韓國 컴퓨터情報學會 論文誌
第9卷 第3號, 2004. 9.

2004-9-3-1-9

타일맵에서 A* 알고리즘을 이용한 유닛들의 길찾기 방법 제안

이 세 일*

Units' Path-finding Method Proposal for A* Algorithm in the Tilemap

Se-Il Lee*

요 약

게임을 하다보면 유닛들이 목표를 찾아 가야하는데 알고리즘에 따라 시간과 거리가 많이 차이 나게 된다. 본 논문에서는 깊이 우선 탐색과 너비 우선 탐색 그리고 거리 값을 주어 개선된 알고리즘과 A* 알고리즘의 각각을 비교하며 특징들을 기술하고 알고리즘을 논하였다. 또한 A* 알고리즘에서 실제로 추정값을 구하여 가장 개선된 값을 찾는다. 마지막으로 A* 알고리즘과 다른 알고리즘의 비교를 통하여 A* 알고리즘을 우수성을 입증하고 A* 알고리즘을 이용한 간단한 길찾기를 제안하였다.

Abstract

While doing games, units have to find goal. And according to algorism, there is great difference in time and distance. In this paper, the researcher compared and described characteristics of each of the improved algorism and A* algorism by giving depth-first search, breadth-first search and distance value and then argued algorism. In addition, by actually calculating the presumed value in A* algorism, the researcher finds the most improved value. Finally, by means of comparison between A* algorism and other one, the researcher verified its excellence and did simple path-finding using A* algorism.

▸ Keyword : Path-Finding, A*, DFS, BFS

• 제1저자 : 이세일
• 접수일 : 2004.08.28, 심사완료일 : 2004.09.08
* 공주대학교 컴퓨터공학과 박사과정

Ⅰ. 서 론

우리가 게임을 만들게 되면 모든 게임이 유저들에게 환영 받는 것은 아니다. 가장 재미있는 게임을 만들기 위해서는 흥미로운 스토리와 훌륭한 그래픽[1], 사운드 등이 뒷받침이 되어야 한다. 또한 게임을 하면서 유저는 지루함이 없어야 하며 게임의 주인공이나 유닛들이 멍청하다는 생각이 들게 되면 그 게임에는 관심이 적어질 것이다. 위와 같은 생각이 들지 않도록 하는 방법 중에 하나가 바로 길찾기 알고리즘을 올바르게 적용하는 것일 것이다. 만약에 게임상의 유닛들이 장애물도 별로 없는 아주 가까운 거리의 목표를 멀리 돌아오거나 또는 찾지 못한다면 유저들의 생각은 너무나도 뻔할 것이다. 좋은 길찾기 알고리즘을 사용한다고 하여도 여러 가지 변수에 의하여 반드시 최고의 결과가 나오지 않을 수 있다.

대부분의 게임 프로그래밍에서는 맵을 사용하며 맵상에서 유닛들이 돌아다니기 위해서는 두 지점의 경로를 찾아야 한다. 실시간 전략이나 1인칭 슈팅, 롤플레잉 게임들은 길찾기[2]를 필요로 한다. 특히 롤플레잉 유저들은 길찾기에 대하여 대단한 불만을 가지고 있다. 본 논문의 2장은 탐색 알고리즘의 종류에 대하여 논한다. 3장에서는 탐색 알고리즘을 비교해보고 A* 알고리즘을 이용한 길찾기의 최적화에 대하여 논하고 4장에서는 비교적 간단한 타일맵[3]을 이용하여 어느 곳에서나 유닛들이 원하는 목표를 찾아가는 길찾기 알고리즘에 대하여 논한다.

Ⅱ. 길찾기 알고리즘의 종류

2.1 맵의 공간 분할 종류

2.1.1 정사각형 분할

동일한 크기의 정사각형(그림 1)으로 분할하는 것으로 가장 간단한 방법이다[4]. 검색의 위치는 사각형의 모서리일수도 있고 사각형의 중심일 수도 있다.

2.1.2 쿼드트리 분할

쿼드트리(Quad Tree)분할(그림 2)은 공간상의 객체에 대하여 저장 맵을 4등분하여 색인을 구성하는 방법으로 2D나 3D에 사용할 수 있다. 각 사각형이 동일한 지형이 될 때까지 순환한다. 불균형한 분포의 공간객체에 대한 성능이 저하된다.

2.1.3 가시점

가시점(Point Of Visibility)은 공간을 분할하지 않고 장애물을 피하는 것에 역점을 두고 직접 위치들을 결정[4]하는 것이다. 이 방식은 장애물의 모양을 따라가는 방식으로 최단 경로를 찾아 갈 수 있다. 당연한 이야기지만 장애물의 모양이 복잡하면 가시점의 개수도 증가한다(그림 3).

2.1.4 일반화된 원통

일반화된 원통(Generalized Cylinder) 방식은 옆에 있는 장애물들 사이의 공간을 하나의 2D 원통형으로 생각하고 그 원통의 중심축을 이루는 선들의 교점을 검색의 위치로 설정한다[4]. 이 방식은 장애물을 피하는데 중점을 두고 있다(그림 4).

2.1.5 볼록 다각형

동일한 지형을 덮는 볼록 다각형(Convex Polygon)들로 공간을 분할하는 것이다(그림 5).

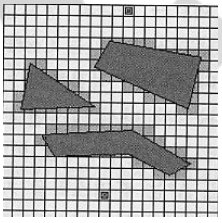


그림 1. 정사각형 격자
Figure. 1 Square Lattice

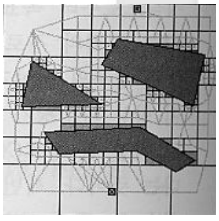


그림 2. 쿼드트리
F igure. 2 Quad Tree

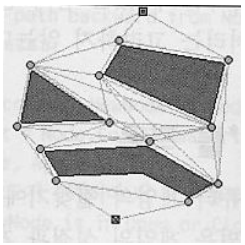


그림 3. 가시점
Figure. 3 Point Of Visibility

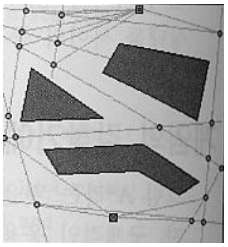


그림 4. 일반화된 원통
Figure. 4 Generalized Cylinder

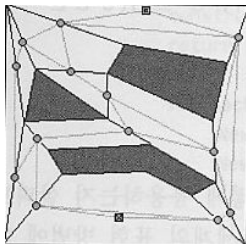


그림 5. 볼록 다각형
Figure. 5 Convex Polygon

다각형 내부의 어떤 두 점을 연결하여도 다각형을 이루는 선분과 교차하지 않는 다각형을 말한다. 여러 가지 공간 분할 방식이 존재하지만 본 논문에서는 정사각형 격자 방식을 사용하였다.

2.2 길찾기 알고리즘의 종류

2.2.1 깊이 우선 탐색

깊이 우선 탐색(Depth First Search)은 출발 노드로부터 시작하여 노드를 계속적으로 확장하여 가장 최근에 생성된 노드로 먼저 확장시키는 탐색기법이다. 만약 자식 노드가 없다든지 어떤 노드의 자식 노드를 방문했지만 목표 노드를 방문하지 못하였을 경우에는 즉각적으로 바로 직전 노드로 돌아가게 된다. 깊이 우선 노드는 목표 노드가 없는 곳을 계속 따라갈 수 있으므로 상황에 따라 상위 노드로 되돌아가는 백트래킹(Backtracking)이 필요하다[5]. 깊이 우선 알고리즘은 다음과 같다.

A. 현재 노드를 큐에 추가한다.
B. 현재 노드를 체크한다.
C. 큐에서 처음 노드를 뽑고 이 뽑힌 노드는 현재 노드가 된다.
   ① 현재 노드가 목표 노드이면 종료한다.
   ② 현재 노드의 자식들 중에서 표시되지 않으면, 그 노드들의 이전 포인터를 현재 노드로 설정하고 그 노드들을 큐의 앞부분에 넣는다.
D. 알고리즘이 종료되거나 큐가 비지 않으면 C를 계속 반복한다.

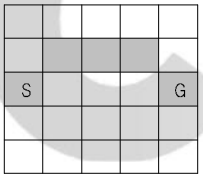
실제 게임 프로그램에서는 재귀적인 방법이나 스택구조를 사용하기 때문에 속도가 떨어져 거의 사용하지 않는다.

2.2.2 너비 우선 탐색

너비 우선 탐색(Breadth First Search)은 맵상에서 사용하는 알고리즘 중에서 가장 기본적인 알고리즘이다. 너비 우선 탐색은 현재 노드에서 한 칸 거리에 있는 모든 노드들을 먼저 처리하고[5], 다음에 두 칸, 세 칸 이런 식으로 길찾기에 성공하여 목표 노드에 도착할 때까지 하게 된다. 이와 같은 과정을 정리하면 다음과 같다.

A. 현재 노드를 큐에 추가한다.
B. 현재 노드를 체크한다.
C. 큐에서 처음 노드를 뽑고 이 뽑힌 노드는 현재 노드가 된다.
   ① 현재 노드가 목표 노드이면 종료한다.
   ② 현재 노드의 자식들 중에서 표시되지 않으면, 그 노드들의 이전 포인터를 현재 노드로 설정하고 그 노드들을 큐에 추가한다.
D. 알고리즘이 종료되거나 큐가 비지 않으면 C를 계속 반복한다.

이 알고리즘을 적용하면 최종 경로를 찾는 것은 문제가 되지 않는다. 그러나 문제는 속도가 느리고 맵상에서 수평, 수직, 대각을 모두 같은 값으로 인식하여 (그림 6)과 (그림 7)은 같은 것으로 인식한다. 물론 깊이 우선 탐색도 같은 문제가 발생하게 된다. 속도의 문제는 완전하다고는 할 수 없지만 A* 알고리즘이 해결하고 대각의 문제는 다음과 같이 해결하면 된다[2].



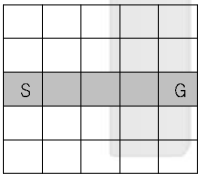


그림 6. 대각 길찾기 그림 7. 일반적인 길찾기
Figure. 6 The Opposite Angle Path-finding
Figure. 7 General Path-finding

대각의 문제를 해결하는 방법은 그림 8처럼 중심을 기준으로 거리 값을 주고, 거리 값이 적은 부분부터 찾게 되면 문제를 해결할 수 있다.

2.828	2.414	2.0	2.414	2.828
2.414	1.414	1.0	1.414	2.414
2.0	1.0	0.0	1.0	2.0
2.414	1.414	1.0	1.414	2.414
2.828	2.414	2.0	2.414	2.828

그림 8. 거리 값이 있는 노드
Figure. 8 The node with distance value

대각 값은 피타고라스의 정리에 의해 구했다.

2.2.3 A* 알고리즘

A* 알고리즘은 g(Goal, 골), h(Heuristic, 휴리스틱), f(Fitness, 적합도)의 3가지 속성[6]을 가지고 있다. g, h, f의 목적은 현재 노드까지의 경로가 얼마만큼이나 가능성을 가지고 있는지를 평가하기 위한 것이다. f를 구하기 위해서는 g의 값과 h의 값을 구해야 한다. g는 현재 노드까지 온 거리를 계산하고 h는 정해진 값이 아니라 추정치[7]를 계산하여야 한다.

A*는 오픈 리스트(Open List)와 클로즈 리스트(Close List)를 사용한다. 오픈 리스트는 아직 탐색하지 않은 노드들을 클로즈 리스트는 이미 탐색한 노드[6]들로 구성되어 진다. 알고리즘을 정의하면 다음과 같다.

A. 시작 노드를 현재로 하고 g, h, f를 배정한다.
B. 현재 노드를 오픈 리스트에 추가한다.
C. f값이 가장 작은 노드를 ㉠라 한다.
  a. ㉠이 목표 노드이면 종료한다.
  b. 오픈 리스트가 비었으면 종료한다.
D. ㉠에 연결된 유효한 노드를 ㉡라 한다.
  a. ㉡에 g, h, f 값들을 배정한다.
  b. ㉡이 오픈 리스트나 클로즈 리스트에 들어 있는지 점검한다.
    i. 만일 들어있다면, 새 경로가 f값이 작은지 점검하여 작다면 경로를 갱신한다.
    ii. 들어있진 않다면 ㉡을 열린 목록에 추가한다.
  c. 단계 D를 ㉠에 연결된 모든 유효한 자식 노드들에 대해 반복한다.
E. C부터 다시 반복한다.

A*가 길찾기에 많이 쓰이는 이유는 깊이 우선 탐색과 너비 우선 탐색보다 훨씬 빠르며 이 둘의 단점도 가지고 있지 않기 때문이다. 실제 게임에서 원하는 목적 노드를 빨리 찾을 뿐만 아니라 길이도 최단 경로가 된다. 그러나 A*는 중요한 문제점을 가지고 있다. 출발 노드에서 현재 노드까지의 경로는 알고 있으나 현재 노드에서 목적 노드를 찾기 위해서는 경로의 길이를 추정하는 방법밖에 없다는 것이다. 그러나 그 추정값들을 계산하여 가장 적은 값을 찾아가게 된다. (그림 9)를 이용하여 추정값을 구한다.

먼저 실선의 추정값을 구하면 다음과 같다.
- 5.09 + 3.60 = 8.69

점선의 추정값을 구하면 다음과 같다.
- 6.32 + 3.60 = 9.92

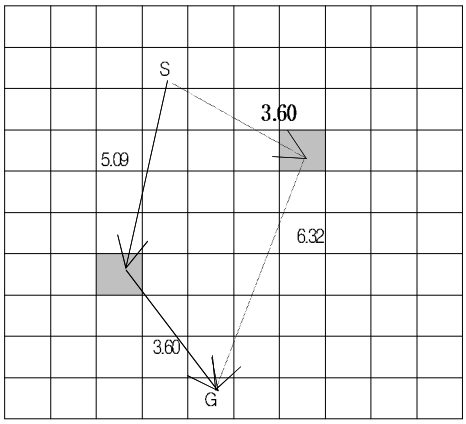


그림 9. 추정값 구하기
Figure. 9 Calculating the presumed value

점선보다는 실선의 추정값이 작으므로 출발 노드에서 목표 노드까지의 이동은 실선 방향으로 이동하게 된다.

III. 실험에 의한 비교 분석 및 활용

3.1 비교 분석

깊이 우선 탐색과 너비 우선 탐색은 길찾기 탐색에서 많이 뒤떨어지므로 깊이 우선 탐색을 개선한 거리 값을 준 탐색 방법 (그림 10)과 A* 알고리즘을 비교한다.

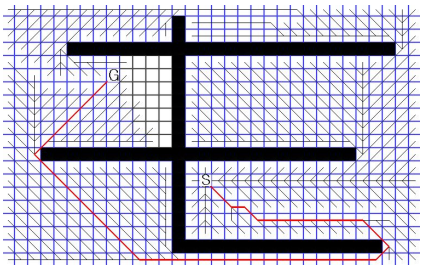


그림 10. 거리 값을 준 탐색
Figure. 10 Search with given Distance Value

다음에는 길찾기에서 가장 많이 사용하고 검색 시간이 빠른 A* 탐색 알고리즘의 탐색 과정이다(그림 11).

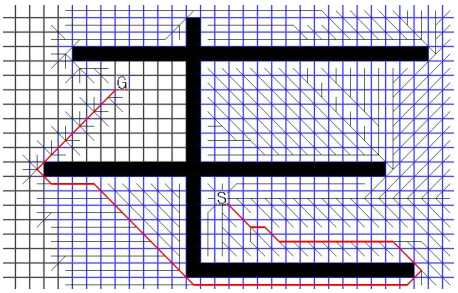


그림 11. A* 탐색
Figure. 11 A* Search

두 그림을 비교하면 똑같은 방법으로 출발 노드에서 목표 노드까지 찾아갔지만 A*는 더 적게 검색 한 것을 알 수 있다. 이 비교는 단순한 비교이지만 실제 게임에서의 지형을 이용해도 A* 알고리즘이 빠른 것을 보여 준다.

3.2 A*의 활용

A* 알고리즘을 이용하여 실제 활용하기 위해서는 다음과 같은 조건이 있다[2]. 첫 번째, 옆 노드로 이동하는 방식인데 맵의 표현이나 방식에 따라 서로 달라진다. 일정한 사각형의 맵으로 이루어 졌다면 네 개의 옆 노드가 나올 것이다. 육각형의 맵으로 이루어 졌다면 여섯 개의 옆 노드가 나올 것이다. 두 번째는 비용에 대하여 알아야 한다. 비용은 보통 이동거기, 이동시간, 이동에너지 등을 말한다. 예를 들어 유닛이 산을 오르거나 바다를 건너면 많은 비용을 들게 하고 평면을 이동하면 적은 비용을 들게 한다. 본 논문에서는 거리만 사용하였다. 세 번째는 목표를 찾기 위해서는 추정으로 밖에는 알 수 없다. 가장 좋은 방법은 유닛의 현 위치에서 목표지점까지 직선으로 연결하는 방법이지만 여러 가지 비용 변화 요인들이 계산에 포함되기 때문에 복잡해져 직선으로 찾을 수 있다하더라도 검색 속도가 느리면 다시 한번 생각해야 할 것이다. 목표 노드를 향한 올바른 방향 가중치(값을 숫자로 표기)가 존재하지 않으면 A* 알고리즘은 모든 방향을 검색한 후에 원하는 목표 노드를 찾아가게 될 것이다. 그러나 가중치를 부여하게 되면 불필요한 방향을 검색하지 않고 목표 노드를 향하게 되어 시간과 메모리에 모두 도움이 된다. 네 번째는 다중의 목표가 존재할 때, 당연한 이야기겠지만 가장 가까운 목표 노드나 최적의 노드를 택해야 한다. 다섯 번째는 알고리즘의 단점을 잘 파악하여 약점에 잘 대응하여야 한다. A* 알고리즘은 오픈 리스트나 클로즈 리스트에 많은 노드들이 존재하게 되면 메모리를 많이 소비하게 되고 CPU의 사용 시간을 낭비하게 된다. 특히 목표 노드가 존재하지 않을 경우 모든 맵을 검사한 후 목표가 없는 것을 알게 됨으로 미리 맵에 정보를 추가하는 것이 좋다.

이제는 A* 알고리즘을 이용하여 직접적인 경로를 만들어 본다. 직접적인 경로를 만들기 위해서는 계통적인 경로 만들기(Hierarchical Path)[2]를 알아야 한다. 계통적인 경로 만들기는 큰 범위 공간에서 경로를 찾은 후 그 보다 하위 범위 공간들에서 경로를 찾는 방식이다. 먼저 4개의 지역이 존재하는데 유닛이 1번 지역에서 출발하여 2번, 3번을 지나 4번 지역에 도착한다. 밑에 나와 있는 (그림 12)를 보면 이해가 될 것이다.

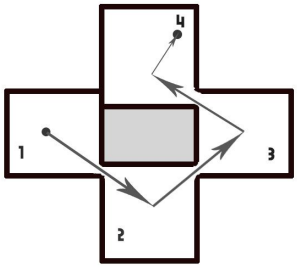


그림 12. 1번에서 4번까지의 단순한 경로
Figure. 12 Simple Path from Number 1 to 4

(그림 12)는 어딘가 모르게 자연스럽지 못하다는 느낌이 든다. 계층적 경로 방법은 A* 알고리즘을 이용하여 각 지역들 사이의 경로를 찾은 후 경로가 거쳐 나가는 각 지역 안에서 경로를 찾는 것이다. 이렇게 되면 실제로 사람들이 이동하는 경로와 비슷하게 될 것이다. 이 방법은 다음 입구 너머의 입구를 목표로 설정해서 경로를 검색한다는 것으로 유닛이 첫 번째 입구를 지나면 나머지 경로는 버리고 다음 입구 너머의 목표를 검색한다. 이 방법은 시각적으로 자연스러운 경로를 얻을 수 있지만 찾은 경로의 반을 버려야 하므로 속도에 약간은 문제가 존재한다. 다음은 실제 동작하는 과정을 나열하였다.

① 지역에서 지역으로 A*를 적용한다.
② 지역 1, 2, 3, 4를 차례로 이동하려 한다.
③ 시작에서 부목표1을 찾는다(그림 13).
④ 유닛을 지역2 입구까지 이동한다.
⑤ 나머지를 버리고 부목표2를 찾는다(그림 14).
⑥ 유닛을 지역3 입구까지 이동한다.

⑦ 나머지를 버리고 부목표3으로 이동한다(그림 15).
⑧ 유닛을 지역4 입구까지 이동한다.
⑨ 나머지를 버리고 목표를 찾는다.
⑩ 유닛의 목표로 이동한다(그림 16).

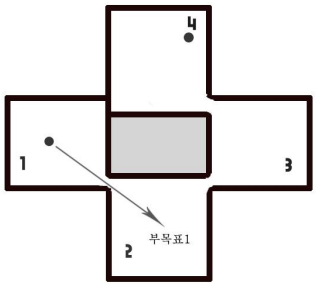


그림 13. 부목표1로 접근
Figure. 13 Approach to Sub-Goal 1

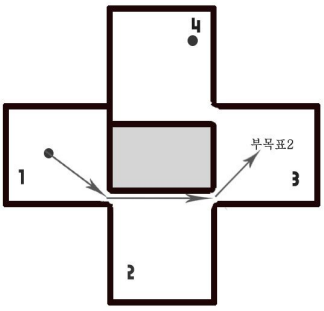


그림 14. 부목표2로 접근
Figure. 14 Approach to Sub-Goal 2

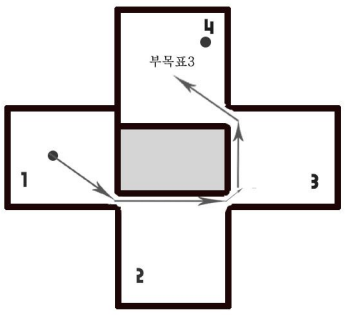


그림 15. 부목표3으로 접근
Figure. 15 Approach to Sub-Goal 3

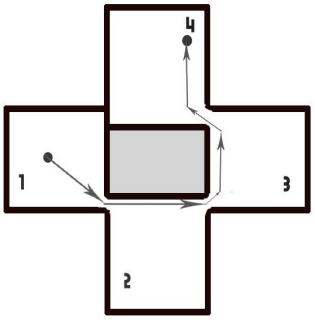


그림 16. 최종 목표에 접근
Figure. 16 Approach to the Final Goal

A*를 이용한 계통적인 경로 만들기로 목표를 탐색하니 일반적인 길찾기 방법보다는 사람들의 실제 움직임과 같다.

Ⅳ. 결론

다양한 길찾기 알고리즘을 제시하였고, 가장 속도가 빠른 A* 알고리즘에 대하여 시뮬레이션 하였다. 깊이 우선 탐색과 너비 우선 탐색 그리고 A* 알고리즘을 비교 분석하였다. 또한 A* 알고리즘을 이용하여 직접적인 경로 탐색을 제안하였다. A* 알고리즘이 완벽한 것은 아니다. 상황에 따라서는 아주 느려 질수도 있고 있기 때문에 그 상황에 따라 적절히 변경해야하는 변경 A* 알고리즘도 많다. 또한 너무 어렵다는 것이 문제가 되기도 한다. 물론 게임작성의 중요도로 따지면 그래픽, 사운드, 스토리 등에 비하여 길찾기 알고리즘은 순위 밖의 내용일 수도 있다. 그러나 사용자들의 재미를 위하여 또한 현실감을 위하여 길찾기 알고리즘은 꼭 필요하다. 앞으로의 과제는 A*를 개선하여 완전한 알고리즘을 개발하여 게임에 적용하는 일이다.

참고문헌

[1] Jonathan S. Harbour, DirectX 비주얼 베이직 게임 프로그래밍, 정보문화사, 예승철 역, pp 38-42, 2003
[2] Ron Penton, DATA STRUCTURES FOR GAME PROGRAMMING, 정보문화사, 류광 역, pp 789-870, 2002
[3] Mark Deloura, GAME PROGRAMMING GEMS 2, 정보문화사, 류광 역, pp 362-370, 2002
[4] Mark Deloura, GAME PROGRAMMING GEMS, 정보문화사, 류광 역, pp 340-361, 2001
[5] 이상용, 인공지능, 상조사. 77-87, 2003
[6] Steve Rabin, AI GAME PROGRAMMING WISDOM, 정보문화사, 류광 역, 189-247, 2003
[7] Dante Treglia, GAME PROGRAMMING GEMS 3, 정보문화사, 류광 역, pp 369-380, 2003

저 자 소 개

이 세 일
1993년 대전공업대학교 전자계산학과 졸업(공학사)
2001년 청운대학교 대학원 전산전자정보공학과(공학석사)
2004 ~ 현재
공주대학교 대학원 컴퓨터공학과 (박사과정)
<관심분야> Ubiquitous Computing, Context Awareness, 신경망, 게임 알고리즘