Geometry algorithms

Damon Wischik, Computer Laboratory, Cambridge University. Lent Term 2022

Contents

1 A* algorithm **1**

2 Segment intersection **3**

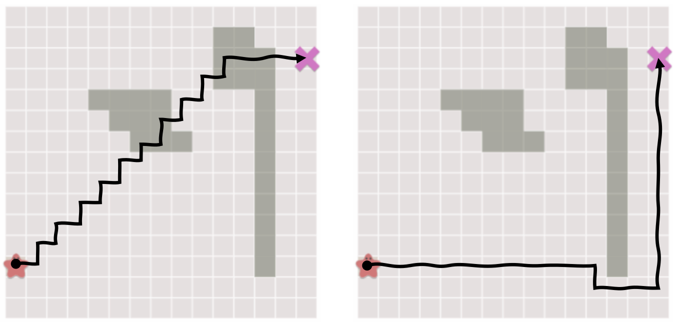
3 Jarvis's march **5**

4 Graham's scan **8**

5 Solutions to exercises **10**

1. A* algorithm

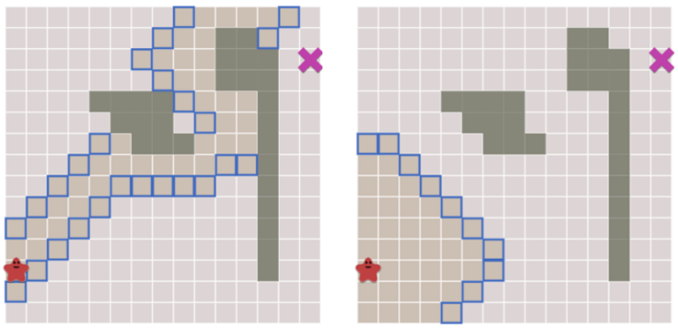
Suppose we're searching for the shortest path between two vertices, on a graph which is embedded in space. In other words, suppose that every vertex has spatial coordinates—meaning that we can measure spatial distances and directions between them. The spatial distance won't be the correct graph distance, since it ignores reachability.



spatial distance = 24 graph distance = 26

Image credit: Red Blob Games[1]

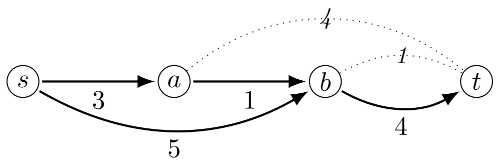
Nonetheless, can we use spatial knowledge to speed up path-finding? A simple greedy strategy is to prioritize the search in the direction of our destination. We can use a priority queue just as in Dijkstra's algorithm, but prioritize vertices by their spatial distance to the destination, in contrast to Dijkstra's algorithm which prioritizes by graph distance from the origin and thus searches blindly in all directions.



greedy search blind search

Greedy search does *not* work, in the following sense: if we stop the search as soon as we reach the destination vertex, we might end up with a suboptimal path.

Here's a simple example to illustrate. The solid lines are graph edges with weights, and the dotted lines show spatial distance to the destination vertex t. The greedy algorithm, starting from s, will first add a and b to the queue, pick b next because it's nearer the destination, add t to the queue, then pop t. So it will stop with the path $s \to b \to t$, when we really want $s \to a \to b \to t$.



[1] There is an excellent tutorial about A^* at Red Blob Games, `http://www.redblobgames.com/pathfinding/a-star/introduction.html`

PROBLEM STATEMENT AND ALGORITHM

The setting is the same as for Dijkstra's algorithm. Consider a directed graph with edge costs ≥ 0, and suppose we're given a start vertex s and a destination vertex t. We seek a shortest path from s to t. 'Shortest' has its usual meaning — the cost of a path is the sum of its edge weights, and we seek a path of minimum cost.

The A^* algorithm requires that we have available an heuristic distance function $h(v \text{ to } t)$. (This might for example be the straight-line distance from v to t.) It proceeds exactly like Dijkstra's algorithm, except that it uses a different key to sort the priority queue:

We must use the version of dijkstra as given in lecture ntoes, which allows vertices to re-enter toexplore after they've been popped. (Some textbooks present a different version of Dijkstra's algorithm, which does not allow re-entry.)

$$\texttt{key}(v) = v.\texttt{distance} + h(v \text{ to } t)$$

where $v.$`distance` is its best guess so far for the distance from s to v. It terminates as soon as it pops t.

Theorem (Correctness). *Assume that the heuristic distance is always less than or equal to the graph-theoretic distance, i.e. that*

$$h(v \text{ to } t) \leq d(v \text{ to } t) \qquad \text{for all } v$$

where $d(v \text{ to } t)$ is the cost of the shortest path from v to t. The A^* algorithm terminates and, when it does, $t.\texttt{distance} = d(s \text{ to } t)$.

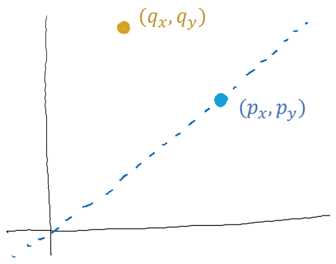
Exercise 1°. Use the A^* algorithm, by hand, to find a path from s to t in the four-node graph above.

Exercise 2*. Prove that when A^* terminates it has found the correct distance to t. *[Hint. It's an induction, but not the same type of induction as used in Dijsktra's algorithm. Look at the proof for case (i) of Bellman's algorithm.]*

2. Segment intersection

Do two line segments intersect? This is a simple question, and a good starting point for many more interesting questions in computational geometry.

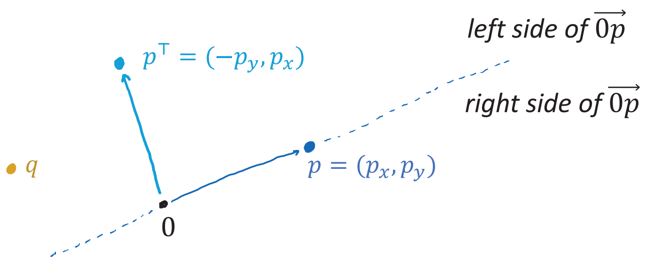
Let's start with a simpler problem. Is the point q above or below the dotted line? The answer doesn't need anything more than basic school maths: if $q_y > \left({}^{p_y}/_{p_x}\right) q_x$ then it's above.



But it's easy to get tangled up thinking through all the cases (e.g. if $p_x > 0$ and $p_y < 0$ do I need to flip the sign?) We can use slightly cleverer maths, namely dot products, to get a cleaner answer:

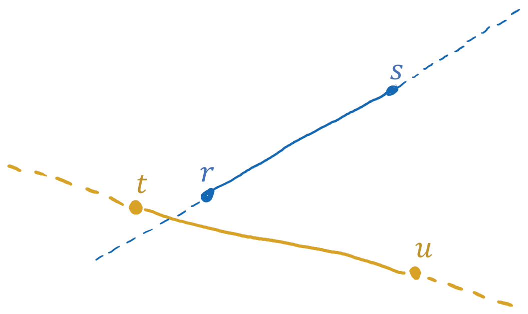
Let $p^\intercal = (-p_y, p_x)$. If we rotate the vector $\overrightarrow{0\,p}$ by 90° anticlockwise, we get $\overrightarrow{0\,p^\intercal}$. Now, the sign of $p^\intercal \cdot q$, i.e. of $-p_y q_x + p_x q_y$, tells us which side of the dotted line q is on. The dotted line is called the *extension* of $\overrightarrow{0\,p}$.

$p^\intercal \cdot q > 0$:	q is on the left, as you travel along the dotted line in direction $\overrightarrow{0\,p}$
$p^\intercal \cdot q = 0$:	q is on the line itself
$p^\intercal \cdot q < 0$:	q is on the right


left side of $\overrightarrow{0p}$
right side of $\overrightarrow{0p}$
$p^\intercal = (-p_y, p_x)$
q
$p = (p_x, p_y)$
0

This gives us all the tests we need to decide if two line segments $\overline{r\,s}$ and $\overline{t\,u}$ intersect:

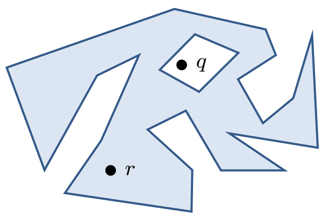
1. If t and u are both on the same side of the extension of $\overrightarrow{r\,s}$, i.e. if $(s-r)^\intercal \cdot (t-r)$ and $(s-r)^\intercal \cdot (u-r)$ have the same sign, then the two line segments don't intersect.
2. Otherwise, if r and s are both on the same side of the extension of $\overrightarrow{t\,u}$, then the two line segments don't intersect.
3. Otherwise, they do intersect.



Well-written code should test all the boundary cases, e.g. when $r = s$ or when t or u lie on the extension of $\overrightarrow{rs}$. It is however a venial sin to test equality of floating point numbers[2], because of the vagaries of finite-precision arithmetic, and so the question “How should my segment-intersection code deal with boundary cases?” depends on “What do I know about my dataset and what will my segment-intersection code be used for?”

Exercise 3. Two line segments are moving. The first line segment has endpoints $(-0.2 + 0.1t, -0.1 + 0.1t)$ and $(0.8 + 0.1t, -0.4 + 0.2t)$, the second has endpoints $(-0.36 + 0.52t, 1.1 - 0.3t)$ and $(2.1 - 0.3t, 0.1 - 0.3t)$, and $t \geq 0$. Do they collide? If so, at what time t? Write the most concise pseudocode you can, to solve this problem for arbitrary coefficients.

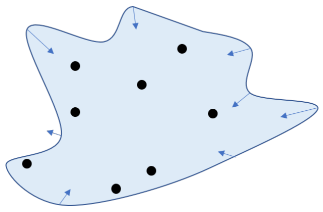
Exercise 4. Consider a horizontal line starting from q and going infinitely to the right. How many times does it cross an edge? What about from r? Devise an algorithm to detect whether a given point is outside a given shape (like q) or inside (like r), where the shape is specified as a list of non-intersecting polygons and each polygon is specified as a list of points. Discuss the handling of corner cases.



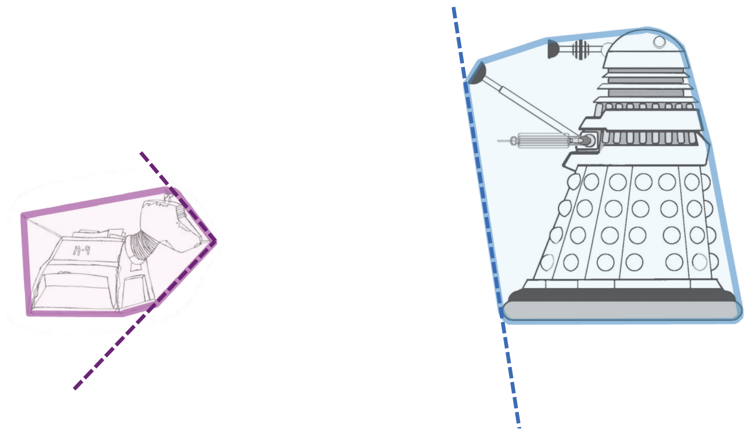
[2]In Python, $0.1 + 0.1 + 0.1 - 0.3$ gives 5.55×10^{-17}, which is not equal to 0

3. Jarvis's march

Given a collection of points, a *convex hull* is what you get if you throw a lasso around all the points, then pull it tight.



Convex hulls are used, among other things, for collision detection. Given a straight line segment on the convex hull of a complex object, all points in that object must lie on the same side of the line. If we can get away with checking the sides of just a few lines, we can drastically speed up checks for collision.

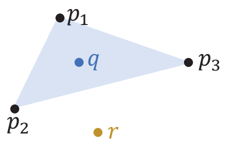


MATHEMATICAL DEFINITION

We may as well define convex hull properly, even though for algorithmic discussion that follows we'll be working informally. Formally speaking, given a collection of points $P = \{p_1, \dots, p_n\}$, any vector q is called a *convex combination* if it can be written as

$$q = \alpha_1 p_1 + \cdots + \alpha_n p_n \quad \text{where } \alpha_i \geq 0 \text{ for all } i, \text{ and } \sum_{i=1}^{n} \alpha_i = 1.$$

The *convex hull* of a collection of points is the set of all convex combinations.

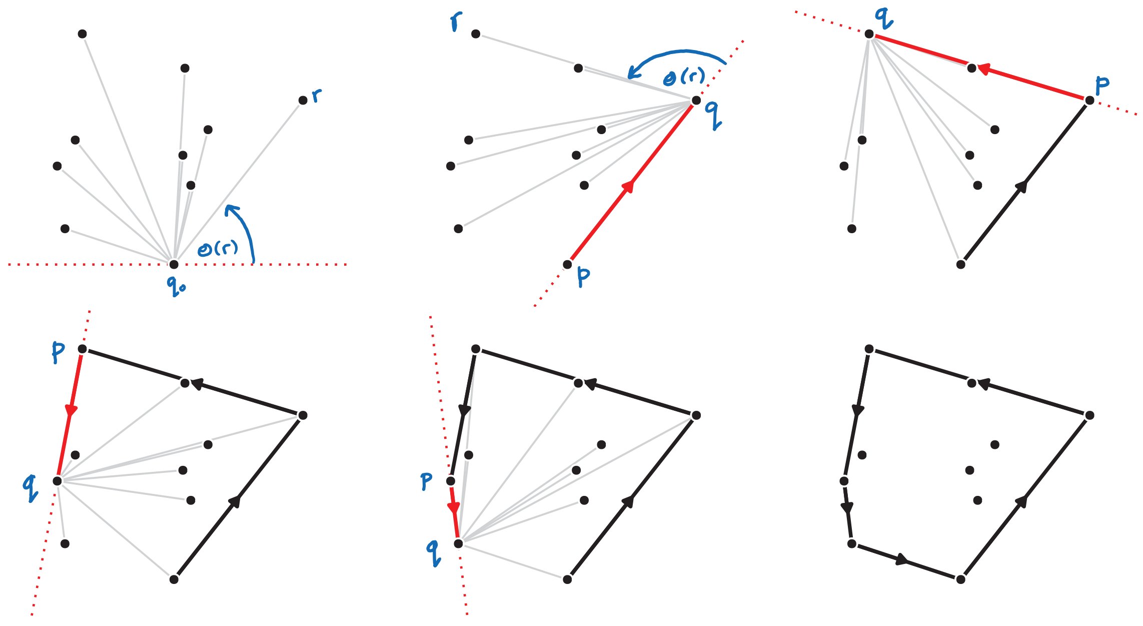


$q = 0.6p_1 + 0.16p_2 + 0.24p_3$: is a convex combination
$r = -0.5p_1 + 0.9p_2 + 0.6p_3$: not a convex combination

The *corner points of the convex hull* are the points $p \in P$ such that $p \notin \text{convexhull}(P \setminus \{p\})$. (These are the points that catch when the lasso is pulled tight—but to turn the informal idea of a lasso pulled tight into precise symbolic logic is rather hard!)

THE ALGORITHM

Here is an algorithm to compute the corner points of the convex hull of a collection of points P. It is due to Jarvis (1973), and was discovered independently by Chand and Kapur (1970).



```
let $q_0$ be the point with lowest $y$–coordinate
(in case of a tie, pick the one with the largest $x$–coordinate)

draw a horizontal (left→right) line through $q_0$
for all other points $r \in P$:
    find the angle $\theta(r)$ from the horizontal line to $\overrightarrow{q_0 r}$, measured ↺
let $q_1$ be the point with the smallest angle
(in case of a tie, pick the one furthest from $q_0$)

$h = [q_0, q_1]$
repeatedly:
    let $p$ and $q$ be the last two points added to $h$ respectively
    for all other points $r \in P$:
        find the angle $\theta(r)$ from the extended $\overrightarrow{pq}$ line to $\overrightarrow{qr}$, measured ↺
    pick the point with the smallest angle, and append it to $h$
    (in case of a tie, pick the one furthest from $q$)
    stop when we return to $q_0$
```

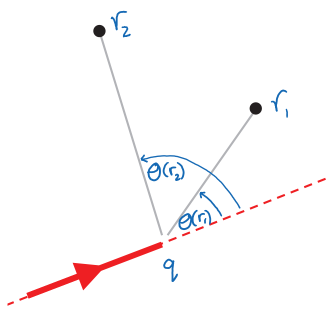
ANALYSIS

At each step of the iteration, we search for the point $r \in P$ with the smallest angle $\theta(r)$, thus the algorithm takes $O(nH)$ where n is the number of points in P and H is the number of points in the convex hull. (As with Ford-Fulkerson, running time depends on the content of the data, not just the size.)

Performance note. The algorithm says "find the point r with the smallest angle $\theta(r)$". We could use trigonometry to compute θ — but there is a trick to make this faster. (Faster in the sense of 'games get more frames per second', but no difference in the big-O sense.) If all the points we're comparing are on the same side of dotted line, as they are at all steps of Jarvis's march, then

$$\theta(r_1) < \theta(r_2) \iff r_2 \text{ is on the left of the extended line } \overrightarrow{q\, r_1}$$

and we've seen how to compute this true/false value with just some multiplications and additions.



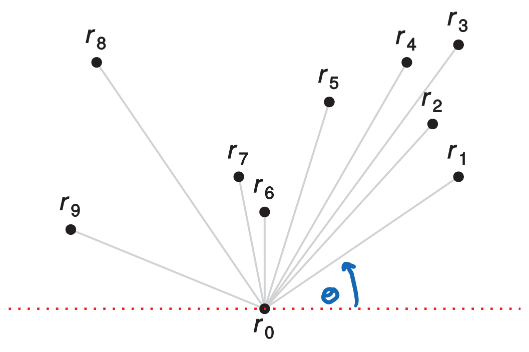
* ∗ *

Jarvis's march is very much like selection sort: repeatedly find the next item that goes into the next slot. In fact, most convex hull algorithms resemble some sorting algorithm.

Exercise 5°. Jarvis's march starts by picking a point $q_0 \in P$ with the lowest y-coordinate, breaking ties by choosing the larger x-coordinate. Give an example to show that, without this tie breaking rule, q_0 might not be a corner point.

4. Graham's scan

Here is another algorithm for computing the convex hull, due to Ronald Graham (1972). It builds up the convex hull by scanning through all the points in a fan, backtracking when necessary.



```
let $r_0$ be the point with lowest $y$–coordinate
(in case of a tie, pick the one with the largest $x$–coordinate)

draw a horizontal (left→right) line through $r_0$
for all other points $r$:
    find the angle from the horizontal line to $\overrightarrow{r_0 r}$, measured ↺
let $r_1, \dots, r_{n-1}$ be the sorted list of points, lowest angle to highest

$h = [r_0, r_1]$
for each $r_i$ in the sorted list of points, $i \geq 2$:
    if $r_i$ isn't on the left of the extension of the final segment of $h$:
        # backtrack
        repeatedly delete points from the end of $h$ until $r_i$ is
    append $r_i$ to $h$
```

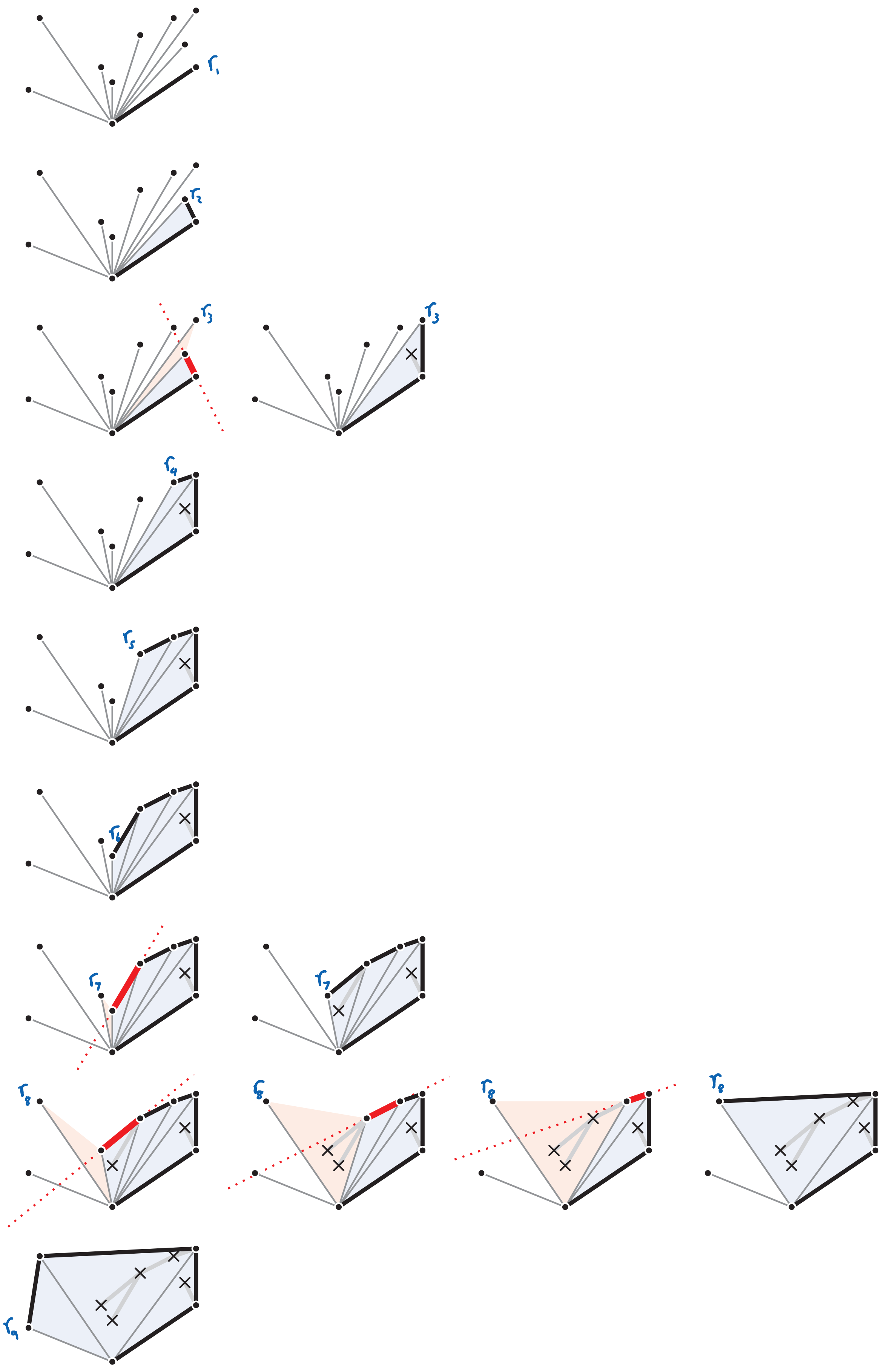
The diagram on the next page shows how the algorithm proceeds. Each row in the diagram shows it working on a new r_i, and the side-by-side panels show steps in backtracking.

ANALYSIS

The initial sort takes time $O(n \log n)$, where n is the number of points in the set. During the scan, each point r_i is added to the list once, and it can be removed at most once, so the loop is $O(n)$.

To save some trigonometrical calculations, the same trick as in Section 3 works.

Exercise 6. Graham's algorithm scans points in order, according to a certain angle. Explain carefully what happens when two points have exactly the same angle. Does the code still work? *[Hint. The code can crash! Make sure you identify all the cases where the code crashes or gives an incorrect answer.]*



r_1
r_2
r_3
r_3
r_4
r_5
r_6
r_7
r_7
r_8
r_8
r_8
r_8
r_9

5. Solutions to exercises

Exercise 1. It's worth noting that the dotted line distances do indeed satisfy the requirement for correctness: $h(a \text{ to } t) = 4 \leq d(a \text{ to } t) = 5$, and $h(b \text{ to } t) = 1 \leq d(b \text{ to } t) = 4$.

```
pop s, s.distance=0
  relax a, b
  toexplore = [a(distance=3,key=7), b(distance=5,key=6)]
pop b, b.distance=5
  relax t
  toexplore = [a(distance=3,key=7), t(distance=9,key=9)]
pop a, a.distance=3
  relax b
  toexplore = [b(distance=4,key=5), t(distance=9,key=9)]
pop b, b.distance=4
  relax t
  toexplore = [t(distance=8,key=8)]
pop t, t.distance=8
```

Exercise 2. *This is a delicate proof. There is obviously some sort of induction involved... but the Dijkstra-style argument ("prove by induction that when a vertex is popped it has the correct distance") doesn't work, and the example from Exercise 1 demonstrates why. Instead, the induction has to be along the vertices of the* correct *path, just like the proof of correctness of Bellman's algorithm.*

At the instant t is first popped, suppose $t.\mathsf{distance} \neq d(s \text{ to } t)$. Consider a true shortest path s to t. There are two cases for the status of the vertices on this path at the instant in question: (A) all the vertices apart from t have been popped with the correct distance, (B) there is a sequence of vertices $s, \ldots, u$ which have been popped with the correct distance, then some vertex v that hasn't (i.e. either v not yet popped, or popped with the wrong distance), then zero or more extra vertices before t. We'll analyse case (B). Case (A) is similar but simpler.

In case (B), the path is

$$\underbrace{s \to \cdots u}_{\text{popped, correct dist.}} \to \underbrace{v}_{\text{not}} \to \cdots t$$

When u was popped with the correct distance, it relaxed $u \to v$, hence at that time

$$\begin{aligned}
\text{key}(v) &= v.\mathsf{distance} + h(v \text{ to } t) \\
&\leq u.\mathsf{distance} + c(u \to v) + h(v \text{ to } t) \quad \text{since } u \to v \text{ just relaxed} \\
&= d(s \text{ to } u) + c(u \to v) + h(v \text{ to } t) \quad \text{by choice of } u \\
&= d(s \text{ to } v) + h(v \text{ to } t) \quad \text{since } u \to v \text{ is on a shortest path to } t \\
&\leq d(s \text{ to } v) + d(v \text{ to } t) \quad \text{by assumption on } h \\
&= d(s \text{ to } t) \quad \text{since } v \text{ is on a shortest path}
\end{aligned}$$

Some time after u was popped, t gets popped; and when t was popped it had the wrong distance, hence

$$\text{key}(v) = t.\mathsf{distance} > d(s \text{ to } t)\,.$$

We just showed that, when u was popped earlier in the execution, $\text{key}(v) \leq d(s \text{ to } t)$; and since keys can only ever decrease it must be that v gets popped after u and before t. But since u was popped with the correct distance (by construction), and u relaxes the edge $u \to v$, and that edge is part of a shortest path $s \rightsquigarrow t$, then v must have the correct distance. This contradicts our choice of v, therefore case (B) is impossible.

A similar but simpler argument shows that case (A) is impossible. Thus, our premise (that when t was first popped, $t.\mathsf{distance}$ is incorrect) must be false.

Exercise 3. First, define two utility classes: Point which is just a tuple with two fields x and y, and Polynomial which supports evaluation, multiplication, and root-finding. Let Polynomial(a_0,a_1,...,a_n) denote the polynomial $a_0 + a_1 x + \cdots + a_n x^n$.

```python
# Transpose a point
def T(p): return Point(-p.y, p.x)

# Define the four coordinates.
p = Point(x=Polynomial(-0.2, 0.1), y=Polynomial(-0.1, 0.1))
q = Point(x=Polynomial(0.8, 0.1), y=Polynomial(-0.4, 0.2))
r = Point(x=Polynomial(-0.36, 0.52), y=Polynomial(1.1, -0.3))
s = Point(x=Polynomial(2.1, -0.3), y=Polynomial(0.1, -0.3))

# Because of the cunning Polynomial class, these are all Polynomials too:
rside = T(q-p)*(r-p)
sside = T(q-p)*(s-p)
pside = T(s-r)*(p-r)
qside = T(s-r)*(q-r)

# At a given instant in time t, do the two line segments intersect?
# This is a concise implementation of the method from lectures.
def intersect(t):
    return ((rside(t)>0) != (sside(t)>0)) and ((pside(t)>0) != (qside(t)>0))

# Find all the instants in time at which one of the "side" tests switches value.
events = rside.roots() + sside.roots() + pside.roots() + qside.roots()
events = sorted(t for t in events if t>0)

# LEFT AS AN EXERCISE:
# dealing properly with (i) empty events list, (ii) time before first event,
# (iii) time after last event.

# For each interval (u,v), do the two lines intersect at time (u+v)/2?
# If they do then return u
collisions = [u for u,v in zip(events[:-1],events[1:]) if intersect((u+v)/2)]

collisions  # answer: [2.352]
```

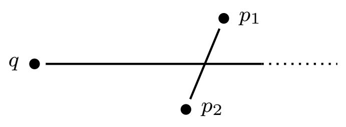
Here is the full-blown idiomatic Python code to implement Point and Polynomial.

```python
import collections
import math

class Point(collections.namedtuple('Point', ['x','y'])):
    def __sub__(self, other): return Point(x=self.x-other.x, y=self.y-other.y)
    def __neg__(self): return Point(x=-self.x, y=-self.y)
    def __str__(self): return 'Point({}, {})'.format(str(self.x), str(self.y))
    def __mul__(self, other): return self.x*other.x + self.y*other.y
    def __call__(self, x): return Point(x=self.x(x), y=self.y(x))

class Polynomial:
    def __init__(self, a0=0, *ais):
        self._coef = []
        for ai in reversed(ais):
            if ai==0.0 and len(self._coef)==0: continue
            self._coef.insert(0, ai)
        self._coef.insert(0, a0)
    def __len__(self):
        return len(self._coef)
    def __getitem__(self, i):
        return self._coef[i] if i<len(self._coef) else 0
    def __neg__(self):
        return Polynomial(*[-ai for ai in self._coef])
    def __add__(self, other):
        return Polynomial(*[self[i]+other[i] for i in range(max(len(self),len(other)))])
    def __sub__(self, other):
        return Polynomial(*[self[i]-other[i] for i in range(max(len(self),len(other)))])
    def __mul__(self, other):
        f,g = self,other
        return Polynomial(*[sum(f[j]*g[i-j] for j in range(i+1))
            for i in range(len(f)+len(g)-1)])
    def __call__(self, x):
        return sum(a*math.pow(x,i) for i,a in enumerate(self._coef))
    def __str__(self):
        terms = ["{a} {e}".format(a=a, e="" if i==0 else "x"+("" if i==1 else "^"+str(i)))
                for i,a in enumerate(self._coef)]
        return " + ".join(terms)
    def roots(self):
        if len(self) > 3:
            raise Exception("No solution for cubics or higher")
        p = self
        if p[1] == 0 and p[2] == 0:
            return []
        elif p[2] == 0:
            return [-p[0]/p[1]]
        elif p[0] == 0:
            return [0, -p[1]/p[2]]
        elif p[1]**2 < 4*p[0]*p[2]:
            return []
        else:
            x = math.sqrt(p[1]**2 - 4*p[0]*p[2])
            return [(-p[1]+x)/(2.0*p[2]), (-p[1]-x)/(2.0*p[2])]
```

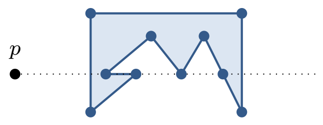
Exercise 4. Algorithm: count the total number of segments that cross the infinite horizontal line. Odd $\Rightarrow$ inside, even $\Rightarrow$ outside.



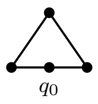
To test if a segment $\overline{p_1 p_2}$ crosses a half-infinite horizontal line: we can actually just apply the test from lecture notes without thinking, and plug in the coordinate $(+\infty, q.\text{y})$ for the other end of the horizontal line 'segment'. This can be justified by considering $(x, q.\text{y})$ as $x \to \infty$, and asking what side it's on as $x \to \infty$. Alternatively, if we don't like $+\infty$ in code, here is what the tests expand to. First, for simplicity, subtract q from each coordinate so that we're considering the half-infinite line from $(0, 0)$. Then,

```
if $p_1$.y and $p_2$.y have the same sign:
    doesn't cross
else if $p_1$.y $==$ $p_2$.y:
    doesn't cross
else if $p_2$.y$-p_1$.y and $-p_1$.x($p_2$.y$-p_1$.y) $-$ $p_1$.y($p_1$.x$-p_2$.x) have the same sign:
    doesn't cross
else:
    does cross
```

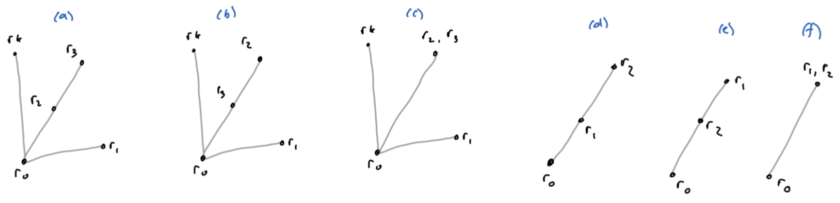
What about corner cases? In the picture, the line from p actually goes through the edge of the polygon, so the number of crossings is (mathematically) not well-defined. What should the algorithm do in such cases? What about the literal corner case, where the line goes precisely through a corner? (Ha ha hah.) I am not aware of any algorithm that deals properly with these issues, as well as coping with finite-precision arithmetic.



Exercise 5.



Exercise 6.



(a) deletes r_2 as soon as it considers r_3; this is the desired behaviour

(b) deletes r_3 when it considers the next point, which is the desired behaviour (except that if there is no next point, r_3 will be included in the output, which is incorrect)

(c) like (a)

(d) the algorithm crashes in this case: it deletes r_1 then has no "final segment" for comparison

(e) like (d) but even worse because we don't actually want to delete r_1

(f) like (d)