 수행된 다익스트라의 확장 노드 수를 모두 합산한 총량입니다.





A* 알고리즘은 휴리스틱을 활용하여 목표와 관련 없는 경로는 많이 배제하므로, 확장한 노드의 수가 상대적으로 적었습니다. 노드 수가 200개인 그래프의 경우 (200, 600 설정), A*는 상대적으로 적은 수의 노드를 확장했습니다. 이는 A*가 목표 지향적 특성을 가짐을 명확히 보여줍니다.

반면 **Johnson** 알고리즘은 (그 자체로는 직접적인 확장 노드 수를 계산하지는 않지만) 내부적으로 각 노드마다 다익스트라 탐색을 수행하므로, 전체 확장된 노드 수(V x Dijkstra의 확장량)는 노드 수의 제곱에 비례하여 매우 급격하게 증가하는 모습을 보입니다. Johnson 알고리즘의 방문 노드 수 곡선이 대략 y=x2 형태로 상승하는 반면, A*의 곡선은 거의 선형에 가깝게 증가하는 모습입니다. 물론 이 비교는 Johnson 알고리즘이 모든 쌍을 구하기 때문에 발생하는 필연적 결과이며, 하나의 경로 탐색만 놓고 보면 Johnson 내부에서 수행된 한 번의 Dijkstra도 평균적으로는 A*와 유사한 수의 노드를 방문할 수 있습니다. 하지만 여러 출발지를 모두 탐색해야 한다는 Johnson 알고리즘의 특성상 누적된 작업량이 방대해지는 것을 확인할 수 있습니다. 이

실험은 목표 지향적 탐색인 A*와 모든 쌍을 대상으로 하는 Johnson의 근본적인 차이를 수치로 잘 보여줍니다.

종합 결론

종합하면, **A*** 알고리즘은 특정한 한 쌍의 최단 경로를 빠르고 효율적으로 찾는 데 매우 뛰어난 반면, **Johnson** 알고리즘은 그래프 전체의 거리 정보를 한꺼번에 산출할 때 유용합니다. 이 실험에서 Johnson 알고리즘은 양수 가중치 그래프에서 테스트되었기 때문에 그 주요 이점인 음수 가중치 처리 능력이 발휘되지 못했으며, 단일 SSSP 문제에 대해서는 A*보다 훨씬 비효율적인 결과를 보였습니다.

따라서 사용 목적에 따라 적합한 알고리즘을 선택하는 것이 중요합니다. 예를 들어 경로 탐색 게임이나 길찾기 애플리케이션처럼 한 번에 하나의 최단 경로만 필요하면 A*가 적합하고, 교통망 분석처럼 모든 지점 간 최단 거리를 미리 계산해야 하는 경우 Johnson 알고리즘(또는 Floyd-Warshall 등)이 적합합니다. 이번 비교를 통해 문제의 특성(단일 쌍 vs 모든 쌍, 그래프의 크기와 밀도, 가중치 특성 등)에 따라 두 알고리즘의 성능과 효율이 크게 달라짐을 확인할 수 있었습니다. 각 알고리즘은 설계 목적이 다르므로, 실제 활용 시에는 이러한 특징을 고려하여 가장 효율적인 방법을 선택해야 할 것입니다.

출처 및 참고자료:

1. https://starrykss.tistory.com/1522
2. https://dad-rock.tistory.com/652
3. https://stlee321.tistory.com/entry/%EA%B7%B8%EB%9E%98%ED%94%84-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98%EC%A1%B4%EC%8A%A8-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98
4. https://www.dbpia.co.kr/Journal/articleDetail?nodeId=NODE10597118
5. https://velog.io/@gang_shik/%EC%B5%9C%EB%8B%A8%EA%B2%BD%EB%A1%9C-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98
6. https://www.dbpia.co.kr/journal/detail?nodeId=T15775418
7. https://scienceon.kisti.re.kr/srch/selectPORSrchArticle.do?cn=NART31803847
8. https://support.minitab.com/ko-kr/minitab/help-and-how-to/quality-and-process-improvement/quality-tools/how-to/johnson-transformation/interpret-the-results/key-results/
9. https://en.wikipedia.org/wiki/A*_search_algorithm
10. https://www.geeksforgeeks.org/a-algorithm-and-its-heuristic-search-strategy-in-artificial-intelligence
11. https://www.geeksforgeeks.org/comparison-between-shortest-path-algorithms/
12. https://en.wikipedia.org/wiki/Johnson%27s_algorithm
13. https://medium.com/@christinejcob/johnsons-algorithm-the-swiss-army-knife-of-graph-theory-b05e
14. https://www.geeksforgeeks.org/a-search-algorithm/