7주차 2차시 기하알고리즘 이해

【학습목표】

1. 기하알고리즘을 이해할 수 있다.
2. 볼록 껍질을 이해할 수 있다.

학습내용1 : 기초적인 기하알고리즘

기하학의 기본요소로는 점, 선, 다각형, 다면체가 있다. 기하문제를 풀기 위해서는 기본요소인 점, 선, 다각형, 다면체등을 어떻게 표현할 것 인가를 결정해야 하는데 간단한 표현법으로 좌표를 사용한다. 기하 문제는 우리가 직접 종이에 그려서 간단하게 풀 수 있는 문제도 컴퓨터를 사용하여 풀려고 하면 그리 쉽지 않은 경우가 많은데 예를 들면, 종이에 그려진 두 직선이 교차하는지의 문제는 사람은 쉽게 결정할 수 있지만, 컴퓨터로 교차하는지를 알아보려면 복잡한 여러 가지 연산을 거쳐야 하기 때문이다.

1. 기하알고리즘(geometric algorithm)

기하 문제를 푸는 알고리즘으로서 점, 선, 다각형과 관련되어 있는 그와 같은 문제를 풀기 위한 알고리즘으로 아래와 같은 내용을 살펴본다.

☞ 두 선분이 서로 교차하는지?
☞ 여러 개의 점들을 꼭지점으로 하는 단순 폐쇄 다각형 만들기.
☞ 주어진 점이 다각형 내부에 존재 하는지?
☞ 주어진 점들을 둘러싸는 가장 작은 볼록 다각형.
☞ 주어진 점들 중에서 최단 경로 찾기.

* 기하요소의 표현 방법
① 점(1차원 물체)
☞ x축과 y축의 좌표로 표현 →(x, y)

```
struct point{
        int x;                  /* 점의 x좌표 */
        int y;                  /* 점의 y좌표 */
  } Firstpoint, SecondPoint;   /* 변수들 */
```

② 선분
☞ 양 끝점의 좌표로 표현
☞ 두 점은 직선으로 연결

```
struct line {
struct point pt1;
struct point pt1;
 } TestLine;
```

③ 다각형

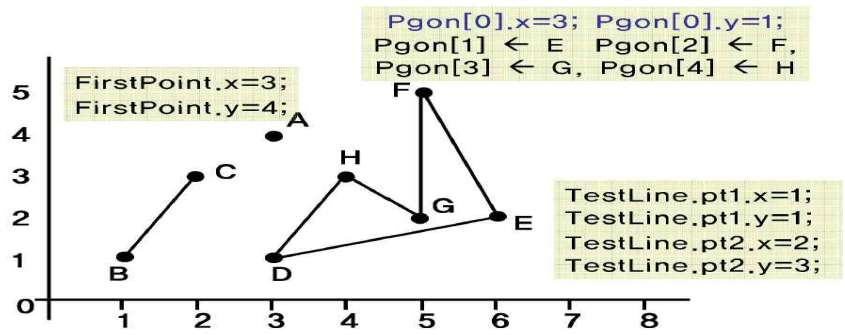
☞ 점의 집합으로 표현

☞ 이웃한 점들은 직선으로 연결

☞ 시작점과 끝점도 직선으로 연결된 닫힌 모양의 도형

struct point Pgon[n];

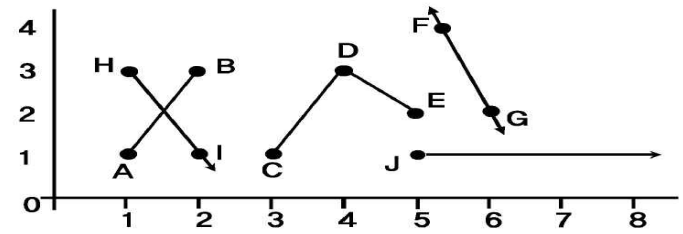
* 기하요소의 표현



* 기본용어

선분 AB 꺾은선 CDE 무한 직선 FG

반직선 HI 반직선 Jx



◎ 기본용어 정의

☞ 선분 AB

양 끝 점이 A와 B인 선분

☞ 꺾은선 CDE

점 C에서 시작하여 점 D를 지나 점 E에 도착하는 꺾은선

☞ 무한직선 FG

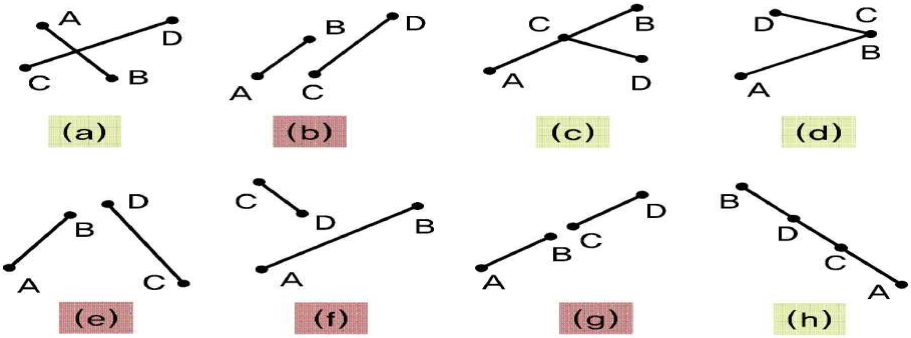
점 F와 G를 지나는 양방향으로 무한한 직선

☞ 반직선 Jx

점 J에서 시작하여 점 x축과 평행한 반직선

* 두 선분의 교차 검사

두 선분이 주어졌을 때, 이 둘이 서로 교차하는지 아닌지를 검사하는 것은 가장 기본적인 기하 알고리즘 중의 하나이다. 여기서 두 선분이 교차한다는 것은 그 두 선분이 적어도 한 점을 서로 공유함을 말하는 것 이다. 우선 두 선분이 직교 좌표 평면에 놓여 있을 수 있는 여러 가지 경우들을 살펴보면 다음과 같다.



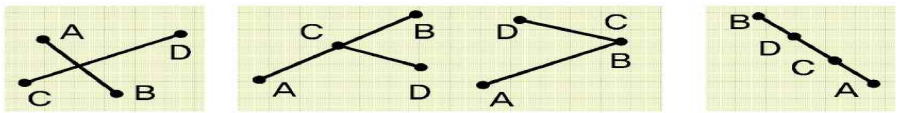
* 두 선분의 교차 여부를 검사하는 방법

① 방법1

☞ 각 선분에 대하여 그 선분을 포함하는 양방향으로 무한한 연장선을 긋고 그 두 연장선의 일차방정식을 구한 후에 그로부터 교점을 계산

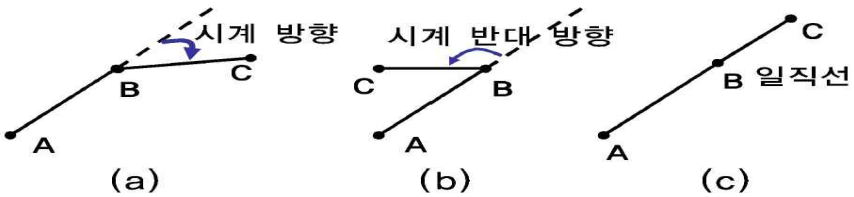
☞ 교점이 원래의 두 선분상에 있으며 두 선분은 교차.

② 방법2



- 위의 교차하는 경우를 살펴보면
  - 선분AB를 기준으로 볼 때 점C와 점D가 서로 반대편에 존재 → **'꺾은선의 방향 조사'**
  - 한 선분의 끝점이 다른 선분 상에 존재
  - 일직선상에 있으면서 한 선분의 끝점이 다른 선분 상에 존재

* 꺾은선 ABC의 꺽이는 방향



두 점 C,D가 선분 AB를 기준으로 하였을 때 서로 반대편에 있으려면?

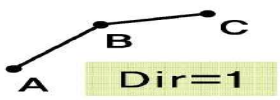
➔ 꺾은선 ABC 방향 ≠ 꺾은선 ABD의 방향

* 꺽은선 ABC의 방향을 결정하는 함수를 정의 (Direction[A, B, C])

⊙ A, B, C가 일직선상에 있지 않을 때, 꺽은선 ABC의 방향

☞ 시계 방향인 경우 : 1

☞ 시계 반대 방향인 경우 : -1



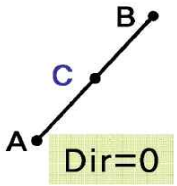
* 점 A, B, C가 일직선상에 있을 때

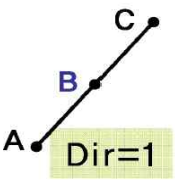
☞ C가 가운데 있는 경우 : 0

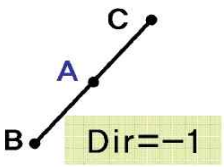
C=A 또는 C=B인 경우

☞ B가 가운데 있는 경우 : 1

☞ A가 가운데 있는 경우 : -1





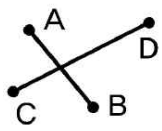


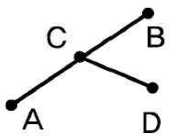
* 두 선분의 교차 조건

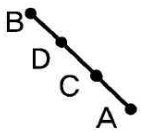
$$Direction(A,B,C) \times Direction(A,B,D) \leq 0$$

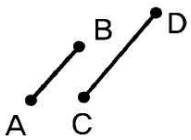
AND

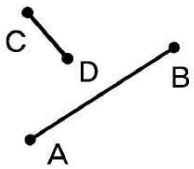
$$Direction(C,D,A) \times Direction(C,D,B) \leq 0$$







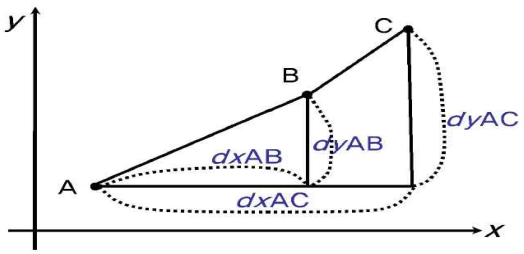




* 두 선분의 교차검사 알고리즘

```
int Intersection (struct line AB, struct line CD) {
  int LineCrossing;
  if ((Direction (AB.A,  AB.B,  CD.C)  *
       Direction (AB.A,  AB.B,  CD.D) <= 0)  &&
      (Direction (CD.C,  CD.D,  AB.A) *
       Direction (CD.C,  CD.D,  AB.B) <= 0) )
     LineCrossing = TRUE;       /* TRUE → 교차*/
  else LineCrossing = FALSE;    /* 교차하지 않음*/
  return LineCrossing;
}
```

* 선분의 기울기(꺽은선의 방향)



선분AB 기울기 $\frac{dyAB}{dxAB} \times (dxAB \times dxAC) = dyAB \times dxAC$

선분AC 기울기 $\frac{dyAC}{dxAC} \times (dxAB \times dxAC) = dxAB \times dyAC$

선분 AB에서 점 C 의 방향 (시계 방향인지 그 반대 방향인지 구하는 방법)
경사도 AB 와 경사도 AC 를 비교해서 AC 가 더 크면 시계 반대방향 이고 경사도 AB 가 더 크면 시계방향 이 된다.
경사도AB 는 (dyAB / dxAB) 이고
(y 의 증가분을 x 의 증가분으로 나눈 값이 선분의 기울기이다.)
경사도AC 는 (dyAC / dxAC) 이다.
경사도AB 는 (dyAB / dxAB)와 경사도AC 는 (dyAC / dxAC)에 (dxAB) * (dxAC) 를 곱하면 경사도AB (dyAB) * (dxAC) 가 되고 경사도 AC 는 (dxAB) * (dyAC) 가 된다.
(dxAC) * (dyAB) 더 크면 시계방향이고 (dxAB) * (dyAC) 더 크면 시계 반대 방향이 된다.

```
int Direction (struct point A, struct point B, struct point C) {
/* 입력: A, B, C : 세 점의 좌표
     출력: Dir의 값
         ① A, B, C가 일직선상에 있지 않을 때, 꺾은 선 ABC의 방향이
                - 시계 방향인 경우 : 1
                - 시계 반대 방향인 경우 : -1
         ② 점 A, B, C가 일직선상에 있을 때
                - C가 가운데 있는 경우 : 0
                - B가 가운데 있는 경우 : 1
                - A가 가운데 있는 경우 : -1   */
  int dxAB, dxAC, dyAB, dyAC, dir;
  dxAB = B.x - A.x;   dyAB = B.y - A.y;
  dxAC= C.x - A.x;    dyAC = C.y - A.y;
  if(dxAB * dyAC < dyAB * dxAC) dir = 1;/* 시계 방향 */
  if(dxAB * dyAC > dyAB * dxAC) dir = -1;/* 시계 반대 방향 */
  /* 일직선상에 있는 경우 */
  if(dxAB * dyAC == dyAB * dxAC)  {
    if(dxAB == 0 && dyAB == 0) dir = 0;    /* A=B */
    if((dxAB * dxAC < 0) || (dyAB * dyAC < 0)) dir = -1;
    else if(dxAB * dxAB + dyAB * dyAB >= dxAC * dxAC + dyAC *
dyAC) dir=0;
    else dir = 1;
    }
  return dir;
  }
```

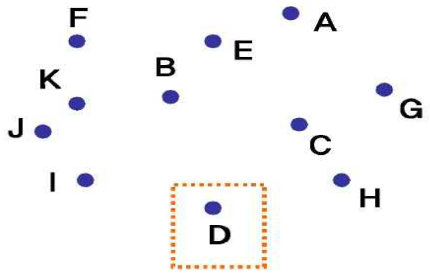
2. 단순 폐쇄 경로

임의의 기준점으로부터 다른 각각의 점까지의 각도를 구한 후 올림차순으로 정렬한 다음 기준점부터 정렬된 순서대로 이으면 그것이 단순 폐쇄경로가 된다. 기준점이 다르면 생성된 단순 폐쇄경로도 다르다.
예를 들어 한명의 택배원이 택배물을 배달하는데 하루에 동안 n개의 집을 방문 할 때 가장 효율적인 경로를 생각 해 보면 택배원이 방문한 집들을 순서대로 직선으로 연결 하여 교차 하지 않도록 하면 한번통과 한 지점은 다시 통과 하지 않으므로 최적의 경로에 가까울 것이다. 이렇게 여러 개의 점이 주어졌을 때 이를 꼭지점으로 하면서 변이 서로 교차하지 않도록 하는 것을 단순 패쇄 경로라 한다.

① n개의 점이 주어 졌을 때 이 점들을 모두 경유하고 출발점으로 되돌아오는 비교차 경로를 말한다,
  ☞주어진 점을 꼭지점으로 하면서 변이 서로 교차하지 않도록 하는 경로/
② 단순 다각형
  ☞ 단순 폐쇄 경로에 의해서 만들어진 다각형.
③ 접근방법
  ☞ 기준점과의 각도 계산과 정렬 연산을 이용한다.

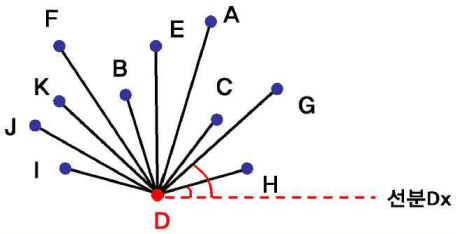
* 단순 다각형 구하는 과정

과정 1)



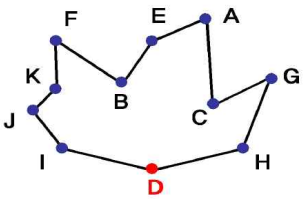
기준점 선정

과정 2)



1. 기준점과 그 외의 각 점을 선분으로 연결
2. 선분 Dx와 각 선분과의 각도 계산
3. 각도를 기준으로 점들을 오름차순으로 정렬

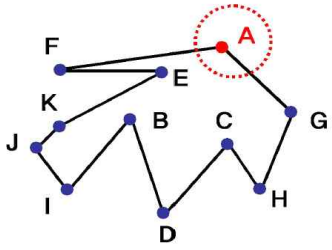
과정 3)



기준점으로부터 시작하여 정렬된 점들을 순서대로 직선으로 연결하고 나서 다시 기준점으로 돌아옴 → 단순 폐쇄 경로, 단순다각형

과정 4)

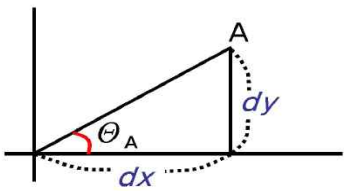
기준점이 다르면 생성된 단순 폐쇄경로도 다르다.



기준점 D가 아니라 A로 선정한 경우에는
다른 모양의 단순 다각형을 형성

3. 점의 각도 계산

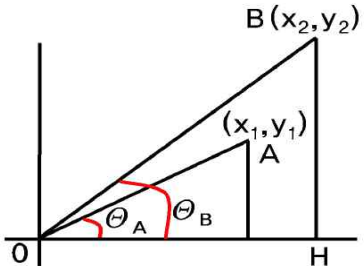
실제 각도 Θ_A



$$\tan\theta_A = \frac{dy}{dx} \qquad \theta_A = \tan^{-1}\frac{dy}{dx}$$

- 실제 각도가 필요한 것이 아니라 상대 각도가 필요하다.

상대 각도 T_A

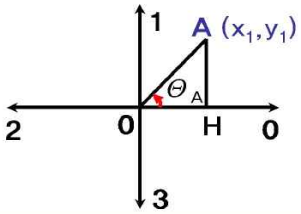


$$\tan\theta_A < \tan\theta_B$$

$$\rightarrow \frac{dy_1}{dx_1} < \frac{dy_2}{dx_2}$$

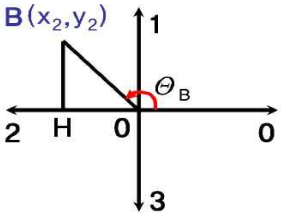
$$\rightarrow \frac{dy_1}{dx_1 + dy_1} < \frac{dy_2}{dx_2 + dy_2}$$

$0 \le T_A \le 1$ $\quad T_B$



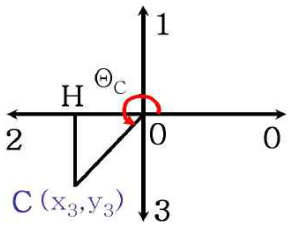
$T_A=|dy_1|/(|dx_1|+|dy_1|)$

점A가 1사분면에 있는 경우



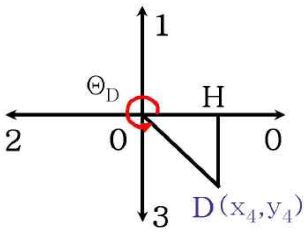
$T_B=2-|dy_2|/(|dx_2|+|dy_2|)$

점B가 2사분면에 있는 경우



$T_C=2+|dy_3|/(|dx_3|+|dy_3|)$

점C가 3사분면에 있는 경우



$T_D=4-|dy_4|/(|dx_4|+|dy_4|)$

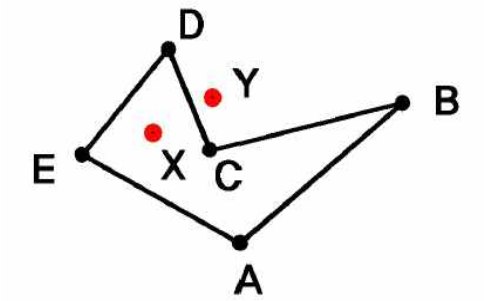
점D가 4사분면에 있는 경우

* 점의 상대 각도 계산 알고리즘

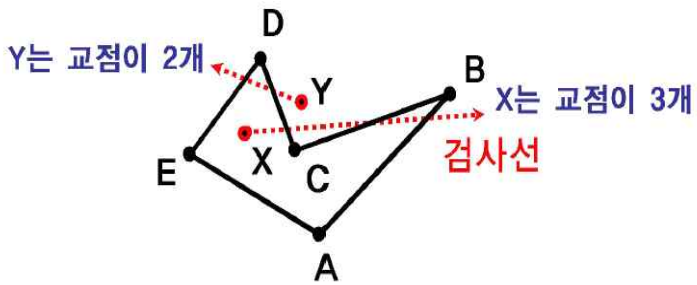
```
float ComputeAngle(struct point A, struct point B) {
/* 출력: 선분 AB와 반직선 Ax가 이루는 상대 각도 */
  int Dx, Dy;  float Angle;
  Dx=B.x-A.x;  Dy=B.y-A.y;
  if ((Dx>=0) && (Dy==0)) Angle= 0;
  else {
     Angle=abs(Dy) / (abs(Dx)+abs(Dy));
     if ((Dx<0) && (Dy>=0)) Angle=2-Angle;
     else if ((Dx<=0) && (Dy<0)) Angle=2+Angle;
       else if ((Dx>0) && (Dy<0)) Angle=4-Angle;
  }
  return(Angle * 90.0);  /* 0.0~360.0 사이의 값 */
}
```

* 점과 다각형의 상대위치 검사

- 점과 다각형이 주어졌을 때 그 점의 위치가 다가형의 내부인지 외부인지를 결정하는 문제.



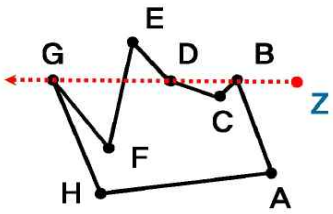
- 점에서 임의의 방향으로 그은 반직선이 다각형의 변과 교차하는 점 개수를 조사.



교점의 개수가 홀수 → 다각형 내부

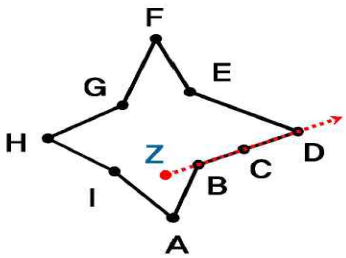
교점의 개수가 짝수 → 다각형 외부

- 다가형의 꼭지점을 지나는 경우.



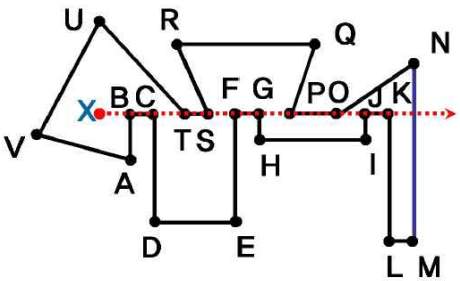
현재 검색 중인 변의 한 꼭지점이 검사선에 닿으면
→ 그 점을 무시 → 다음 꼭지점과 현재 변의 다른 꼭지점이 검사선을 중심으로 같은 편에 있는지 여부를 검사
➔ 같은 편에 있으면 어느 점도 지나지 않은 것으로 간주
➔ 다른 편에 있으면 다각형의 변과 교차한 것을 간주

- 한 변 전체를 통과하는 경우



다각형의 꼭지점을 지나는 경우와 동일하게 처리

- 여러변을 스치며 자나는 경우.



변 MN에서 한 번 교차하기 때문에 X는 내부에 존재

* 점과 다각형의 상대위치 검사 알고리즘

```
int IsPointInside(struct point A,
                  struct point P[ ], int n)  {
 int Count, i, LastPoint;
 struct line TestLine, PolygonLine;
 int PointOnTestLine;
 Count=LastPoint=0; PointOnTestLine=FALSE;
 TestLine.p1 = A;  TestLine.p2 = A;
 Testline.p2.x = MAXINT;
 for (i = 1;  i <= n;  i++)  {
   PolygonLine.p1 = PolygonLine.p2 = P[i];
   /* 다음 슬라이드 */
 }
 return( (Count mod 2) == 1 );
}
```

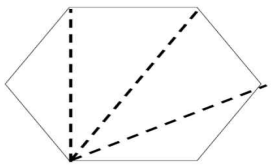
```
 if (Intersection(TestLine,  PolygonLine) )
   PointOnTestLine = TRUE;
 else  {
   PolygonLine.p2 = P[LastPoint]; LastPoint = i;
   if (!PointOnTestLine)  {
     if (Intersection(PolygonLine, TestLine)) Count++;
   } else  {
     if (Direction(TestLine.p1, TestLine.p2,
                                  PolygonLine.p1) *
         Direction(TestLine.p1, TestLine.p2,
                                 PolygonLine.p2) < 0) {
          Count++;
     }
     PointOnTestLine = FALSE;
  }
 }
```

학습내용2 : 볼록 껍질

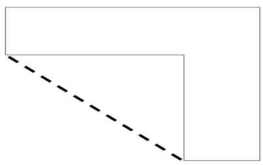
1. 볼록 껍질 찾기

임의의 집합 X에 대한 볼록 껍질이란 X를 포함하는 가장 작은 볼록 집합을 말하며 여러 개의 점이 주어졌을 때 그 점들을 모두 포함하는 가장 작은 볼록 다각형을 그 점집합의 볼록 껍질 (Convex Hull)이라고 한다. 볼록 껍질에는 짐꾸러기, 그레이엄 알고리즘이 있으며, 평균 $O(n^2)$의 실행시간을 갖는다.

⦿ 볼록 다각형
☞ 다각형 내부의 임의의 두 점을 연결하는 선분이 반드시 다각형 내부에 존재하는 다각형.
⦿ 볼록 껍질(Convex Hull)
☞ 점집합의 모든 점을 포함하는 최면적의 볼록 다각형
☞ 주어진 점들을 모두 둘러싸는 최소길이의 경로.
⦿ 볼록 껍질의 꼭지점
☞ 볼록 껍질 외부의 임의의 직선을 껍질 쪽으로 접근시킬 때 이 직선이 처음으로 만나는 점.

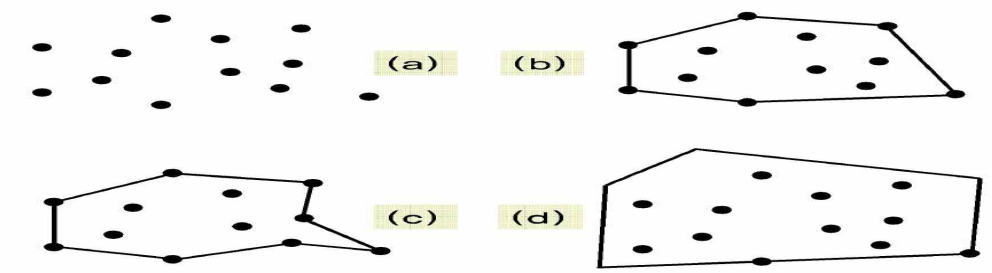


볼록 다각형



오목 다각형

⦿ 점과 볼록 껍질



(a)에 주어진 점집합에 대하여서 (b)는 볼록 껍질이다. 하지만 (c)는 오목 다각형이어서 볼록 껍질이 될 수 없으며 (d)는 최소의 볼록 다각형이 아니므로 볼록 껍질이 될 수 없다. 볼록 껍질은 점들을 모두 둘러싸는 최소 면적의 볼록 다각형이며 볼록 껍질의 꼭지점은 반드시 주어진 점들의 일부이어야 한다. 꼭지점이 아닌 점들은 볼록 껍질의 내부에 속하여야 한다.

* 단순한 방법

점을 하나씩 추가해 나가면서 구하는 방법

- k개의 점에 대한 볼록 껍질을 구했고, k+1번째 점까지를 포함한 볼록 껍질을 구하는 경우

두 가지 작업

- 새 점이 볼록 껍질 내부/외부에 존재하는 지를 결정 **점이 다각형 안에 있는지 확인**
- 밖의 점이라면 해당 점까지 포함하도록 볼록 껍질을 확장 **볼록 껍질의 확장**

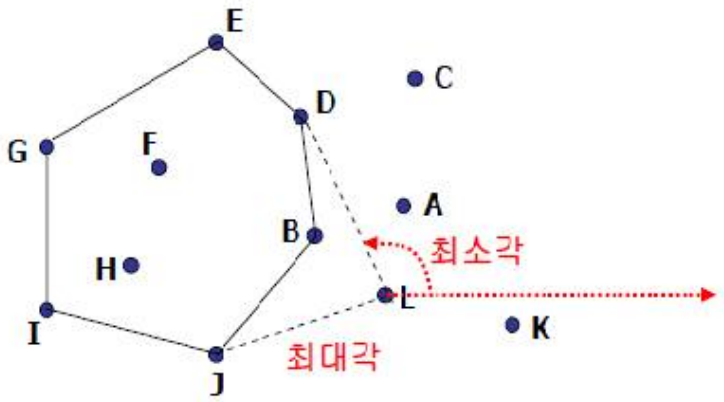
좌표가 최대/최소인 점을 항상 선택하면

- 다각형 안의 점인지를 확인할 필요 없음

➔ 항상 x좌표가 최소인 점을 선택

- x값이 같은 경우 y좌표가 최소인 것 선택

새 점과 현 볼록 껍질의 점들 중에서 최소각의 점과 최대각의 점을 구하고, 이 두 점 사이의 점들은 제거하고 이 두 점과 새점을 연결



【학습정리】

1. 두 선분의 교차 검사
- 교차는 두 선분이 공유하는 점이 두 선분 상에 하나 이상 존재하는 경우를 의미
- 교차 검사 방법 (선분AB와 선분CD에 대해서)
→ 각 선분을 중심으로 다른 선분의 양 끝점이 서로 다른 방향에 있는 지를 검사, 즉 두 점 C와 D가 선분AB를 기준으로 서로 반대편에 있는 지를 검사한다.
→ 서로 반대편에 있는 지를 검사하기 위해서 꺾은선ABC의 방향과 꺾은선ABD의 방향이 서로 다른 지를 조사하면 된다.
→ 꺾은선ABC의 방향을 조사하기 위해서는 선분AB의 기울기와 선분AC의 기울기를 비교한다. 만약 선분AB의 기울기(dyAB*dxAC)가 선분 AC의 기울기(dxAB*dyAC)보다 작으면 꺾은선ABC는 시계반대방향이고, 크면 시계방향이며, 같은 기울기를 갖는 경우는 일직선상에 존재하는 것을 의미한다.

2. 단순 폐쇄 경로 찾기
- 단순 폐쇄 경로란 n개의 점이 주어졌을 때 이 점들을 모두 경유하고 출발점으로 되돌아오는 비교차 경로 (주어진 점을 꼭지점으로 하면서 변이 서로 교차하지 않도록 하는 경로)
→ 단순 다각형: 단순 폐쇄 경로에 의해서 만들어지는 다각형
→ 단순 폐쇄 경로를 찾기 위해 필요한 연산: 기준점과의 각도 계산, 정렬
- 방법
① Y좌표가 최소인 점들 중에서 그 X좌표가 최소인 D를 기준점을 선정
② 선분Dx와 각 선분의 각도를 계산한 후, 이를 기준으로 점들을 오름차순으로 정렬
③ D로부터 시작하여 정렬된 점

3. 점과 다각형의 상대 위치 검사
- 점과 다각형이 주어졌을 때 그 점의 위치가 다각형의 내부인지 외부인지를 결정하는 문제
- 검사 방법
→ 점에서 임의의 방향으로 그은 반직선이 다각형의 변과 교차하는 점의 개수를 조사한다. 교점의 개수가 홀수이면 다각형 내부, 짝수이면 다각형의 외부에 존재하는 것을 판단한다.
→ 검사선(반직선)이 꼭지점 또는 변을 통과하는 경우에는 검사선에 닿기 직전의 꼭지점과 검사선을 벗어난 직후 처음 만나는 꼭지점이 검사선을 기준으로 서로 같은 편에 있는 지 조사
(같은 편 → 다각형의 어느 점도 지나지 않은 것으로 간주, 다른 편 → 다각형의 한 변을 지난 것(교점이 하나 존재)으로 취급)

4. 볼록 껍질 찾기
- 볼록 껍질이란 점집합의 모든 점을 포함하는 최소 면적의 볼록다각형
- 볼록 껍질을 찾는 방법: 단순한 방법, 짐꾸리기 알고리즘