**The snowball algorithm for minimal enclosing disks**

By A.R.J. Marchand

CNRS UMR 5287, Bordeaux

I present here a novel snowball algorithm to compute the minimal enclosing disk for a set of 2-D points. At each step, the set supporting a minimal disk for a sample of four points is determined, and the first one or two points not included in the disk are substituted to the unnecessary points in the sample, always searching forward (cyclically) in the set. I demonstrate that the algorithm always yields the minimal enclosing disks for the set. It runs in linear time and requires little more than two inclusion tests (distance calculations) per point. Extensions to higher dimensions and ellipsoids may be considered.

Over several decades, progress has been made in reducing the complexity of algorithms for the minimal enclosing disk (Elzinga & Hearn, 1972; Megiddo, 1983; Skyum, 1991; Welzl, 1991; Efrat, Sharir & Ziv, 1994; Har-Peled & Mazumdar, 2005; Yildirim, 2008; Gao & Wang, 2018; Smolik & Skala, 2022). A currently standard solution is that proposed by Welzl (1991). Welzl’s algorithm runs in *O*(*n*), meaning that the number of steps required to solve the problem increases linearly with the number of points. Its performance is however hampered by its recursive nature, which stems from the requirement to include at each step all the points that have been examined so far. The snowball algorithm presented here avoids this requirement.

The algorithm relies on the well-known properties of the minimal circle:

* It is unique for a given set of points in the plane;
* It is determined by three points on its circumference if these three points form a triangle that is not obtuse. Otherwise, two points suffice, determining a diameter of the minimum circle.

Therefore, a minimal circle enclosing any sample of four points can always be defined by some subset of two or three of them (forming a diameter or a non-obtuse triangle). One and maybe two points are unnecessary. Adding new points to the necessary subset provides a new sample of four points. This is the heart of the snowball algorithm, as well as of the earlier algorithm of Elzinga & Hearn (1972). Note that in general, the unnecessary points should not be eliminated, as they may be needed at a later step.

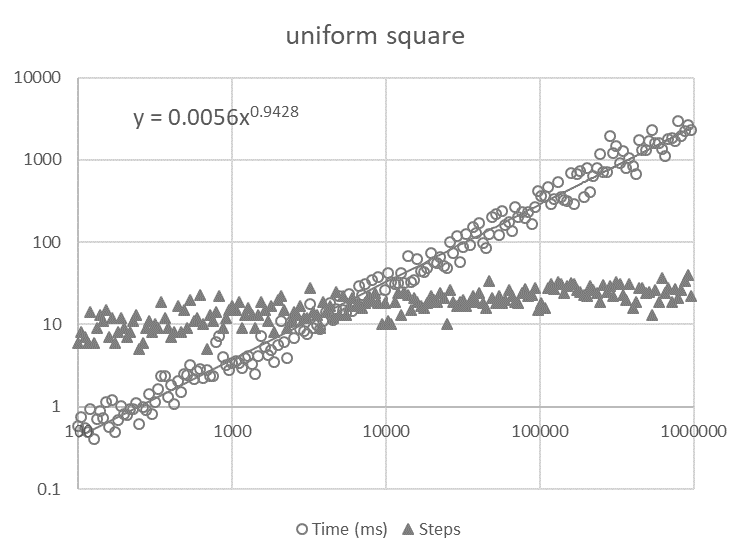
At each step of the snowball algorithm, after a minimal circle has been determined for by two or three points, the sample is complemented to four by the *first point(s) found outside* the circle in the whole set. The set is *always searched forward* and when it is exhausted, the search starts again at the first point. Unlike in Elzinga & Hearn (1972), no point is tested and eliminated. Unlike in Welzl (1991), no attempt is made to recursively ensure that the current circle includes all previously explored points.

**Correctness**

The correctness of the algorithm is proved by the following four properties:

1. At each step, the circle defined by the necessary subset is the minimal circle for the subset as well as the minimal circle for the sample four points. So *if* a circle obtained in this way ultimately encloses all the points in the analyzed set, then it must necessarily be minimal enclosing circle for the whole set;
2. At each step which includes a new point outside the circle, the radius of the new enclosing circle strictly increases;
3. The algorithm only stops when no point can be found outside the circle;
4. The set of possible circles is finite and includes the unique solution.

**Performance**



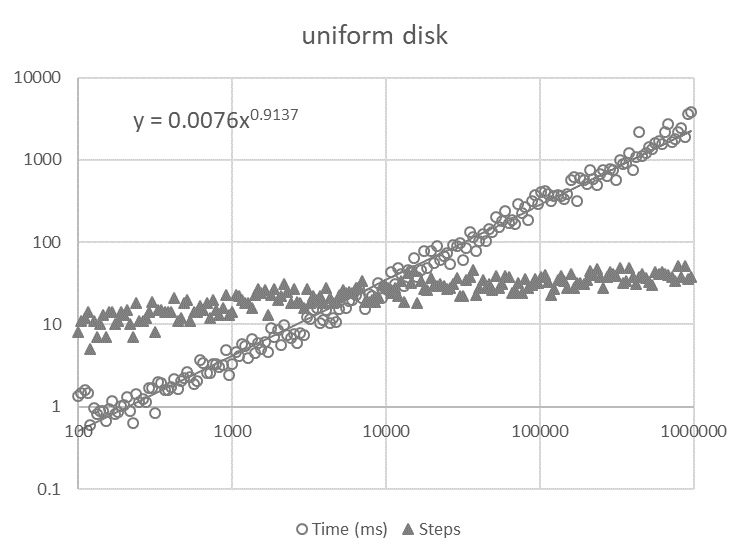


Figure 1. Time complexity and number of steps evaluated for n points uniformly distributed in a square (upper panel) or a disk (lower panel). Each symbol represents a simulation for increasingly larger sets. Execution time increased as n0.94 and n0.91 respectively. The number of steps increased much more slowly to approximately 30 and 45 respectively for N=106 points.

The performance of the snowball algorithm was simulated with the Python script shown below. The efficiency of the method was remarkable. With n points uniformly distributed in a square or a disk (Fig. 1), execution time increased in an approximately linear way in the range N=102 to 106. This is empirical evidence that the algorithm belongs to class *O*(*n*).

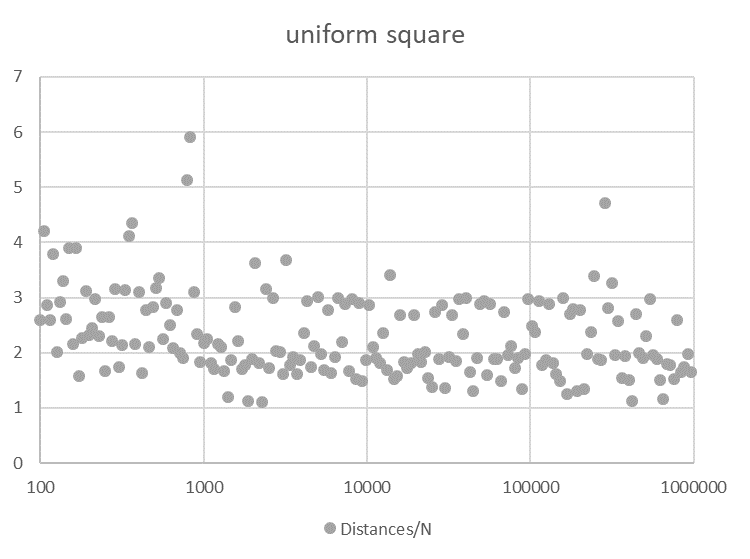


Figure 2. Total number of distances computed relative to the size N of the set. For large sets, this number comes close to 2N.

Moreover, the radius of the circle increased rapidly at each step, so that the number of steps was quite small for large sets. Time performance was thus dominated by the search for points outside the current circle, i.e. computing the distance (squared) from the center to the next points.

The total number of distances computed was close to 2N (Figure 2). This value (2.21 and 2.08 for n=5000) compares favorably with Welzl’s minidisk algorithm, which reports an average of 4.9 and 5.6 distance computations per point (MTFBALL algorithm).

Other trials with Gaussian radial or annular distributions and with elongated distributions yielded essentially the same results.

**Discussion**

It is not immediately obvious that the snowball algorithm is correct or efficient. Some of the unnecessary points at step n may be necessary at a later step. This may have led earlier researchers to disregard this relatively straightforward method. However, it can be proved that all points get eventually enclosed and that the circle found is the minimum one.

Not only is the algorithm correct, but its performance was found to be exceptional. Its time complexity appears to be *O*(*n*) and it requires fewer distance computations than the widely recognized Welzl (1972) algorithm.

The recursive nature of Welzl’s algorithm precludes its use for very large sets because of the depth of recursion and the necessity to recursively store subsets. The snowball algorithm does not suffer from this limitation and can therefore accommodate larger sets. Points may be accessed sequentially which is another simplification.

The algorithm requires little working memory and few operations because only the current subset of two or three points and the center and radius of their enclosing disk need to be kept in memory at each step. The computation of the minimal circle of four points requires more calculations than a simple distance check, but since the radius increases in a small number of steps relative to the total number of points, most of the computing time is taken by distance computations to verify the inclusion of single points (essentially two multiplications and four additions).

Of course, performance will necessarily degrade if the set of points exhibits a sequential spatial trend. This issue was already raised by Welzl who indicated that it is sufficient to randomize the set at the start. Provided that the trend is not periodic, it is even possible to avoid the cost of randomizing by browsing the set with a large step p so that p and N are relative primes and p is not too close to a divisor of N. Combined with the cyclical search, it guarantees that all points will be explored in N steps with no additional cost compared to a unit step.

It should be noted that attempts to speed up the search by eliminating points inside the convex hull, as proposed for instance by Skyum (1991) or Smolik & Skala (2022), were unsuccessful. The snowball algorithm tests each point approximately twice over the whole process. This leaves little room for improvement. Indeed, just to verify that the final circle encloses all points already requires at least one distance measurement per point. So, it is not clear that a point can be eliminated with a cost of less than two distance computations.

Finally, the snowball algorithm holds promises for the same generalizations as Welzl’s to higher dimensions, as well as to ellipsoids.

**References**

Efrat, A., Sharir, M., Ziv, A. (1994). Computing the smallest k-enclosing circle and related problems. Computational Geometry, 4(3), 119–136.

Elzinga, J., Hearn, D. W. (1972). "The minimum covering sphere problem", Management Science, 19: 96–104.

Gao, S.,Wang, C. (2018). A new algorithm for the smallest enclosing circle. In: 2018 8th International Conference on Management, Education and Information (MEICI 2018), Atlantis Press.

Har-Peled, S., Mazumdar, S. (2005). Fast algorithms for computing the smallest k-enclosing circle. Algorithmica, 41(3), 147–157.

Megiddo, N. (1983). Linear-time algorithms for linear programming in R3 and related problems. SIAM Journal on Computing, 12(4), 759–776.

Skyum, S. (1991). A simple algorithm for computing the smallest enclosing circle. Information Processing Letters, 37(3), 121–125.

Smolik, M., Skala, V. (2022). Efficient Speed-Up of the Smallest Enclosing Circle Algorithm. Informatica, 33(3): 623-633.

Welzl, E. (1991). Smallest enclosing disks (balls and ellipsoids). In: Maurer, H. (Ed.), New Results and New Trends in Computer Science, Lecture Notes in Computer Science, Vol. 555. Springer, Berlin, Heidelberg.

Yildirim, E. A. (2008). Two algorithms for the minimum enclosing ball problem, SIAM J. Optim., 19: 1368–1391.

**Annex: Python script for the snowball algorithm**

#====================================== d\_squared

def d\_squared(A, B):

"""

return squared distance between points A and B

"""

return (A[0]-B[0])\*\*2 + (A[1]-B[1])\*\*2

#====================================== inc\_position

def inc\_position(n, position, step, size):

"""

add n steps to position, modulo size

"""

return (position+n\*step) % size

#====================================== inside

def inside(A, O, r\_squared):

"""

check whether point A is in circle (O, r\_squared)

Returns True if inside, False if outside

"""

return d\_squared(A, O)<=r\_squared\*Precision # Precision = 1+1E-12 allows for rounding errors

#====================================== obtuse

def obtuse(subset):

"""

test whether a triangle has an obtuse angle

Parameters: subset, a list of 3 points, tuples of 2 float

Returns: True if an obtuse angle is found, False otherwise

"""

for i,j,k in C\_3\_2:

(x1, y1), (x2, y2), (x3, y3) = subset[i], subset[j], subset[k]

if (x2-x1)\*(x3-x1) + (y2-y1)\*(y3-y1) < 0: return True # negative scalar product

return False

#====================================== point\_outside

def point\_outside(sample, position, O, r\_squared):

"""

look for a point outside circle

cyclic search in sample

stop if found

Parameters: sample, a list of 3 points, tuples of 2 float

position, an integer between 0 and len(sample)-1

O, a point, center of the circle

r\_squared, a float, square of circle radius

Returns: E, point if found, else None

new position, an integer

"""

start=position

while True:

E=sample[position]

position=inc\_position(1, position, Step, len(sample))

if not inside(E, O, r\_squared):

return E, position # a point outside

if position==start:

return None, position # no point outside

#====================================== minimal2

def circle2(A, B, remain):

"""

Test whether a circle based on 2 points encloses a set of other points

Parameters: A, B, two points to use as a diameter

remain: a set of points to test for inclusion in the circle

Returns: O, a point, tuple of 2 float

r\_squared, a float

[A, B], a list of 2 points forming a diameter of the circle

True if all points are in the circle, False otherwise

"""

# calculate circle

O = (A[0]+B[0])/2, (A[1]+B[1])/2 # middle of AB

r\_squared=d\_squared(A, O)

# test remaining point(s)

if not point\_outside(remain, 0, O, r\_squared)[0]:

return O, r\_squared, True # all points are in the circle

return None, None, False

#====================================== circle3

def circle3(subset):

"""

find the circumscribed circle of a subset of 3 points using bisectors

Parameters: subset, a list of 3 points, tuples of 2 float

Returns: (x0, y0), a point, tuple of 2 float

r\_squared, a float, square of circle radius

"""

(x1, y1), (x2, y2), (x3, y3) = subset

if (x3-x2)\*(y2-y1) == (x2-x1)\*(y3-y2) : # check for alignment

return minimal2(subset)[:2] # print(subset, "are aligned")

# mid points

xm, ym = (x1+x2)/2, (y1+y2)/2

xn, yn = (x2+x3)/2, (y2+y3)/2

# slopes of bisectors

sm=(x1-x2)/(y2-y1) if y2!=y1 else None

sn=(x2-x3)/(y3-y2) if y3!=y2 else None

# vertical bisectors

if sm is None:

x0=xm

y0=yn+sn\*(x0-xn)

elif sn is None:

x0=xn

y0=ym+sm\*(x0-xm)

# horizontal bisectors

elif sm==0:

y0=ym

x0=xn+(y0-yn)/sn

elif sn==0:

y0=yn

x0=xm+(y0-ym)/sm

# general bisectors

else:

x0=((sm\*xm-ym)-(sn\*xn-yn))/(sm-sn)

y0=ym+sm\*(x0-xm)

r\_squared=d\_squared(subset[0], (x0,y0))

return (x0,y0), r\_squared

#====================================== minimal2

def minimal2(subset):

"""

try circles based on 2 points, looking for one that encloses all 3 or 4 points of the subset

Parameter: subset, a list of 4 (or 3) points, tuples of 2 float

Returns: O, a point, tuple of 2 float

r\_squared, a float

[A, B], a list of 2 points forming a diameter of the circle

True if all points are in the circle, False otherwise

"""

# generate combinations of two points

if len(subset)==4: # 4 points

combine=C\_4\_2

else:

combine=C\_3\_2\_plus # 3 points

for i, j, k, l in combine:

A, B = subset[i], subset[j] # 3 or 4 points

remain=[subset[k]]

if len(subset)==4: # 4 points

remain+=[subset[l]]

# calculate circle

O, r\_squared, ok = circle2(A, B, remain)

if ok:

return O, r\_squared, [A, B], True # all points are in the circle

return None, None, [None, None], False # no solution with 2 points

#====================================== minimal3

def minimal3(subset):

"""

try circles based on 3 points, find the smallest one that encloses all 4 points of the subset

Parameters: subset, a list of 4 points, tuples of 2 float

Returns: (x0, y0), a point, tuple of 2 float

r\_squared, a float

a list of 3 points defining the minimum circle

"""

best = [None, None], float('inf'), subset[:3] # initialize best

# solve if only 3 points

if len(subset)==3:

O, r\_squared=circle3(subset)

if r\_squared<best[1]:

best=O, r\_squared, subset # store best

# solve with 4 points

else:

# generate combinations of three points

for i, j, k, l in C\_4\_3:

combine = [subset[i], subset[j], subset[k]]

remain = subset[l]

# avoid obtuse triangles

if obtuse(combine):

continue

# compute circle

O, r\_squared=circle3(combine)

if not r\_squared: # aligned points

continue

# test last point in combination

if inside(remain, O, r\_squared):

if r\_squared<best[1]:

best=O, r\_squared, combine # store best

if best[1]!=float('inf'):

return best # give result

#====================================== snowball

def snowball (points, subset, position):

"""

compute minimum circle enclosing points

Parameters: points, a list of points, tuples of 2 float

subset, a list of four points to start with

position, in points list

Returns: (x0, y0), a point, tuple of 2 float

r\_squared, a float

"""

# search for minimum circle

while True:

# try solving with 2 points

P, s, [A, B], ok = minimal2(subset)

if ok:

O, r\_squared=P, s

subset=[A, B]

# solve with 3 points

else:

O, r\_squared, [A, B, C] = minimal3(subset)

subset=[A, B, C]

# try adding one outside point

D, position=point\_outside(points, position, O, r\_squared)

if not D:

return subset, O, r\_squared # all points are inside: return

else:

subset+=[D]

# try adding another outside point

if len(subset)==3:

D, position=point\_outside(points, position, O, r\_squared)

if D:

subset+=[D]