Evolutionary Computation

2023/24

Master Artificial Intelligence

Arno Formella

Departamento de Informática
Escola Superior de Enxeñaría Informática
Universidade de Vigo

23/24

evaluation/assessment

- active participation in class (10%)
- two major up-loads of homework (50%)
  (probably week 19-23 Feb, and week 11-15 Mar)
- final exam (40%)
  (28th May and/or 2th Jul)

bibliography

- D. Simon, *Evolutionary Optimization Algorithms*, ISBN: 978-0-470-93741-9, Wiley, 2013
- A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*, second edition, ISBN 978-3-662-44873-1, Springer, 2015
- my webpage: http://formella.webs.uvigo.es/doc/ec23/index.html

evolutionary computation

- Evolutionary computation is about optimization.
- Evolutionary algorithms are usually randomized or probabilistic heuristic algorithms.
- Many of them are called nature-inspired (or bio-inspired) algorithms as they exhibit some properties observed in nature (especially, but not only, from biology).
- Programs based on evolutionary algorithms are typically used to find approximate solutions to difficult problems.

Disclaimer: we are talking about *nature-inspired optimization algorithms*, we are not copying nature, mostly, because I've no idea what nature is doing. We are interested in mathematical models and certain types of algorithms that serve as powerful optimization methods.

www.nature.com/scientificreports

scientific reports

Check for updates

OPEN

The cheetah optimizer: a nature-inspired metaheuristic algorithm for large-scale optimization problems

Mohammad Amin Akbari[1], Mohsen Zare[2], Rasoul Azizipanah-abarghooee[3], Seyedali Mirjalili[4,5] & Mohamed Deriche[1]✉

Motivated by the hunting strategies of cheetahs, this paper proposes a nature-inspired algorithm called the cheetah optimizer (CO). Cheetahs generally utilize three main strategies for hunting prey, i.e., searching, sitting-and-waiting, and attacking. These strategies are adopted in this work. Additionally, the leave the pray and go back home strategy is also incorporated in the hunting process to improve the proposed framework's population diversification, convergence performance, and robustness. We perform intensive testing over 14 shifted-rotated CEC-2005 benchmark functions to evaluate the performance of the proposed CO in comparison to state-of-the-art algorithms. Moreover, to test the power of the proposed CO algorithm over large-scale optimization problems, the CEC2010 and the CEC2013 benchmarks are considered. The proposed algorithm is also tested in solving one of the well-known and complex engineering problems, i.e., the economic load dispatch problem. For all considered problems, the results are shown to outperform those obtained using other conventional and improved algorithms. The simulation results demonstrate that the CO algorithm can successfully solve large-scale and challenging optimization problems and offers a significant advantage over different standards and improved and hybrid existing algorithms. Note that the source code of the CO algorithm is publicly available at https://www.optim-app.com/projects/co.

12/2022

www.nature.com/scientificreports

scientific reports

Check for updates

OPEN

A new human-inspired metaheuristic algorithm for solving optimization problems based on mimicking sewing training

Mohammad Dehghani, Eva Trojovská✉ & Tomáš Zuščák

This paper introduces a new human-based metaheuristic algorithm called Sewing Training-Based Optimization (STBO), which has applications in handling optimization tasks. The fundamental inspiration of STBO is teaching the process of sewing to beginner tailors. The theory of the proposed STBO approach is described and then mathematically modeled in three phases: (i) training, (ii) imitation of the instructor's skills, and (iii) practice. STBO performance is evaluated on fifty-two benchmark functions consisting of unimodal, high-dimensional multimodal, fixed-dimensional multimodal, and the CEC 2017 test suite. The optimization results show that STBO, with its high power of exploration and exploitation, has provided suitable solutions for benchmark functions. The performance of STBO is compared with eleven well-known metaheuristic algorithms. The simulation results show that STBO, with its high ability to balance exploration and exploitation, has provided far more competitive performance in solving benchmark functions than competitor algorithms. Finally, the implementation of STBO in solving four engineering design problems demonstrates the capability of the proposed STBO in dealing with real-world applications.

2022

www.nature.com/scientificreports

scientific reports

Check for updates

OPEN

A novel hermit crab optimization algorithm

Jia Guo[1,2], Guoyuan Zhou[1], Ke Yan[3], Binghua Shi[1]✉, Yi Di[1,2] & Yuji Sato[4]

High-dimensional optimization has numerous potential applications in both academia and industry. It is a major challenge for optimization algorithms to generate very accurate solutions in high-dimensional search spaces. However, traditional search tools are prone to dimensional catastrophes and local optima, thus failing to provide high-precision results. To solve these problems, a novel hermit crab optimization algorithm (the HCOA) is introduced in this paper. Inspired by the group behaviour of hermit crabs, the HCOA combines the optimal search and historical path search to balance the depth and breadth searches. In the experimental section of the paper, the HCOA competes with 5 well-known metaheuristic algorithms in the CEC2017 benchmark functions, which contain 29 functions, with 23 of these ranking first. The state of work BPSO-CM is also chosen to compare with the HCOA, and the competition shows that the HCOA has a better performance in the 100-dimensional test of the CEC2017 benchmark functions. All the experimental results demonstrate that the HCOA presents highly accurate and robust results for high-dimensional optimization problems.

2023

www.nature.com/scientificreports

scientific reports

Check for updates

OPEN

Pair barracuda swarm optimization algorithm: a natural-inspired metaheuristic method for high dimensional optimization problems

Jia Guo[1,2], Guoyuan Zhou[3], Ke Yan[4], Yuji Sato[5] & Yi Di[1,2]✉

High-dimensional optimization presents a novel challenge within the realm of intelligent computing, necessitating innovative approaches. When tackling high-dimensional spaces, traditional evolutionary tools often encounter pitfalls, including dimensional catastrophes and a propensity to become trapped in local optima, ultimately compromising result accuracy. To address this issue, we introduce the Pair Barracuda Swarm Optimization (PBSO) algorithm in this paper. PBSO employs a unique strategy for constructing barracuda pairs, effectively mitigating the challenges posed by high dimensionality. Furthermore, we enhance global search capabilities by incorporating a support barracuda alongside the leading barracuda pair. To assess the algorithm's performance, we conduct experiments utilizing the CEC2017 standard function and compare PBSO against five state-of-the-art natural-inspired optimizers in the control group. Across 29 test functions, PBSO consistently secures top rankings with 9 first-place, 13 second-place, 5 third-place, 1 fourth-place, and 1 fifth-place finishes, yielding an average rank of 2.0345. These empirical findings affirm that PBSO stands as the superior choice among all test algorithms, offering a dependable solution for high-dimensional optimization challenges.

2023

www.nature.com/scientificreports

scientific reports

Check for updates

OPEN

Mother optimization algorithm: a new human-based metaheuristic approach for solving engineering optimization

Ivana Matoušová[1]✉, Pavel Trojovský[1], Mohammad Dehghani[1], Eva Trojovská[1] & Juraj Kostra[2]

This article's innovation and novelty are introducing a new metaheuristic method called mother optimization algorithm (MOA) that mimics the human interaction between a mother and her children. The real inspiration of MOA is to simulate the mother's care of children in three phases education, advice, and upbringing. The mathematical model of MOA used in the search process and exploration is presented. The performance of MOA is assessed on a set of 52 benchmark functions, including unimodal and high-dimensional multimodal functions, fixed-dimensional multimodal functions, and the CEC 2017 test suite. The findings of optimizing unimodal functions indicate MOA's high ability in local search and exploitation. The findings of optimization of high-dimensional multimodal functions indicate the high ability of MOA in global search and exploration. The findings of optimization of fixed-dimension multi-model functions and the CEC 2017 test suite show that MOA with a high ability to balance exploration and exploitation effectively supports the search process and can generate appropriate solutions for optimization problems. The outcomes quality obtained from MOA has been compared with the performance of 12 often-used metaheuristic algorithms. Upon analysis and

2023

www.nature.com/scientificreports

scientific reports

Check for updates

OPEN

A new human-based metaheuristic algorithm for solving optimization problems based on preschool education

Pavel Trojovský

In this paper, with motivation from the No Free Lunch theorem, a new human-based metaheuristic algorithm named Preschool Education Optimization Algorithm (PEOA) is introduced for solving optimization problems. Human activities in the preschool education process are the fundamental inspiration in the design of PEOA. Hence, PEOA is mathematically modeled in three phases: (i) the gradual growth of the preschool teacher's educational influence, (ii) individual knowledge development guided by the teacher, and (iii) individual increase of knowledge and self-awareness. The PEOA's performance in optimization is evaluated using fifty-two standard benchmark functions encompassing unimodal, high-dimensional multimodal, and fixed-dimensional multimodal types, as well as the CEC 2017 test suite. The optimization results show that PEOA has a high ability in exploration–exploitation and can balance them during the search process. To provide a comprehensive analysis, the performance of PEOA is compared against ten well-known metaheuristic algorithms. The simulation results show that the proposed PEOA approach performs better than competing algorithms by providing effective solutions for the benchmark functions and overall ranking as the first-best optimizer. Presenting a statistical analysis of the Wilcoxon signed-rank test shows that

2023

www.nature.com/scientificreports

scientific reports

Check for updates

OPEN

A new human-based metahurestic optimization method based on mimicking cooking training

Eva Trojovská✉ & Mohammad Dehghani

Metaheuristic algorithms have a wide range of applications in handling optimization problems. In this study, a new metaheuristic algorithm, called the chef-based optimization algorithm (CBOA), is developed. The fundamental inspiration employed in CBOA design is the process of learning cooking skills in training courses. The stages of the cooking training process in various phases are mathematically modeled with the aim of increasing the ability of global search in exploration and the ability of local search in exploitation. A collection of 52 standard objective functions is utilized to assess the CBOA's performance in addressing optimization issues. The optimization results show that the CBOA is capable of providing acceptable solutions by creating a balance between exploration and exploitation and is highly efficient in the treatment of optimization problems. In addition, the CBOA's effectiveness in dealing with real-world applications is tested on four engineering problems. Twelve well-known metaheuristic algorithms have been selected for comparison with the CBOA. The simulation results show that CBOA performs much better than competing algorithms and is more effective in solving optimization problems.

2022

the problems we will look at

- search of minimum in real-valued multi-dimensional functions
- traveling salesman (or nowadays *salesperson*) problem (TSP)
- sorting as an optimization problem (just for fun)
- maybe: (0-1)-knapsack problem (KSP)
- some more problems as examples

algorithms

- An algorithm is a finite sequence of well-defined steps (or instructions) to complete a task or solve a problem.
- In principal, the individual steps must be executable by a human being.
- The steps (or instructions) must realize a finite change of state (or configuration) on the system on which the algorithm is executed.
- Completing a task means that there is an other algorithm that can decide that the final configuration has the required property.
- The time complexity of an algorithm is the number of steps the algorithm executes before it stops.

computational complexity

- Algorithms can be grouped into classes according to their time complexity (same is true for space complexity).
- For an asymptotic upper bound according to some input size n we say:
  function f is in the order of function g, whenever we have:

$$\exists c > 0 \;\; \exists n_0 \;\; \forall n > n_0 : |f(n)| \leq c \cdot g(n)$$

  and write: $f(n) = \mathscr{O}(g(n))$.
- Example: if $f(n) = \mathscr{O}(n^3)$ then f does not grow faster than cubic.
- There are more notations for other asymptotic characterizations: o, ω, Ω, Θ.
- If you like, take a look at the complexity zoo:
  https://complexityzoo.net/Complexity_Zoo

computational complexity

- There are arbitrary difficult problems
  (there is a hierarchy of classes).
- There are problems that don't have a solution at all
  (uncomputable problems)
  examples are: decide whether an arbitrary program stops, decide whether two formal languages are equivalent, among others.
- There are problems where we know that they are computable (i.e., there exists a solution) but we don't know how to compute one. Look into *forbidden graph minors in graph theory*.
- There are problems for which there are known algorithms, but we don't know the *smallest* class they belong to, e.g., unknotting an unknot (take a look into knot theory)
  https://en.wikipedia.org/wiki/Unknot or
  factorization of numbers https://en.wikipedia.org/wiki/Integer_factorization.

runtime examples according to complexity

Assume we can deal with 1 million items in 1 second using a linear time algorithm (i.e., megahertz item processing):

size n	**function**	$\mathcal{O}$**-notation**	**time**
1000000	linear time	$\mathcal{O}(n)$	1 second
	quasi-linear time	$\mathcal{O}(n \log n)$	20 seconds
	quadratic time	$\mathcal{O}(n^2)$	11.6 days
	cubic time	$\mathcal{O}(n^3)$	31710 years
	quartic time	$\mathcal{O}(n^4)$	32 billion year
			(2.3 times age of universe)
	exponential time	$\mathcal{O}(2^n)$	eternal
	factorial time	$\mathcal{O}(n!)$	no words any more

runtime examples according to complexity

Assume we can deal with 1 million items in 1 millisecond using a linear time algorithm (i.e., gigahertz item processing):

size n	**function**	$\mathcal{O}$**-notation**	**time**
1000000	linear time	$\mathcal{O}(n)$	1 millisecond
	quasi-linear time	$\mathcal{O}(n \log n)$	20 milliseconds
	quadratic time	$\mathcal{O}(n^2)$	16 minutes
	cubic time	$\mathcal{O}(n^3)$	31.7 years
	quartic time	$\mathcal{O}(n^4)$	32 million years
	exponential time	$\mathcal{O}(2^n)$	eternal
	factorial time	$\mathcal{O}(n!)$	no words any more

Note: Parallelization on a p-processor machine gives you at most a linear speedup of p (and most of the time not even that). Quantum computation offers sometimes (Grover algorithm) a quadratic speedup regarding input size n.

handable problem sizes according to complexity

Assume 1 nanosecond processing time per item (i.e., 1 GHz operating frequency), problem sizes handable in one hour:

function	$\mathcal{O}$**-notation**	**problem size**
linear time	$\mathcal{O}(n)$	3.6 trillion
quasi-linear time	$\mathcal{O}(n \log n)$	96.6 billion
quadratic time	$\mathcal{O}(n^2)$	1.9 million
cubic time	$\mathcal{O}(n^3)$	15.3 thousand
quartic time	$\mathcal{O}(n^4)$	1377
exponential time	$\mathcal{O}(2^n)$	41
factorial time	$\mathcal{O}(n!)$	15

algorithm types

- deterministic algorithms
- non-deterministic algorithms
- randomized or probabilistic algorithms
- quantum algorithms

Note, all types compute the same set of computable functions, they differ only in time and space complexity (see below).

When do we consider a problem to be difficult?

A problem is difficult whenever we only know deterministic algorithms solving the problem that have at least exponential runtime (or polynomial runtime with a large exponent).

Side note: NP-complete and NP-hard problems

Evolutionary algorithms are often mentioned as a way to tackle NP-complete or NP-hard problems: What does that mean?

- A problem is NP-complete when we know a polynomial time deterministic algorithm that checks a solution, but we know only an exponential time deterministic algorithm to find a solution.
- A problem is, at least, NP-hard when we even don't know a polynomial time deterministic algorithm that checks a solution.
- There are problems where we know that they can be solved in exponential time, but we don't know whether they are NP-complete (or even simpler), e.g., the graph isomorphism problem, or the unknot problem.
- Essentially, we don't know whether the NP-complete problems (those that we know to have a polynomial time algorithm to check them) are the same class as the polynomial time solvable problems.
  In other words, we don't know whether $P = NP$ or $P \neq NP$.

randomized or probabilistic algorithms

two main classes:

- Monte Carlo algorithm:
  for a certain number of iterations do:
  - perform some randomized algorithm step towards a better solution
  - (usually keep track of your best solution found so far)
- Monte Carlo algorithms always terminate and (hopefully) find a somewhat good solution.
- Las Vegas algorithm:
  while a certain end condition still not met do:
  - perform some randomized algorithm step towards a better solution
- Las Vegas algorithms only terminate with a correct solution (or do not terminate at all), but their runtime is probabilistic.

algorithmic paradigms

- backtracking
- branch and bound
- brute-force (or exhaustive) search
- divide and conquer
- dynamic programming
- greedy algorithm
- prune and search
- online algorithms

heuristic algorithms

What are heuristic algorithms?

- Just do something you come up with and be happy with the result.

evolutionary algorithms

What are evolutionary algorithms?

- Evolutionary algorithms are heuristic optimization algorithms usually implemented with the Monte Carlo approach (and possibly a Las Vegas stopping condition when available)
- that exhibit, let's say, at least a tendency to approach a global minimum as solution of the optimization problem.
- (Often they are inspired by some phenomenon observable in nature or a *creative* name has been used.)

optimization

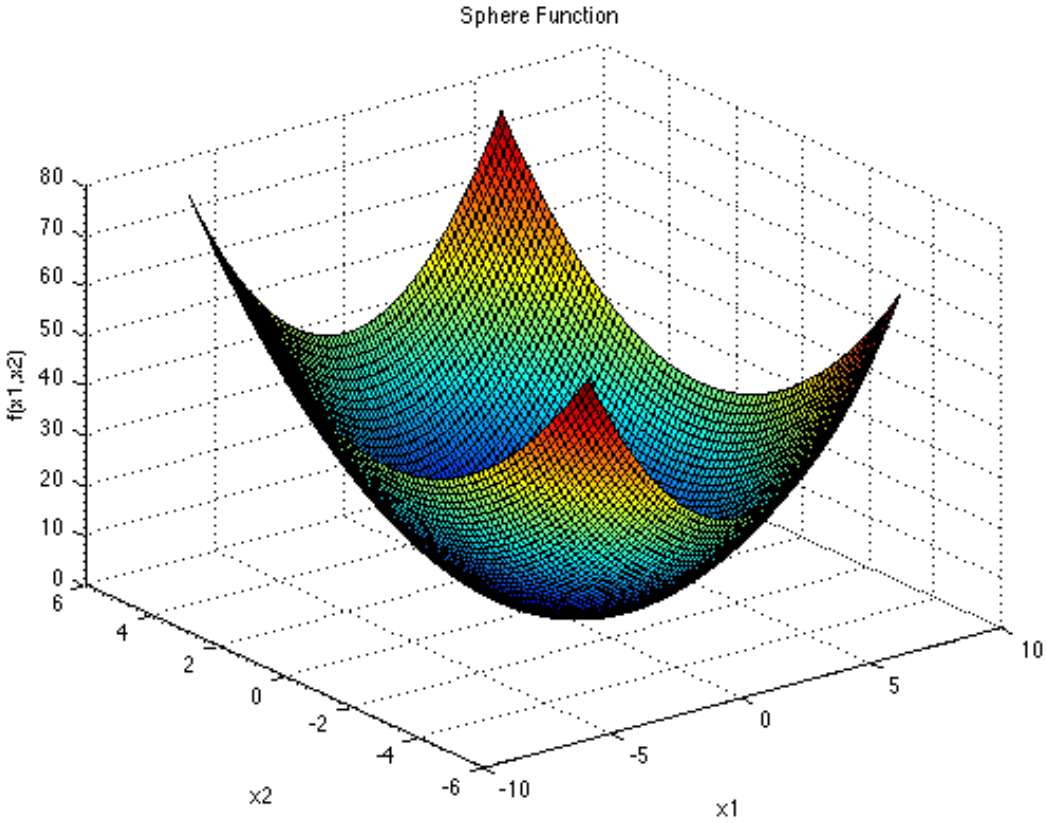
- Given a search space $\mathbb{X}$ (called as well *search domain* or *problem space*) and
- a function f (bounded from below) from the search space to the real numbers (or at least a totally ordered set), e.g. $f : \mathbb{X} \longrightarrow \mathbb{R}$,
- find an element $x^\star \in \mathbb{X}$ such that $f(x^\star) \leq f(x)$ for all $x \in \mathbb{X}$.
- i.e., we look for a global minimum.

Observe: whenever we look for a maximum, we can use just a negative sign and look for a minimum (and f must be bounded from above)!

local versus global minimum

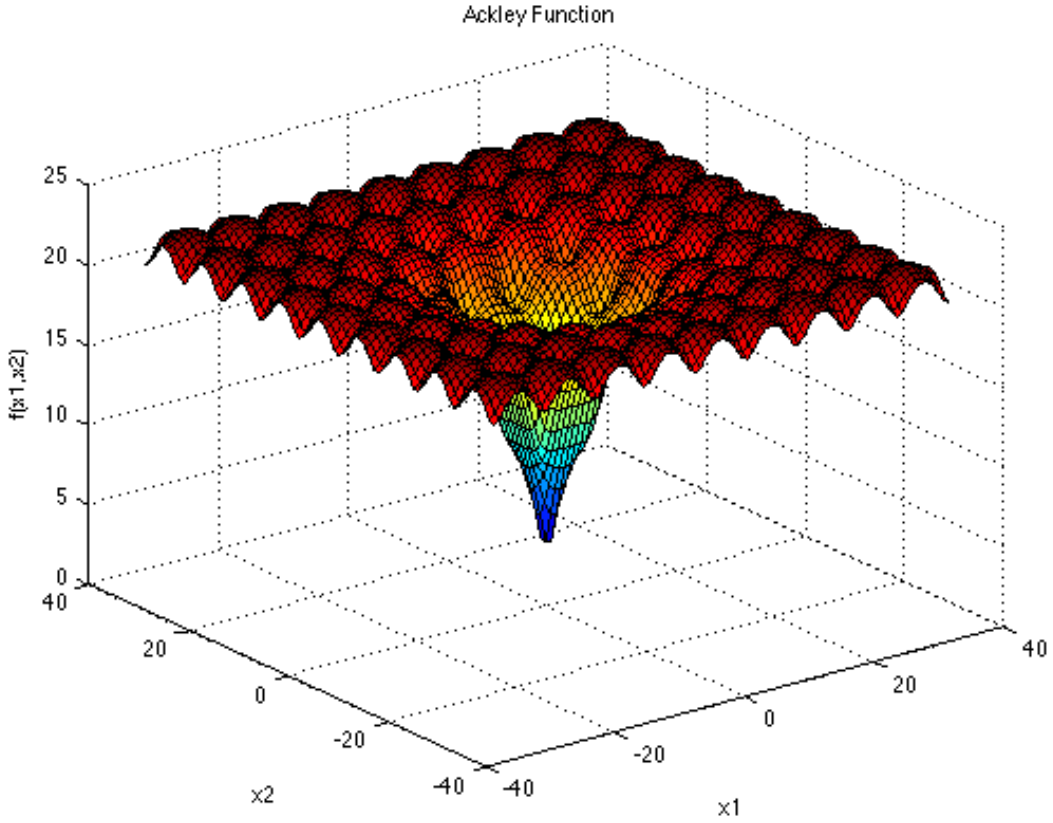
- If we can determine a neighborhood around each element $x \in \mathbb{X}$, we call $\mathcal{N}(x)$ the set of neighbors of x.
- and if we have for all such neighbors $x' \in \mathcal{N}(x)$, that $f(x) \leq f(x')$,
- then we call x a local minimum (sometimes written as $\hat{x}$).
- Reaching a local minimum is often somewhat easier, as we can take advantage of a possibly available gradient (local search algorithms).
- It happens to be an issue in optimization not to get stuck in a local minimum while searching for a global minimum.

optimization test functions: Sphere function



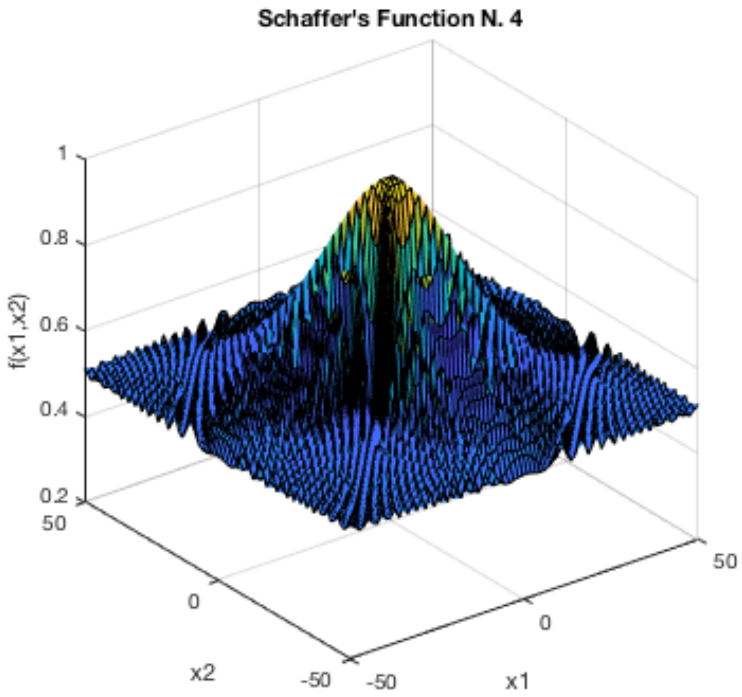
$$f(x) = \sum_{i=1}^{d} x_i^2$$

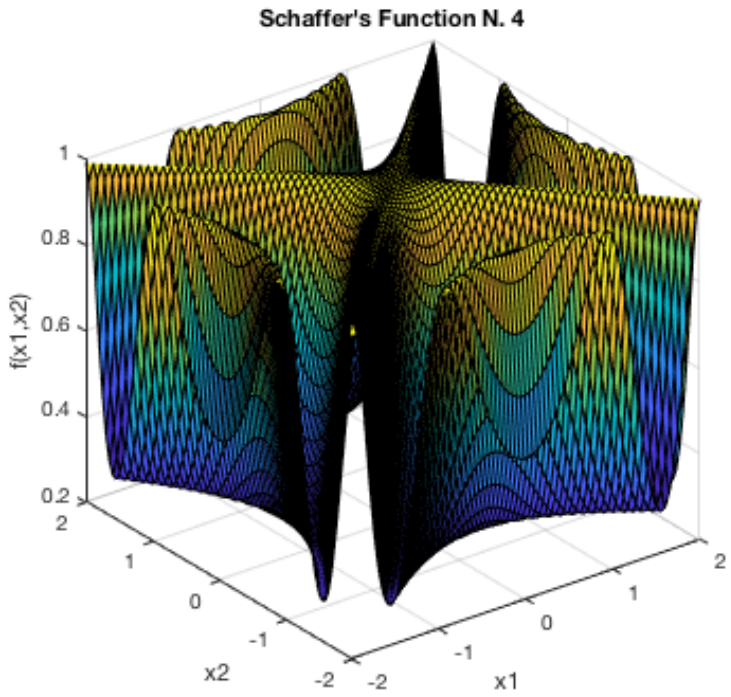
optimization test functions: Ackley function



$$f(x) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + e$$

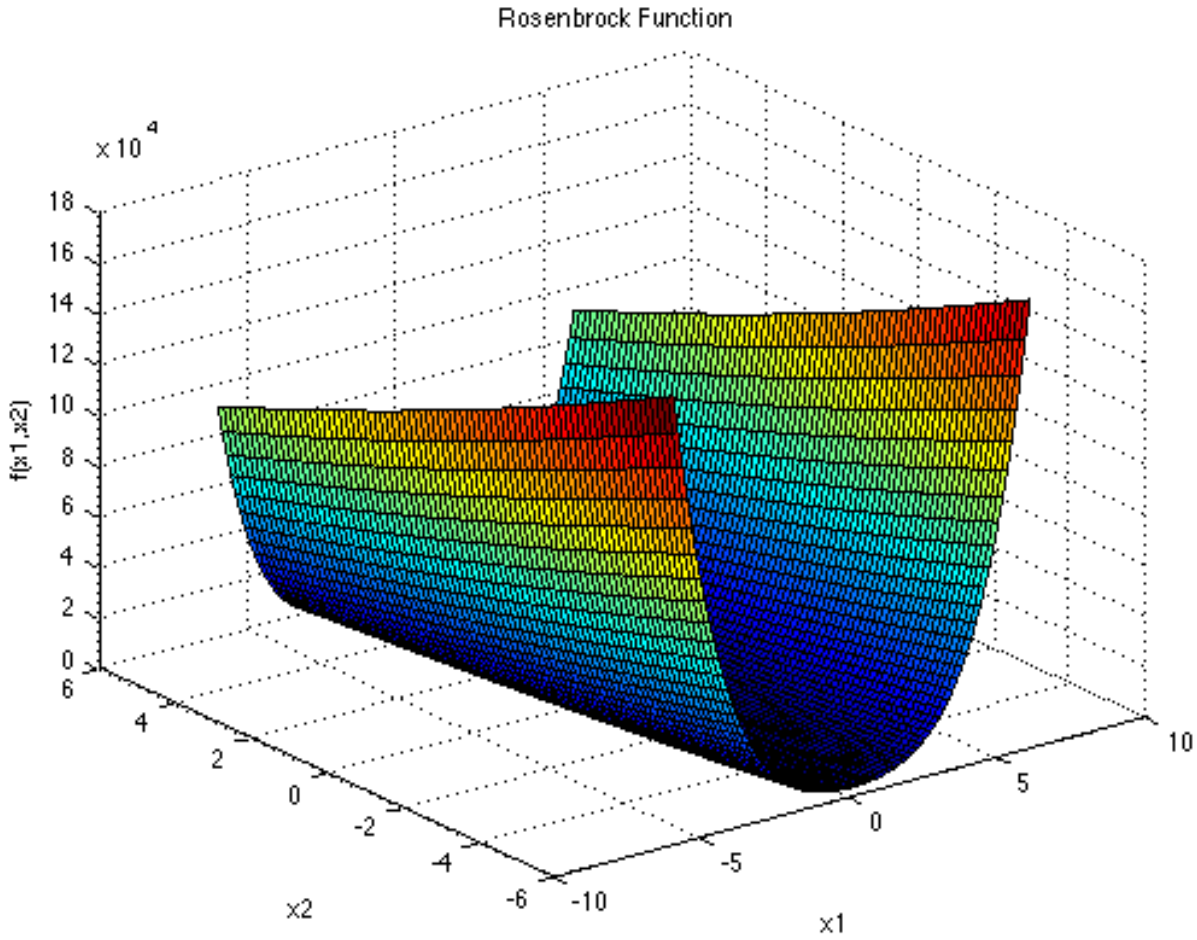
optimization test functions: Schaffer 4 function





$$f(x) = 0.5 + \frac{\cos^2(\sin(|x_1^2 - x_2^2|)) - 0.5}{[1 + 0.001(X_1^2 + x_2^2)]^2}$$

optimization test functions: Rosenbrock function



$$f(x) = \sum_{i=1} d-1[100 \cdot (x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

optimization test functions

For more real-valued functions and common

- parameter settings,
- search areas,
- local/global optima,
- and code examples

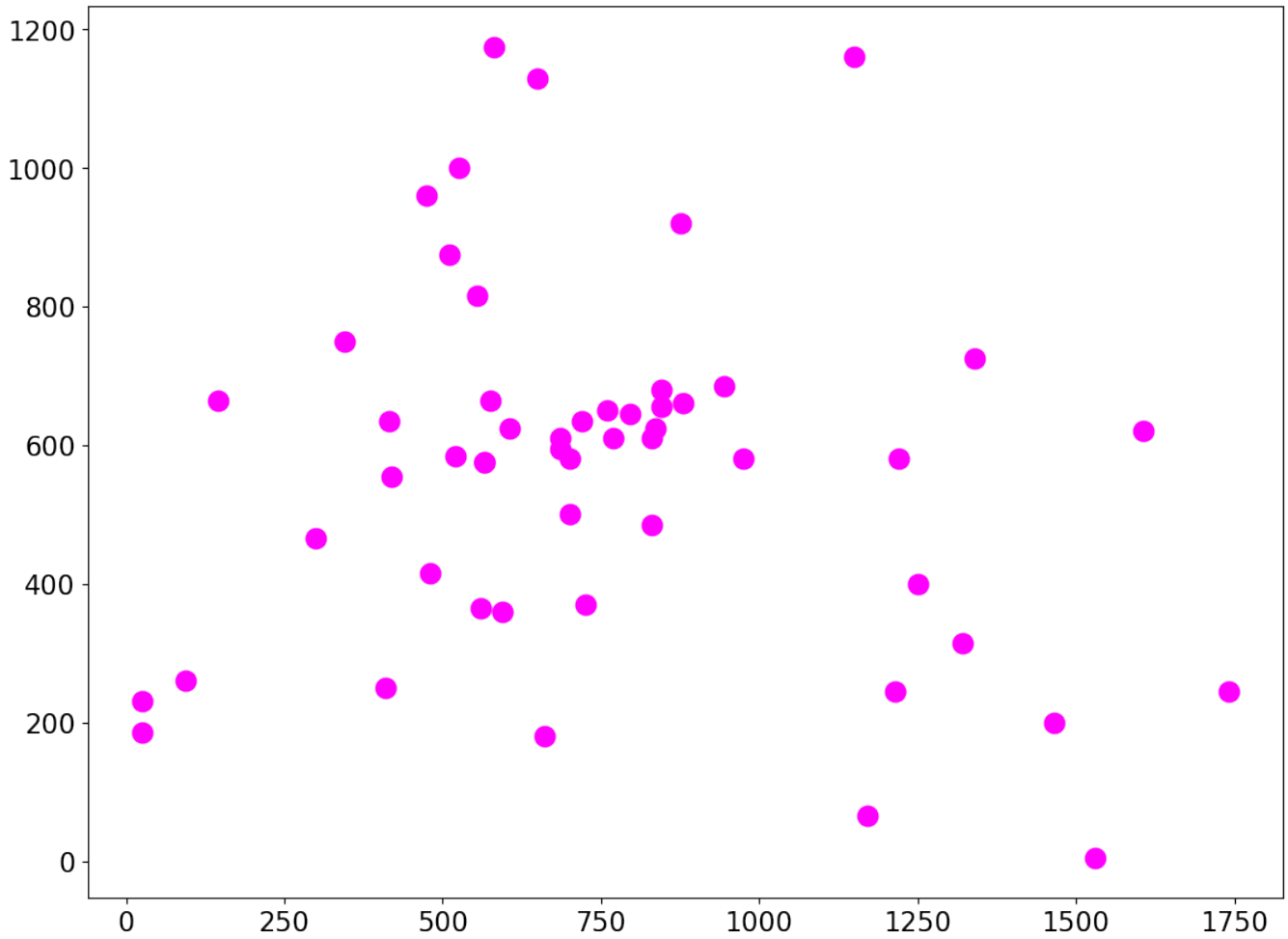
take a look at
https://www.sfu.ca/~ssurjano/optimization.html
or look for CEC2017 benchmark https://www.kaggle.com/code/kooaslansefat/cec-2017-benchmark

the traveling salesperson problem (TSP)

The basic traveling salesperson problems are:

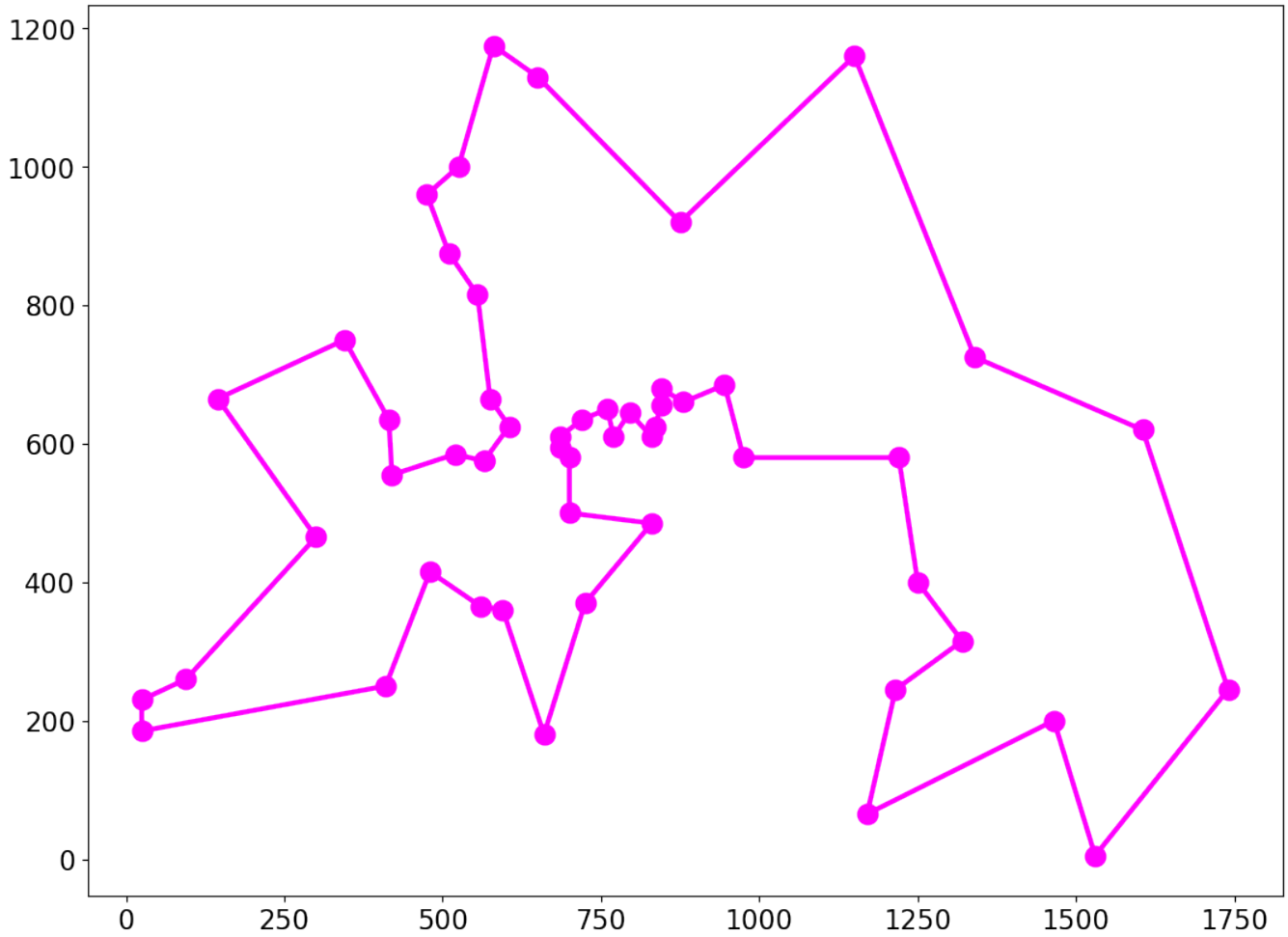
- Given n locations or cities and their interconnections in the 2D plane, tell whether there is a closed tour through all cities that visits each city exactly once and has a length below a certain threshold.
  This is an NP-complete decision problem.
- Given n cities in the 2D plane, find a shortest closed tour through all cities that visits each city exactly once.
  This is an NP-hard search problem.
- Given n cities in the 2D plane, find all shortest closed tours through all cities that visit each city exactly once.
  This is an exponential time solver problem.
- Note, there are $n!$ possible tours through the n cities.
- TSP is one of the best studied problems in computer science.
- There are more varieties of TSPs (some details later).

traveling salesperson problem: first impressions



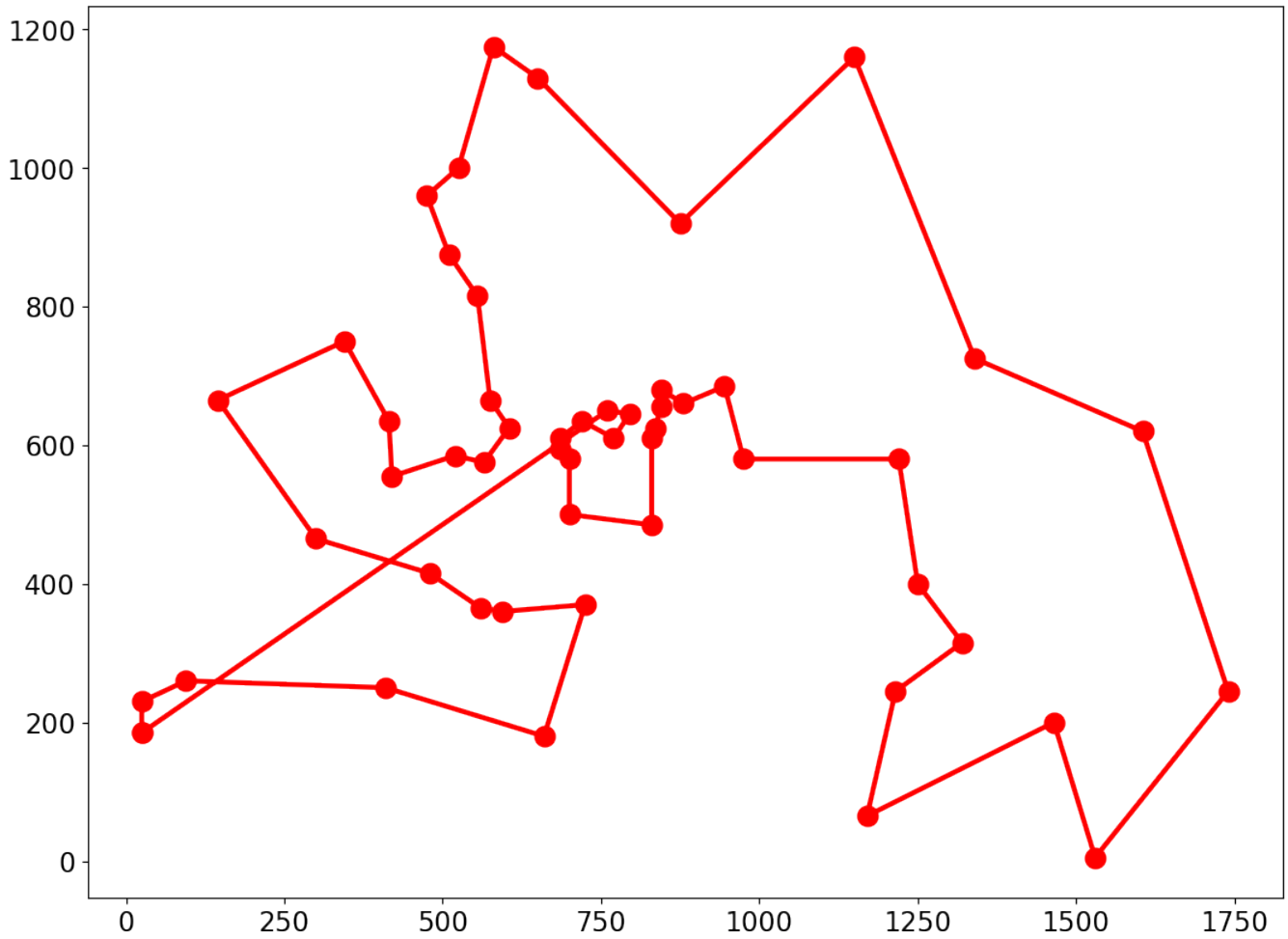
the cities distributed geographically (dataset berlin52)

traveling salesperson problem: first impressions



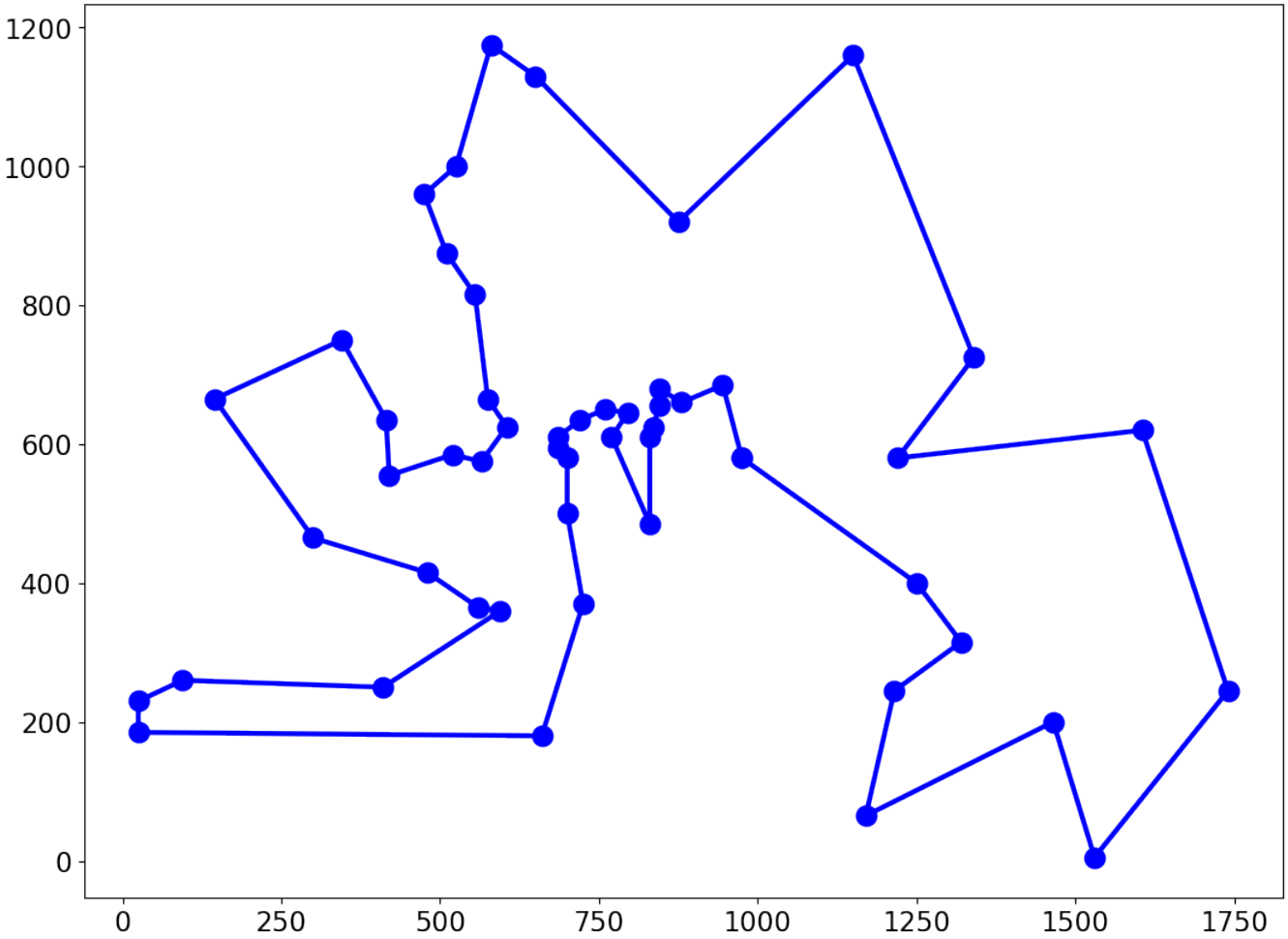
the best tour (known for this example): 0% relative error

traveling salesperson problem: first impressions



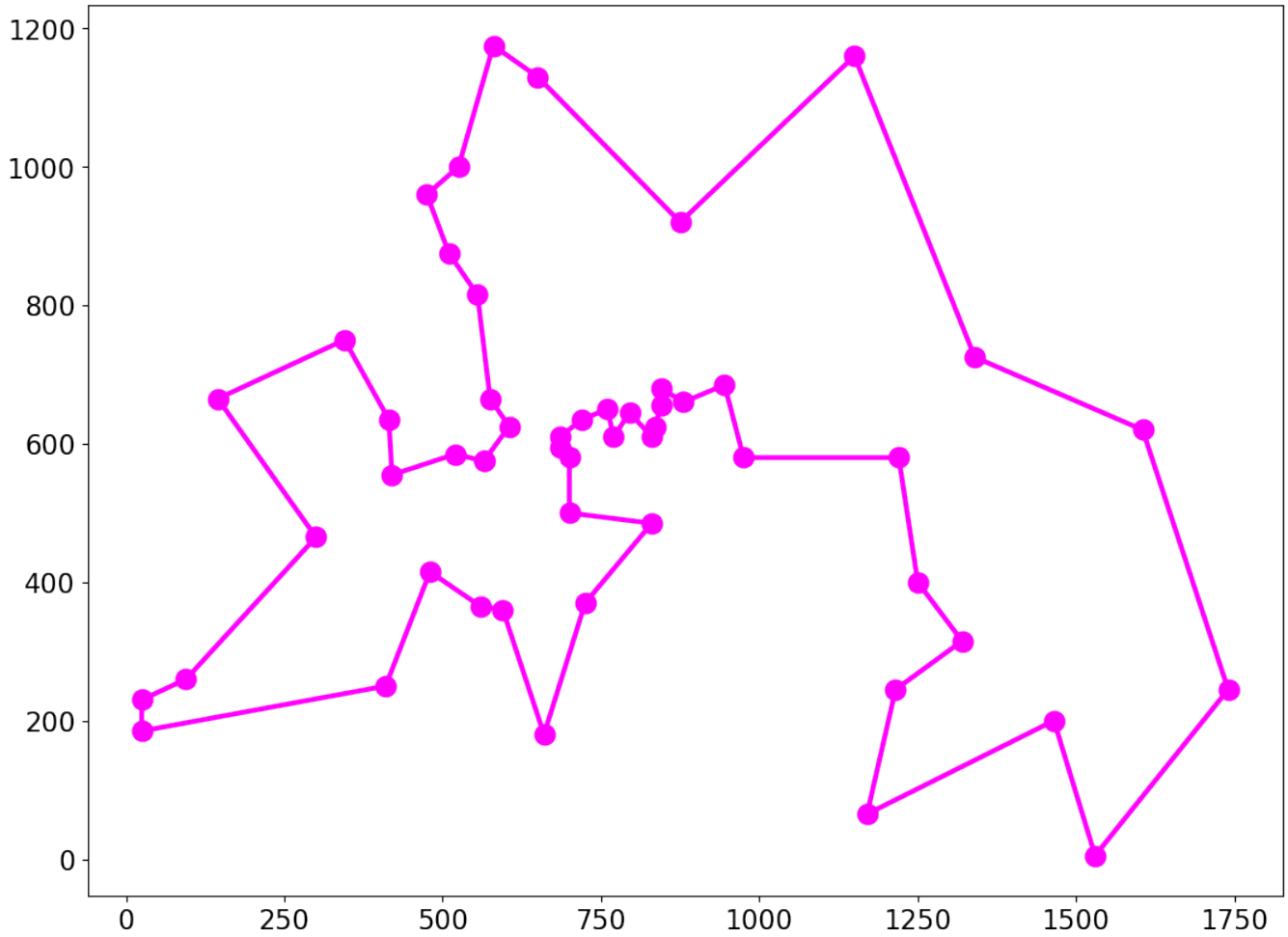
a closest-neighbor tour: 8.49% relative error

traveling salesperson problem: first impressions



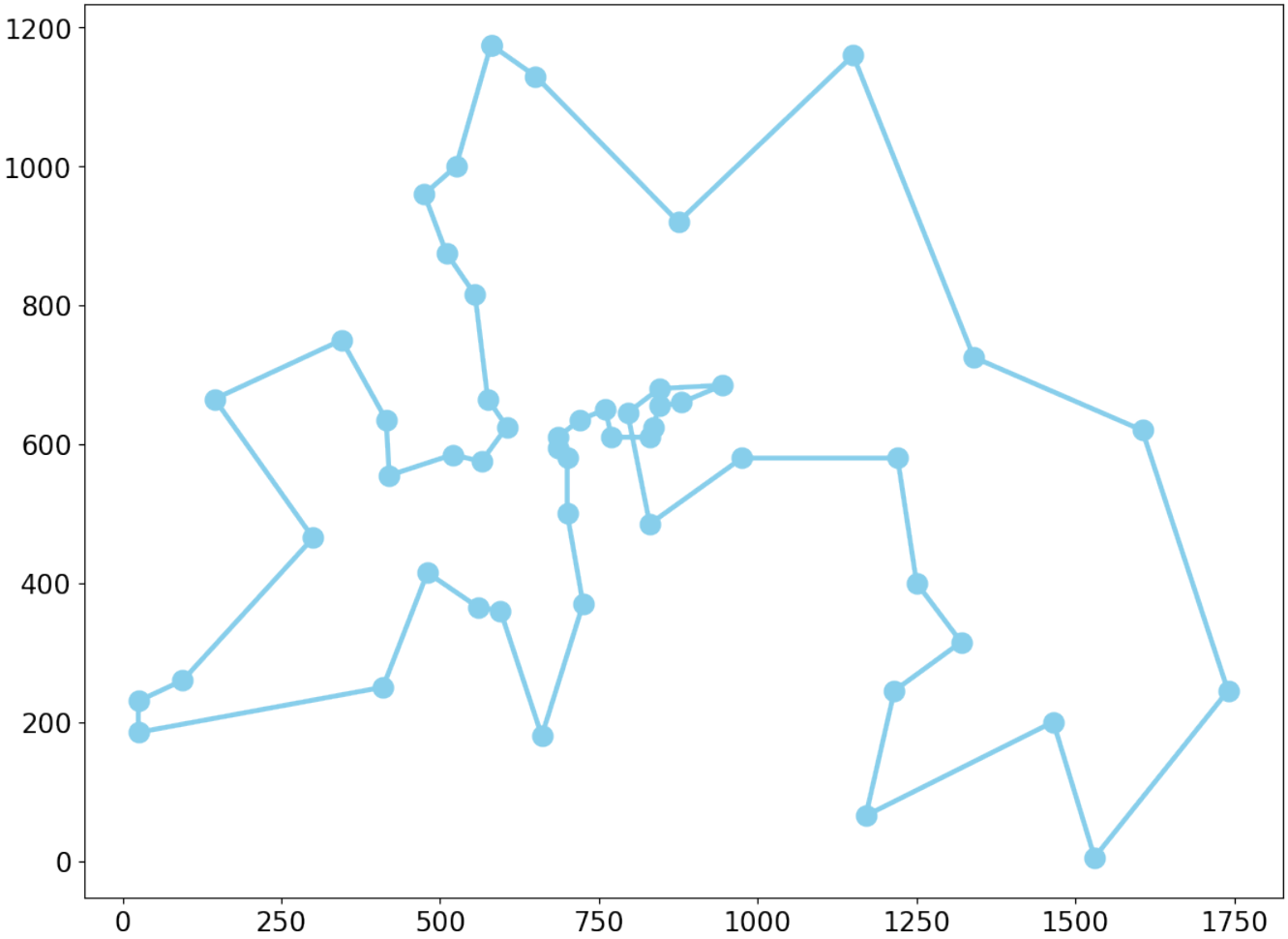
a pair-center tour: 7.28% relative error

traveling salesperson problem: first impressions



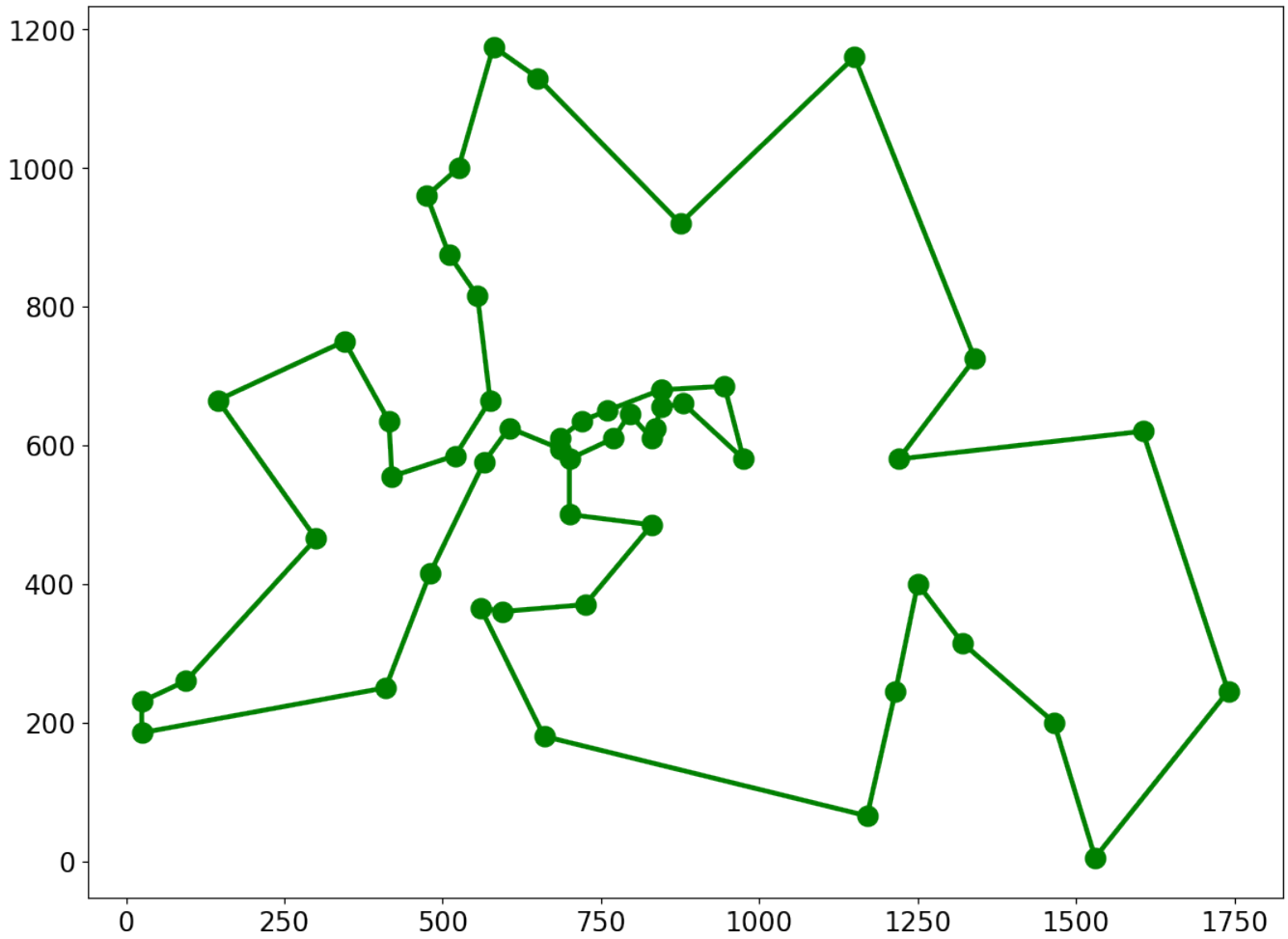
the best tour (known for this example): 0% relative error

traveling salesperson problem: first impressions



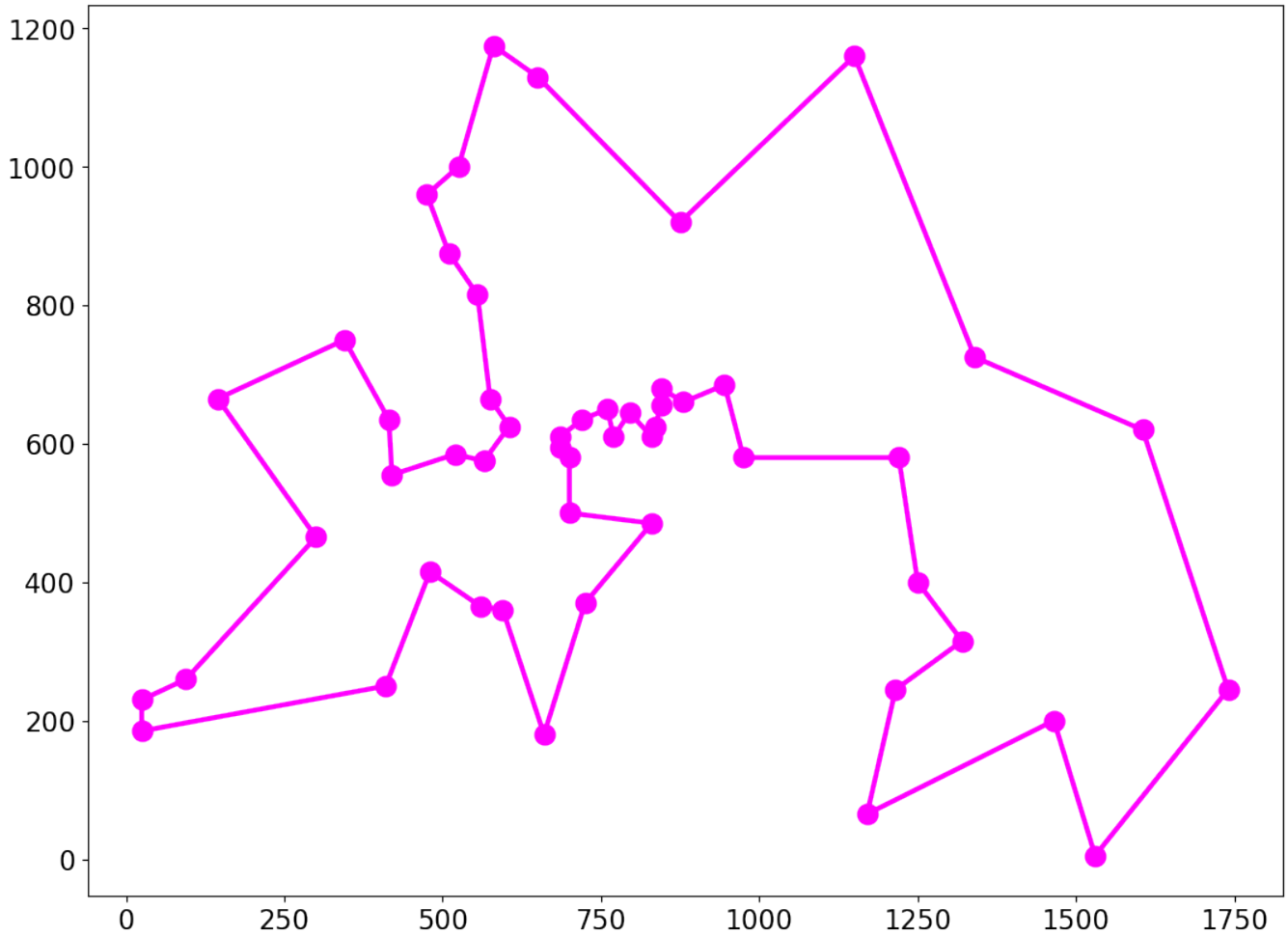
a quick tour (with Monte Carlo): 2.32% relative error

traveling salesperson problem: first impressions



a genetic algorithm tour: 8.37% relative error

traveling salesperson problem: first impressions



the best tour (known for this example): 0% relative error

lower bounds on tour length of TSP

- simple bound: sum of minimal distances to neighbors
- Held-Karp bound
  (typically comes close to 1% on random instances and below 2% on TSPLIB, arguments for the bound are quite complicated)

upper bounds on tour length of TSP

- tour around minimal spanning tree yields $\leq 2 \cdot L_{opt}$
  runtime $\mathcal{O}(n^2)$
- Christofides algorithm yields $\leq 1.5 \cdot L_{opt}$
  runtime $\mathcal{O}(n^2 \log n)$

known solving algorithms for TSP

- brute force exhaustive search $\mathcal{O}(n!)$
  (quite easy to implement)
- Bellman-Held-Karp dynamic programming for Euclidean TSP
  $\mathcal{O}(n^2 2^n)$ time and $\mathcal{O}(n 2^n)$ space
  (not covered here, please refer to advanced algorithms in computer science)
- state-of-the-art solver Concorde.
- state-of-the-art approximative solver LKH.

known approximation heuristics for TSP

start with some small tour generated by a simple heuristic, then:

- use 2-opt moves (modifing tour by changing two edges, for instance, to eliminate crossings in Euclidean TSP);
- or use 3-opt moves (modifing tour by changing three edges);
- or use Lin-Kernighan heuristic algorithm
  (variable mixture of 2-opt and 3-opt moves),
  currently the best known heuristic strategy.

TSP on a grid

let's take a look at a regular $m \times n$ grid (e.g. checker board)

- an optimal tour on a regular grid is easy to build
- optimal length:
  - $L_{opt} = n \cdot m$ if n or m even
  - $L_{opt} = n \cdot m - 1 + \sqrt{2}$ if $n \cdot m$ odd
- there are many! optimal tours

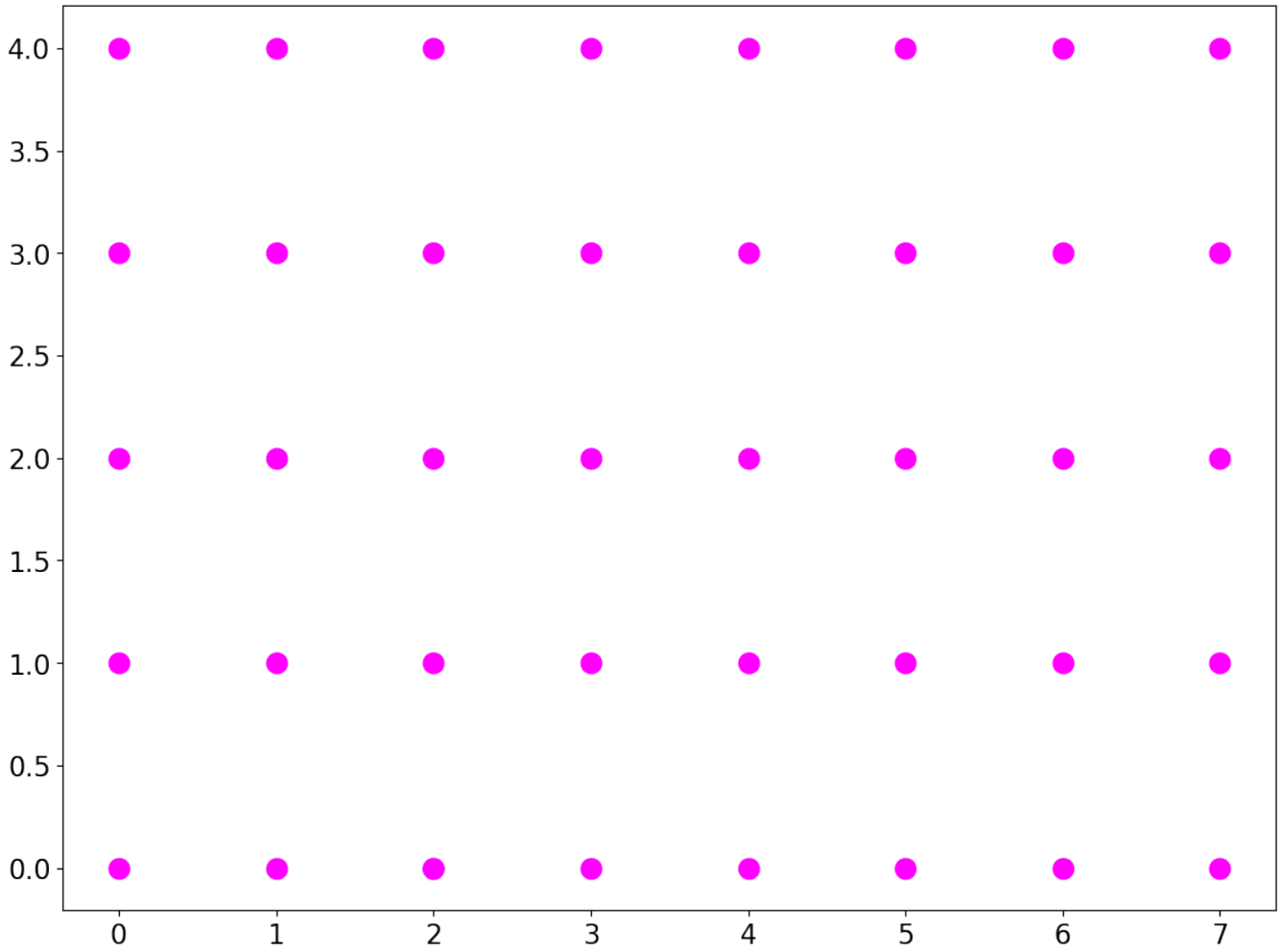
More on traveling salesperson problem

- The version of the TSP problem as shown until now is a special case of a more general problem definition:
- Let $G = (V, E)$ be a graph. V are the nodes (or locations), E are the edges (or connections) with some weight (e.g., distance, time, cost).
- Goal: find a minimal tour through all nodes.
- Particularly we talked about the Euclidean TSP, where the nodes are points in the Euclidean plane and the distance among all pairs, hence complete graph, is just the Euclidean distance.
- One step to be more general is, just require the triangular condition to be met (then, possibly, the pair-center approach cannot be used as we have no distances for the centers), this is called the metric TSP (mTSP).
- Moreover, the distances might be asymmetric, i.e., going in one direction is different from going in the other (ATSP).
- or we have additional conditions: open loop, arrival time windows, asymmetric distances, interrupted tours, etc.

More recent results on the eTSP

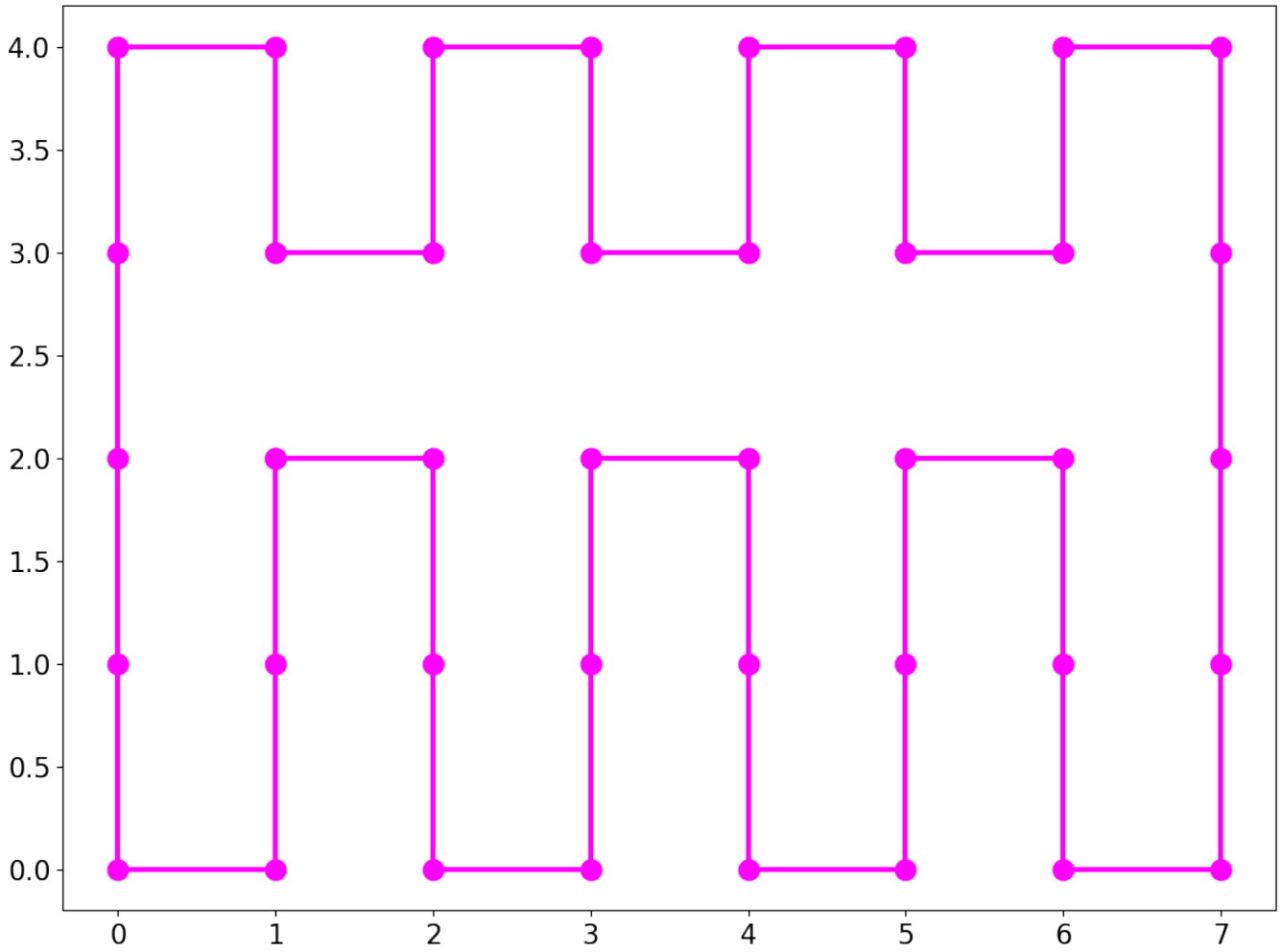
- In the 90's it was shown that eTSP can be solved in $\mathcal{O}(2^{O(\sqrt{n}\log n)})$.
- In the 10's this was improved to $\mathcal{O}(2^{\sqrt{n}})$, and with certain arguments that further improvement may be very unlikely.
- One recent result of complexity theory is that eTSP has a polynomial time approximation scheme (PTAS) of $\mathcal{O}_{\varepsilon,d}(n\log n)$ (with fixed error ε and fixed dimension d), however, an implementation is not available (to my knowledge).
- This has been improved to $\mathcal{O}_{\varepsilon,d}(n)$ with high probability (2013).

traveling salesperson problem: first impressions



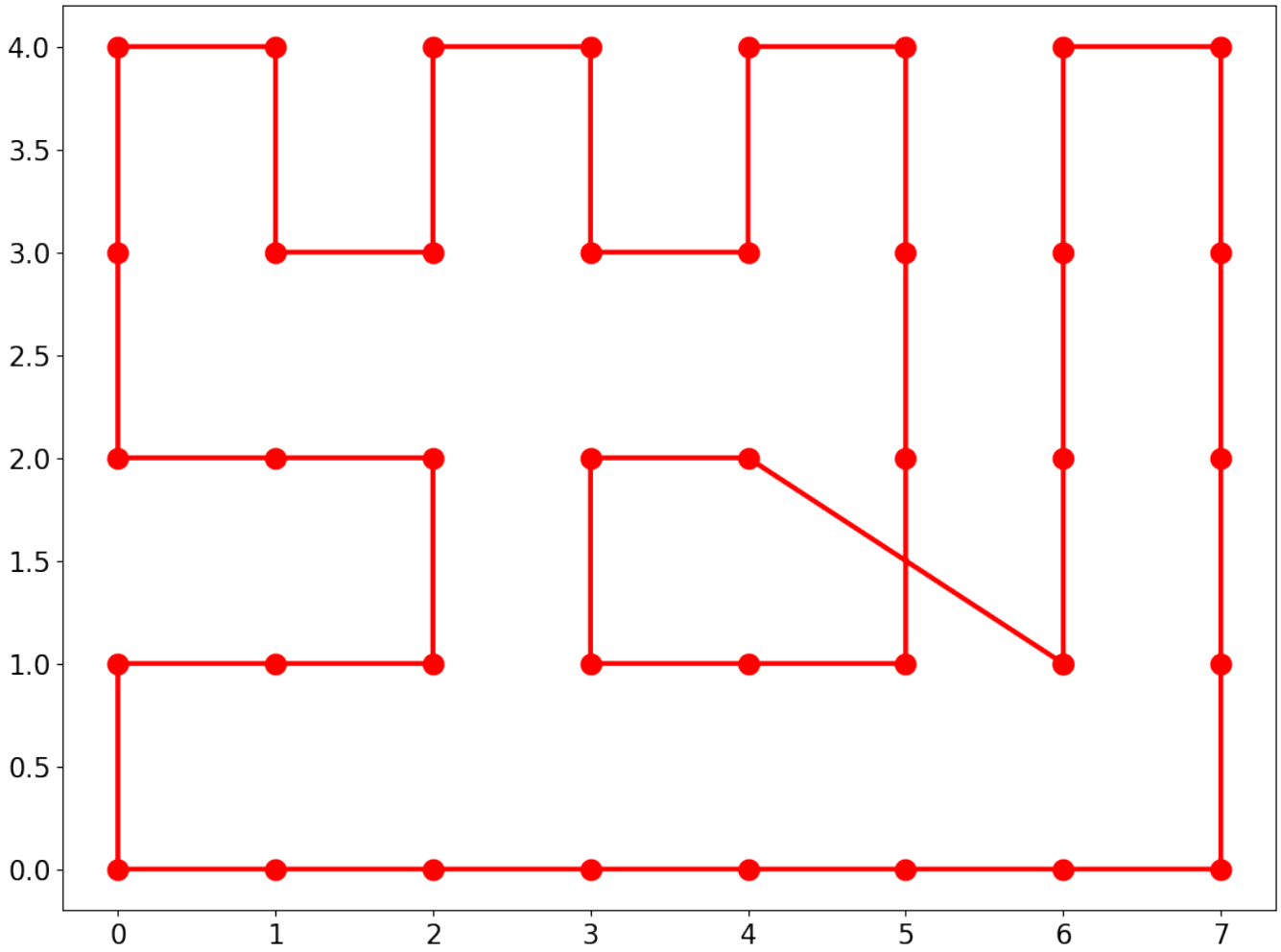
the cities distributed geographically

traveling salesperson problem: first impressions



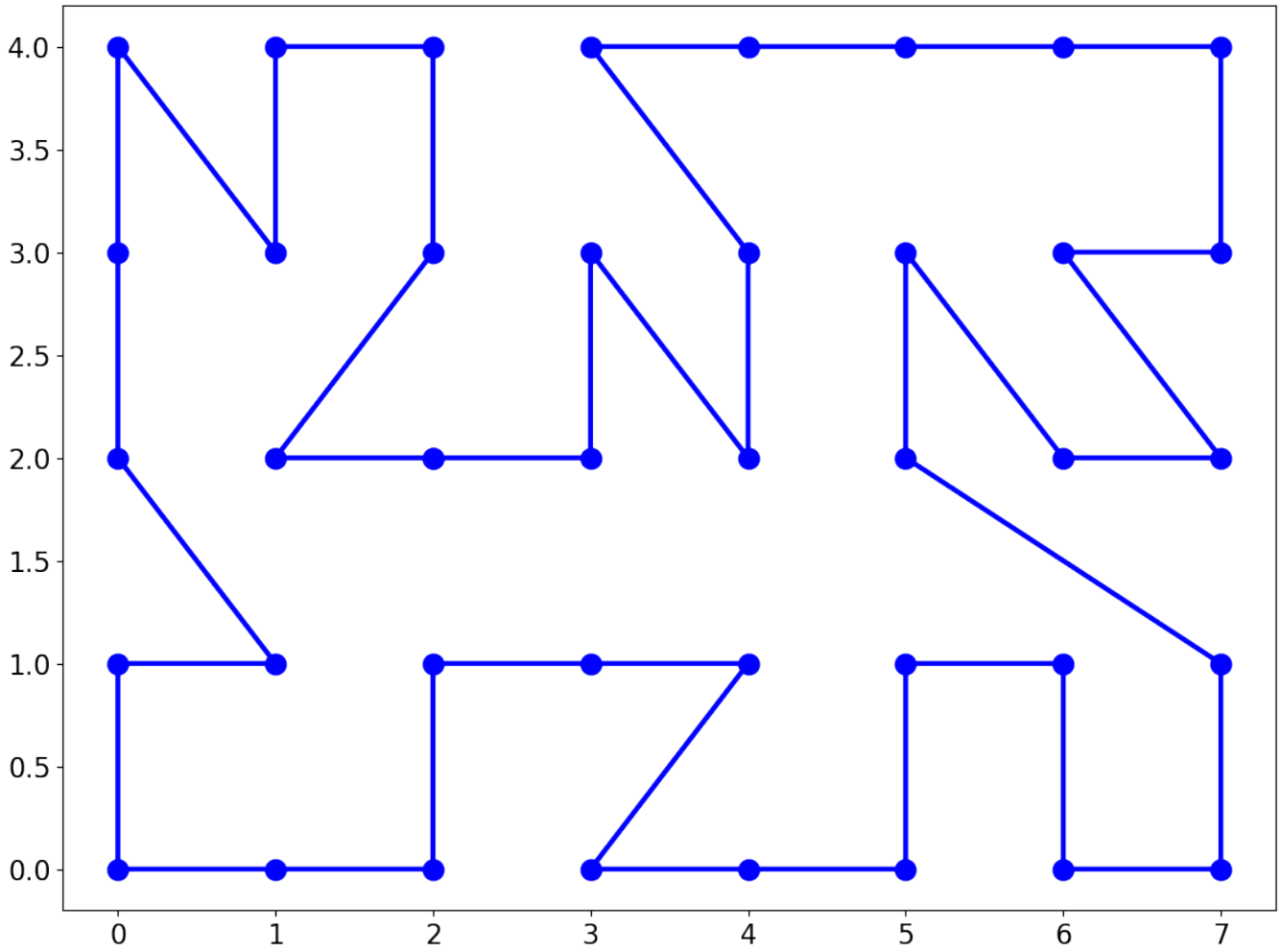
a best tour (trivial for this example): 0% relative error

traveling salesperson problem: first impressions



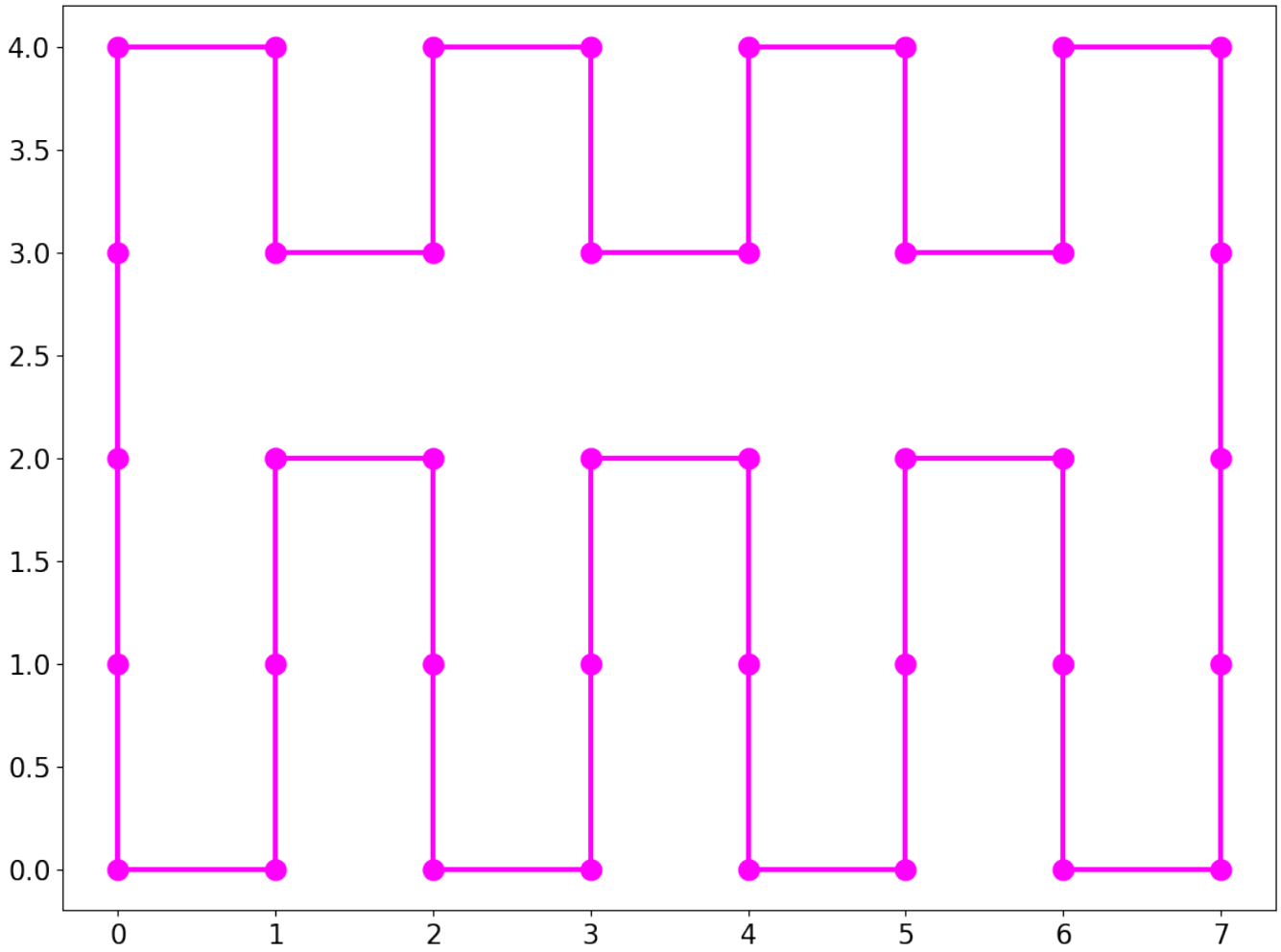
a closest-neighbor tour (with Monte Carlo): 3.09% relative error

traveling salesperson problem: first impressions



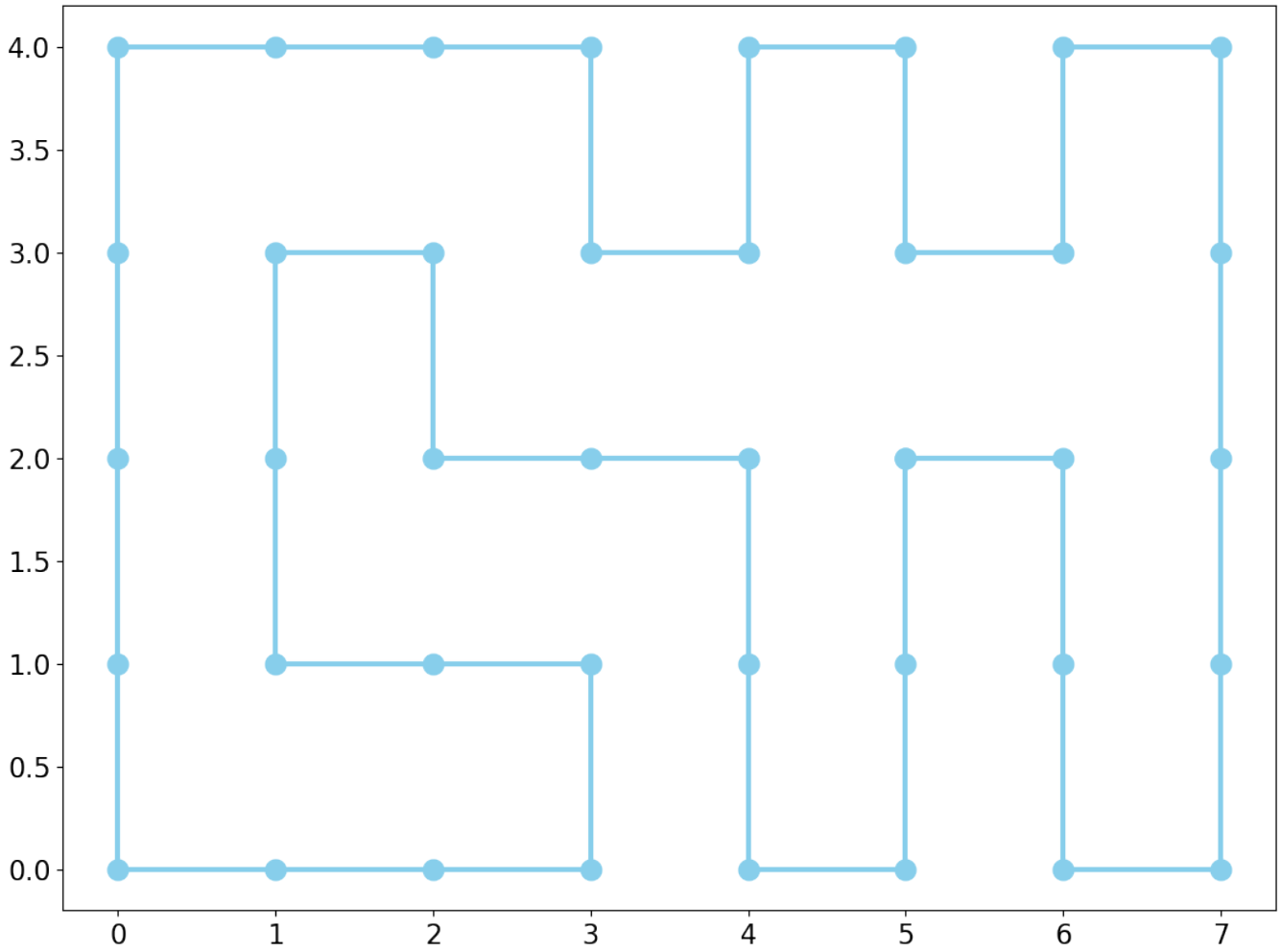
a pair-center tour: 14.46% relative error

traveling salesperson problem: first impressions



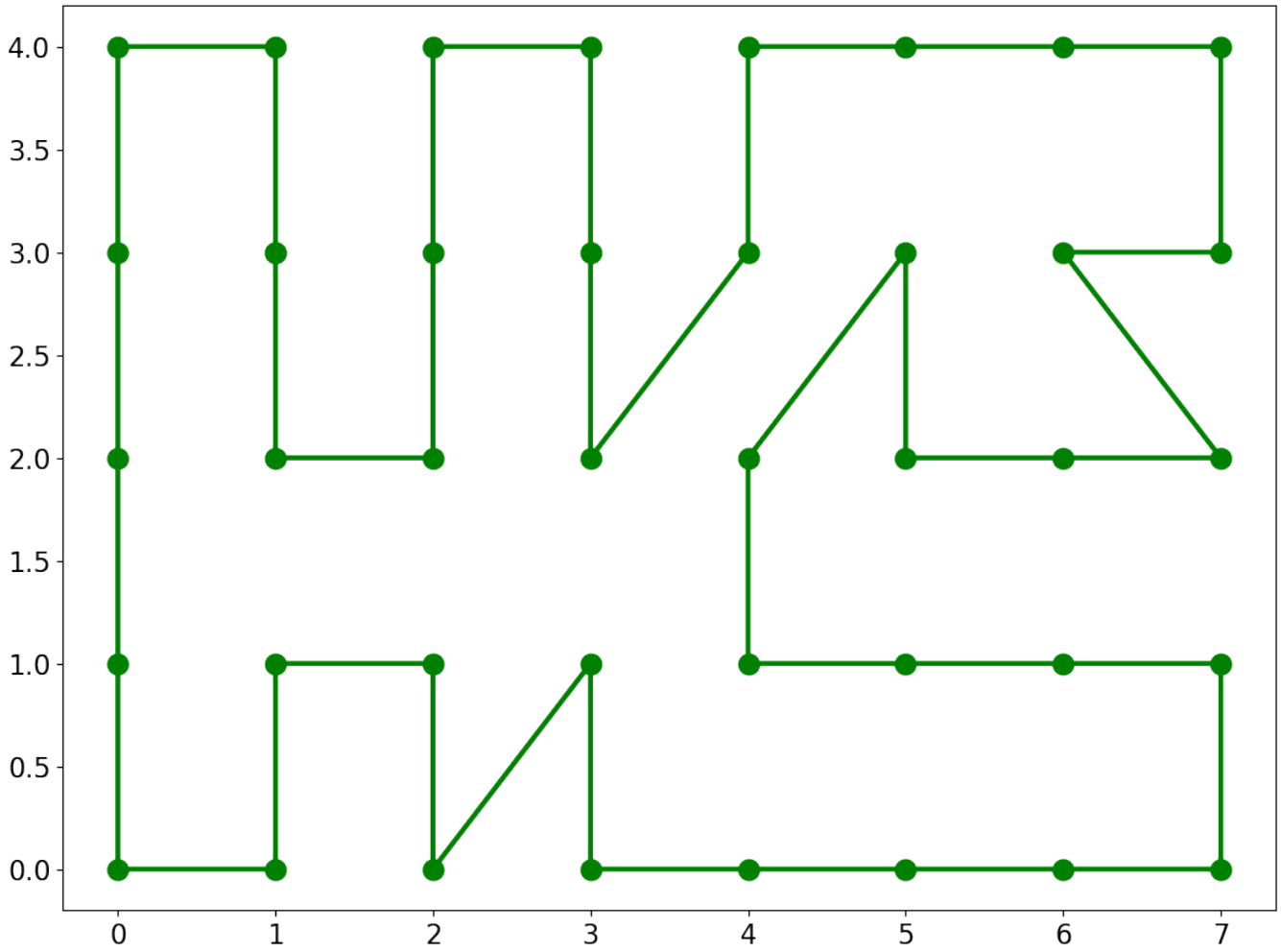
a best tour (trivial for this example): 0% relative error

traveling salesperson problem: first impressions



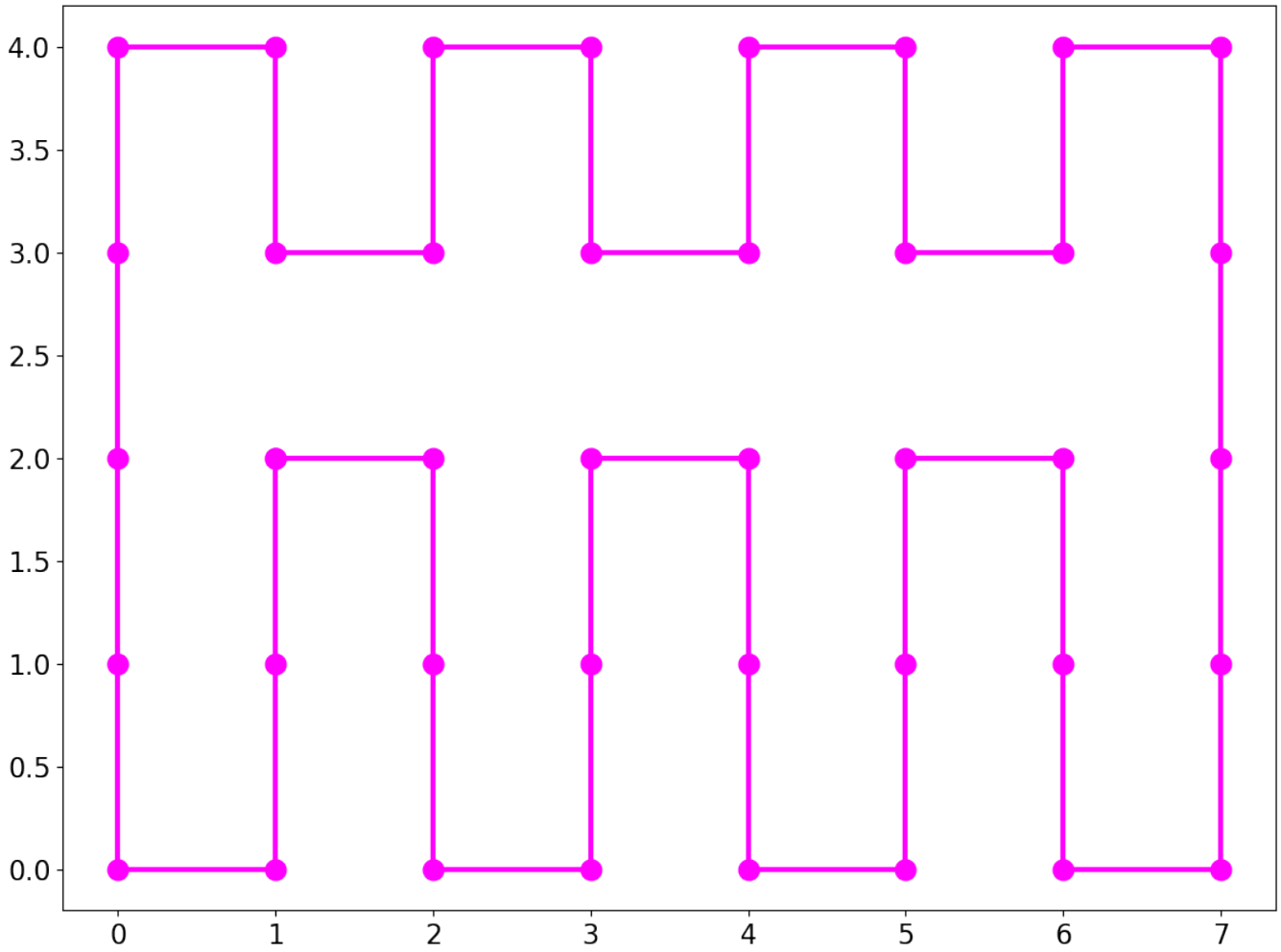
a quick tour (with Monte Carlo): 0.00% relative error

traveling salesperson problem: first impressions



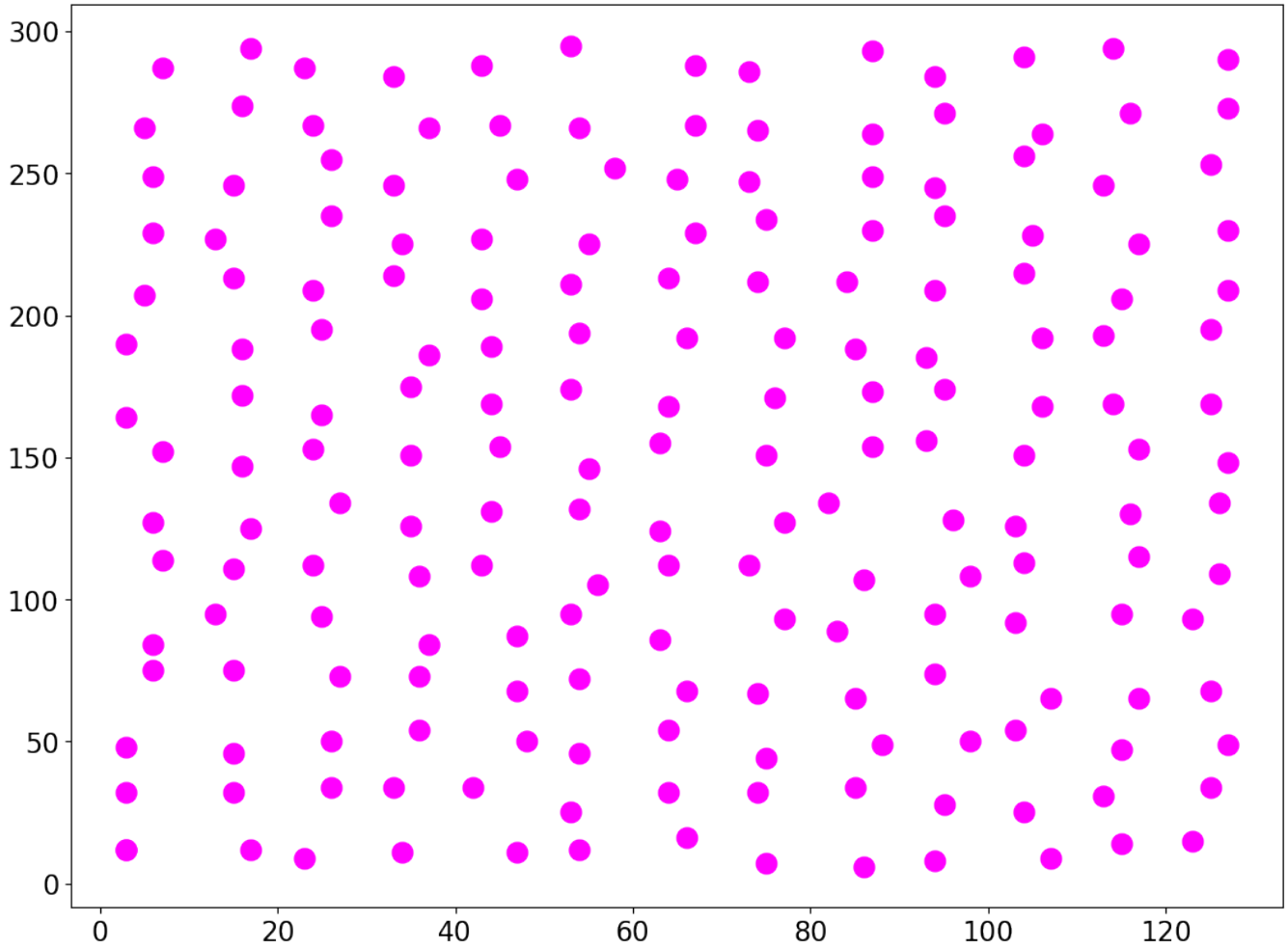
a genetic algorithm tour: 4.14% relative error

traveling salesperson problem: first impressions



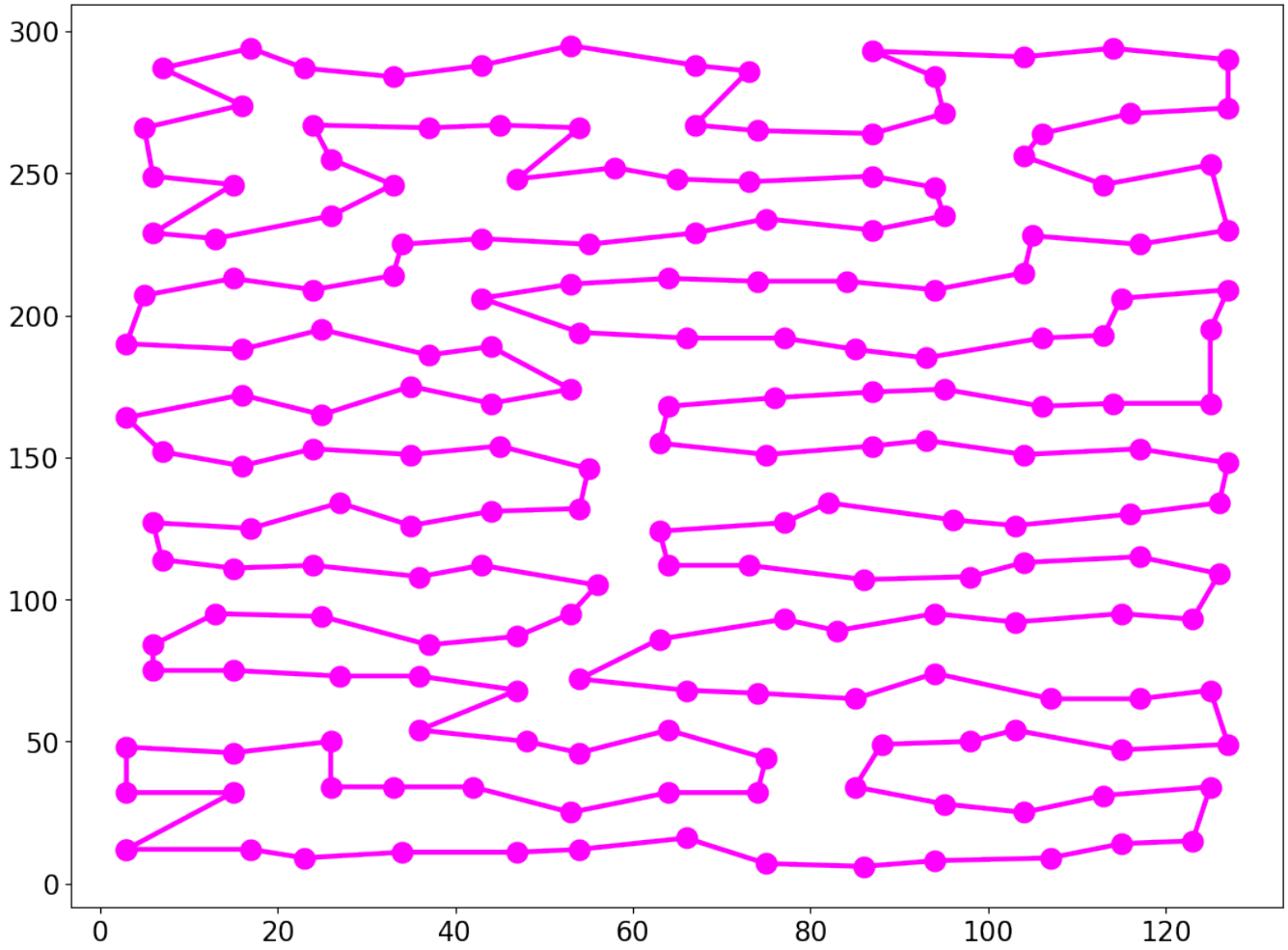
a best tour (trivial for this example): 0% relative error

traveling salesperson problem: first impressions



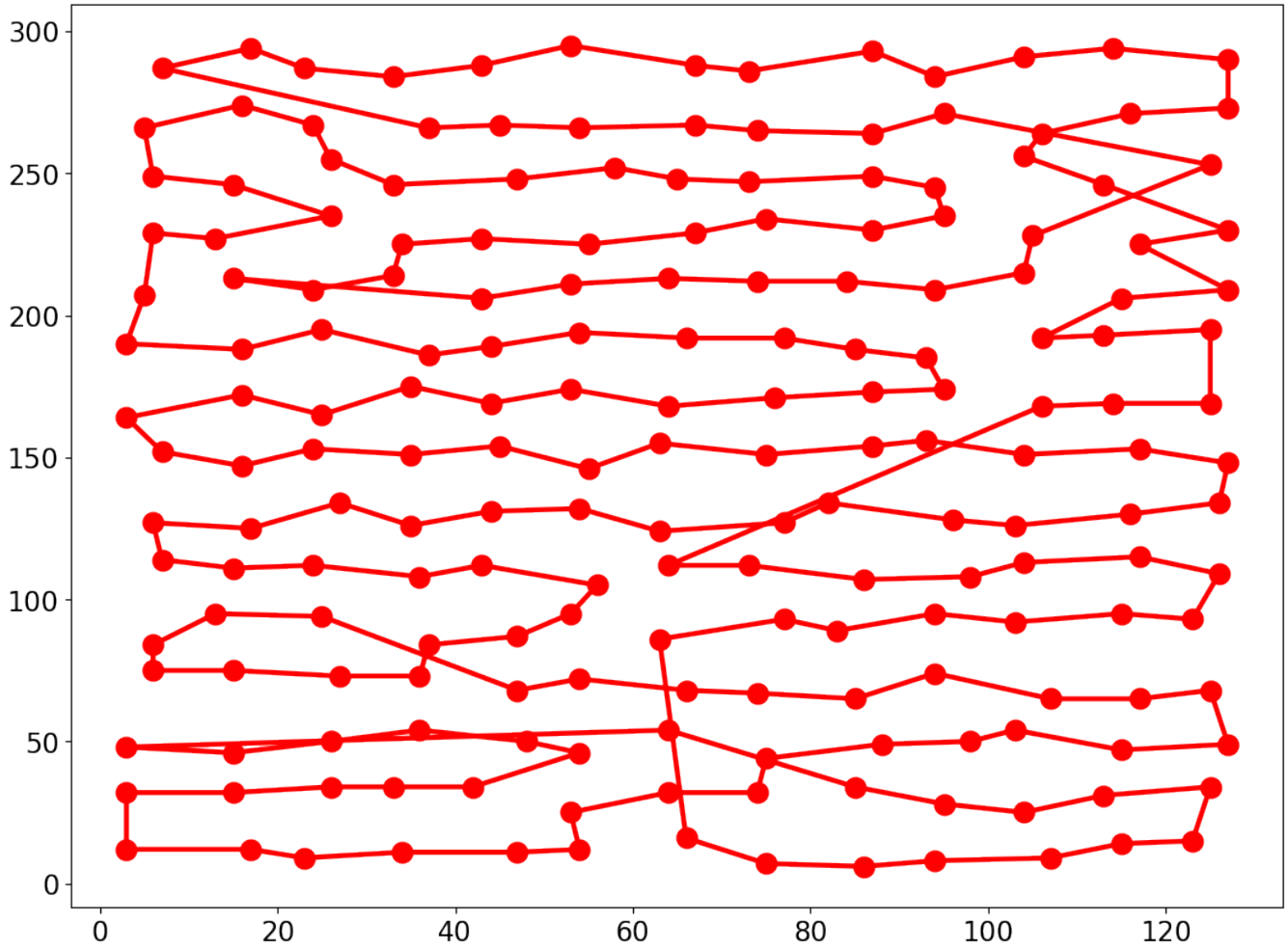
the locations distributed in the plane

traveling salesperson problem: first impressions



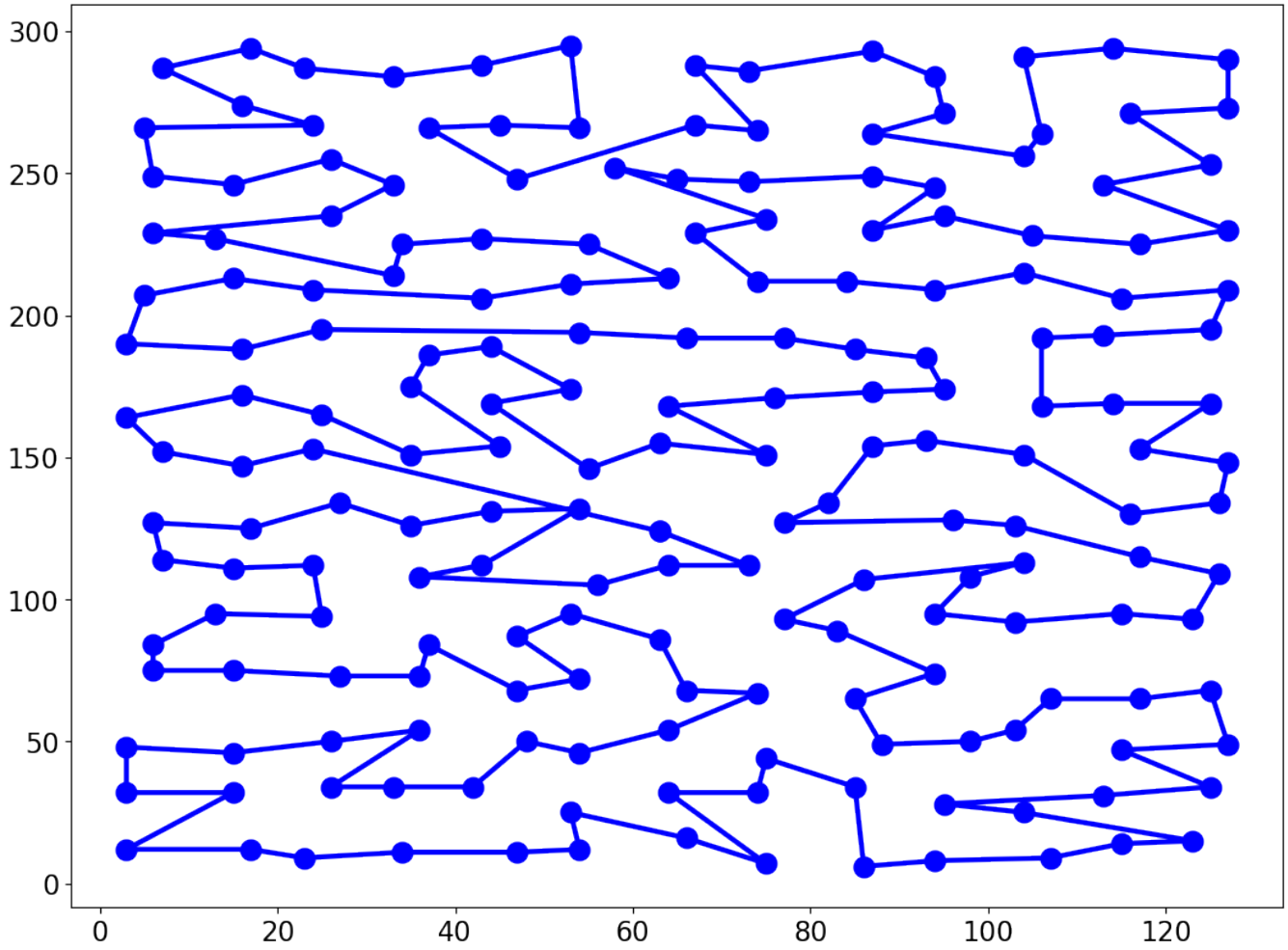
the best tour (known for this example): 0% relative error

traveling salesperson problem: first impressions



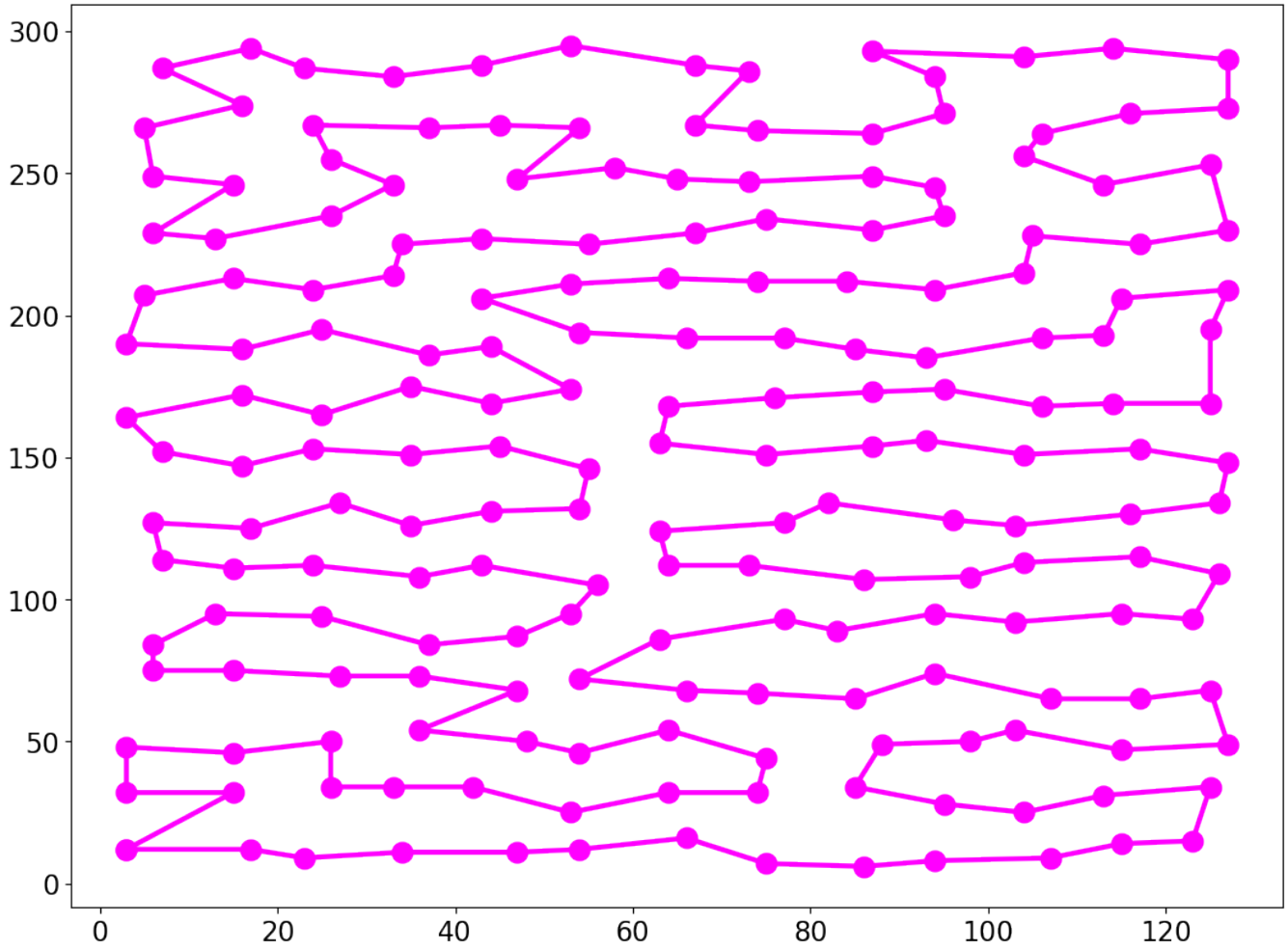
a closest-neighbor tour (with Monte Carlo): 10.23% relative error

traveling salesperson problem: first impressions



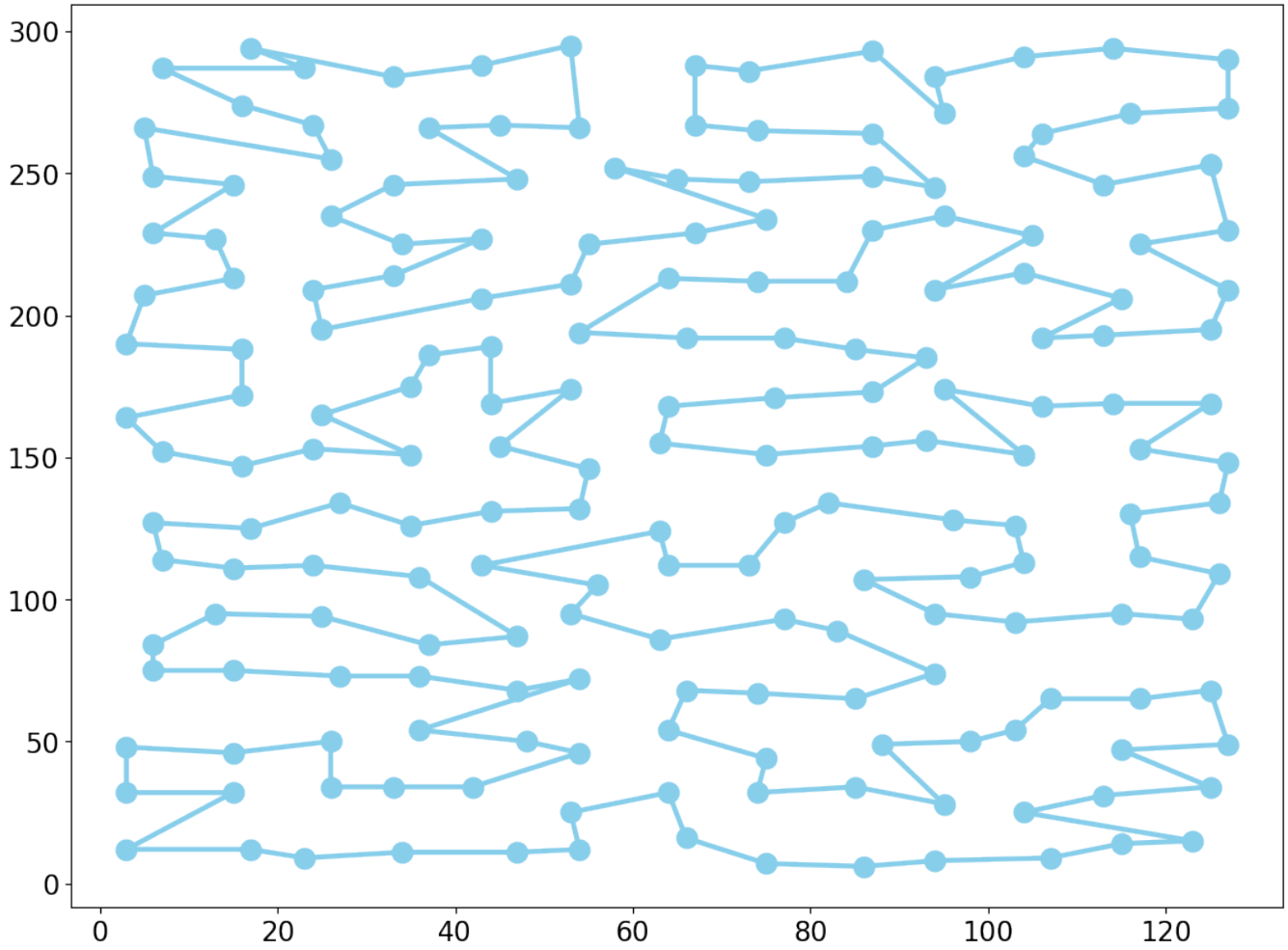
a pair-center tour: 13.93% relative error

traveling salesperson problem: first impressions



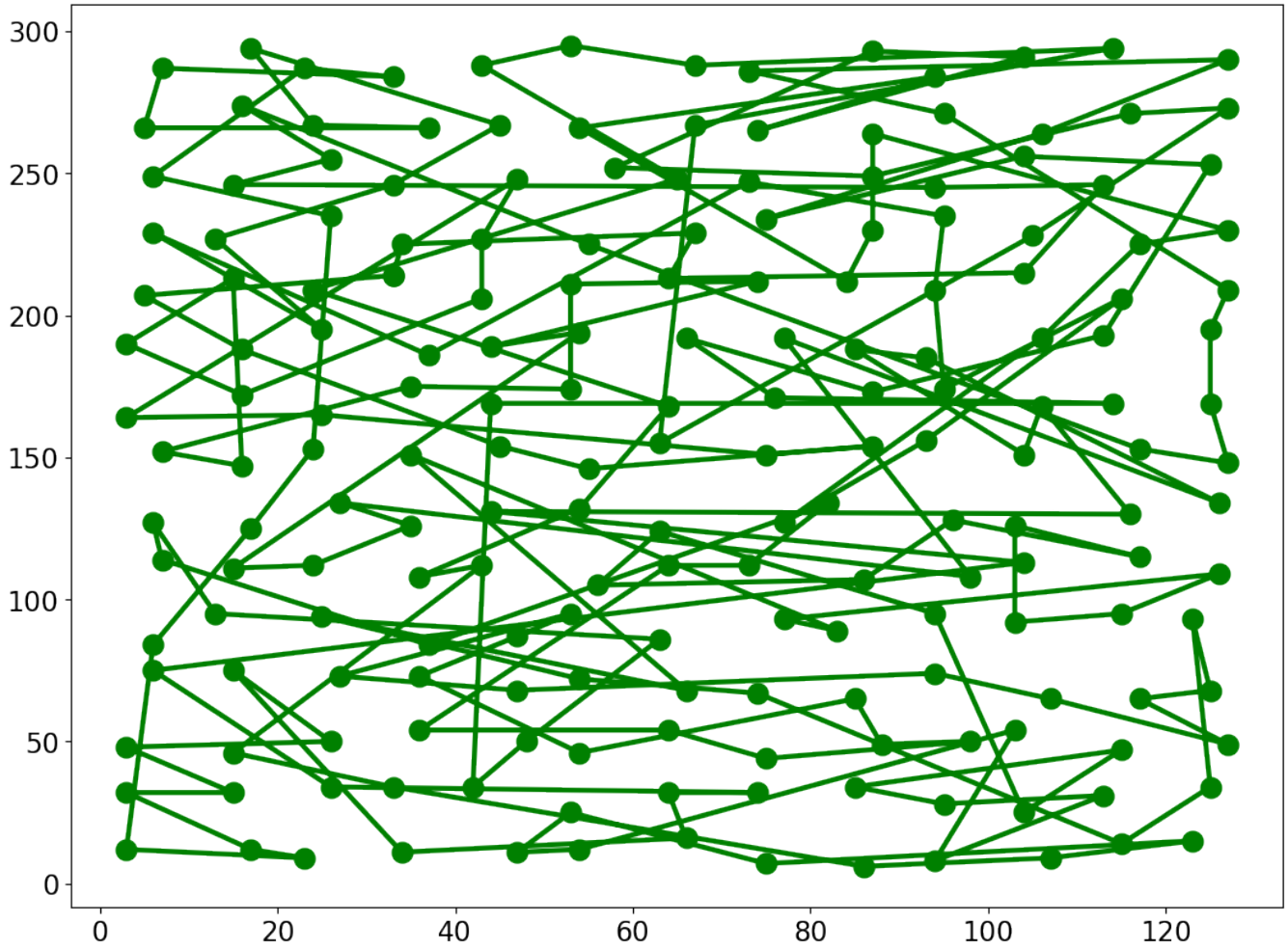
the best tour (known for this example): 0% relative error

traveling salesperson problem: first impressions



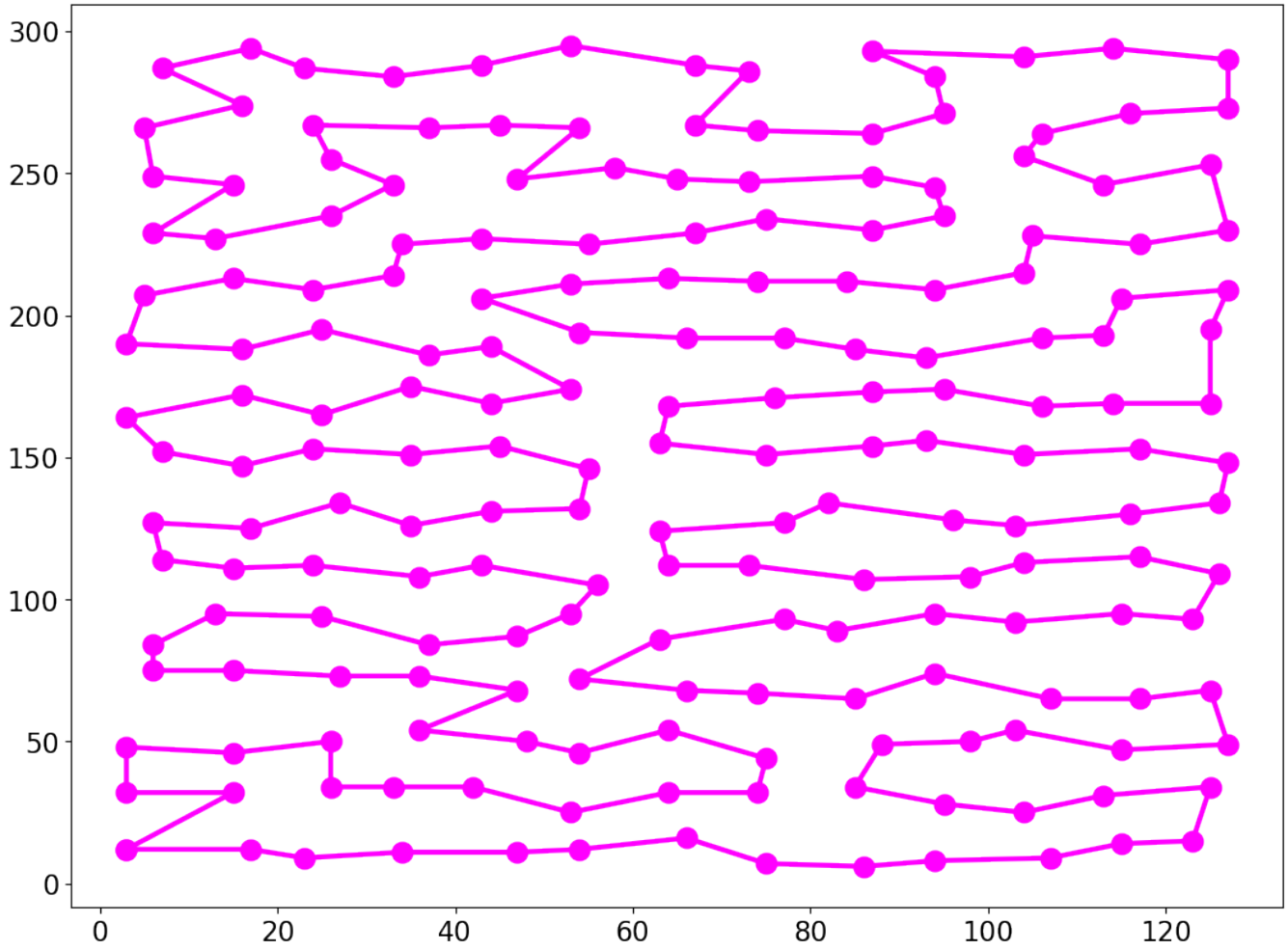
a quick tour (with Monte Carlo): 9.66% relative error

traveling salesperson problem: first impressions



a genetic algorithm (GA) tour: 205.65% relative error

traveling salesperson problem: first impressions



the best tour (known for this example): 0% relative error

errors for the above heuristics

problem	heuristic	relative error
berlin52	closest neighbor tour	8.49
	quick tour	2.32
	pair-center tour	7.28
	genetic algorithm tour	8.37
	pair-center tour improved	0.00
	Lin-Kernighan tour	0.00
rat195	closest neighbor tour	10.23
	quick tour	9.66
	pair-center tour	13.93
	genetic algorithm tour	205.65
	pair-center tour improved	1.16
	Lin-Kernighan tour	0.00
block40	closest neighbor tour	3.09
	quick tour	0.00
	pair-center tour	14.46
	genetic algorithm tour	4.14
	improved pair-center tour	0.00
	Lin-Kernighan tour	0.00

Your goal: make GA-tour consistently better than pair-center tour or quick tour.

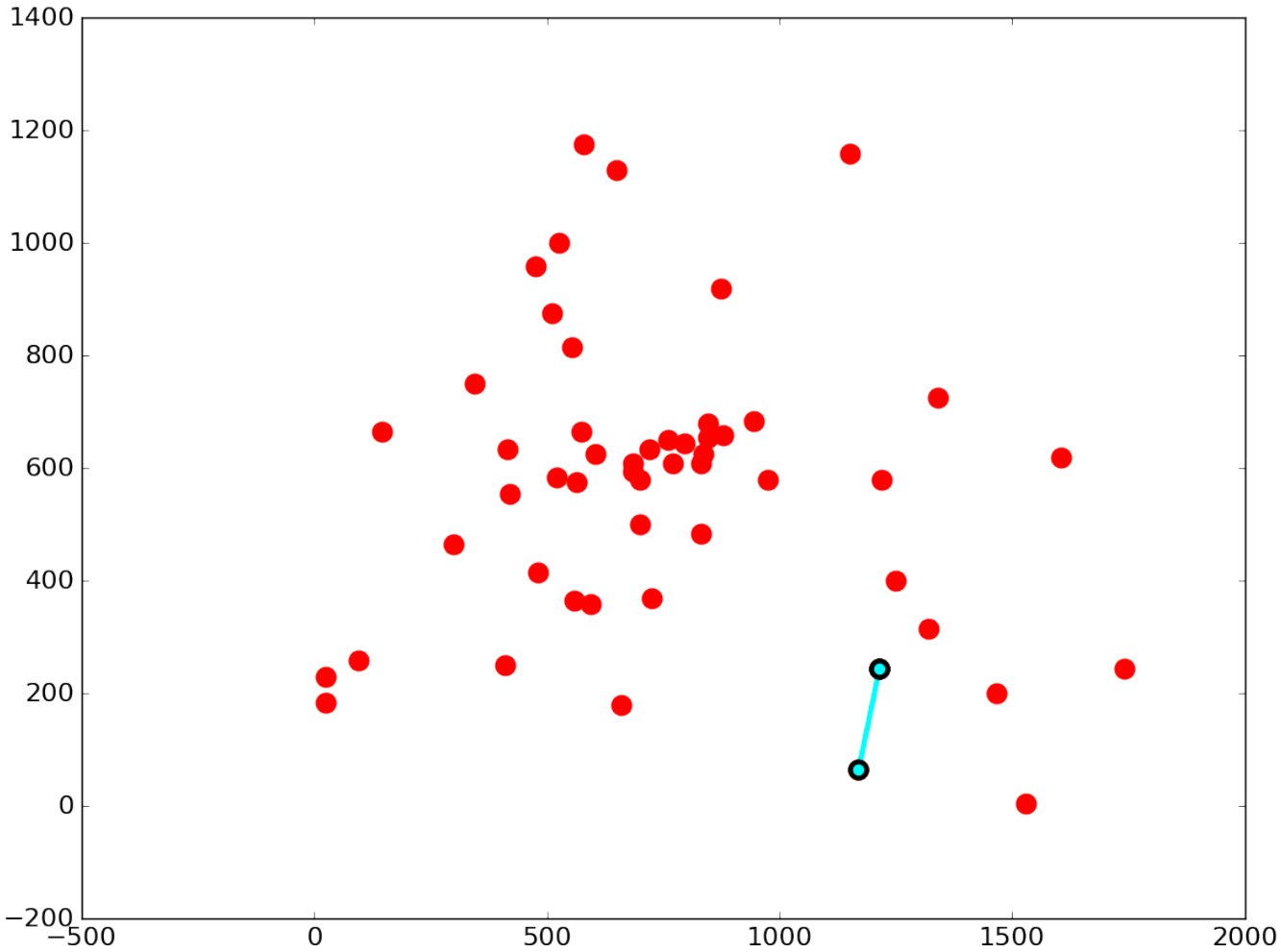
How does the closest-neighbor algorithms work?

The classical closest-neighbor algorithm is a greedy algorithm with a small random component:

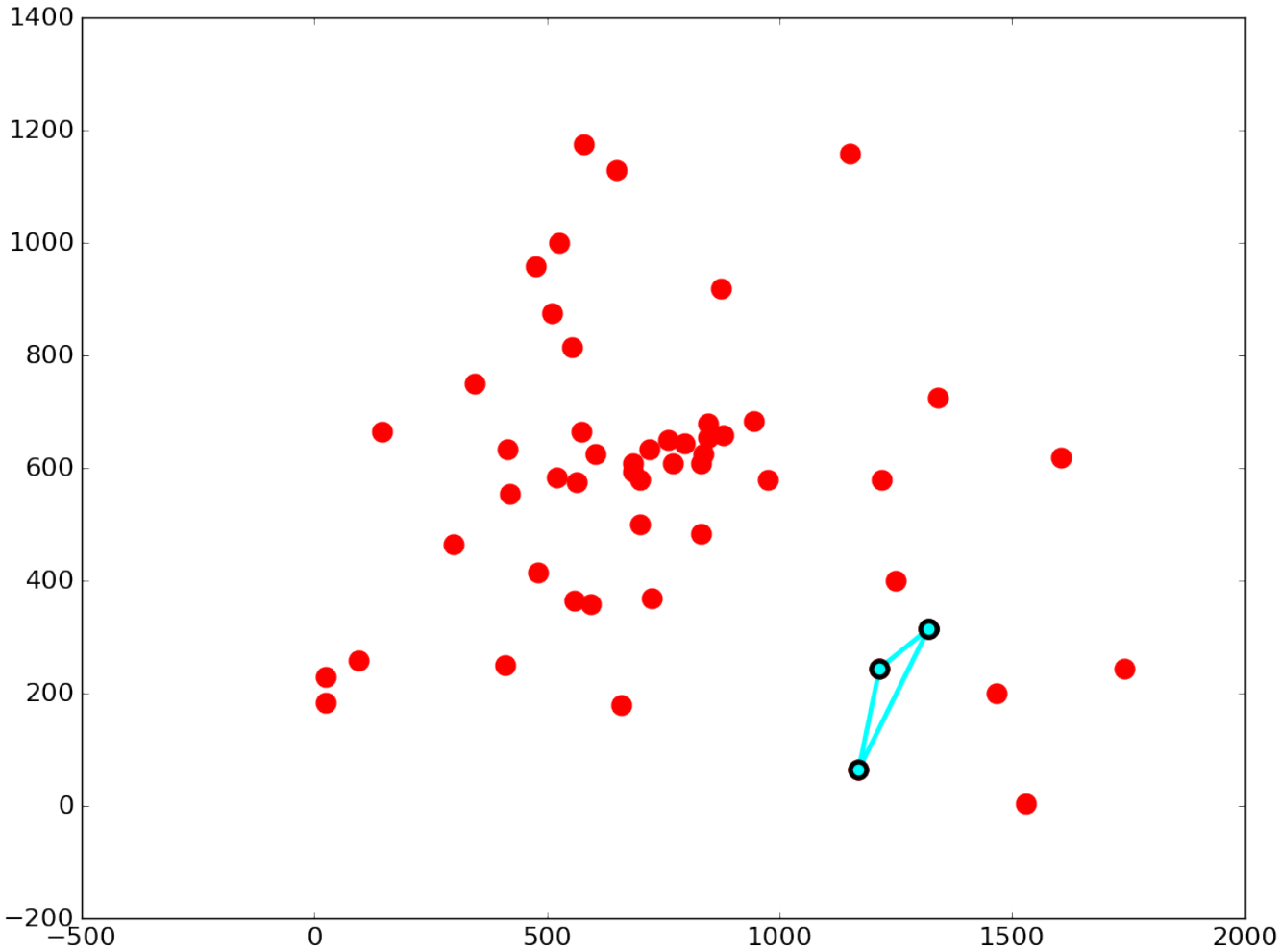
- select a random city
- while there are still unconnected cities
  - connect to the closest unconnected neighbor
  - use a random tie break
- connect the first with the last city

- runtime is $\mathcal{O}(n^2)$,
- can be run in Monte Carlo fashion keeping the shortest tour
- worst tour may have a length up to $0.5 \cdot \log n \cdot L_{opt}$

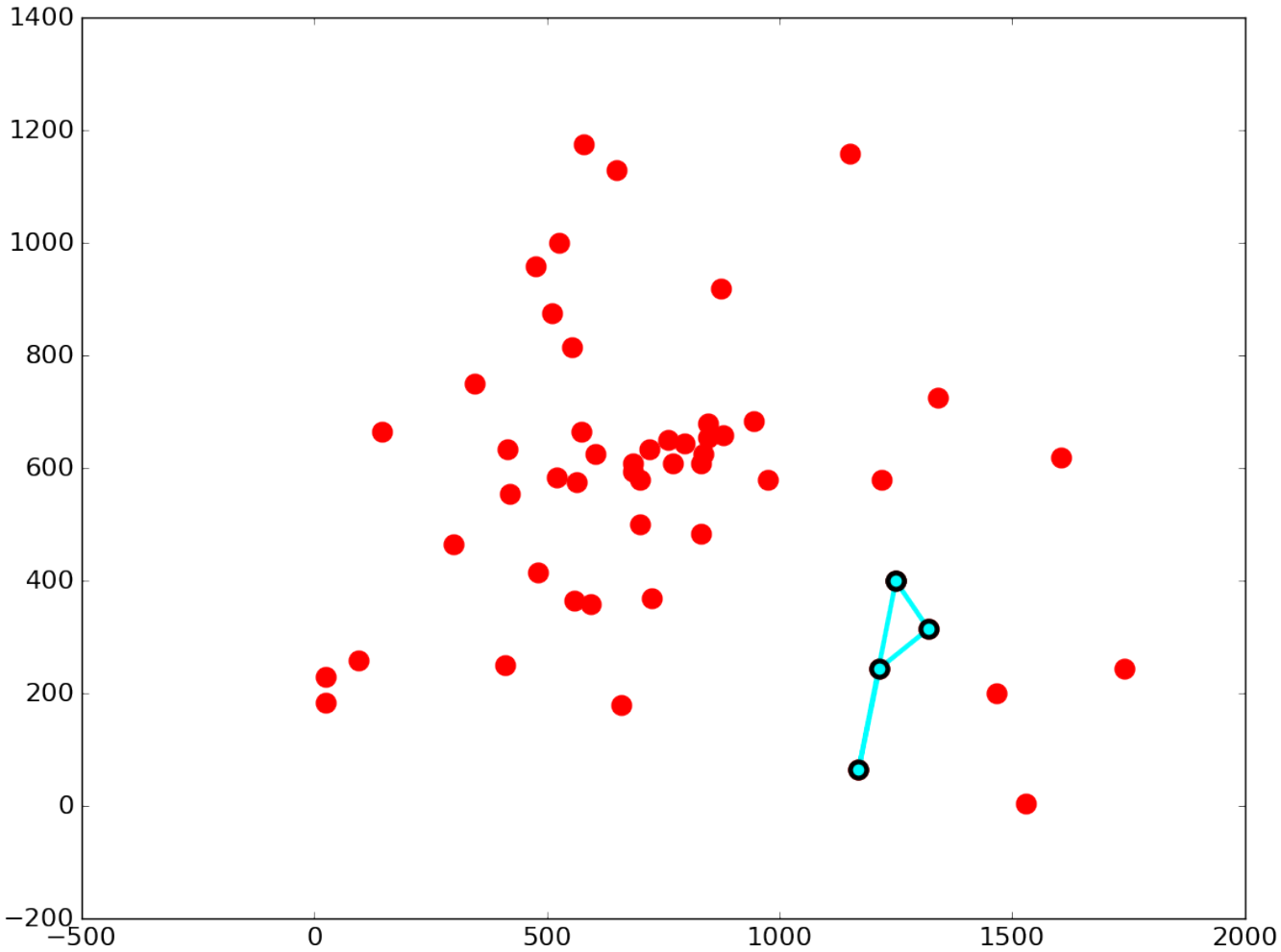
animation of the closest-neighbor algorithm



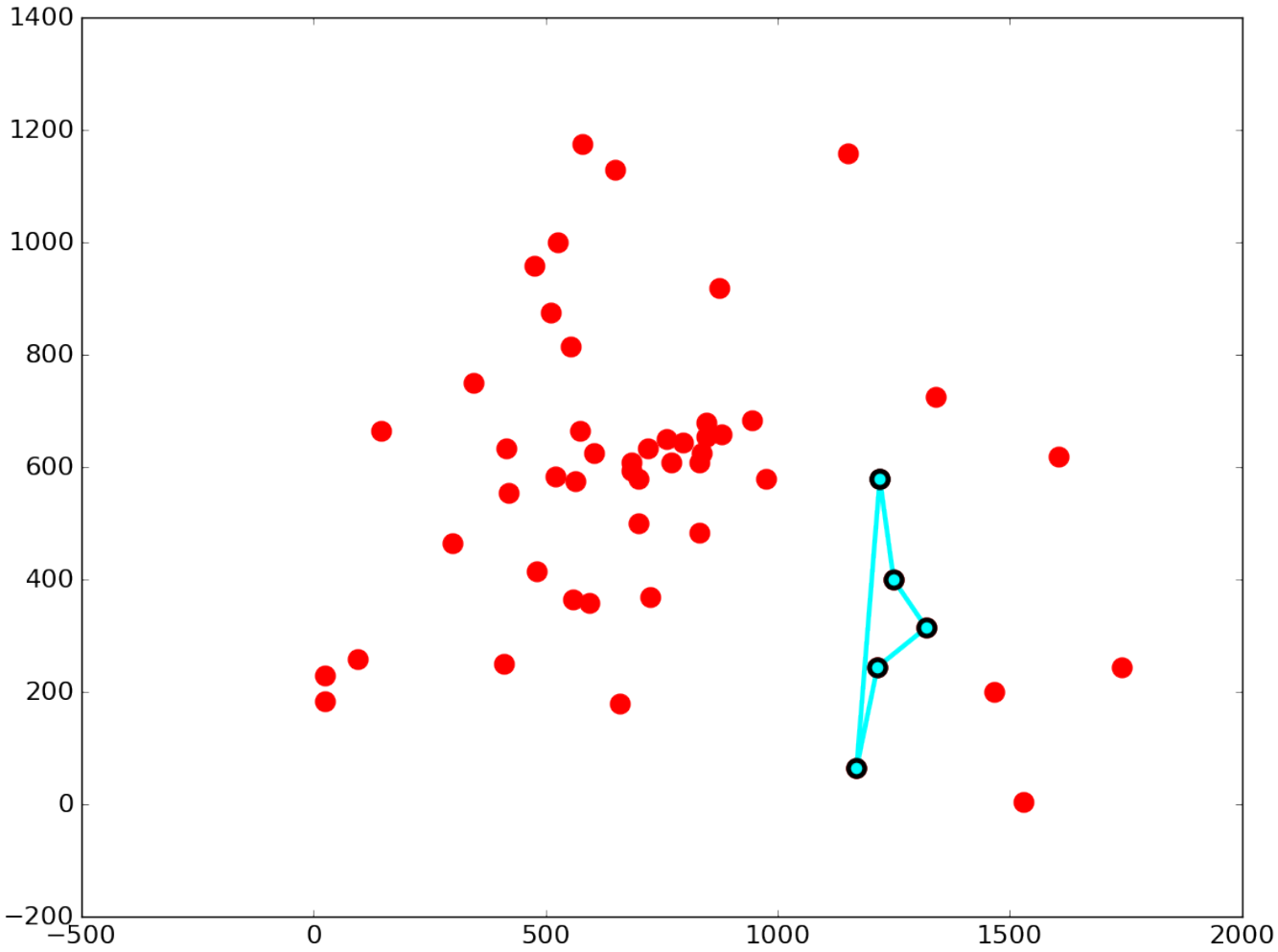
animation of the closest-neighbor algorithm



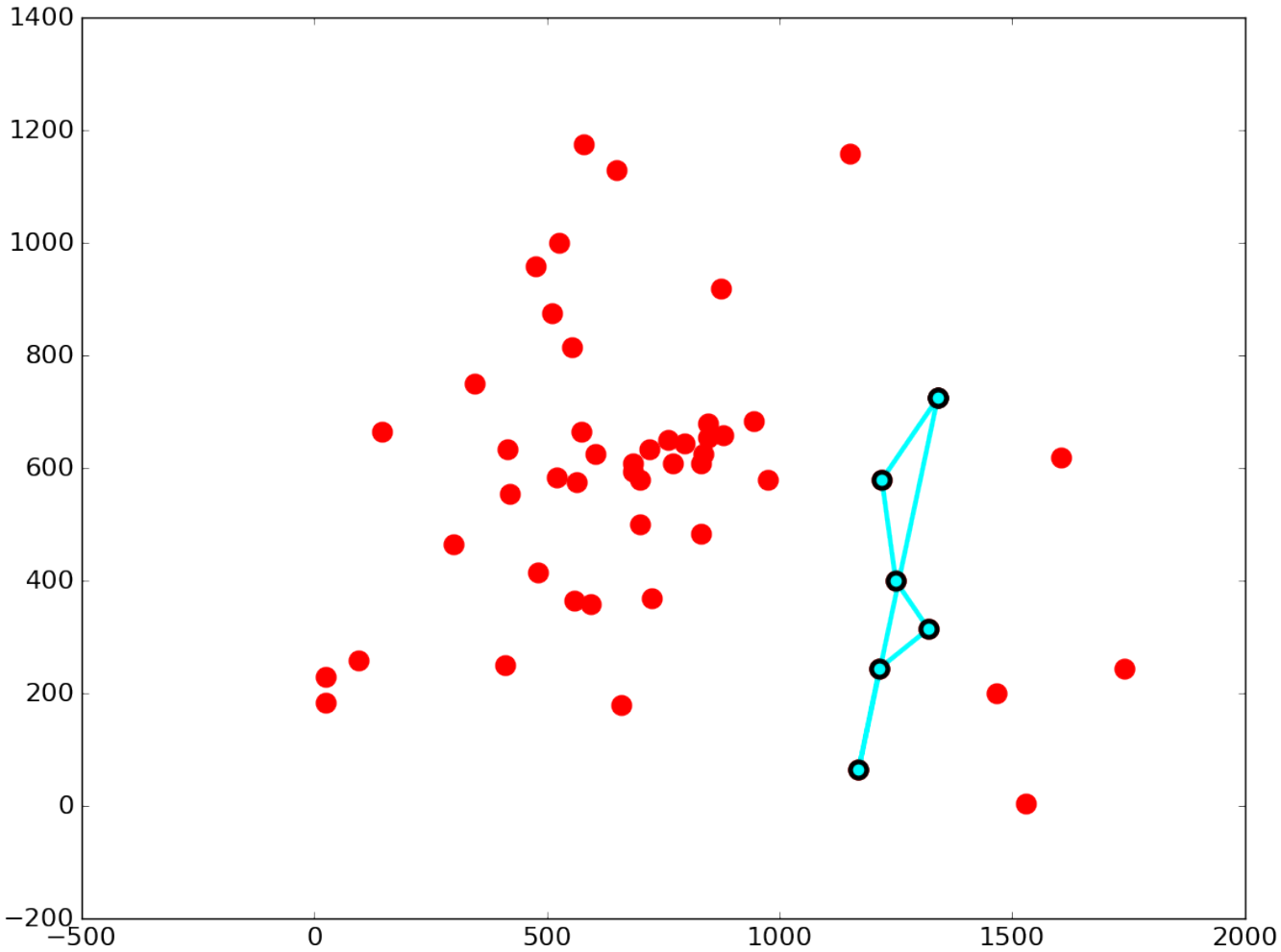
animation of the closest-neighbor algorithm



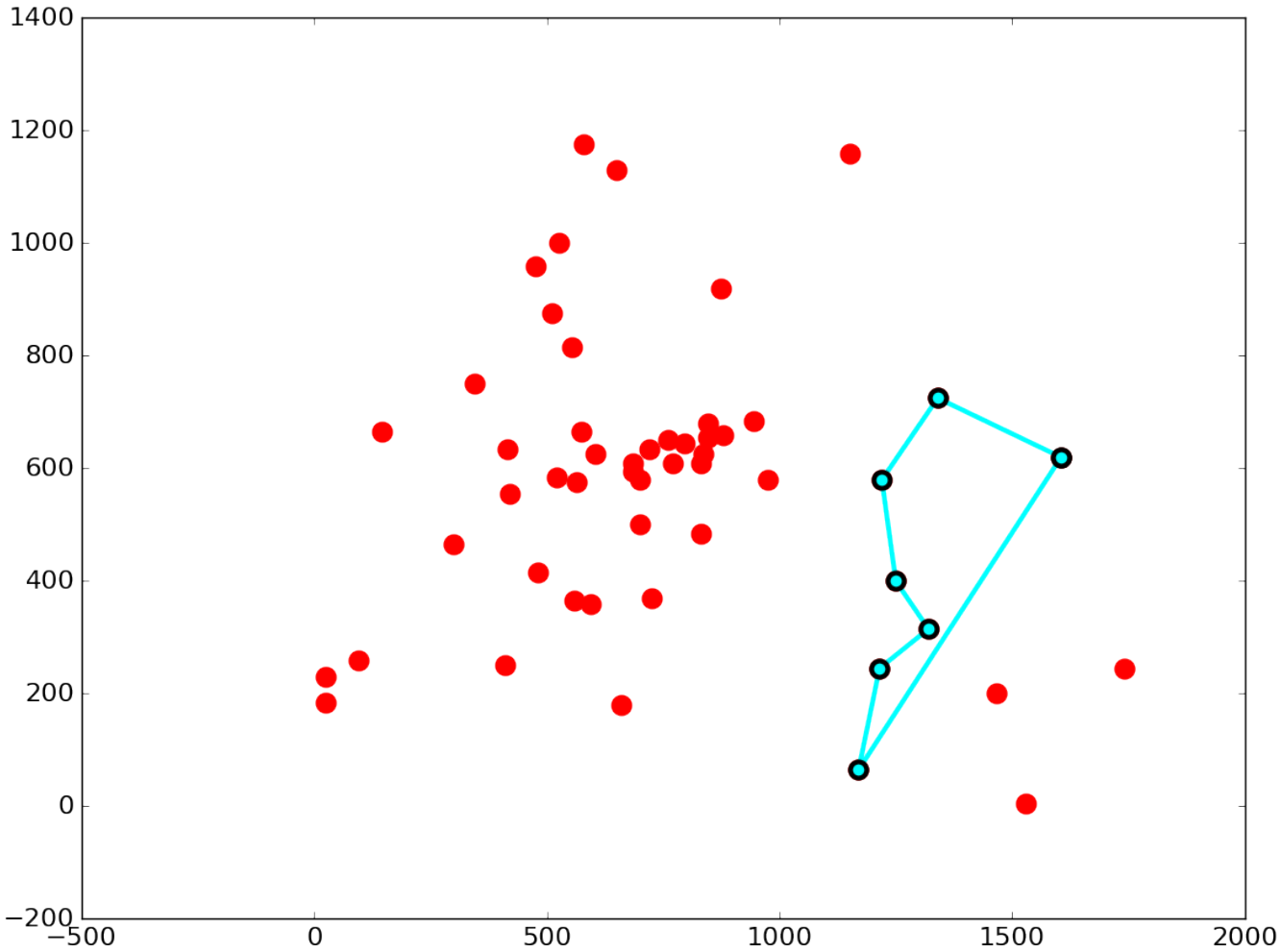
animation of the closest-neighbor algorithm



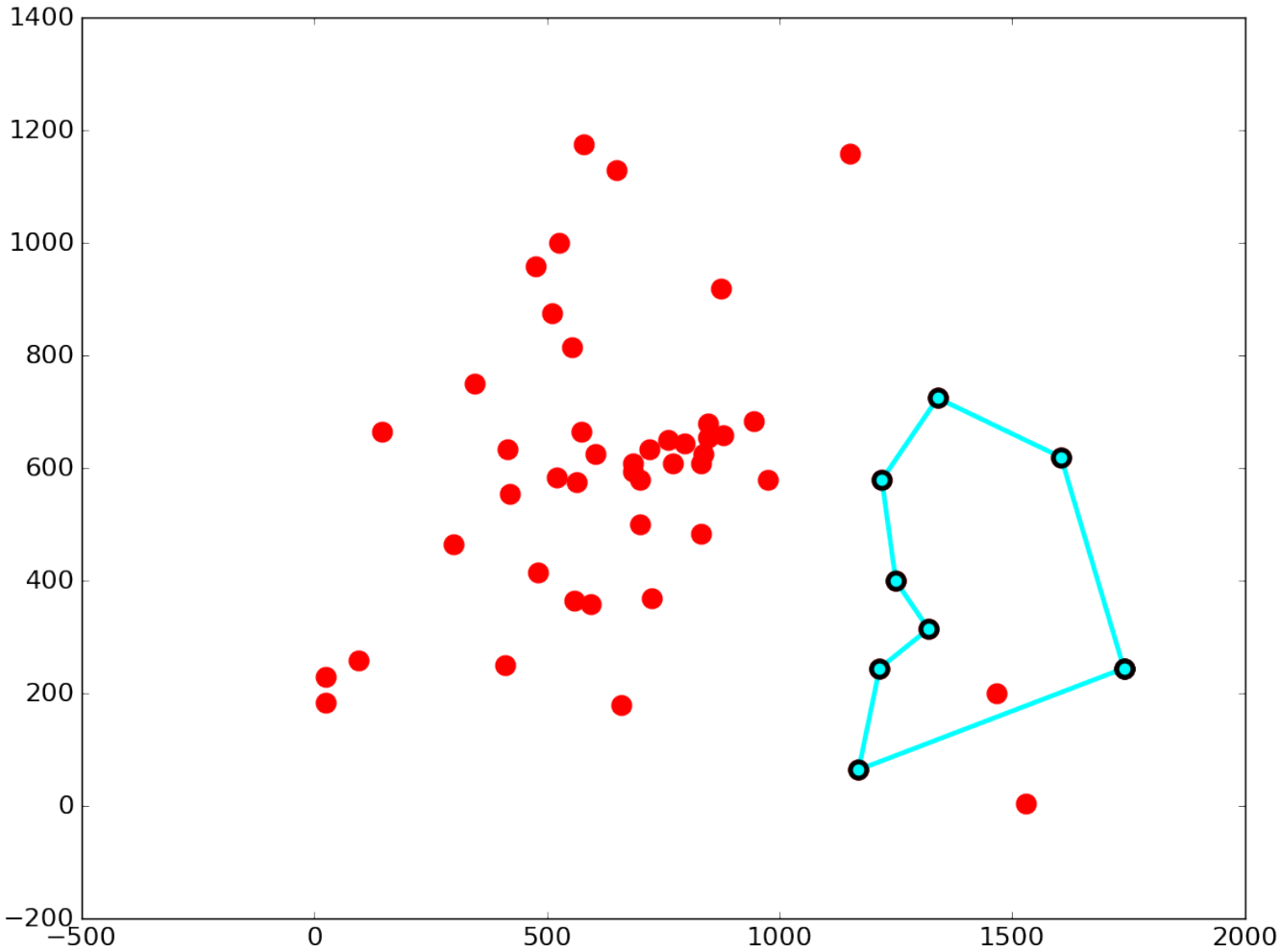
animation of the closest-neighbor algorithm



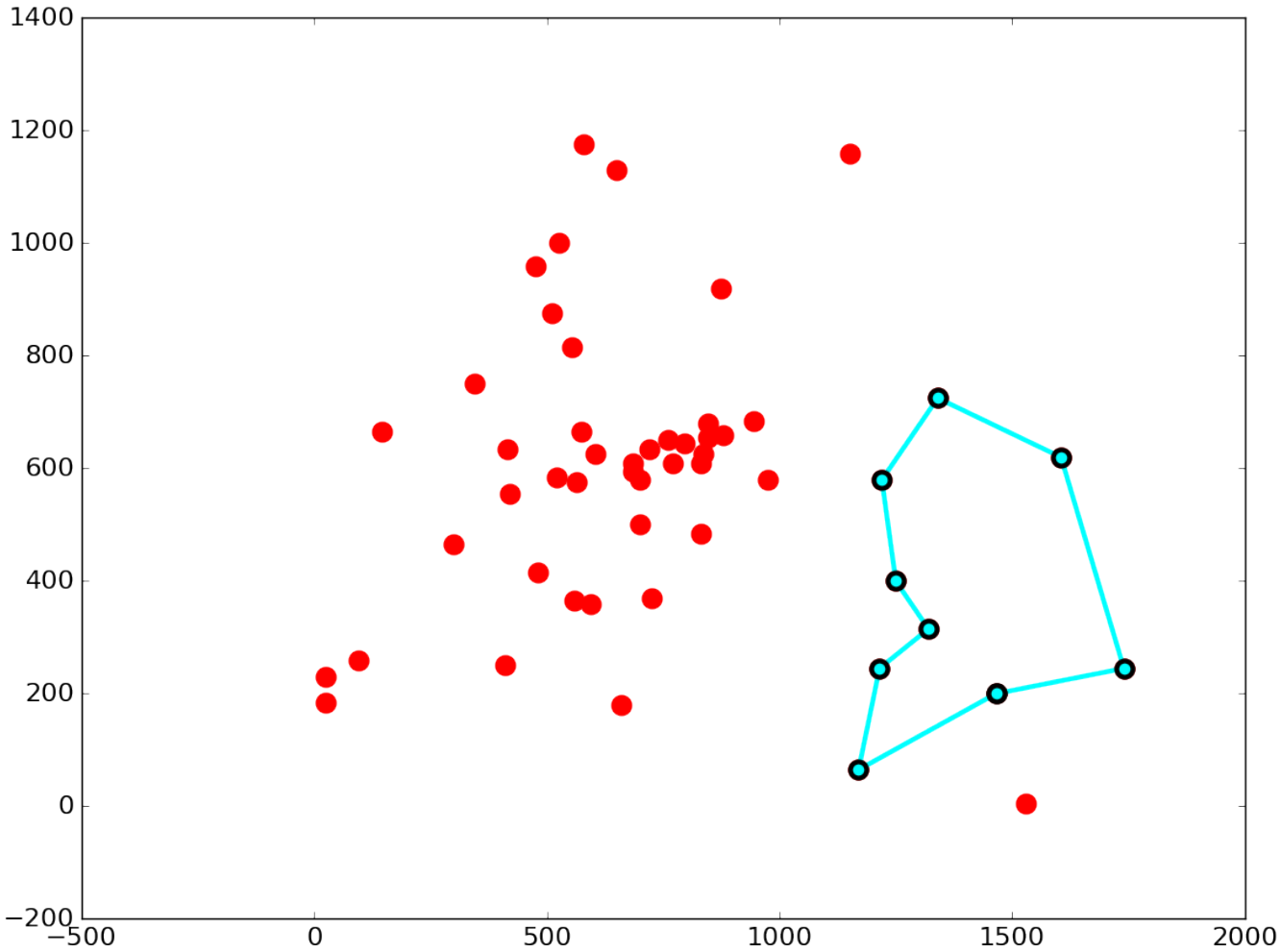
animation of the closest-neighbor algorithm



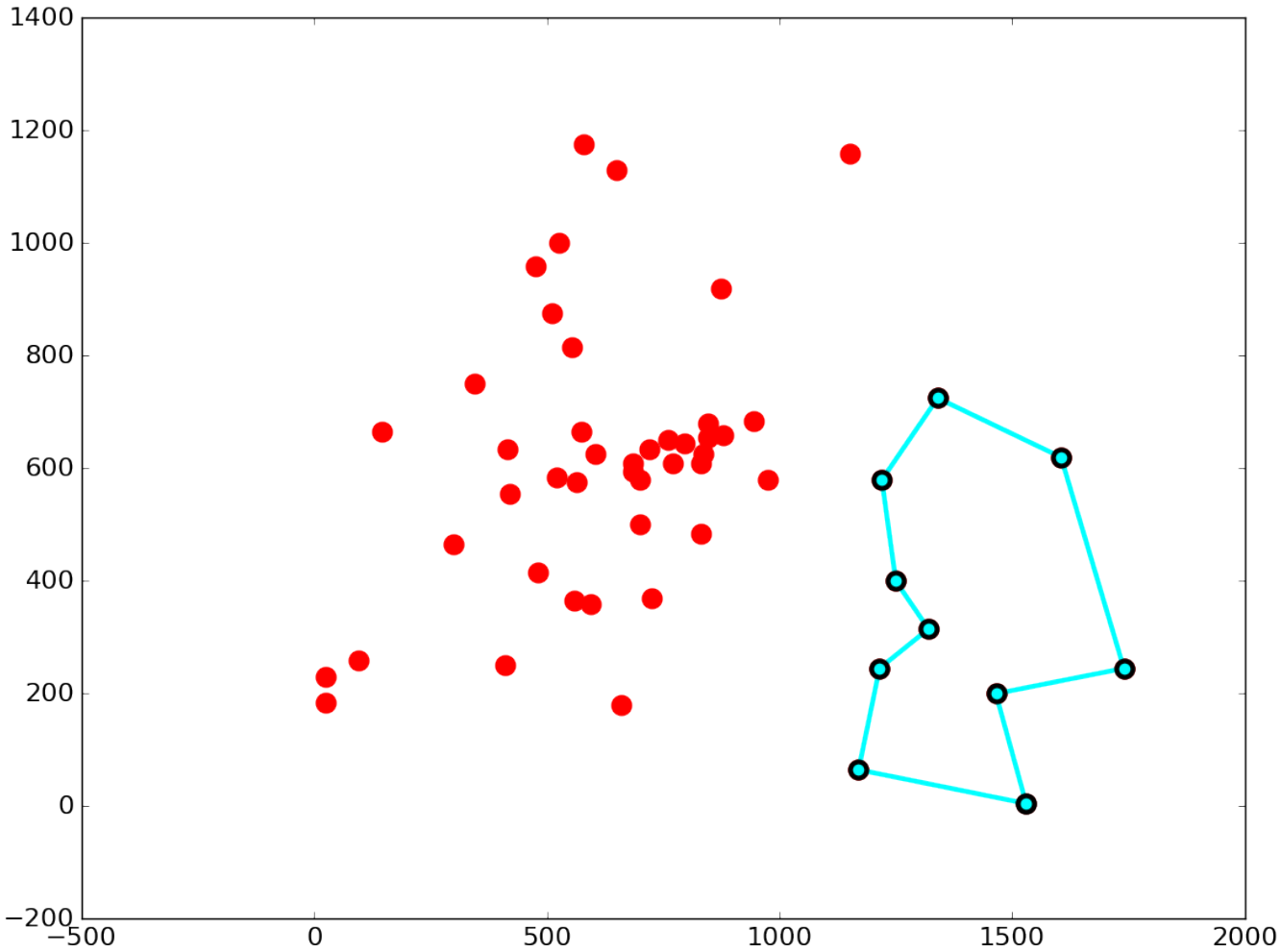
animation of the closest-neighbor algorithm



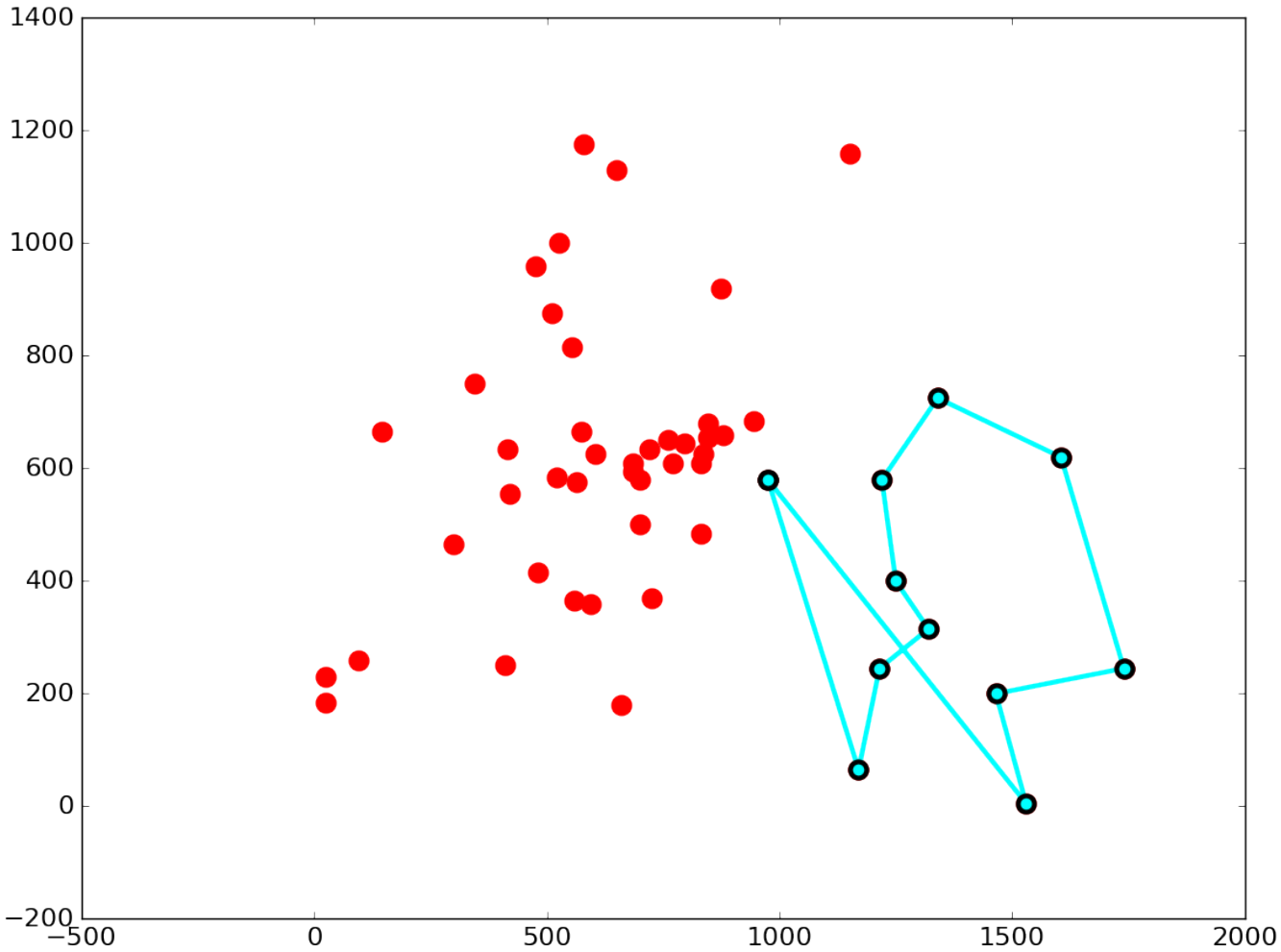
animation of the closest-neighbor algorithm



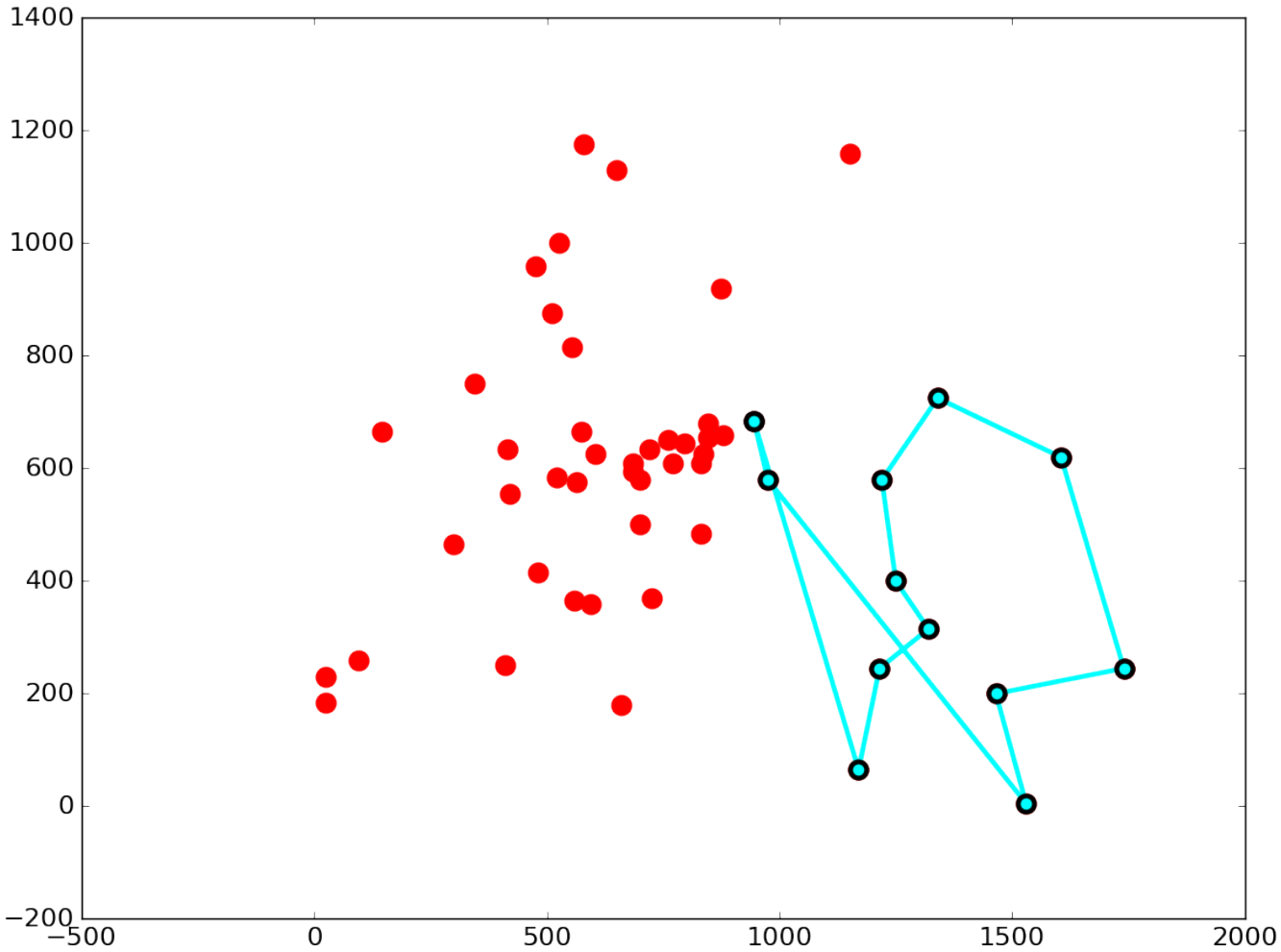
animation of the closest-neighbor algorithm



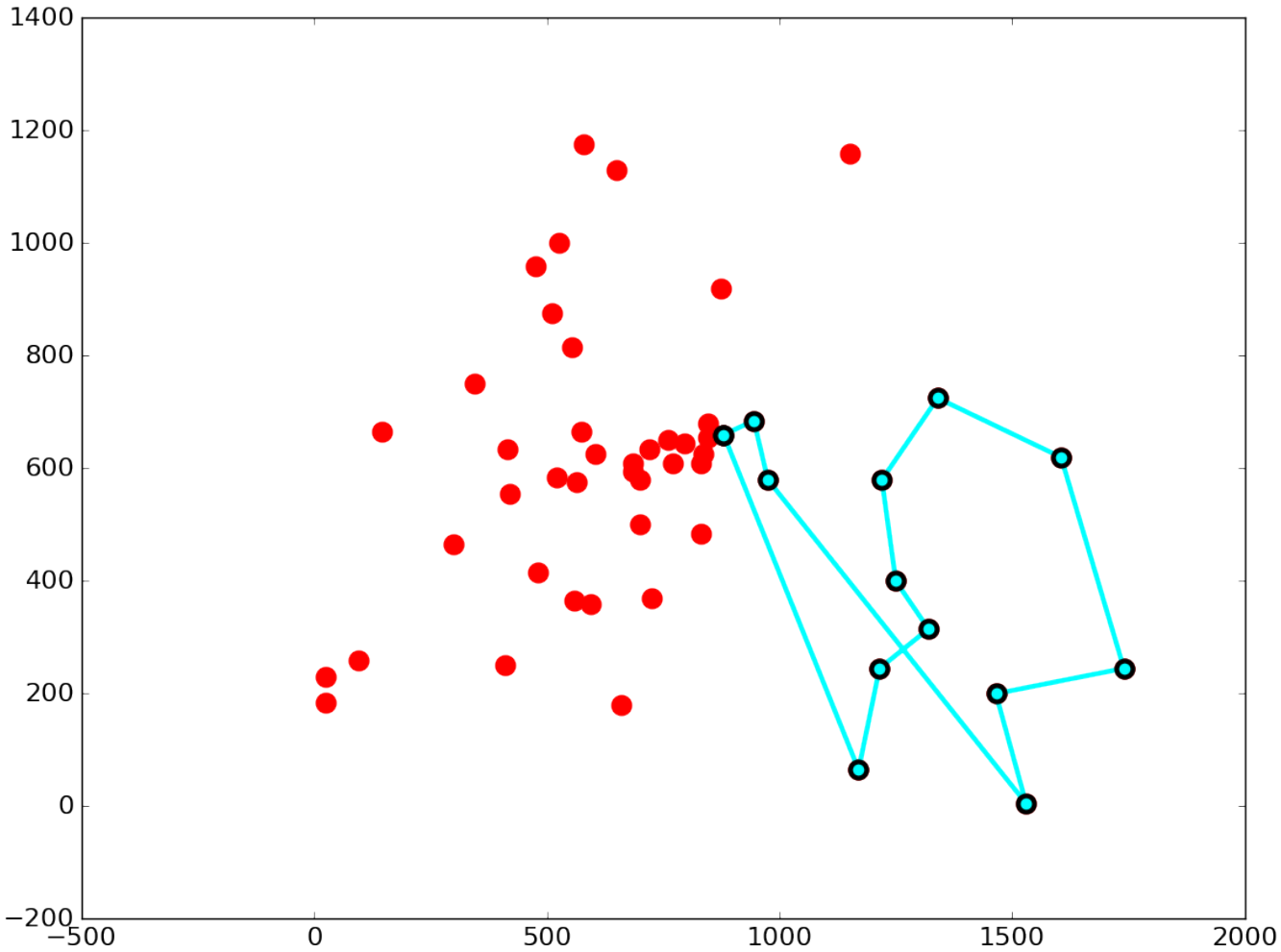
animation of the closest-neighbor algorithm



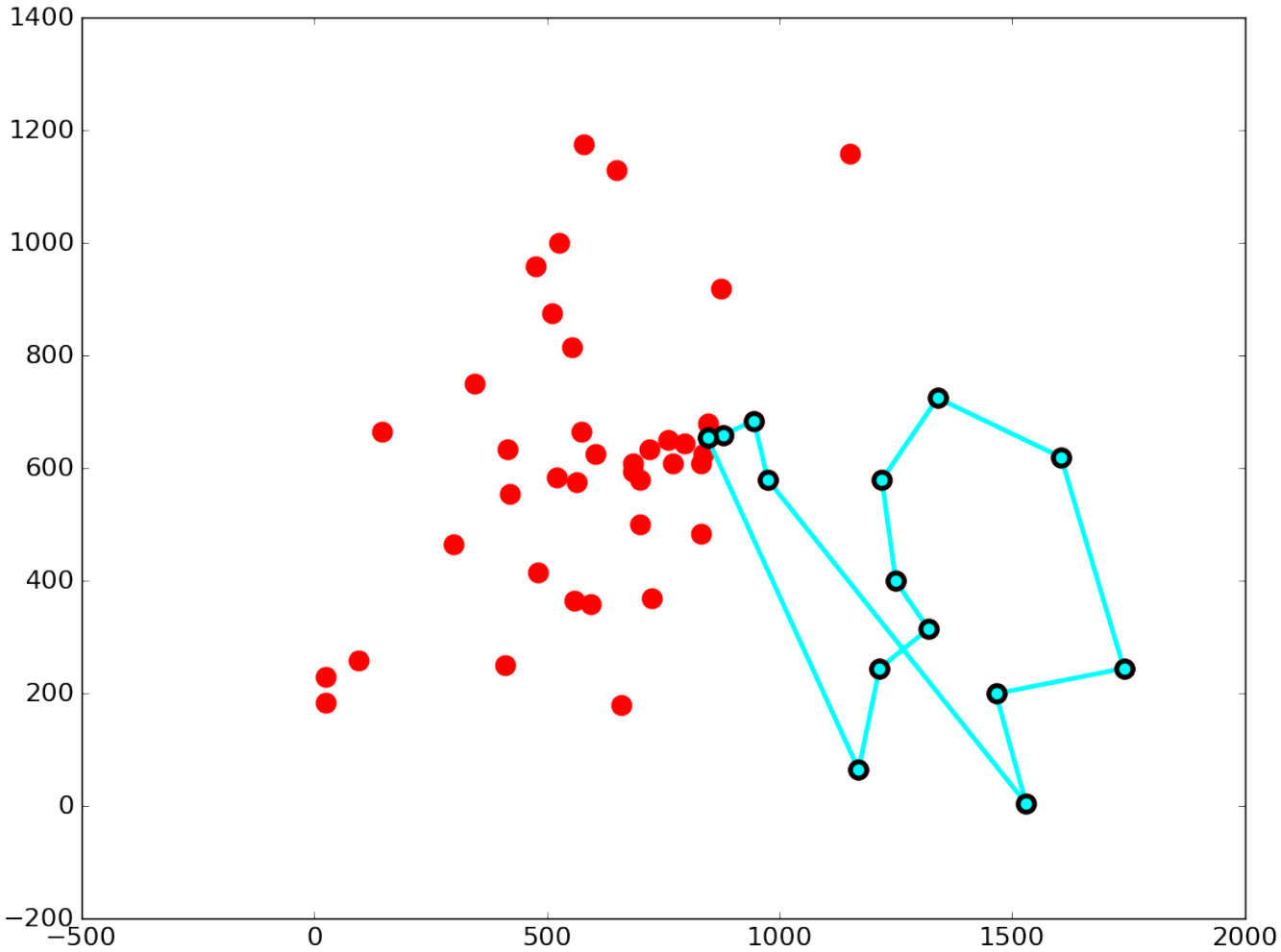
animation of the closest-neighbor algorithm



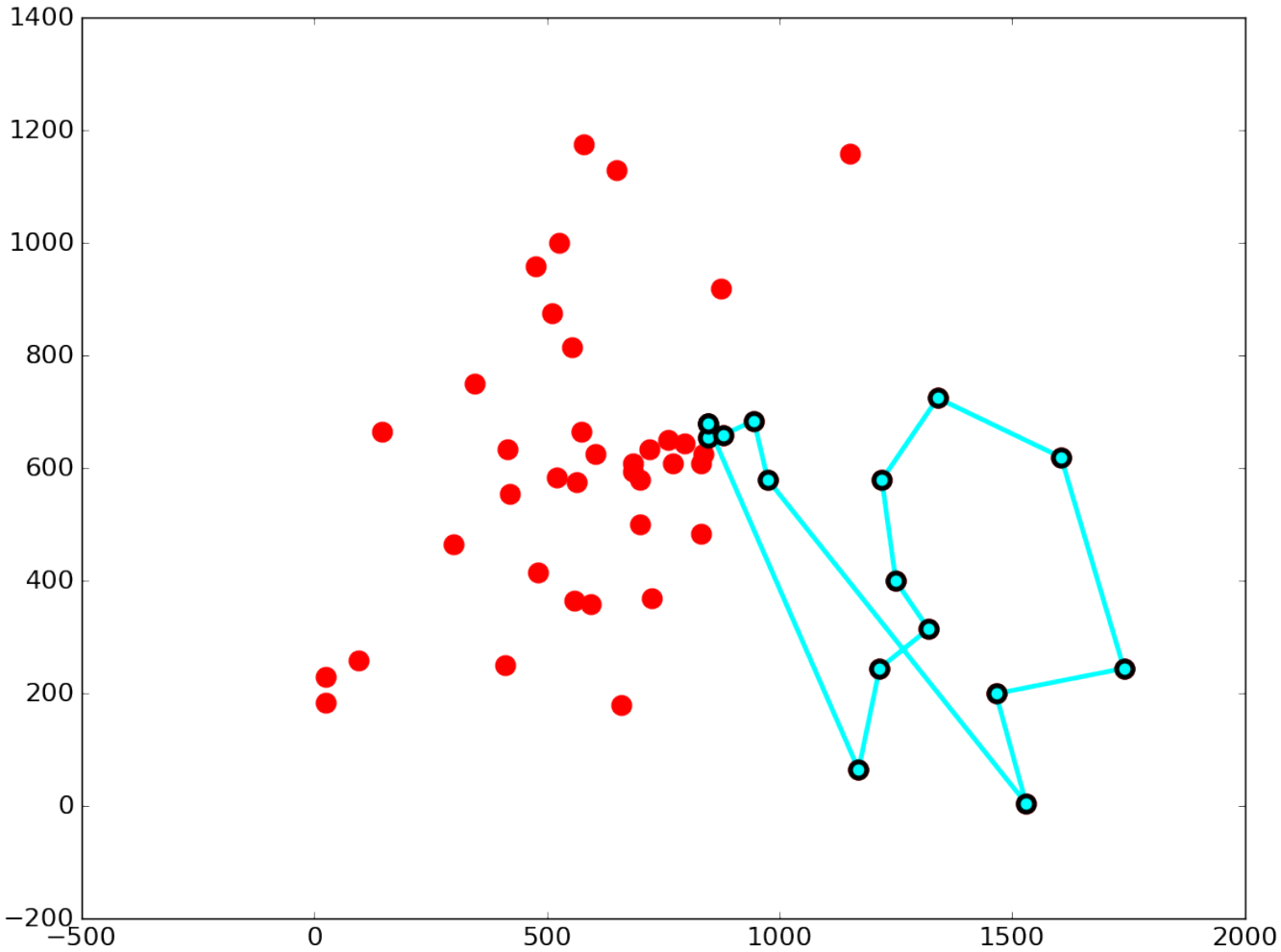
animation of the closest-neighbor algorithm



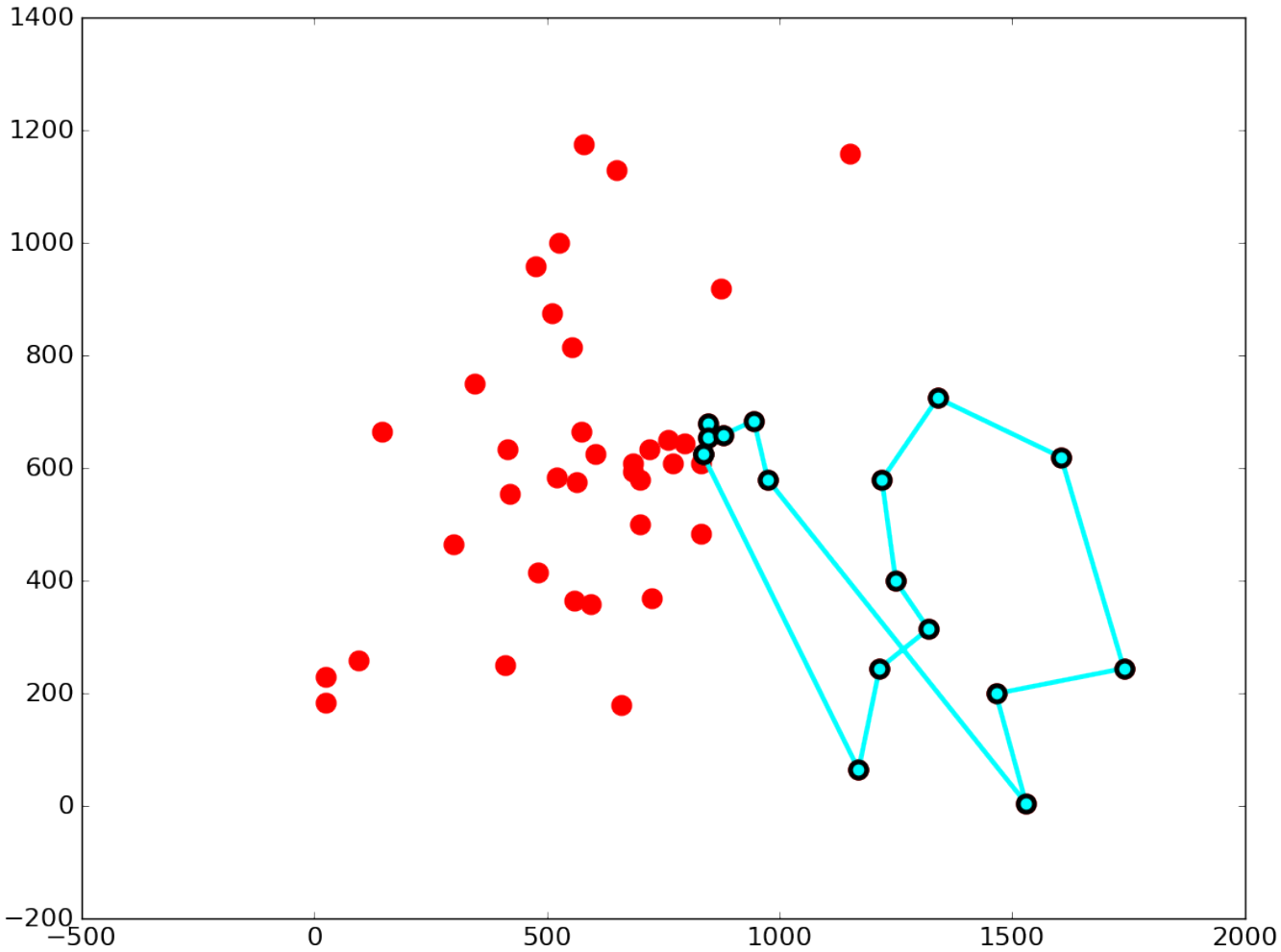
animation of the closest-neighbor algorithm



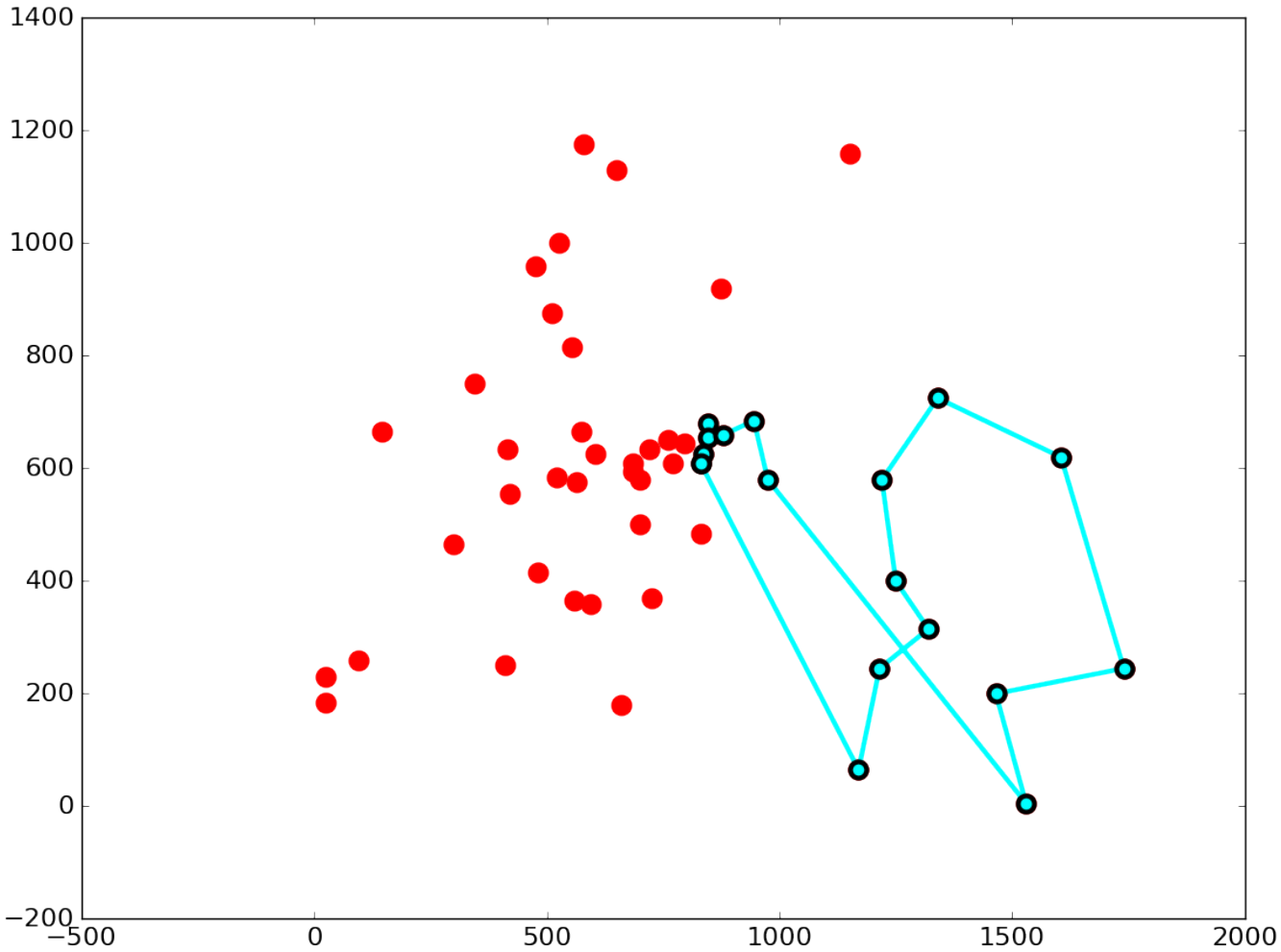
animation of the closest-neighbor algorithm



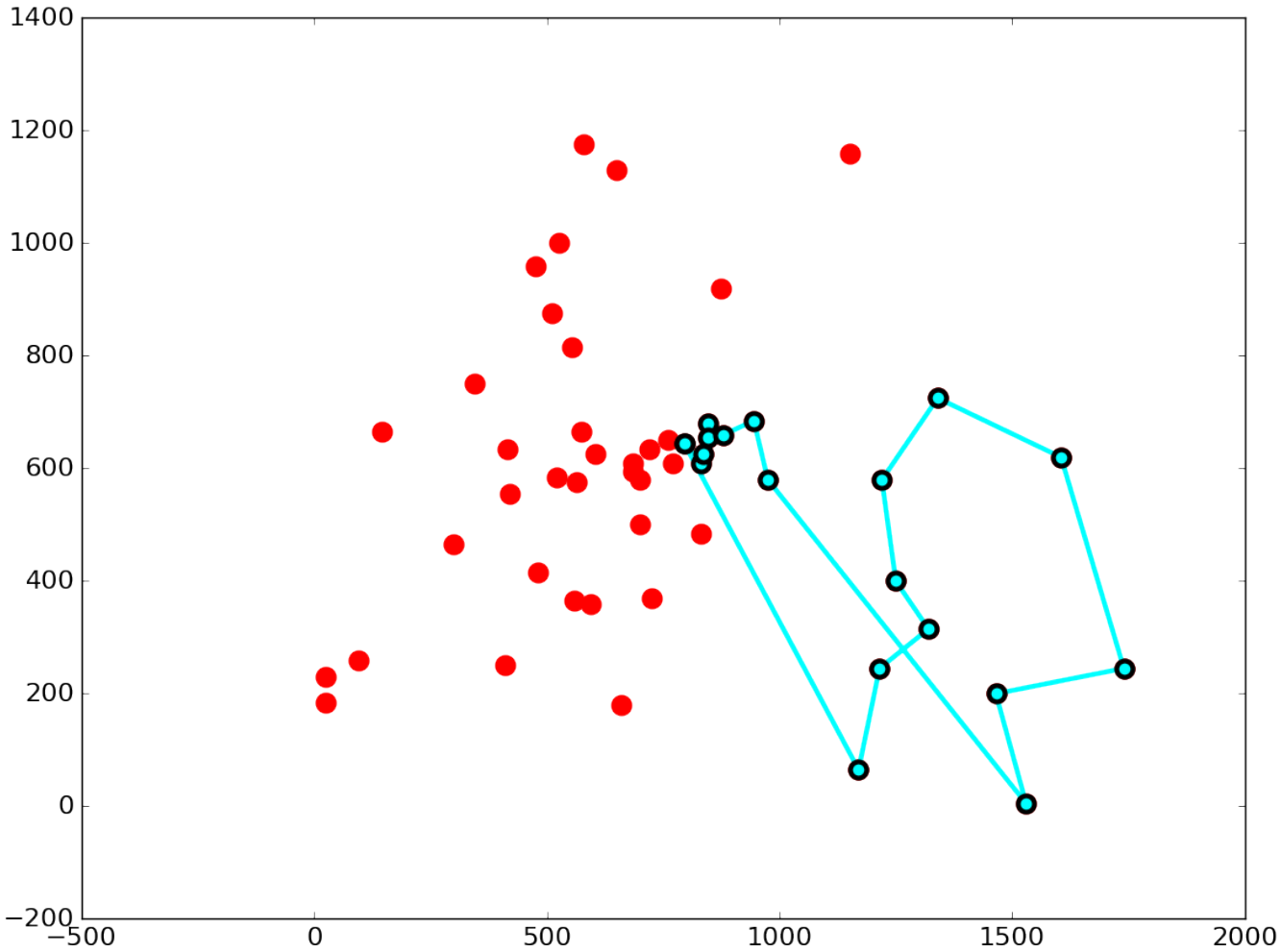
animation of the closest-neighbor algorithm



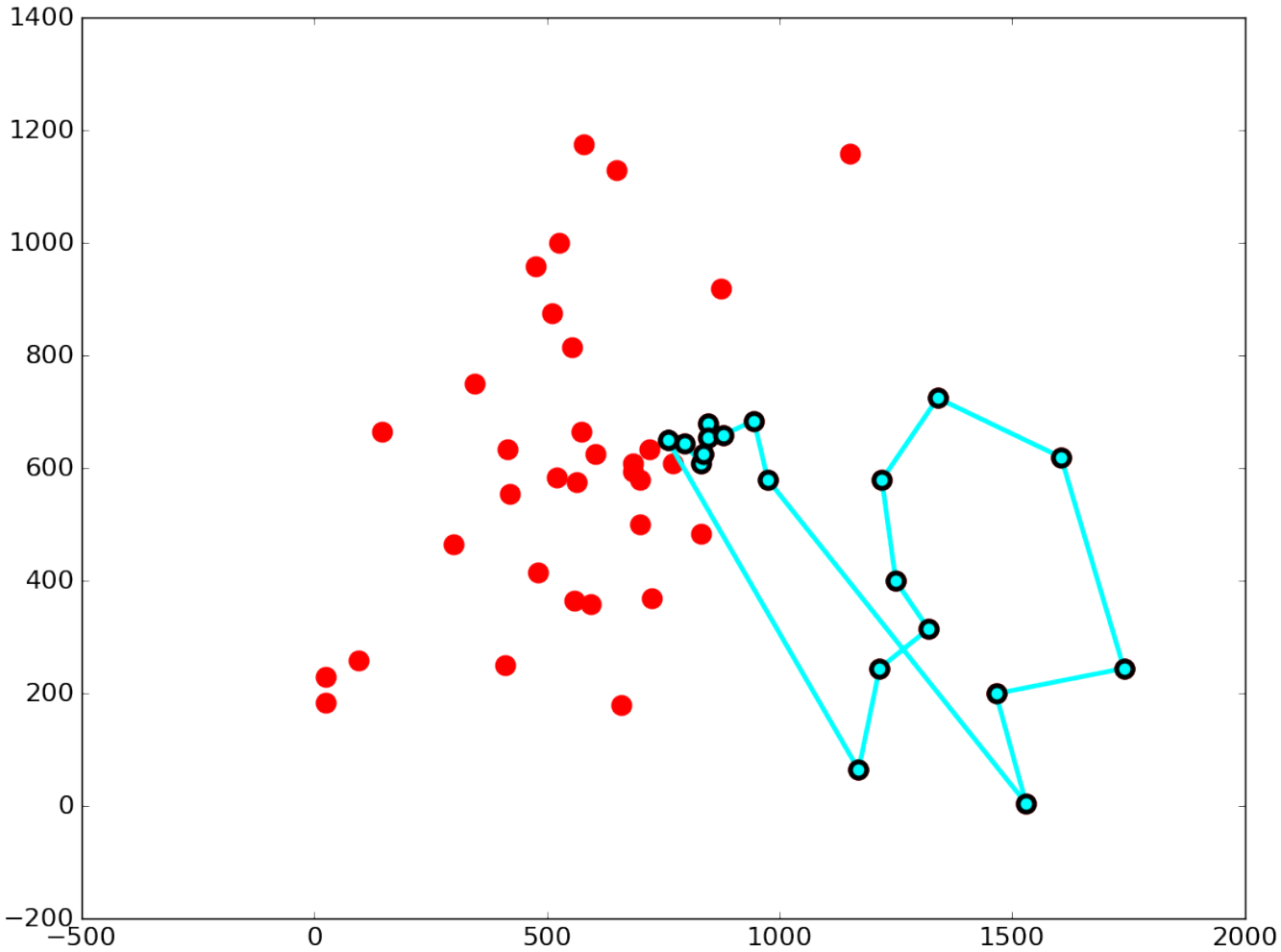
animation of the closest-neighbor algorithm



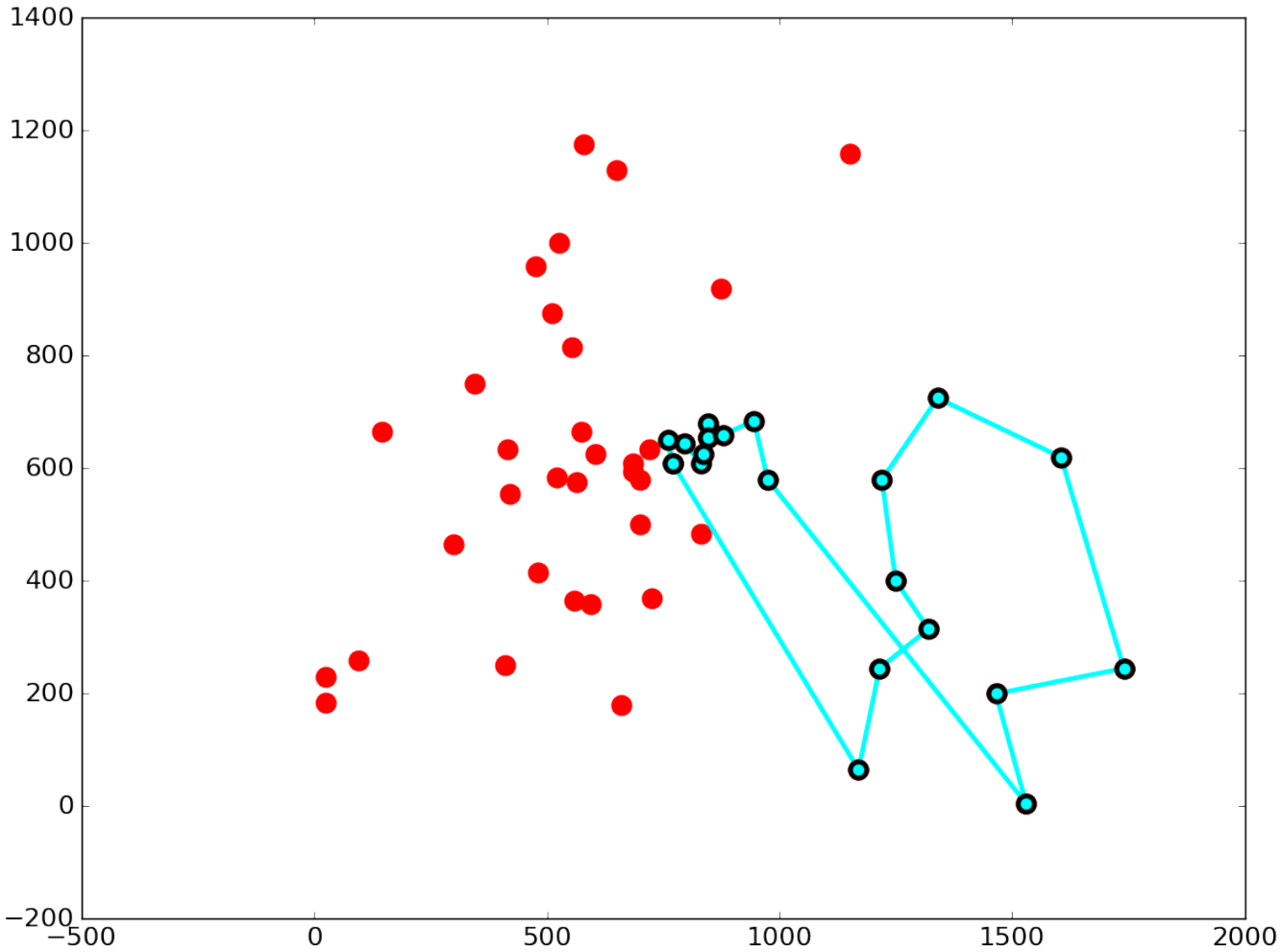
animation of the closest-neighbor algorithm



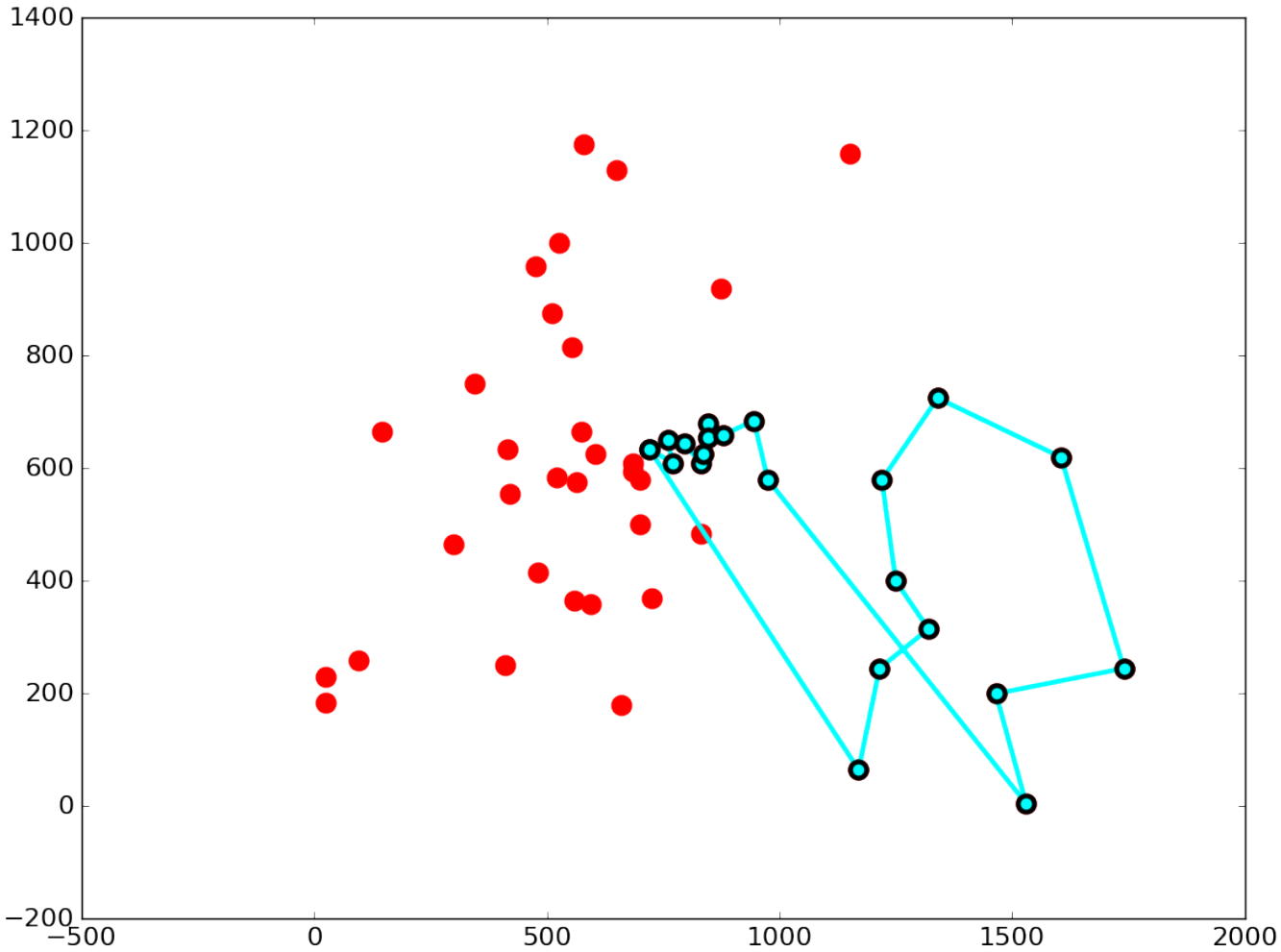
animation of the closest-neighbor algorithm



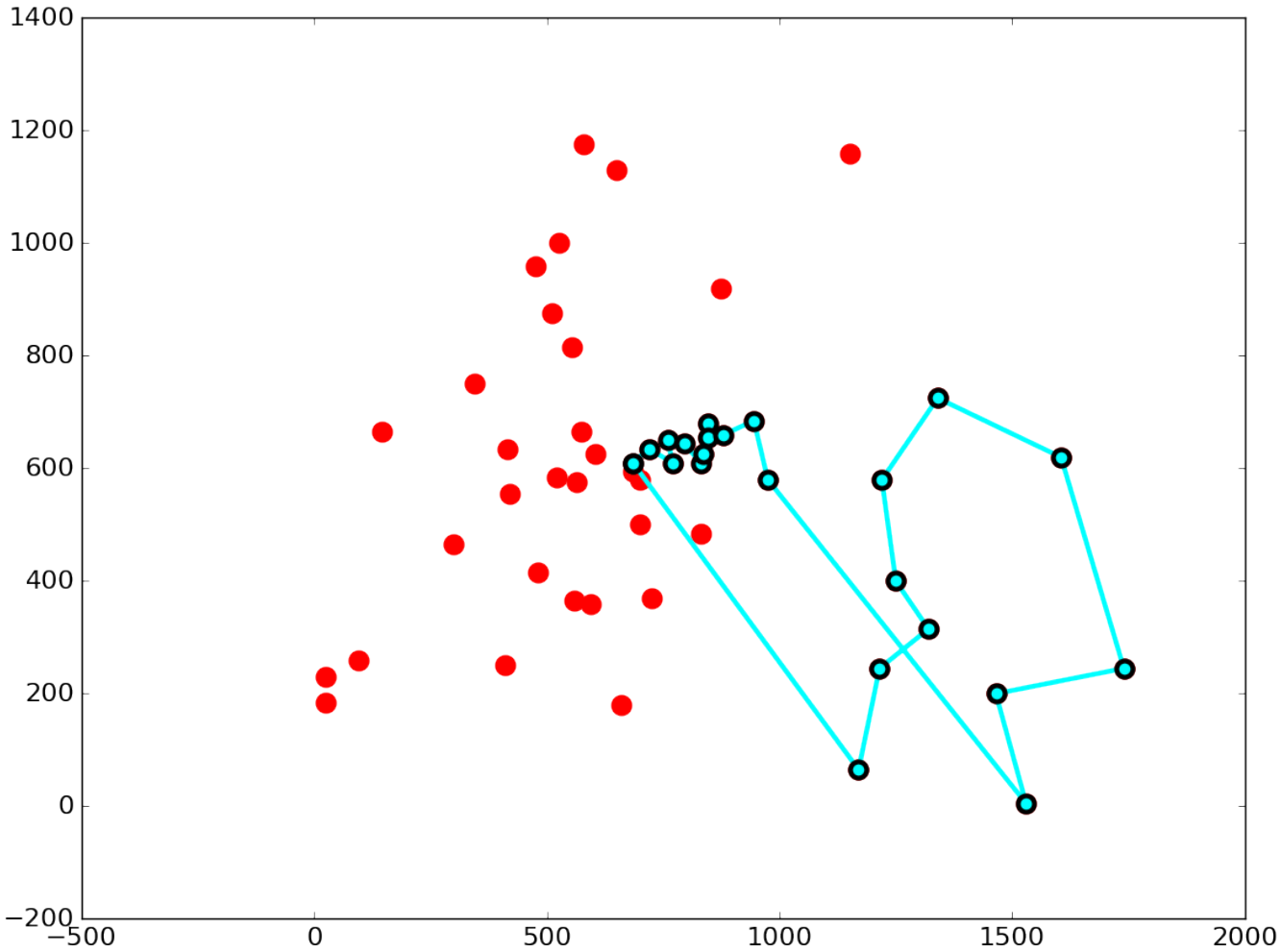
animation of the closest-neighbor algorithm



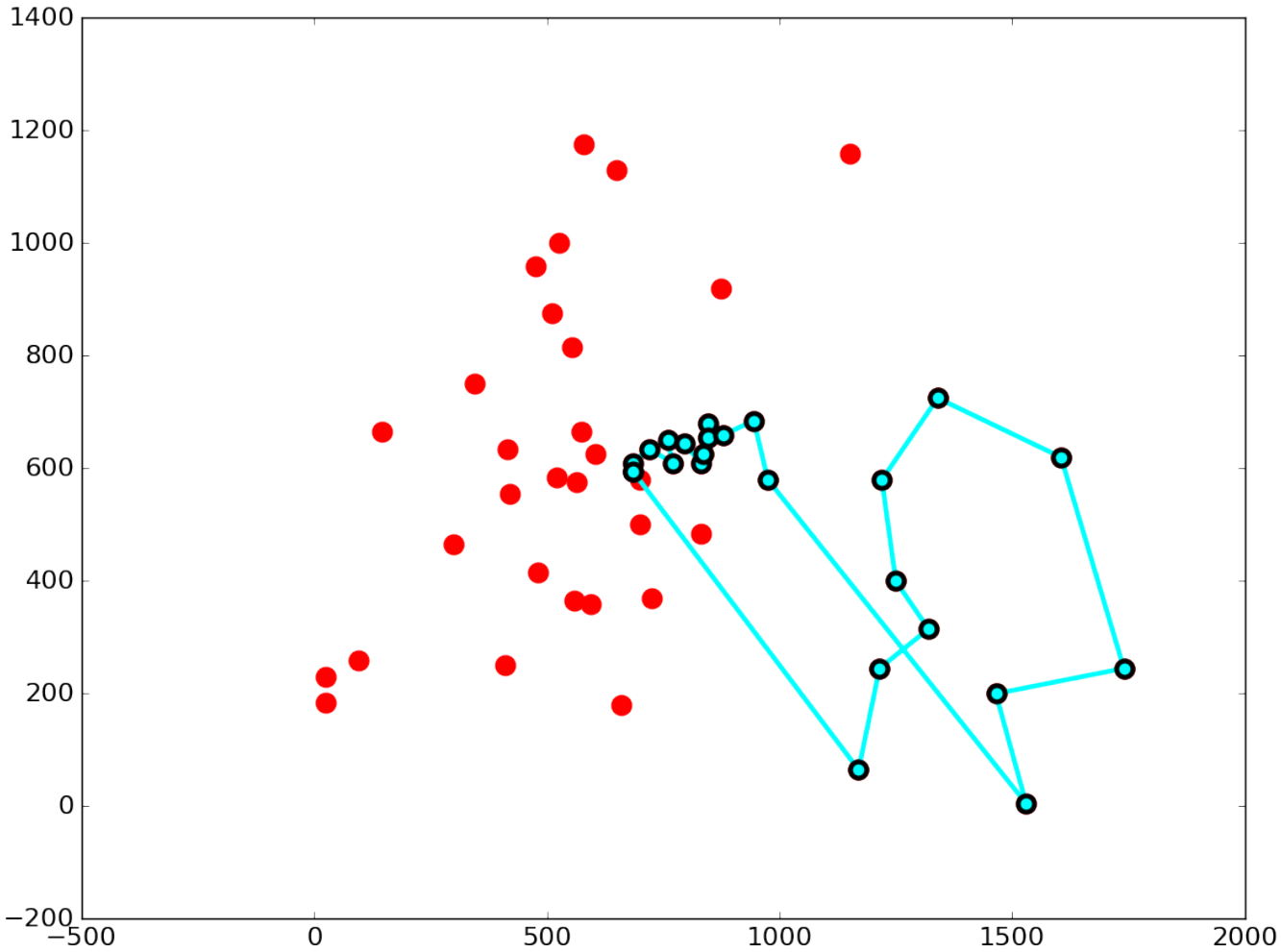
animation of the closest-neighbor algorithm



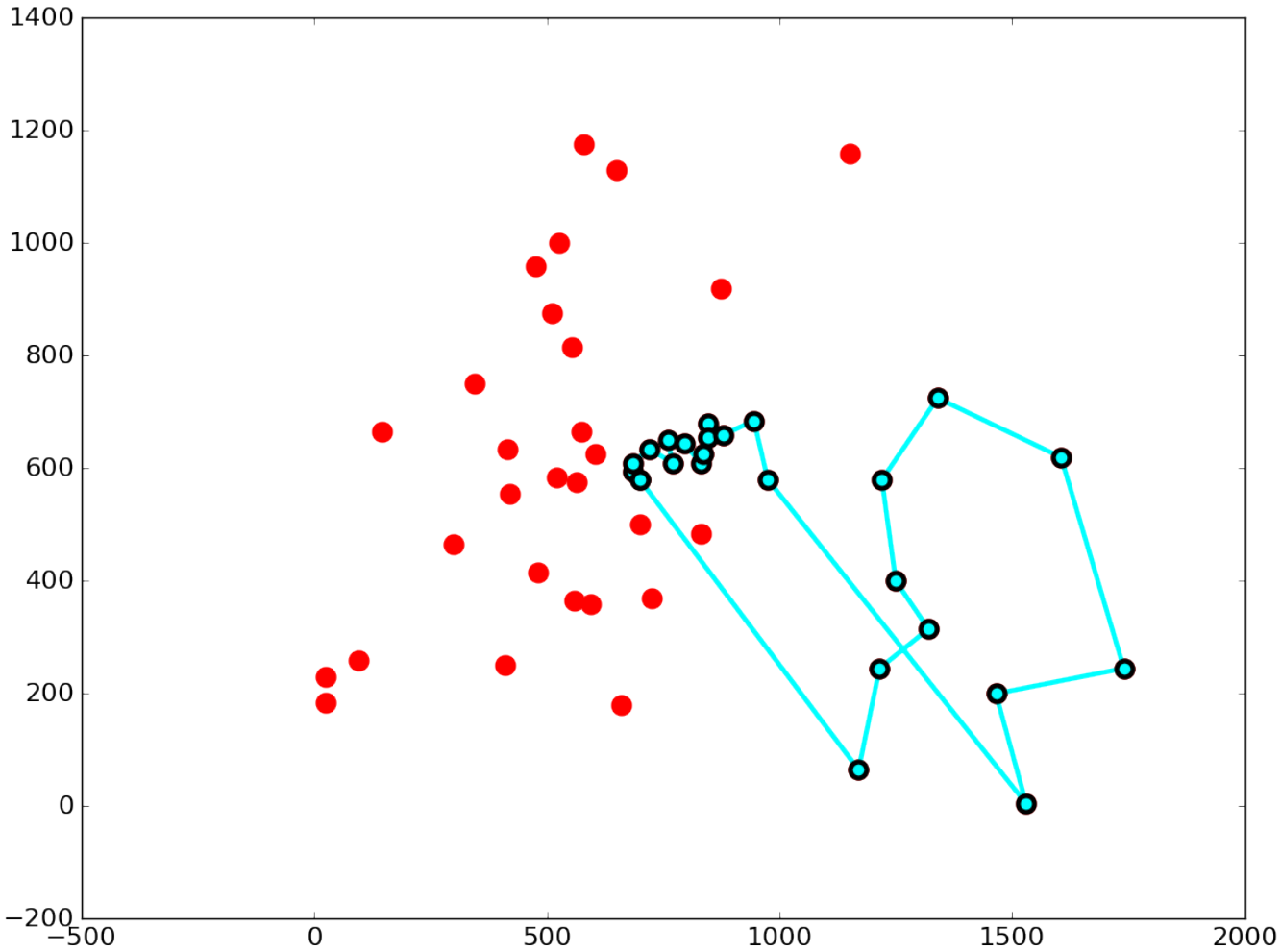
animation of the closest-neighbor algorithm



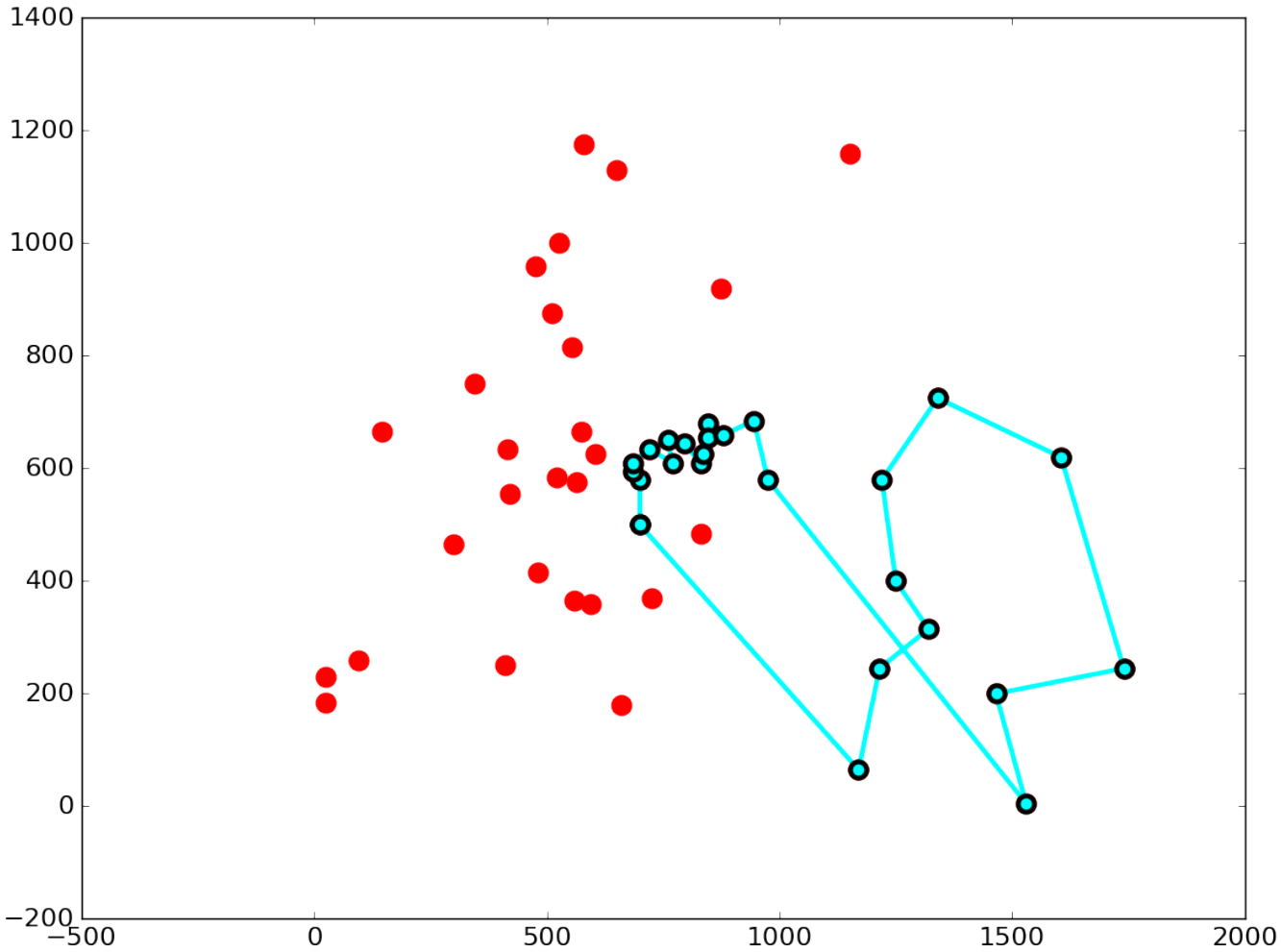
animation of the closest-neighbor algorithm



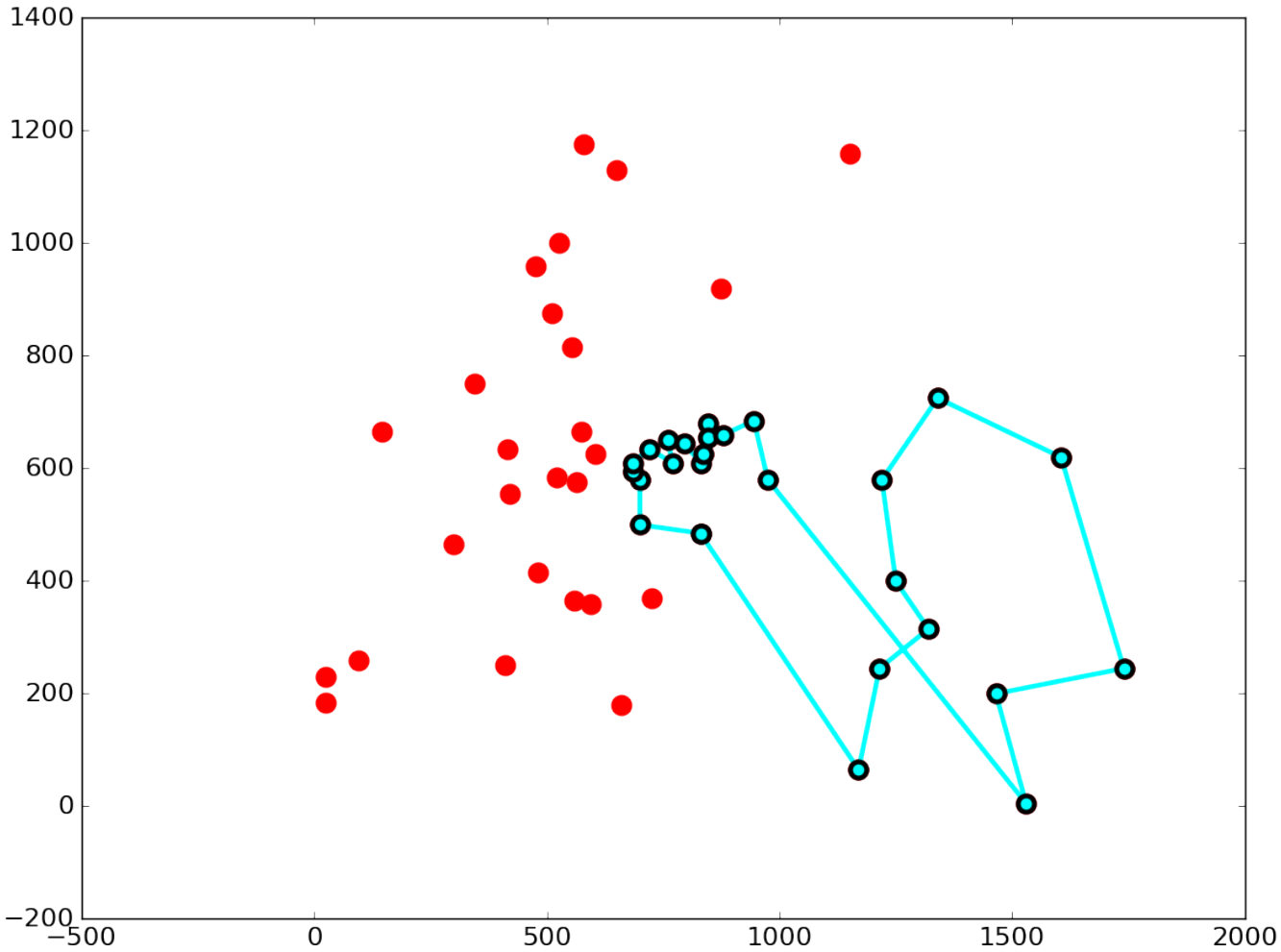
animation of the closest-neighbor algorithm



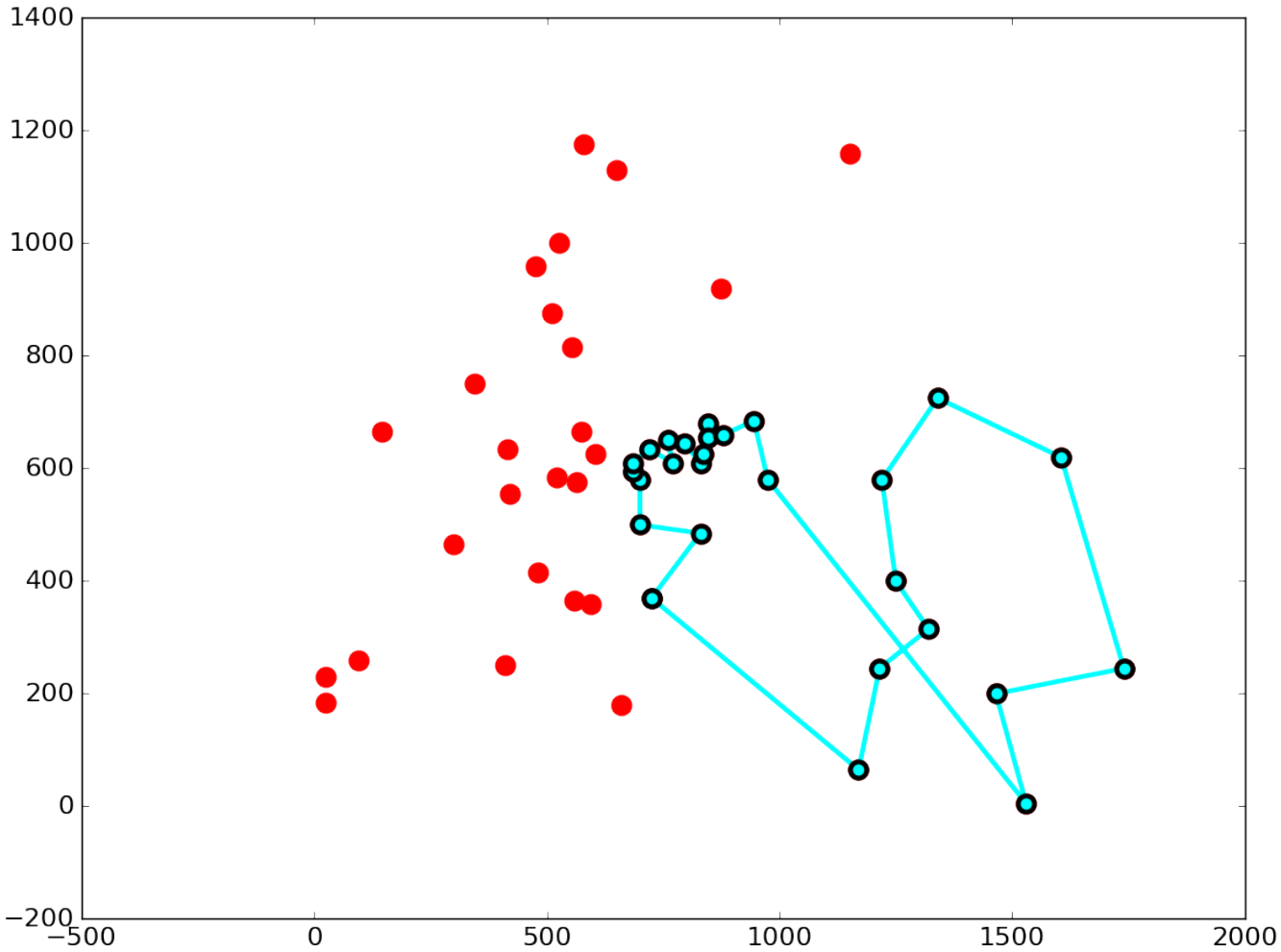
animation of the closest-neighbor algorithm



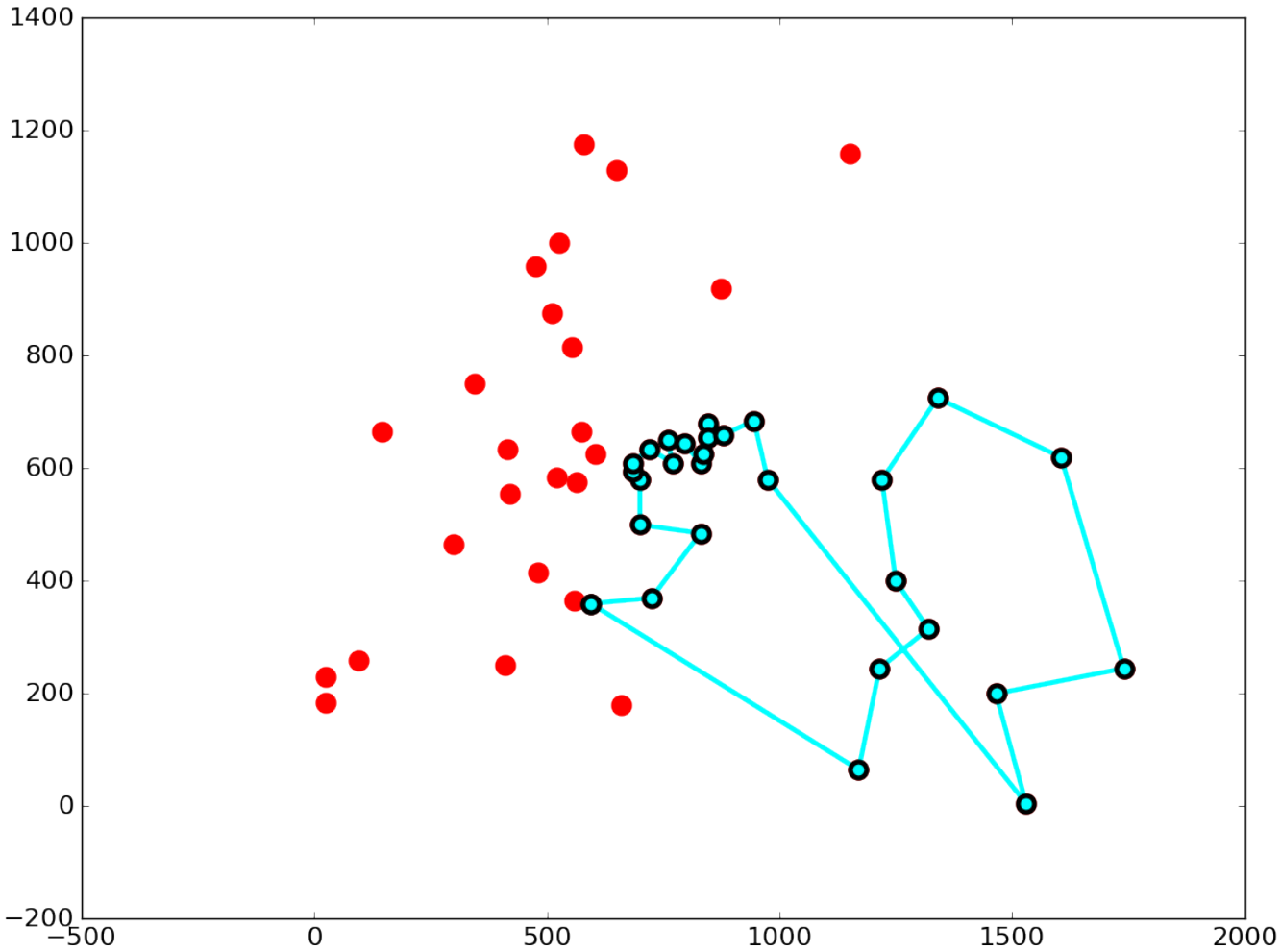
animation of the closest-neighbor algorithm



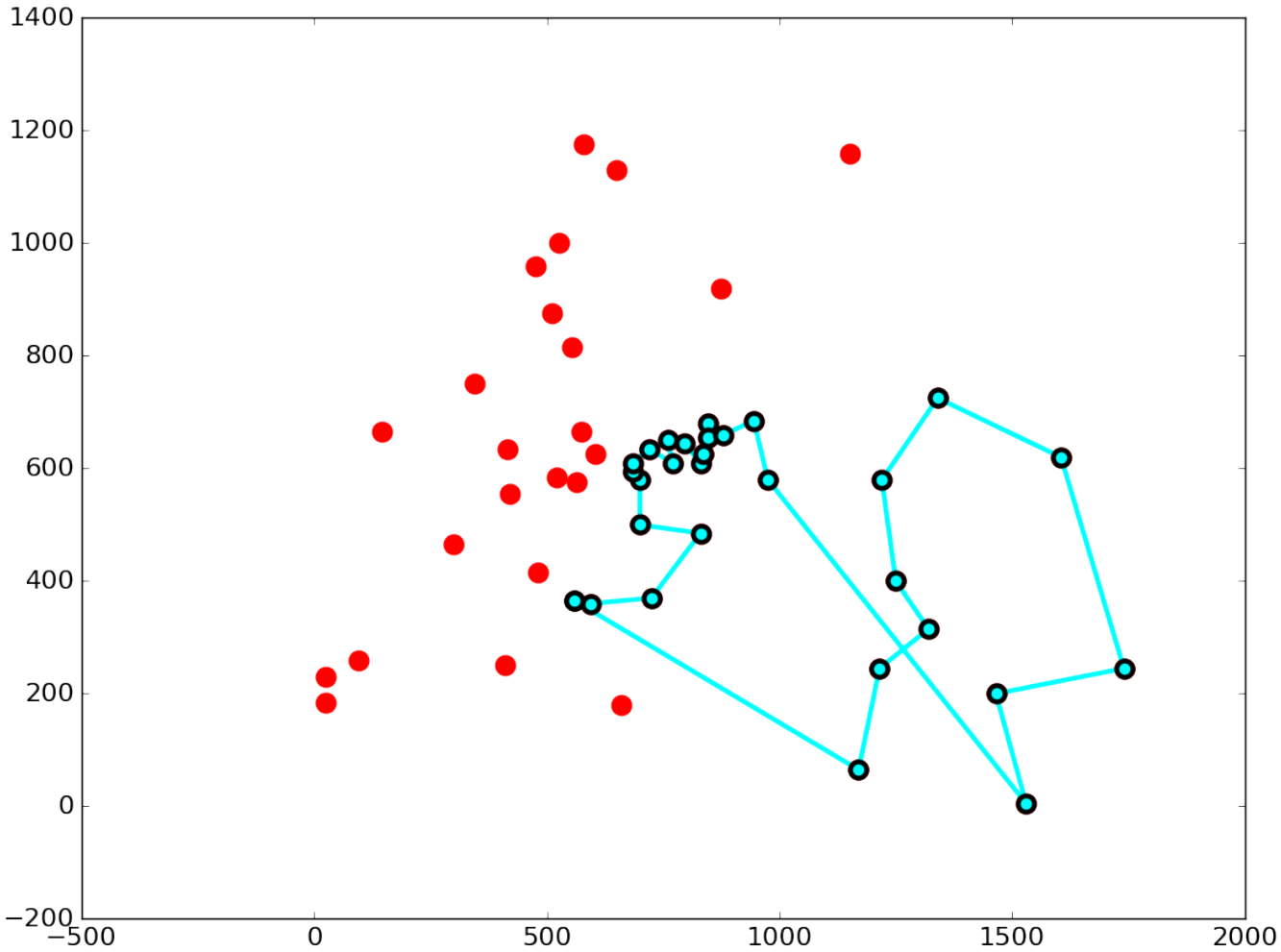
animation of the closest-neighbor algorithm



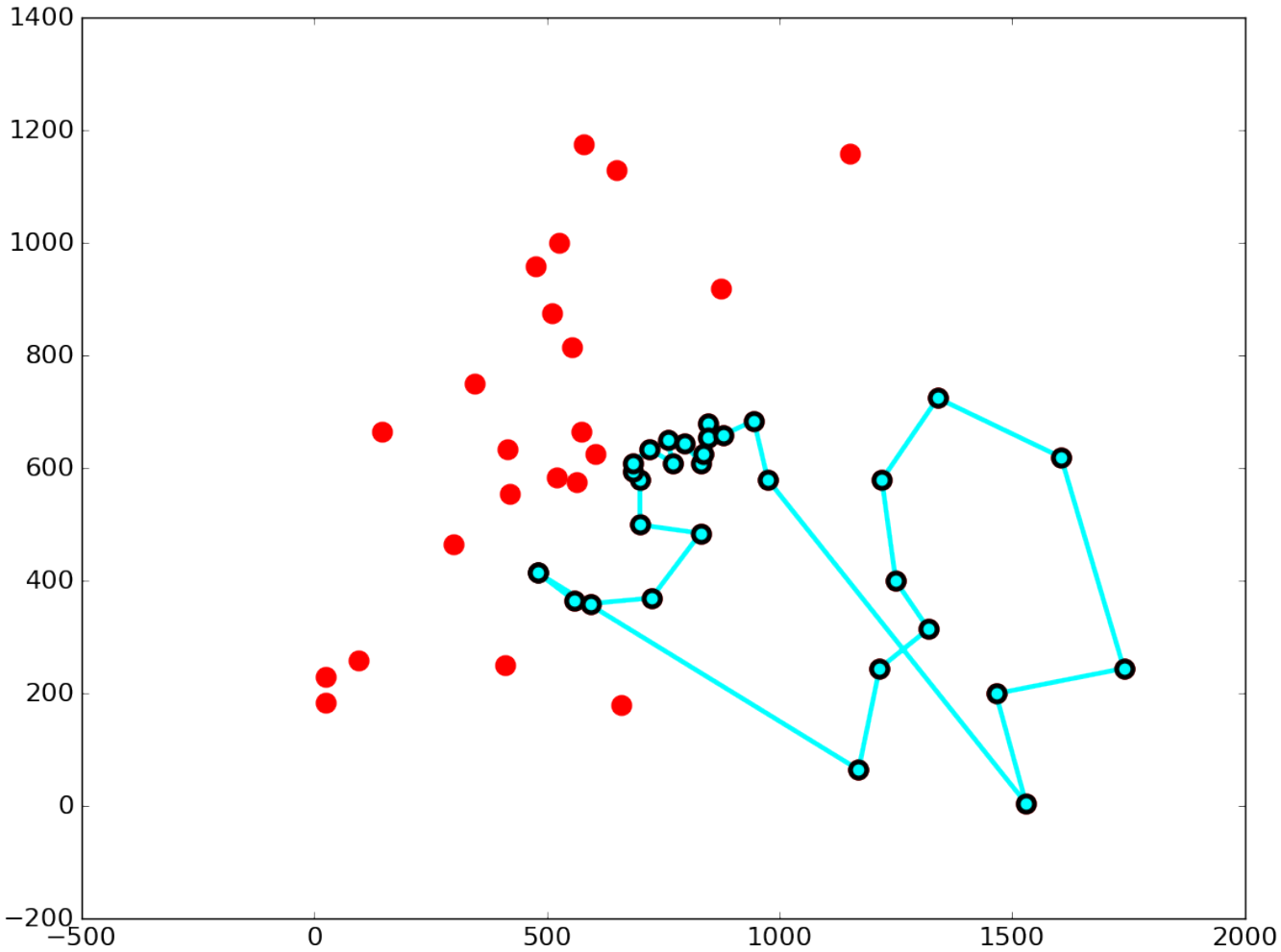
animation of the closest-neighbor algorithm



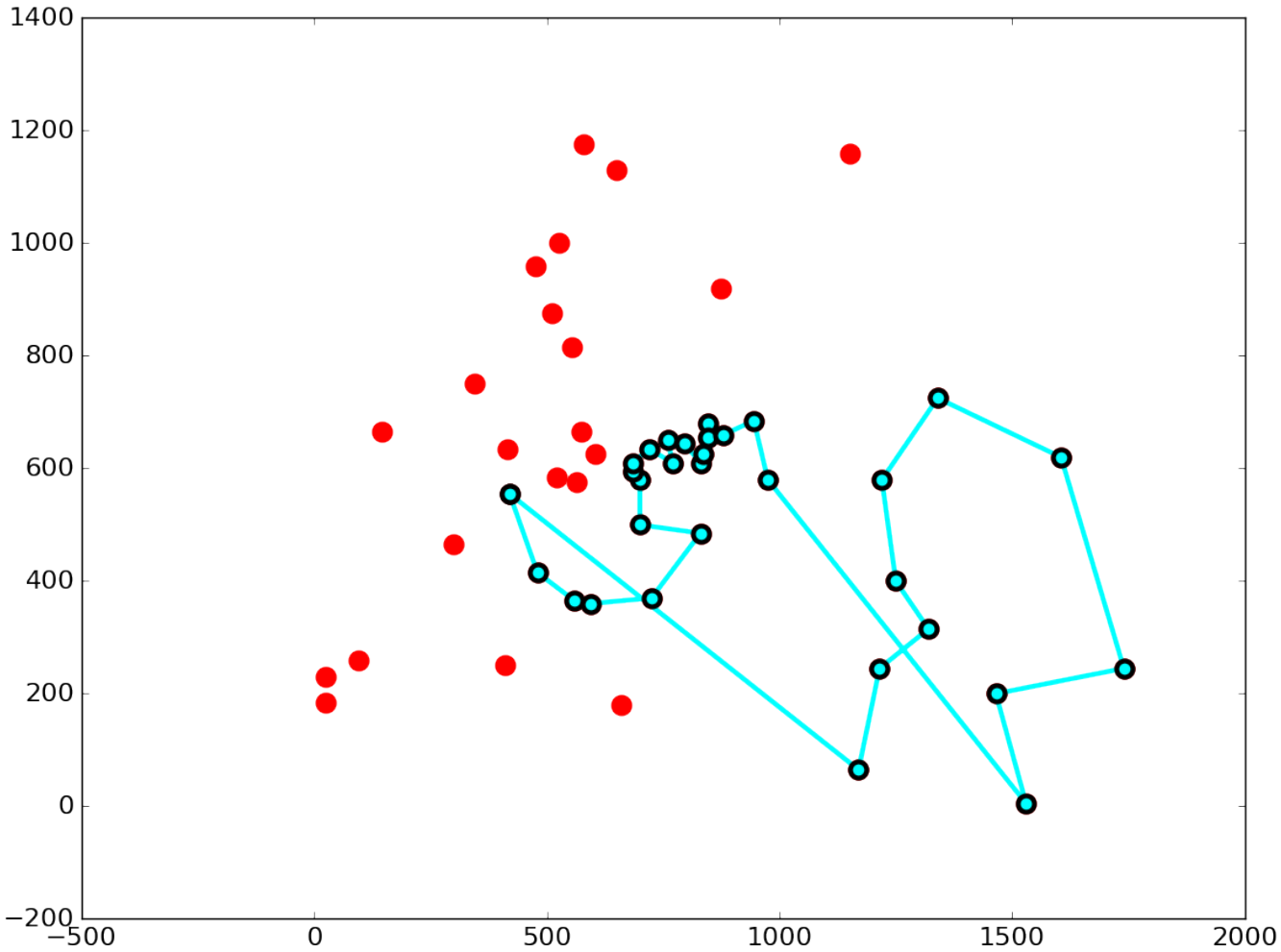
animation of the closest-neighbor algorithm



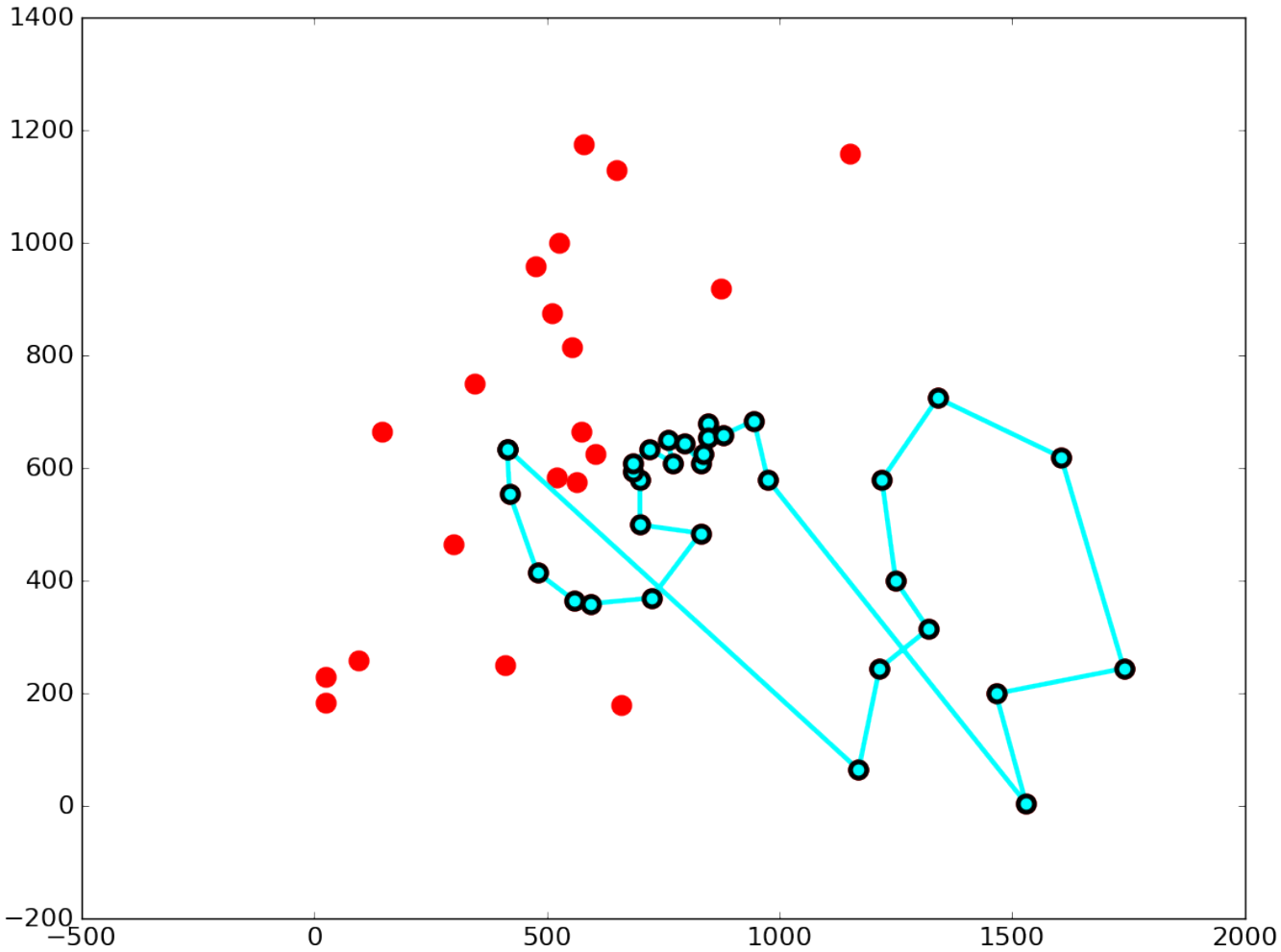
animation of the closest-neighbor algorithm



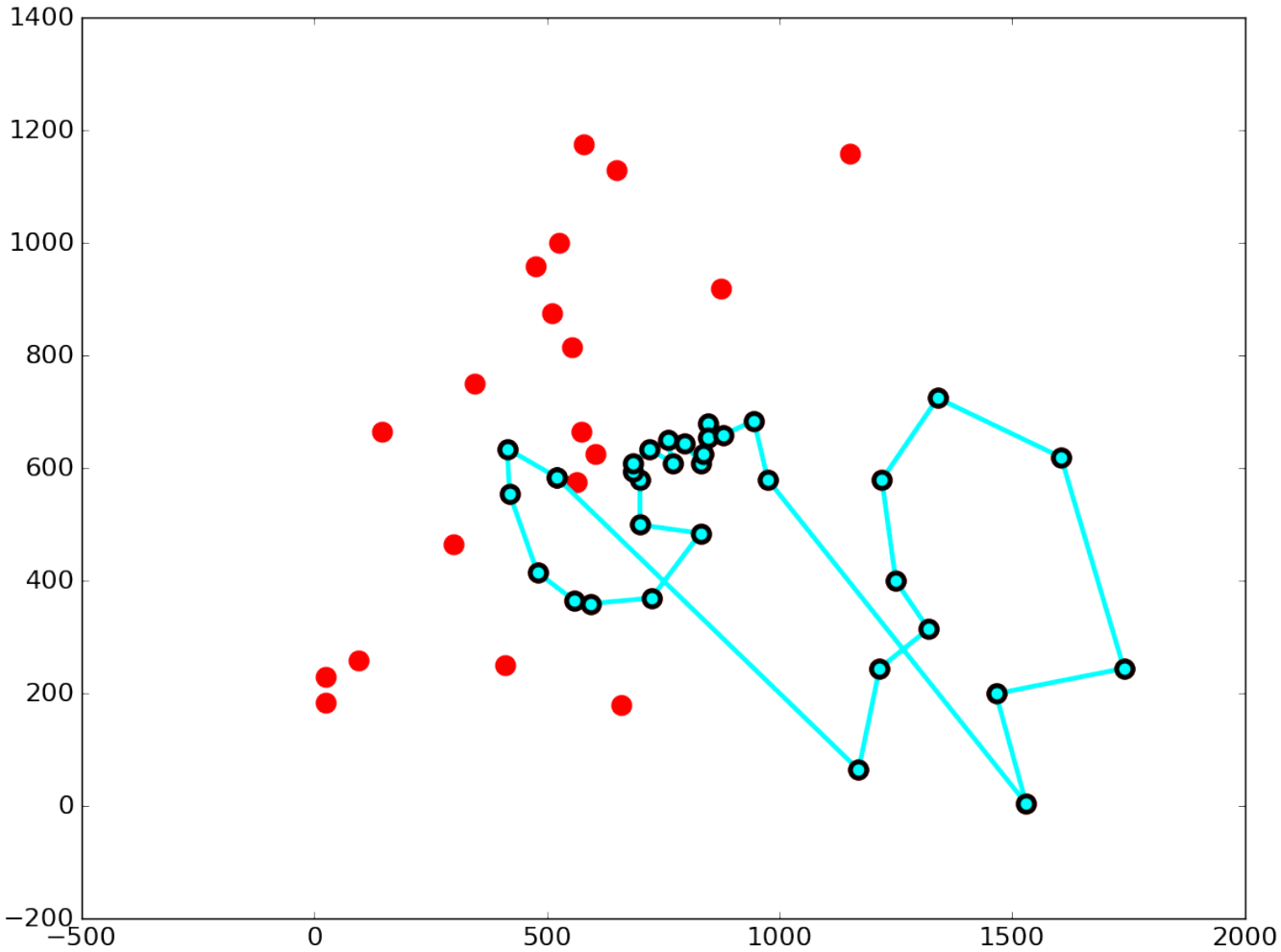
animation of the closest-neighbor algorithm



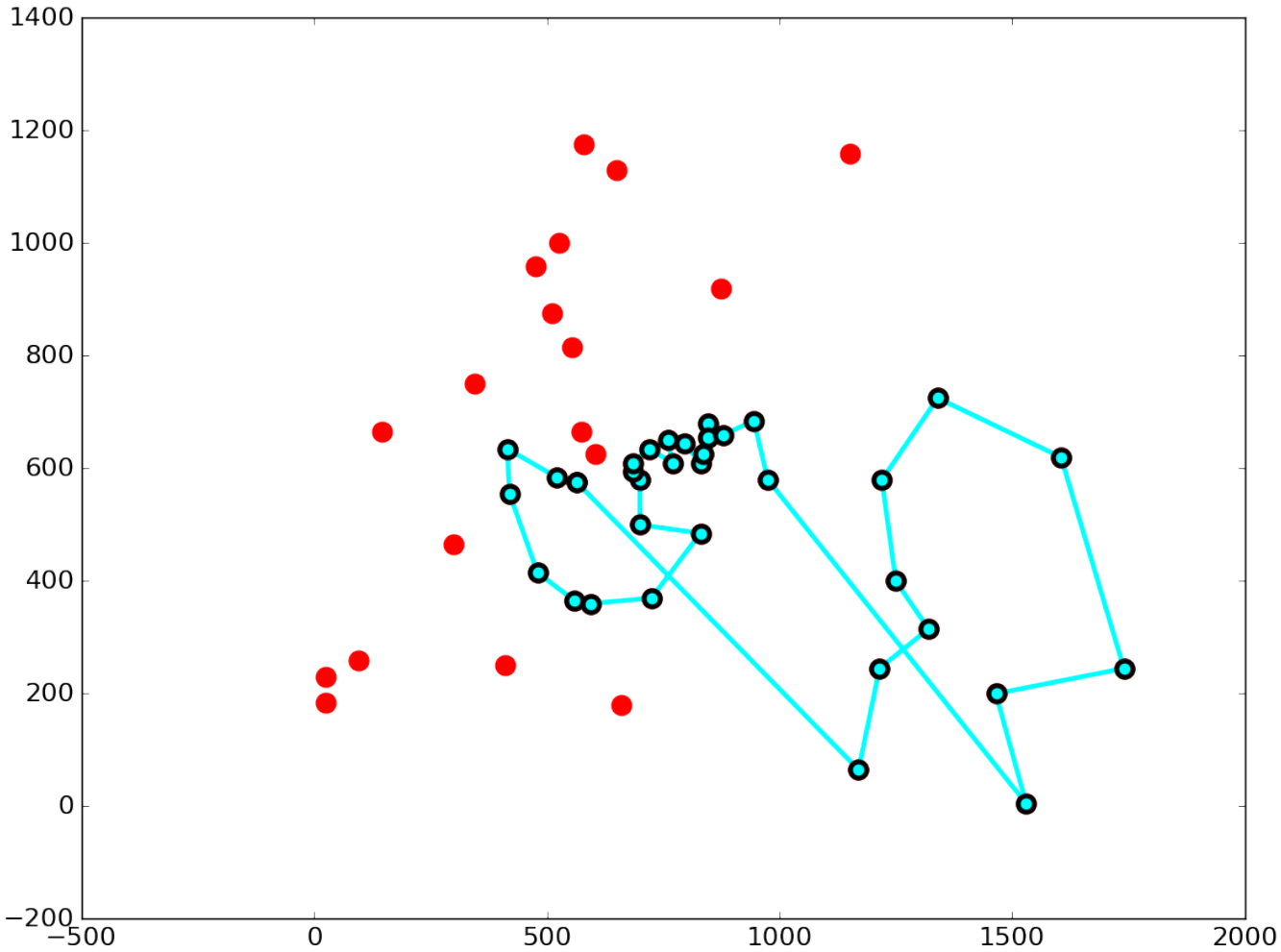
animation of the closest-neighbor algorithm



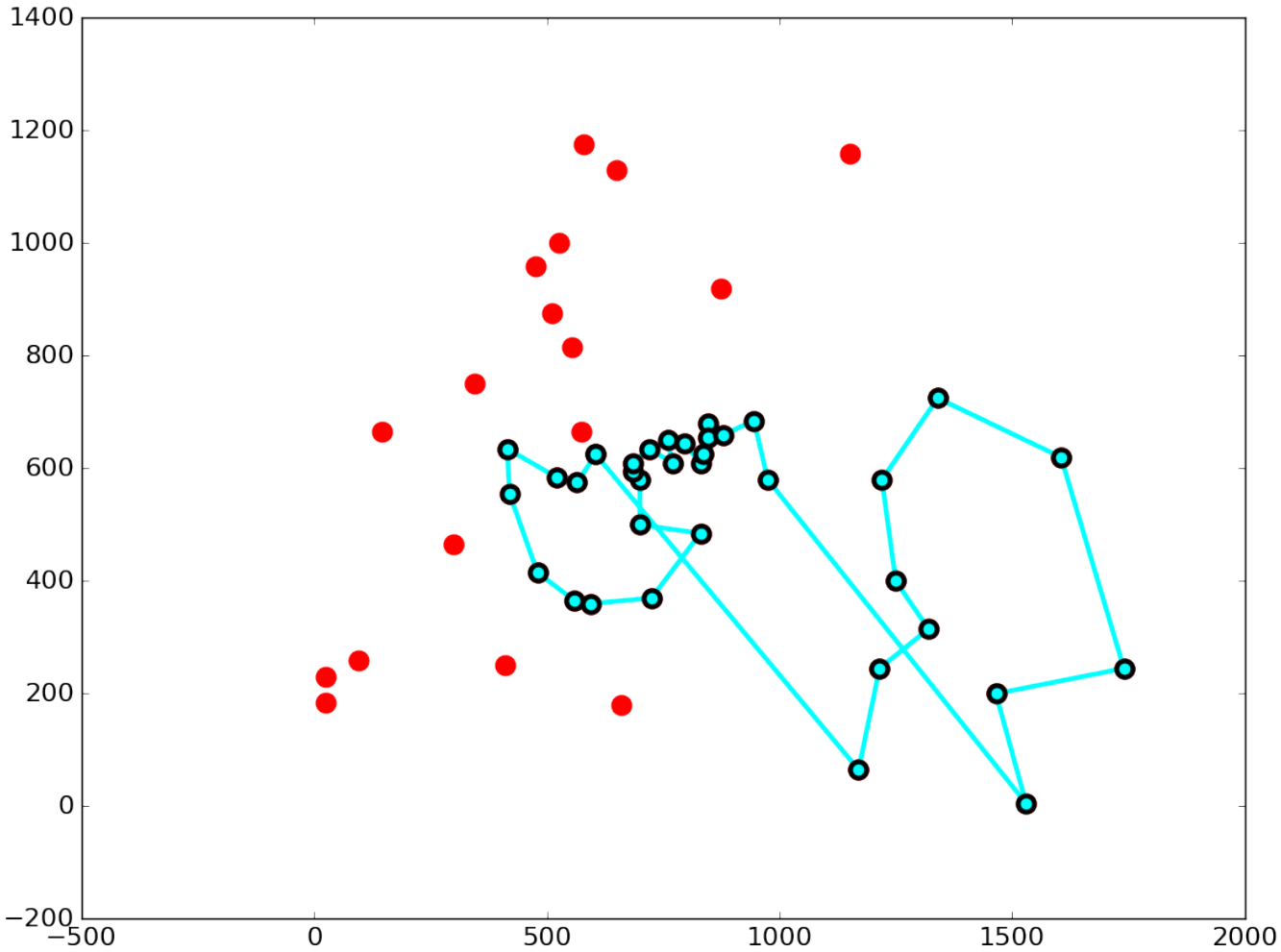
animation of the closest-neighbor algorithm



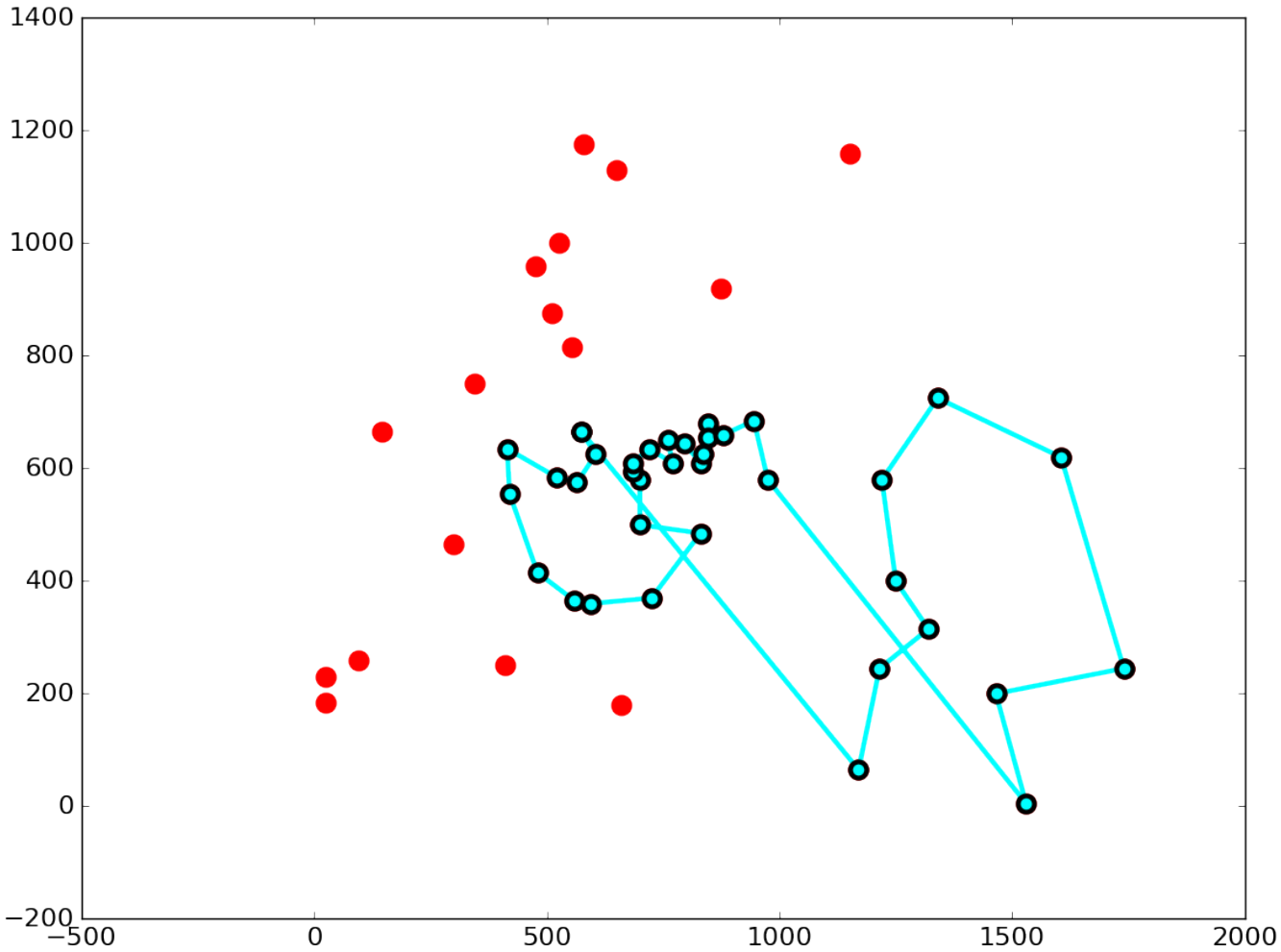
animation of the closest-neighbor algorithm



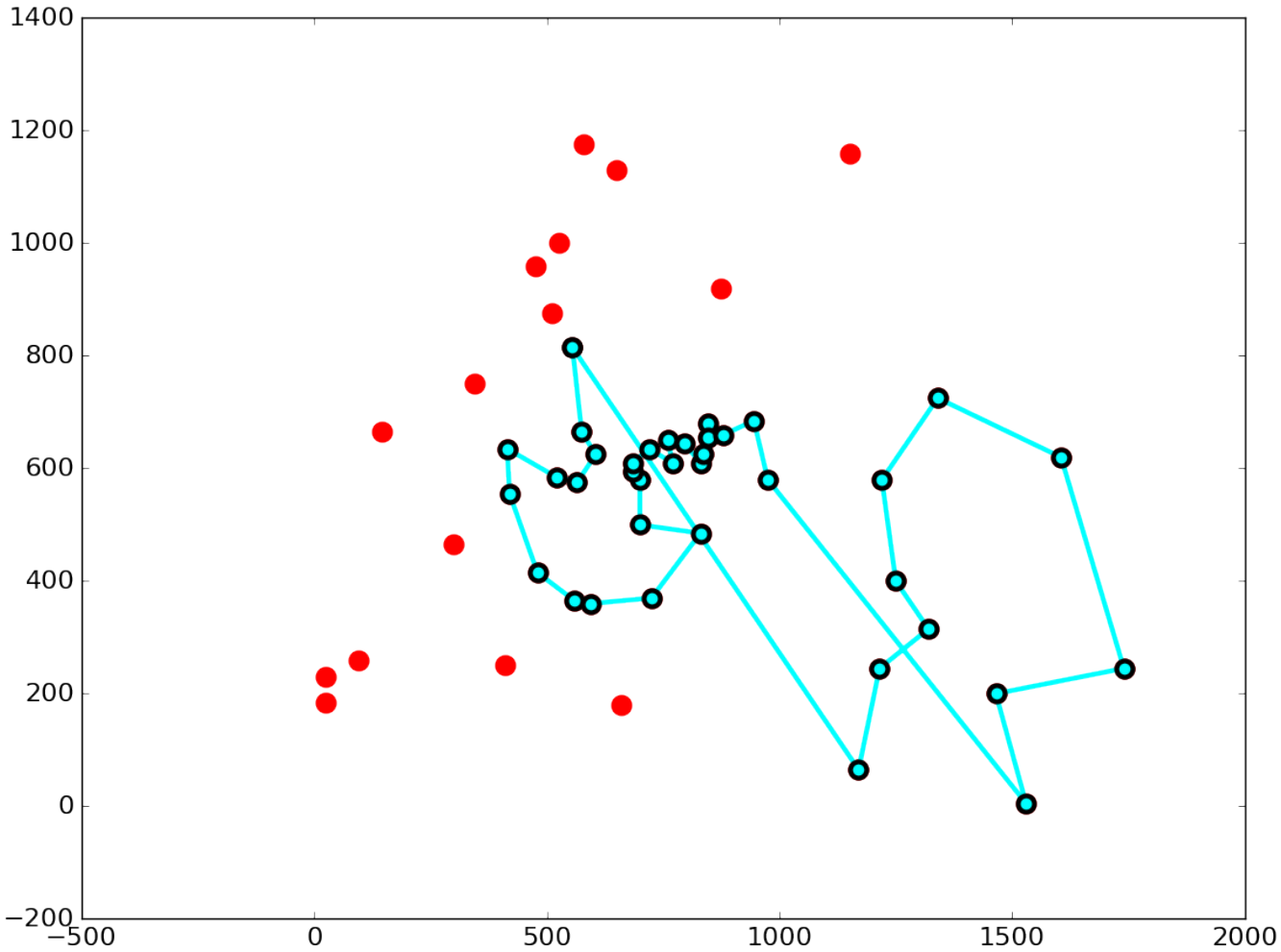
animation of the closest-neighbor algorithm



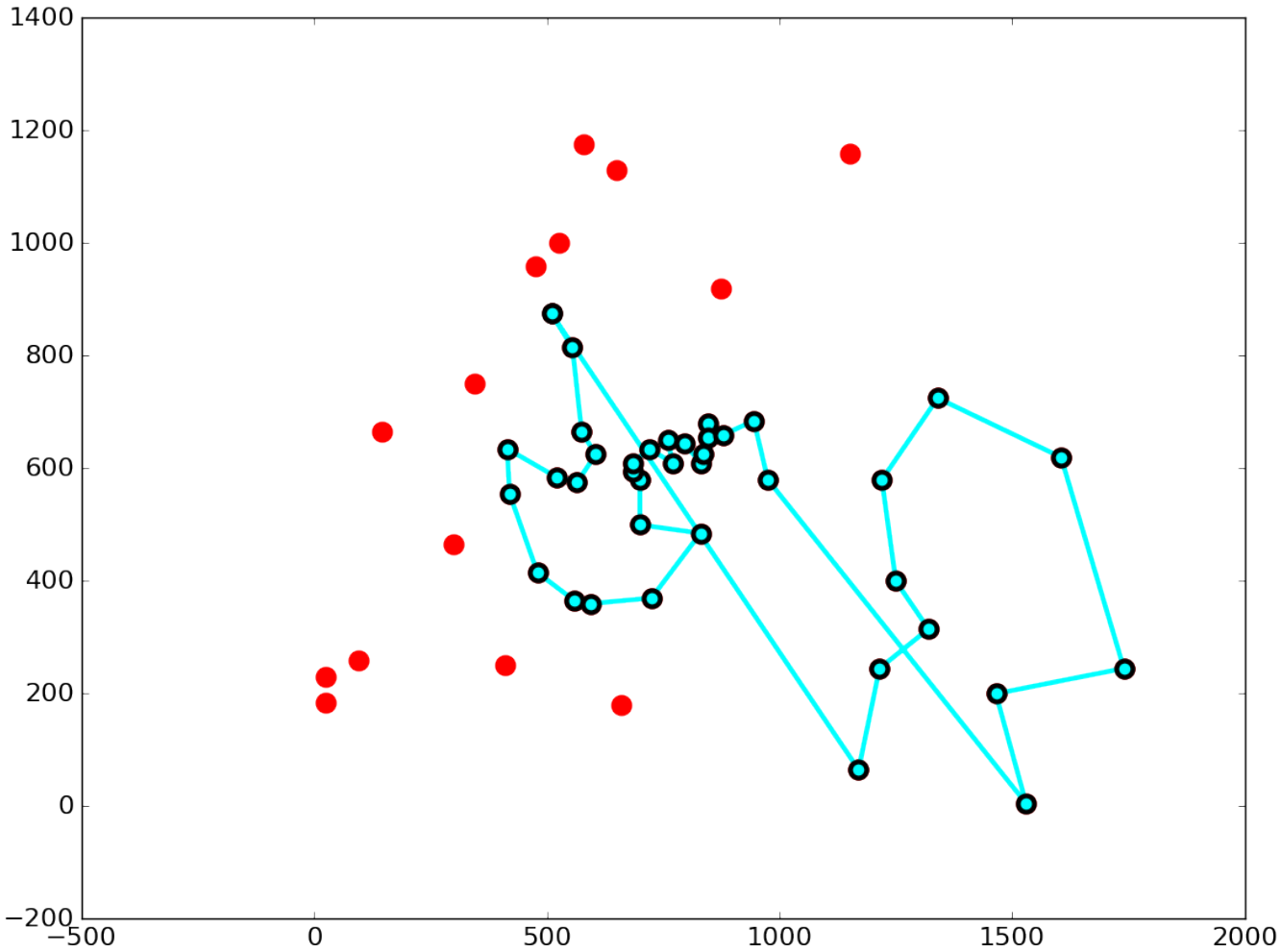
animation of the closest-neighbor algorithm



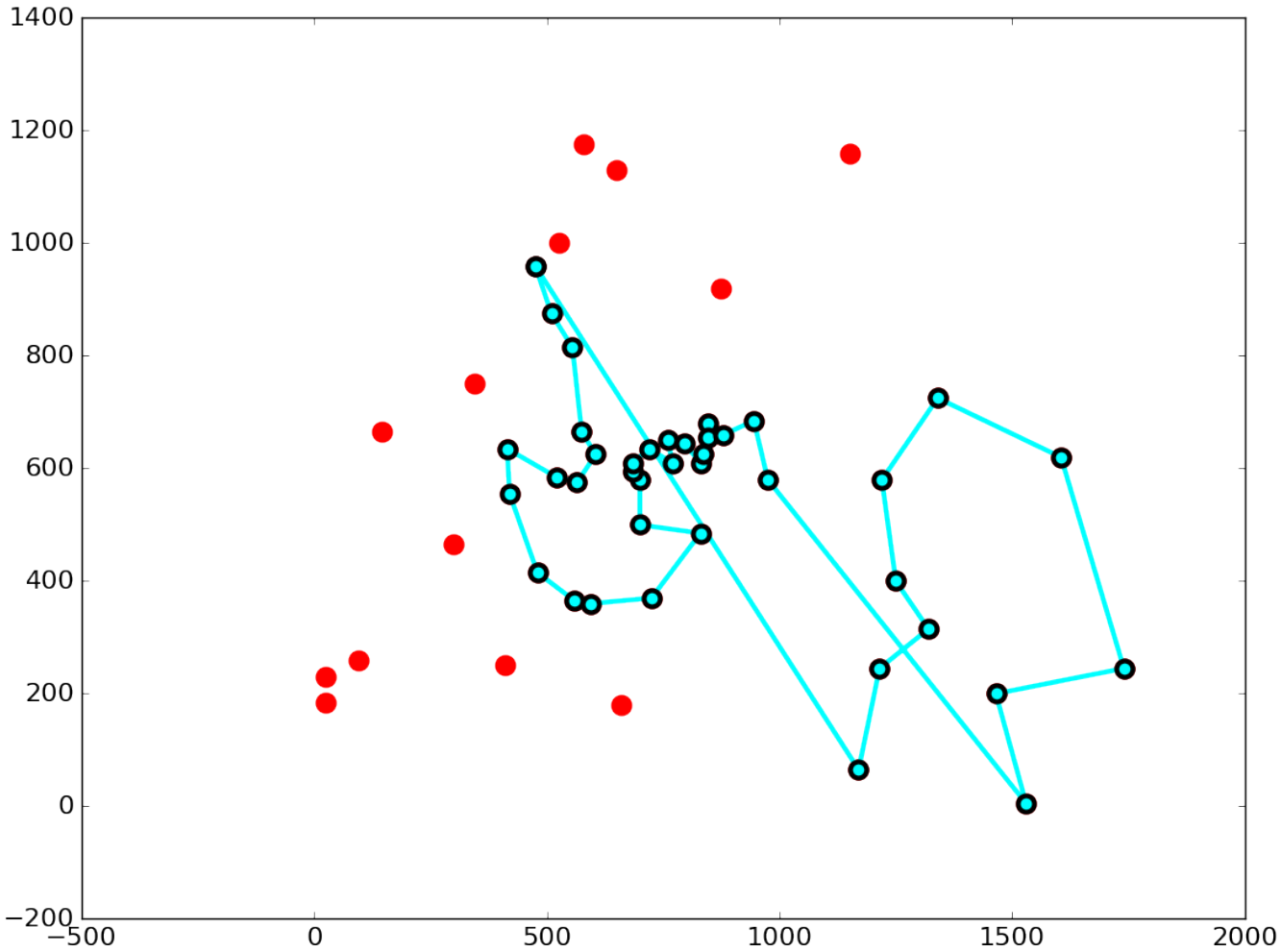
animation of the closest-neighbor algorithm



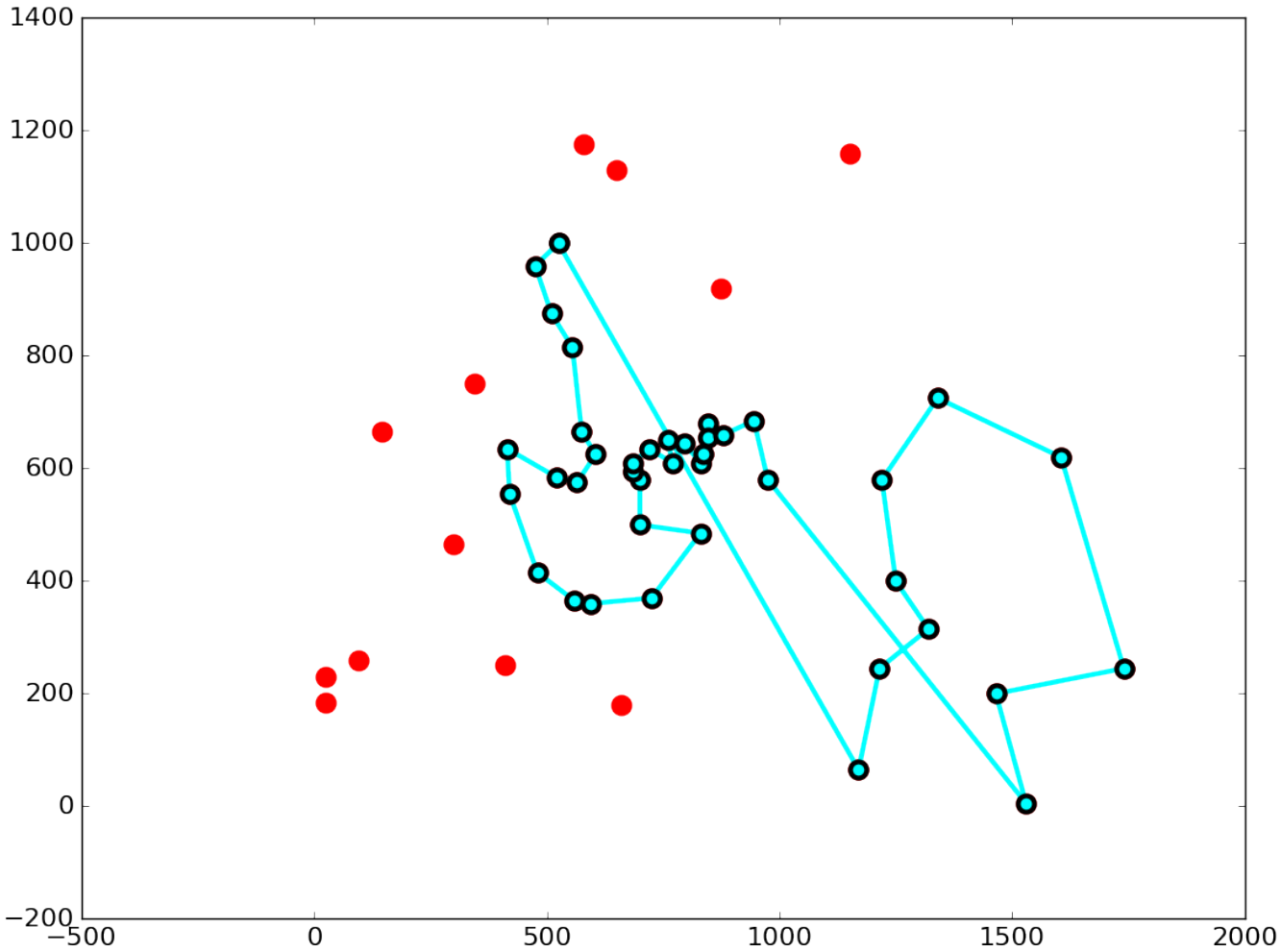
animation of the closest-neighbor algorithm



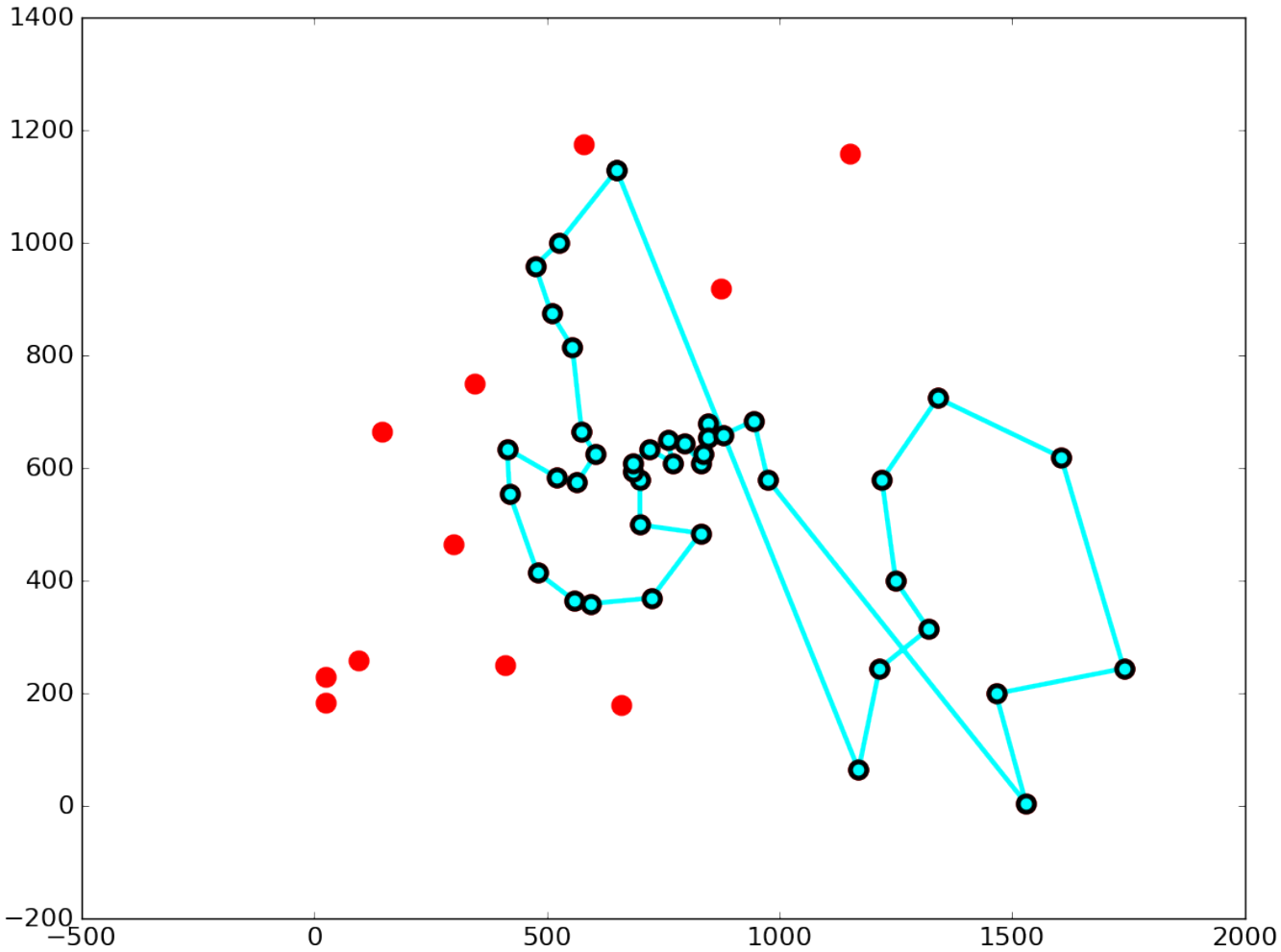
animation of the closest-neighbor algorithm



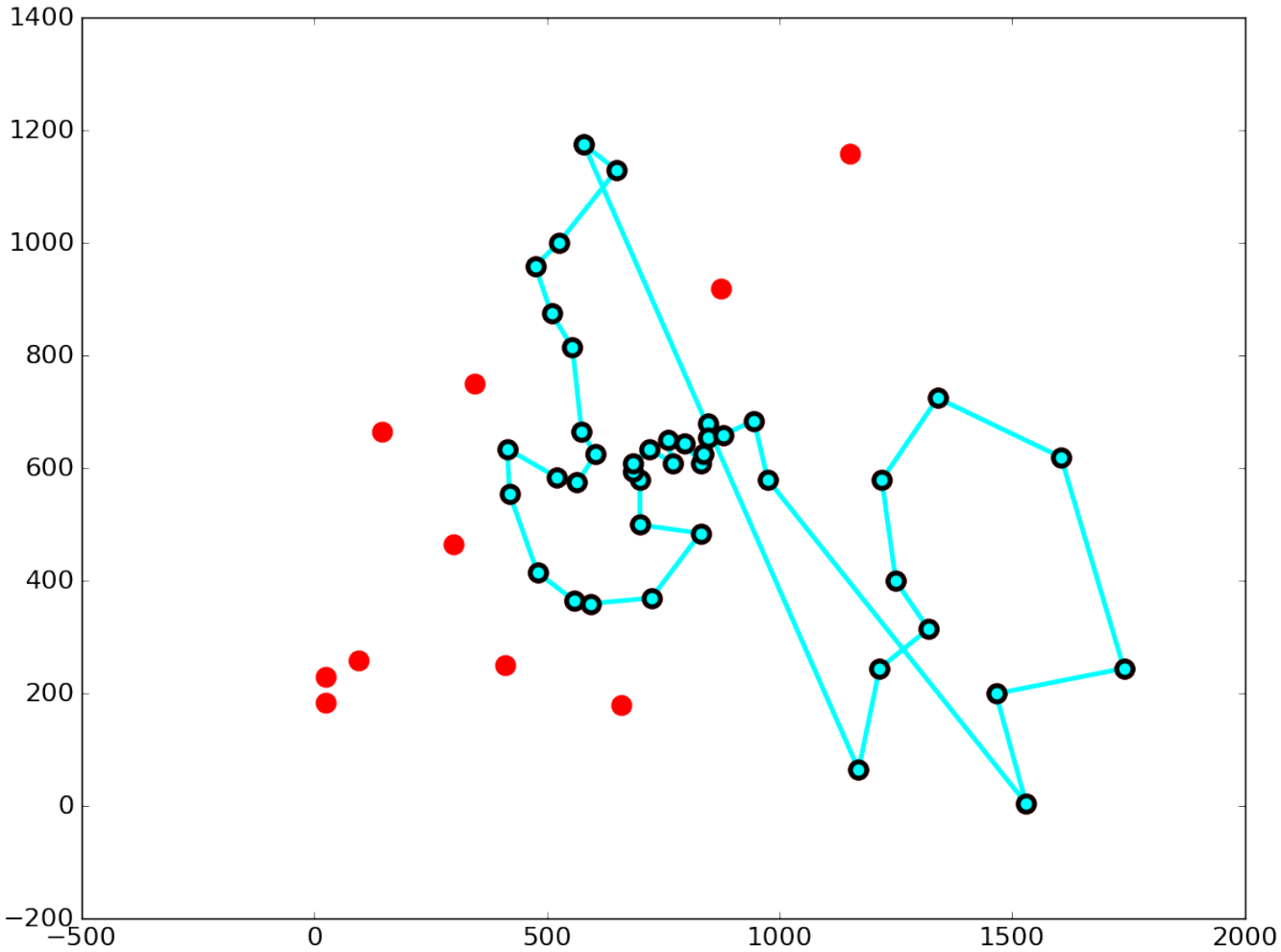
animation of the closest-neighbor algorithm



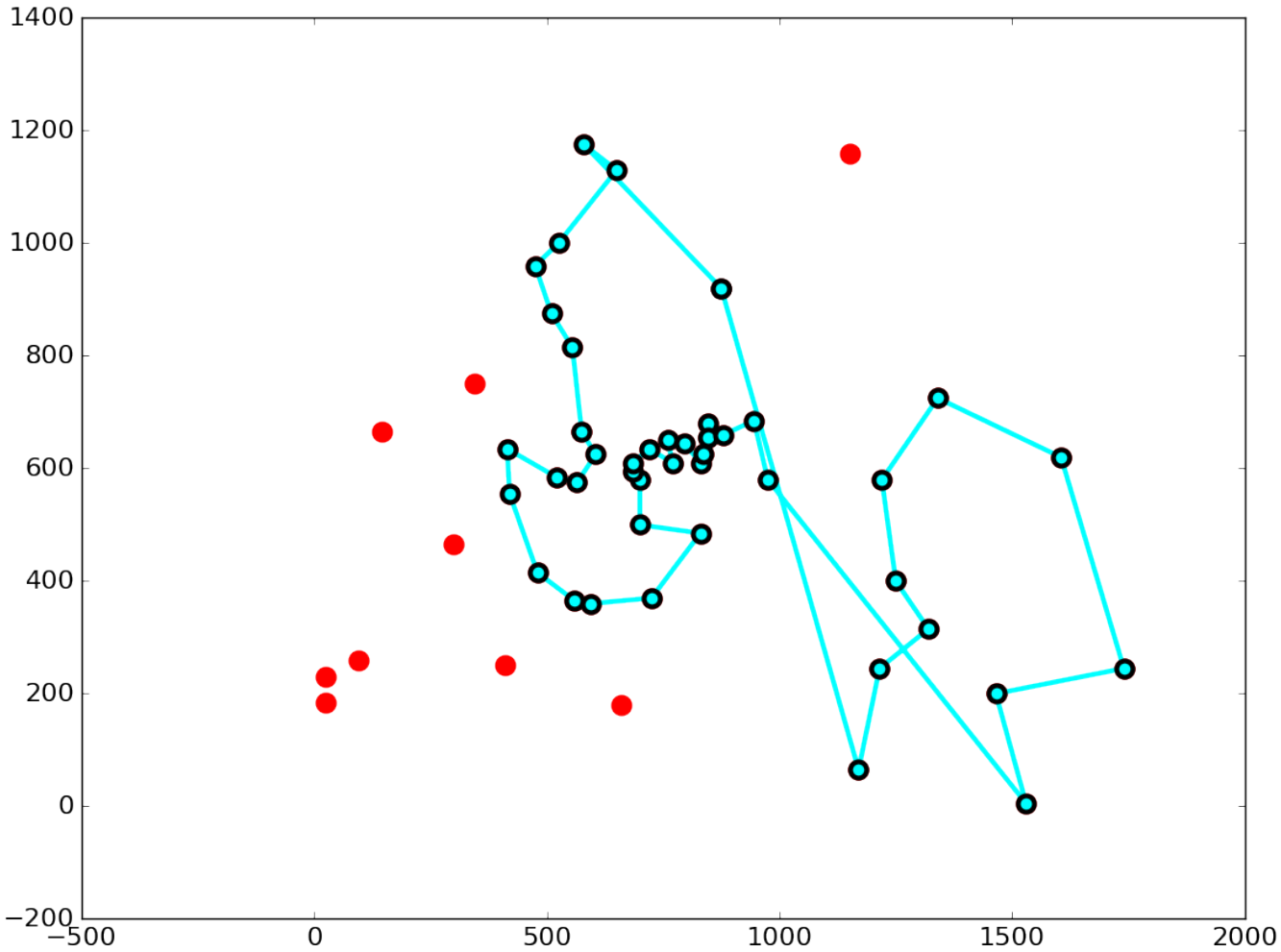
animation of the closest-neighbor algorithm



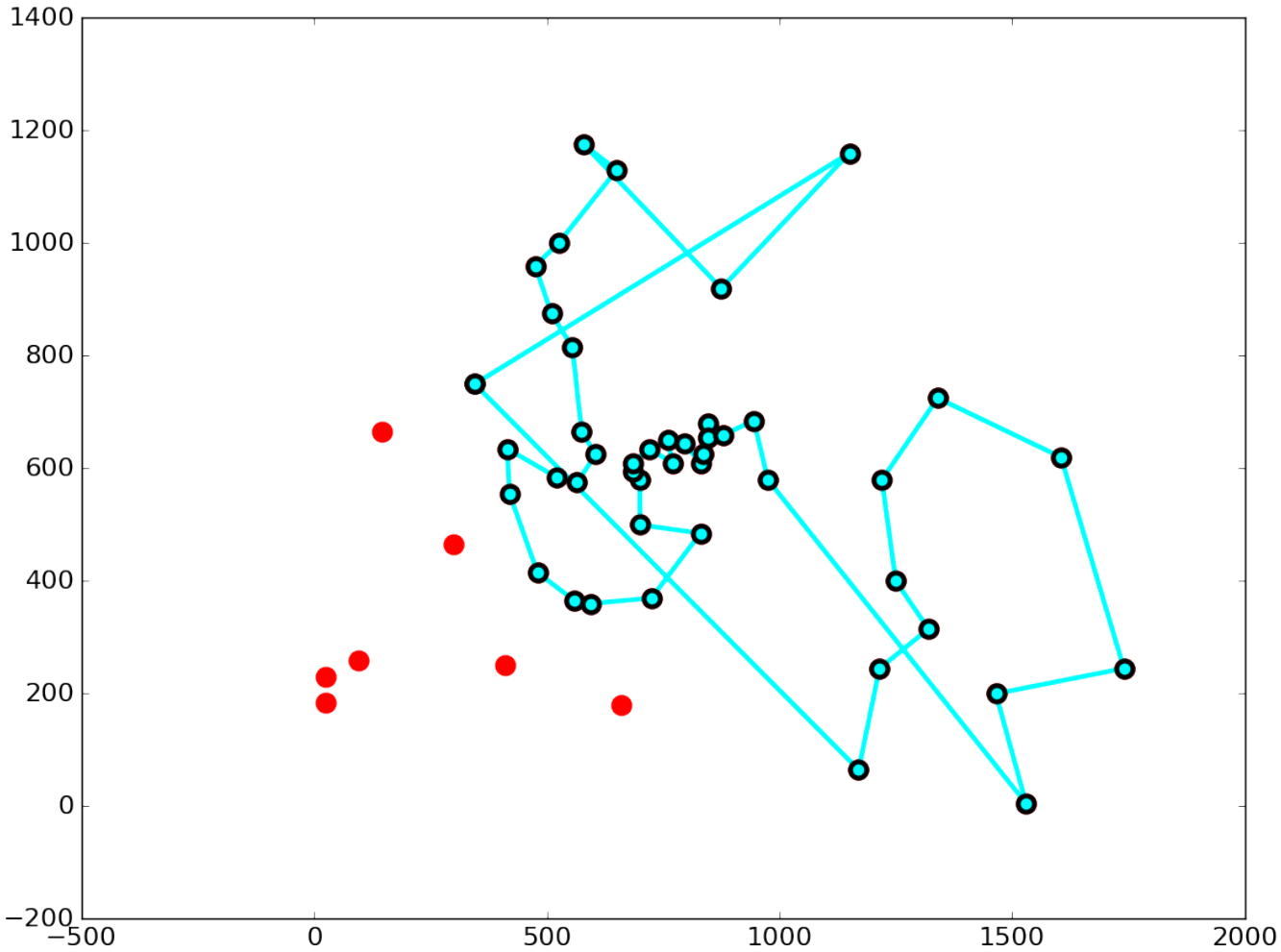
animation of the closest-neighbor algorithm



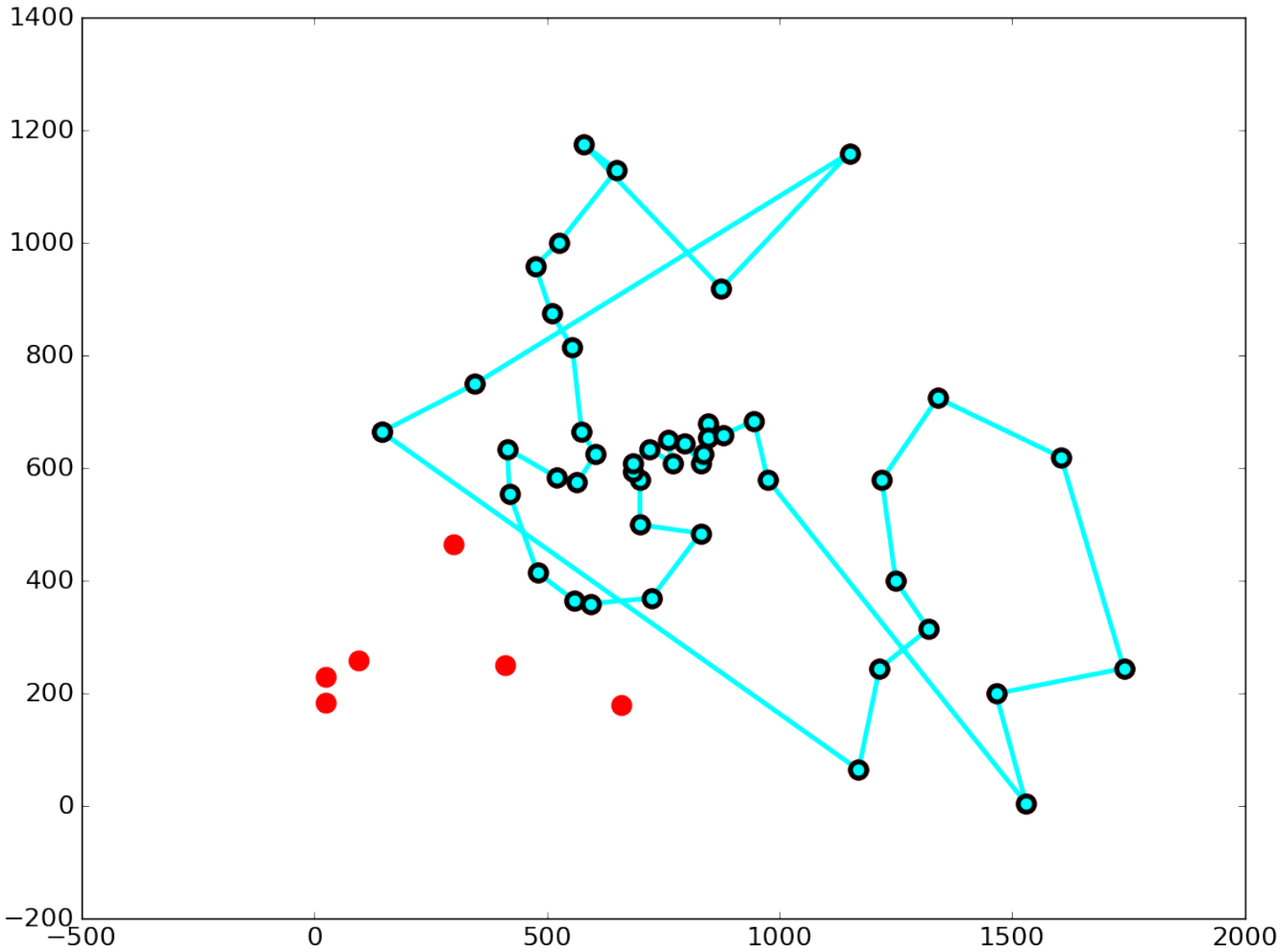
animation of the closest-neighbor algorithm



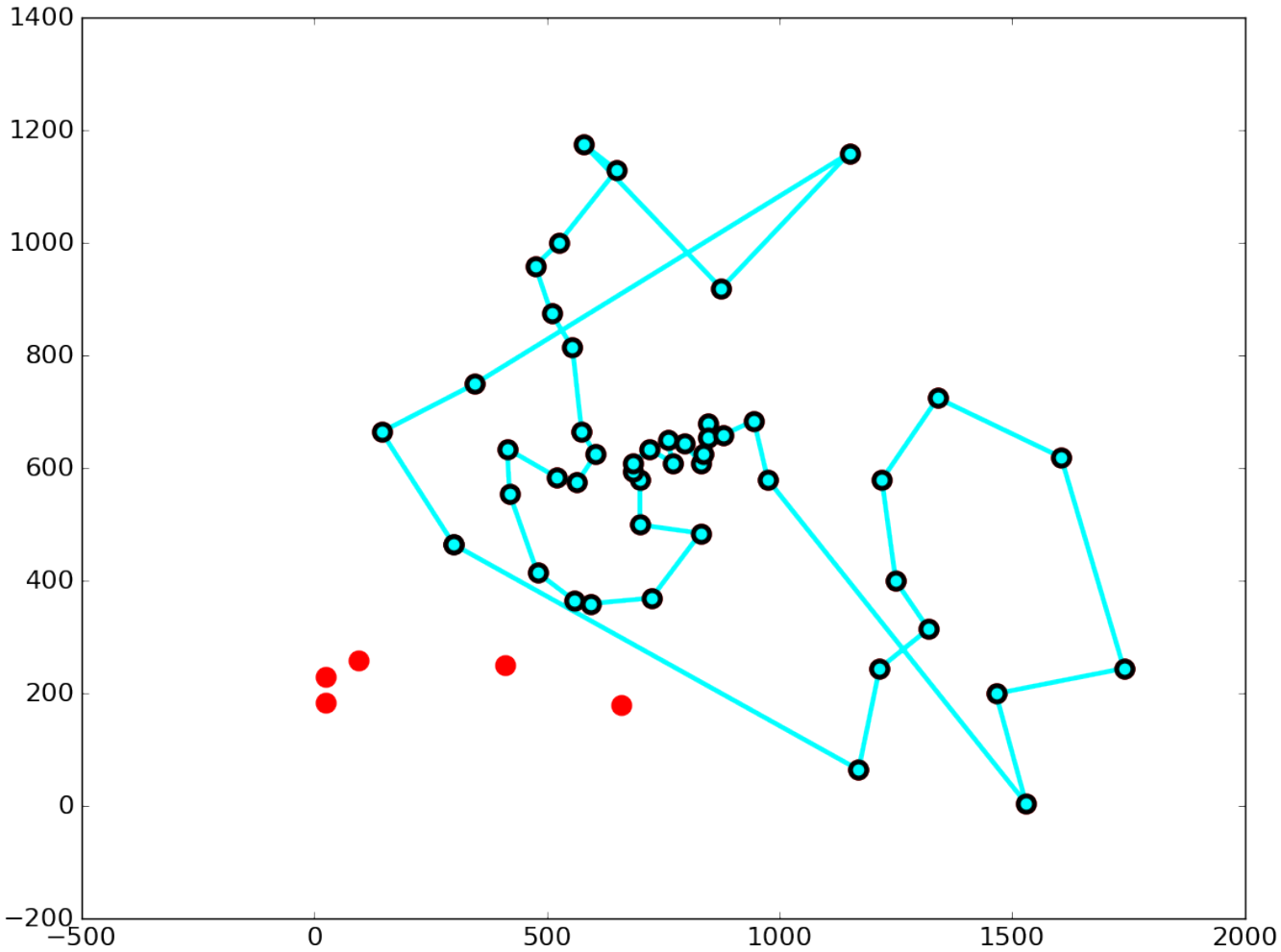
animation of the closest-neighbor algorithm



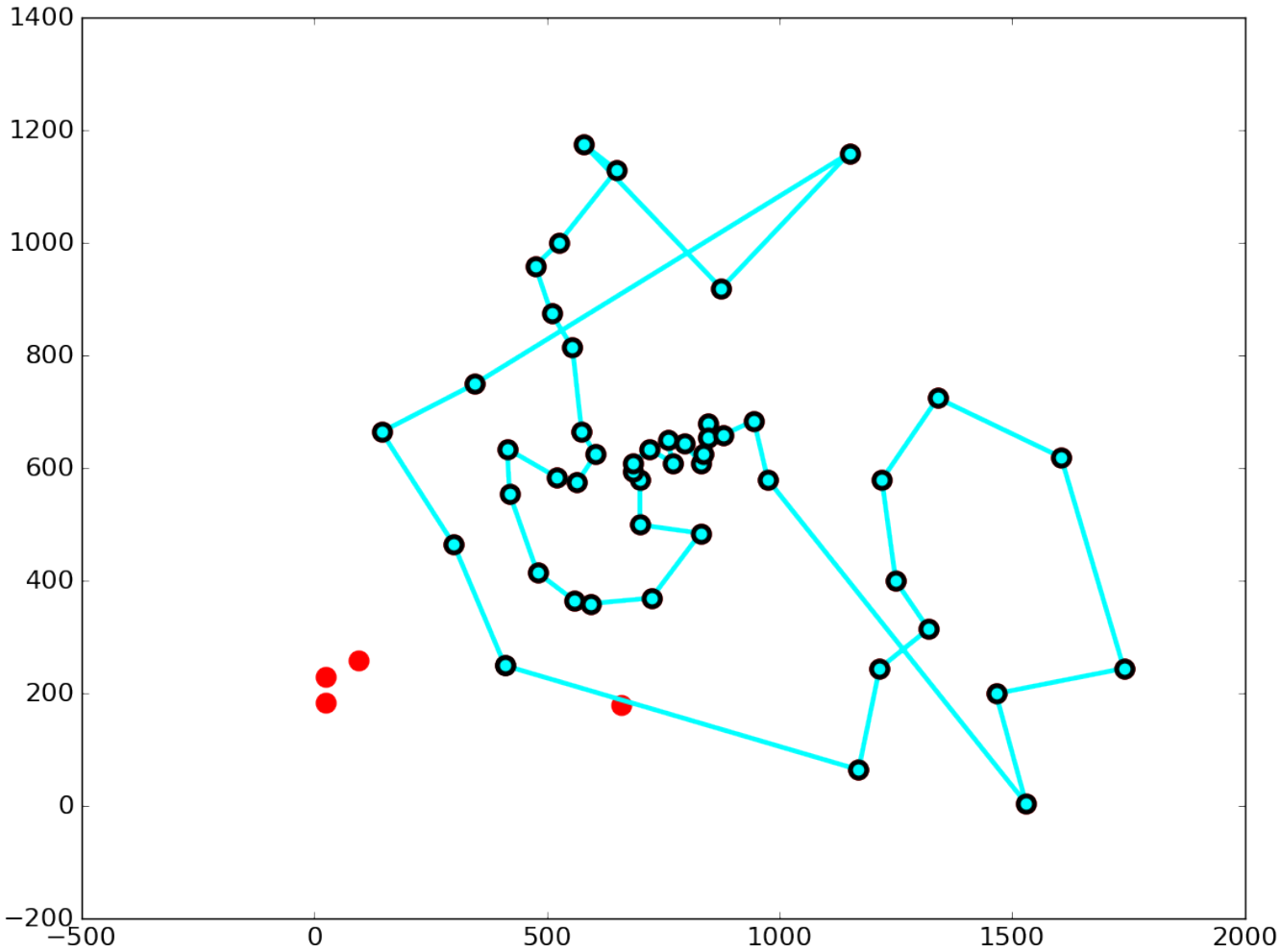
animation of the closest-neighbor algorithm



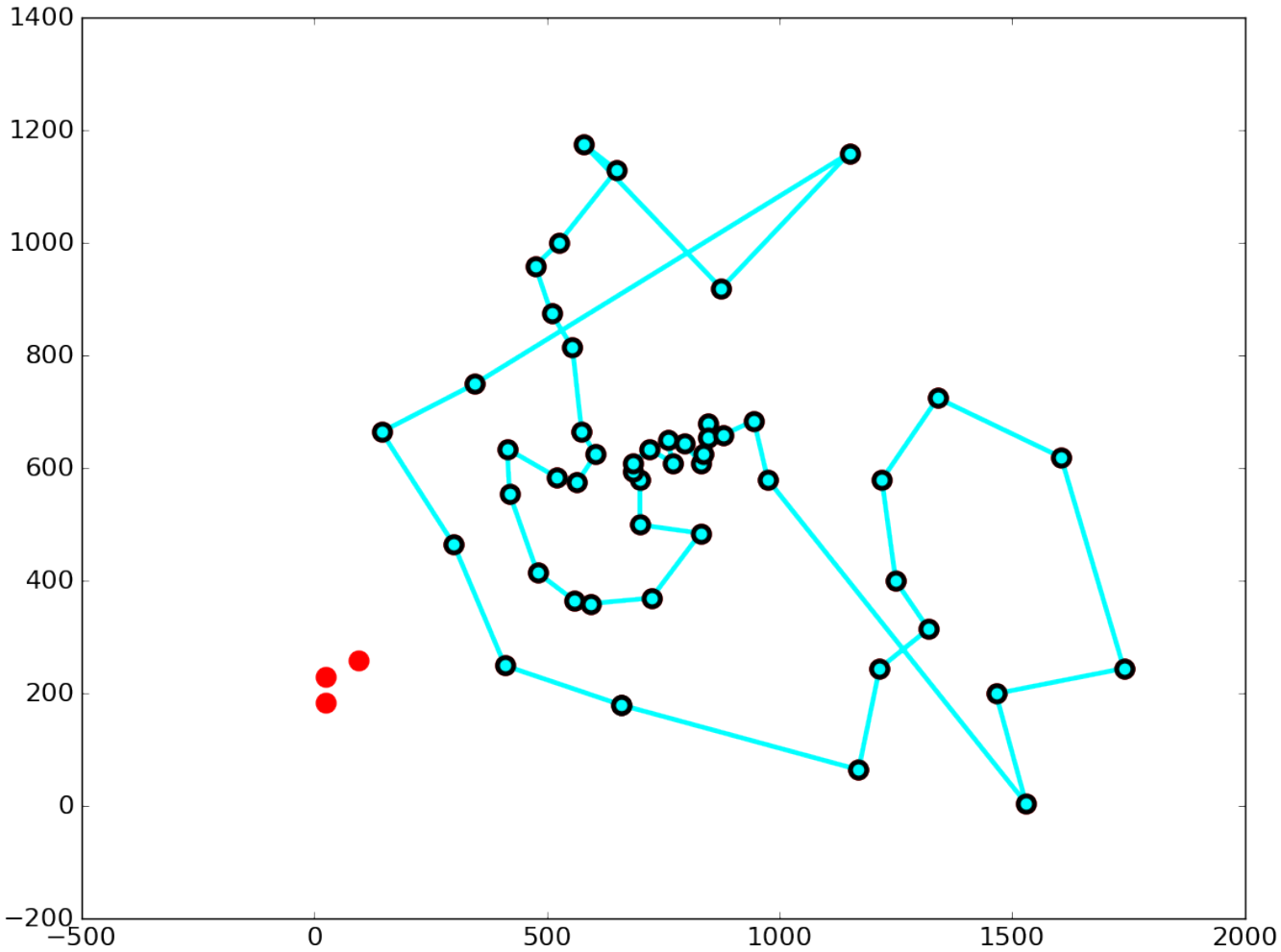
animation of the closest-neighbor algorithm



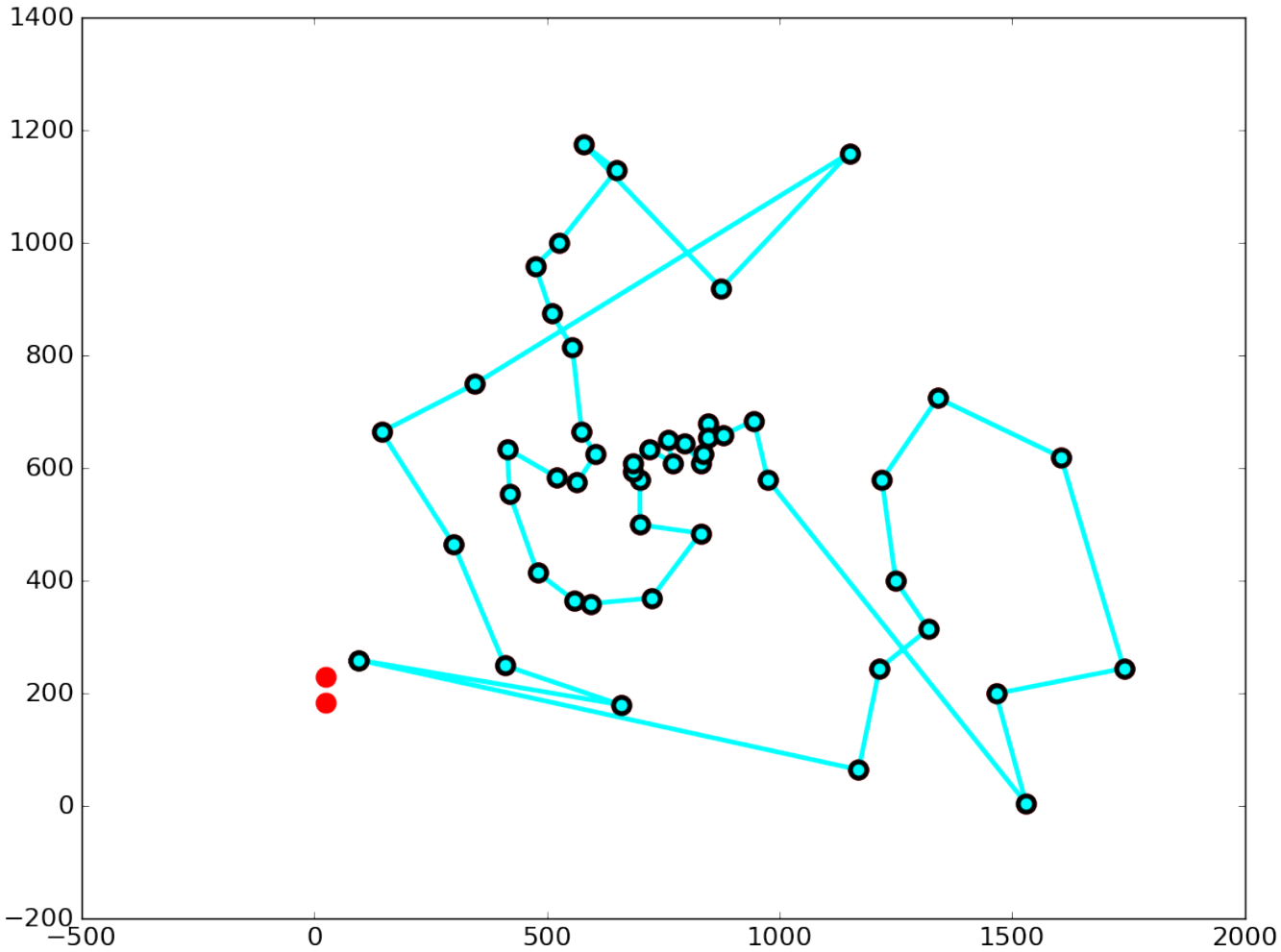
animation of the closest-neighbor algorithm



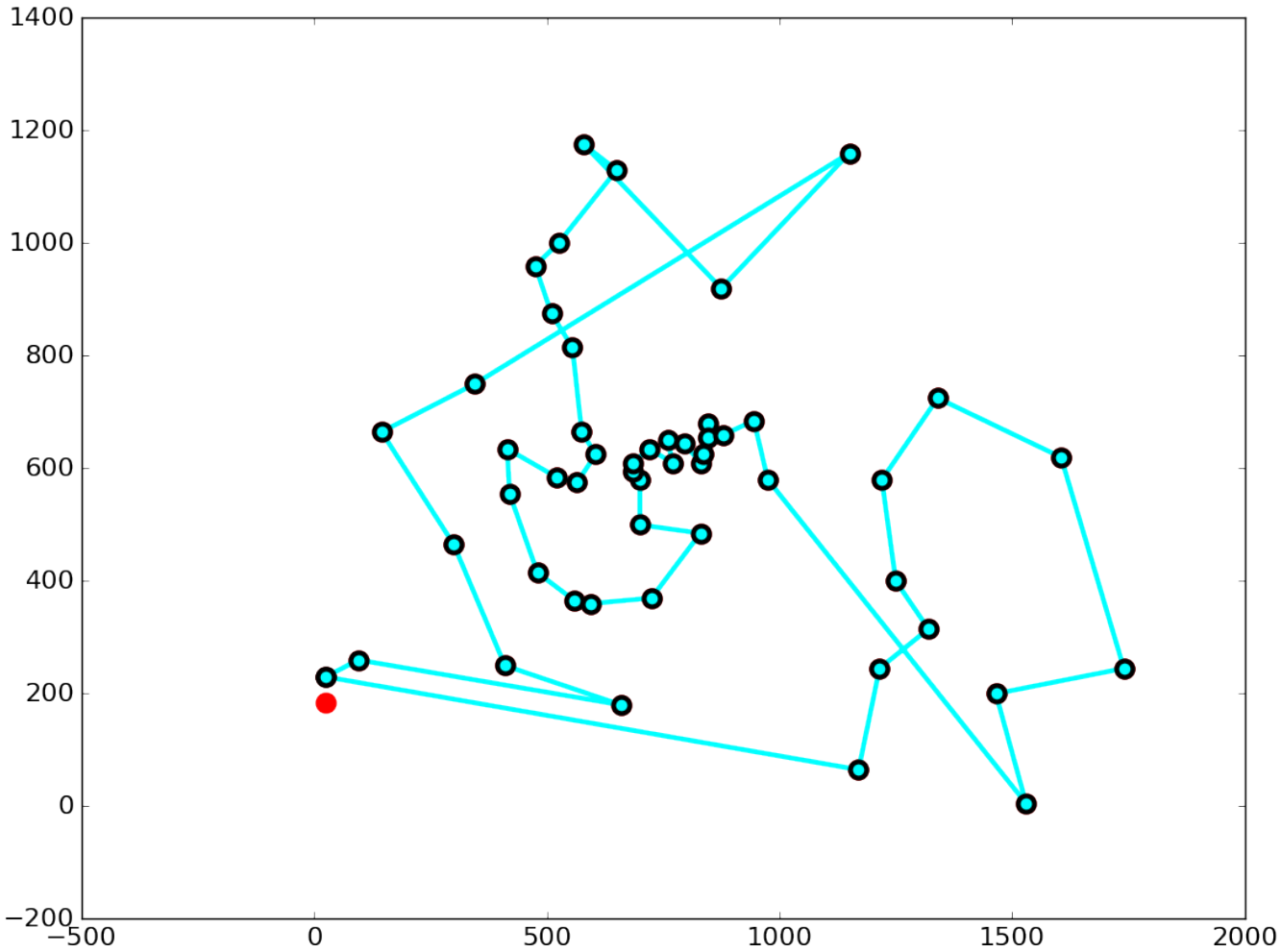
animation of the closest-neighbor algorithm



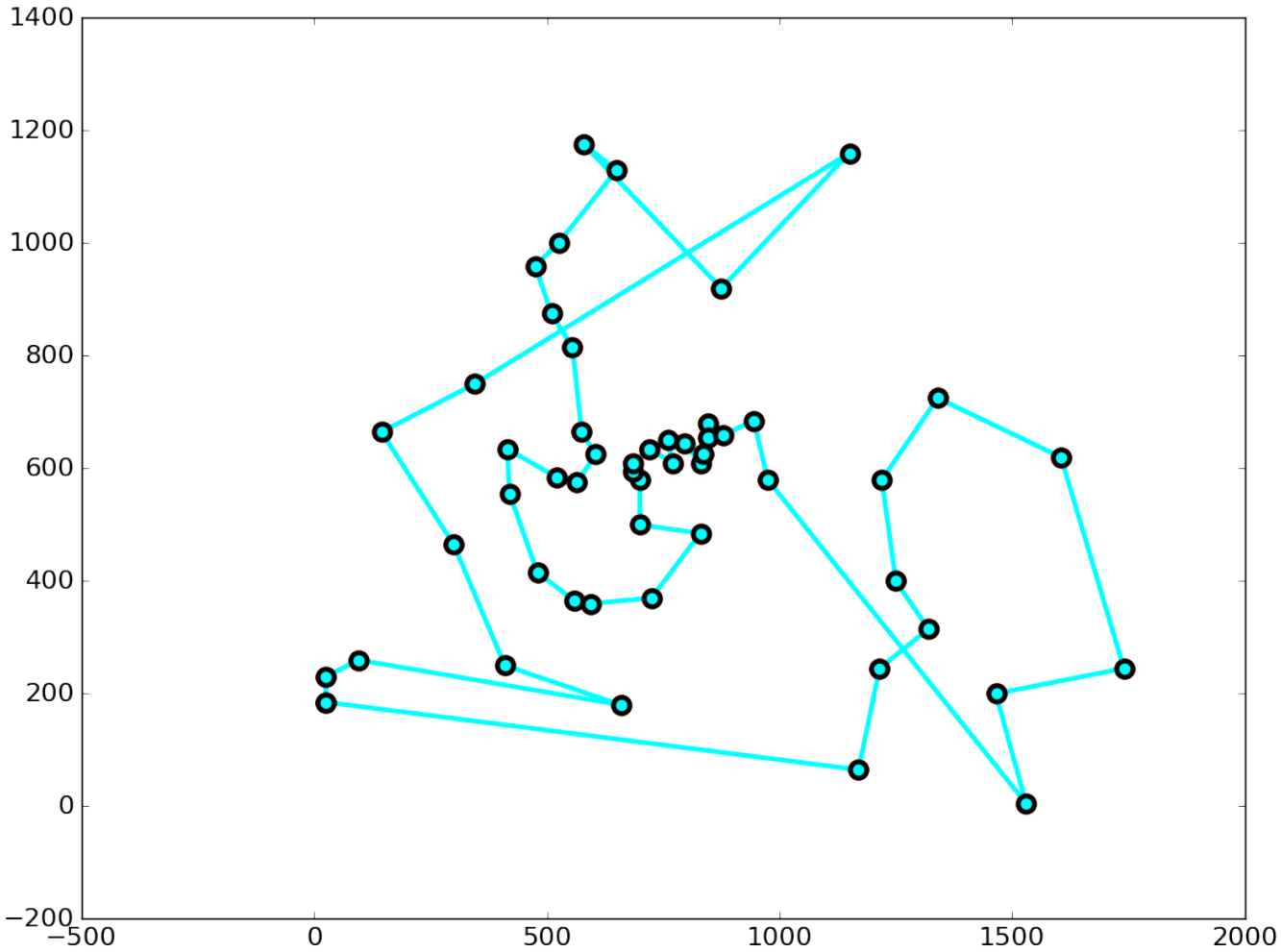
animation of the closest-neighbor algorithm



animation of the closest-neighbor algorithm



animation of the closest-neighbor algorithm



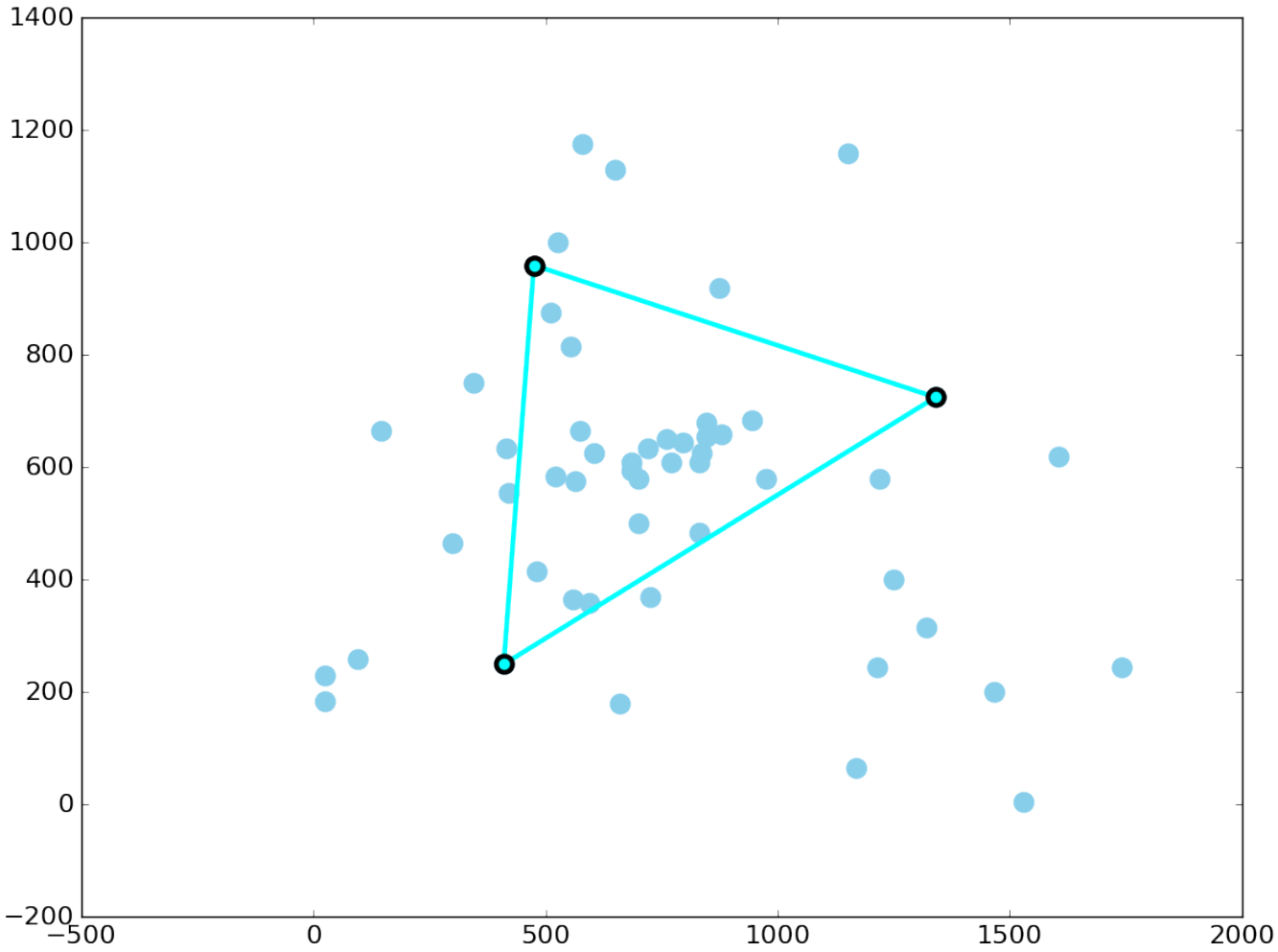
How does the quick tour algorithm work?

The quick tour algorithm is my own contribution (but found to be known as *insertion/addition approach*) for this course. It is a probabilistic greedy algorithm:

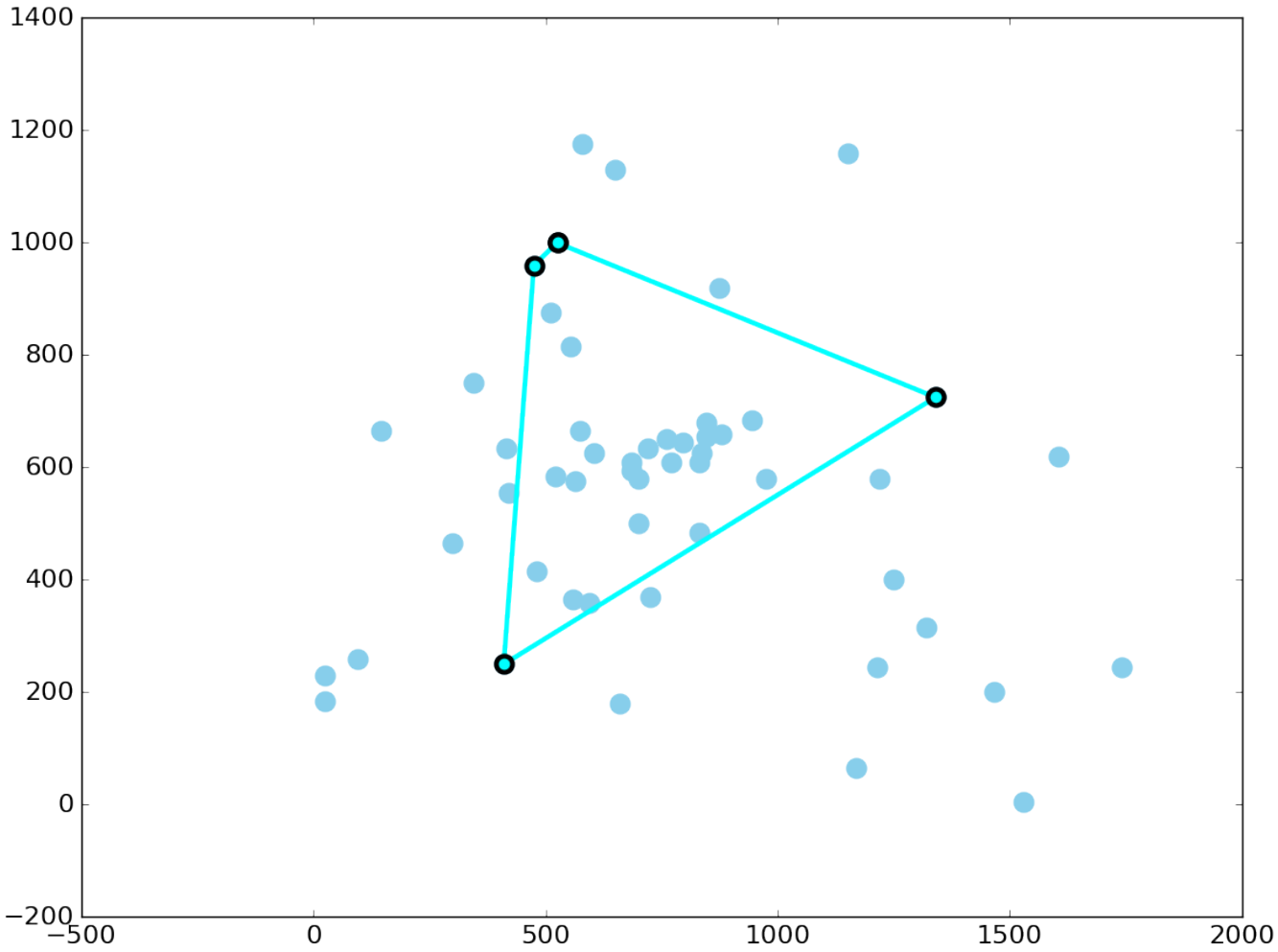
- select three random cities to form an initial triangular tour
- while there are still unconnected cities
  - choose a random (closest) unconnected city
  - expand the current tour by inserting the new city such that the tour increment is minimal
- runtime is in $\mathcal{O}(n^2)$,
- the random version can be run in Monte Carlo fashion keeping the shortest tour
- worst tour may have a length up to $2 \cdot L_{opt}$

There are more similar approaches, e.g., nearest addition or farthest addition.

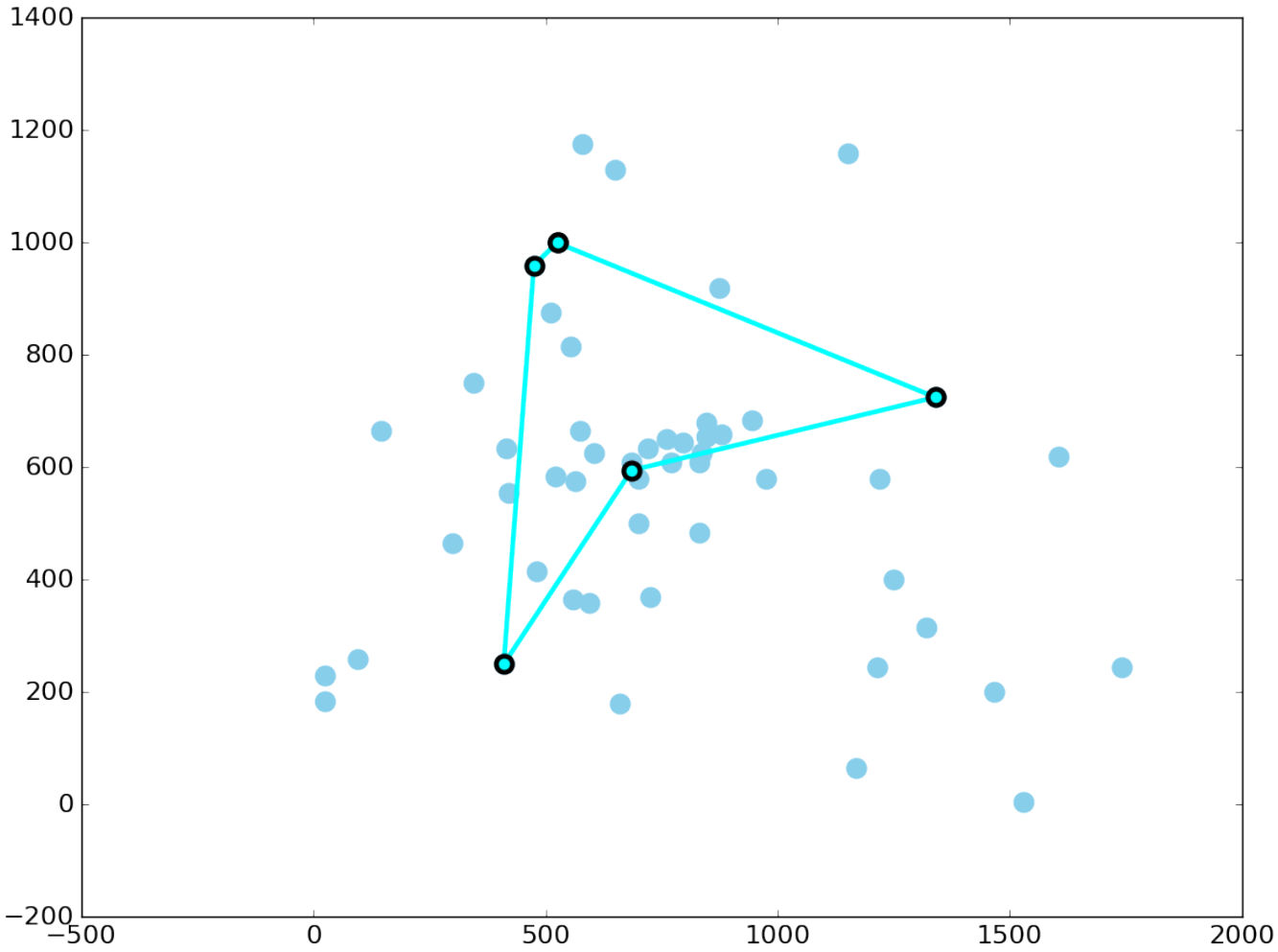
animation of the quick tour algorithm



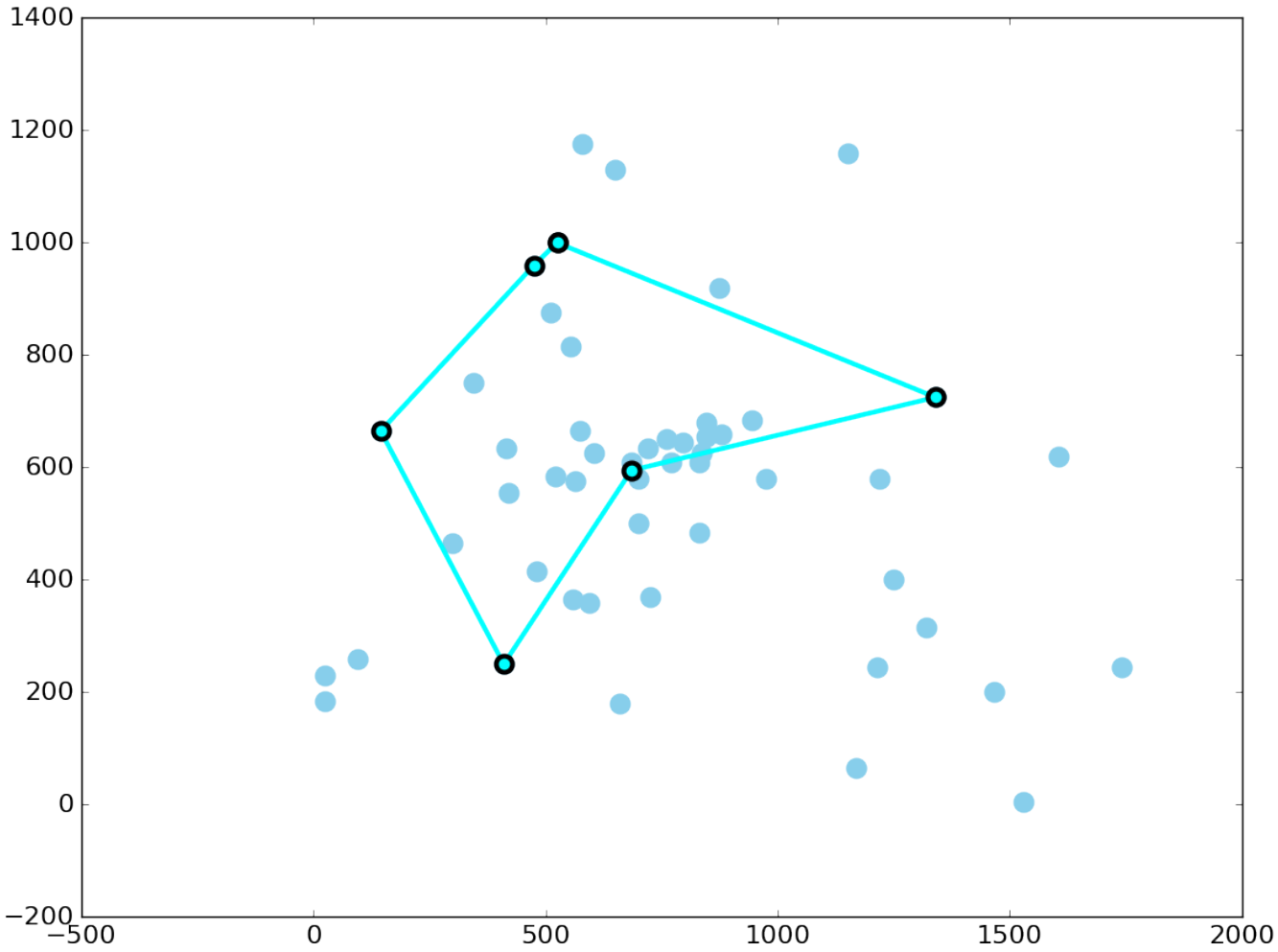
animation of the quick tour algorithm



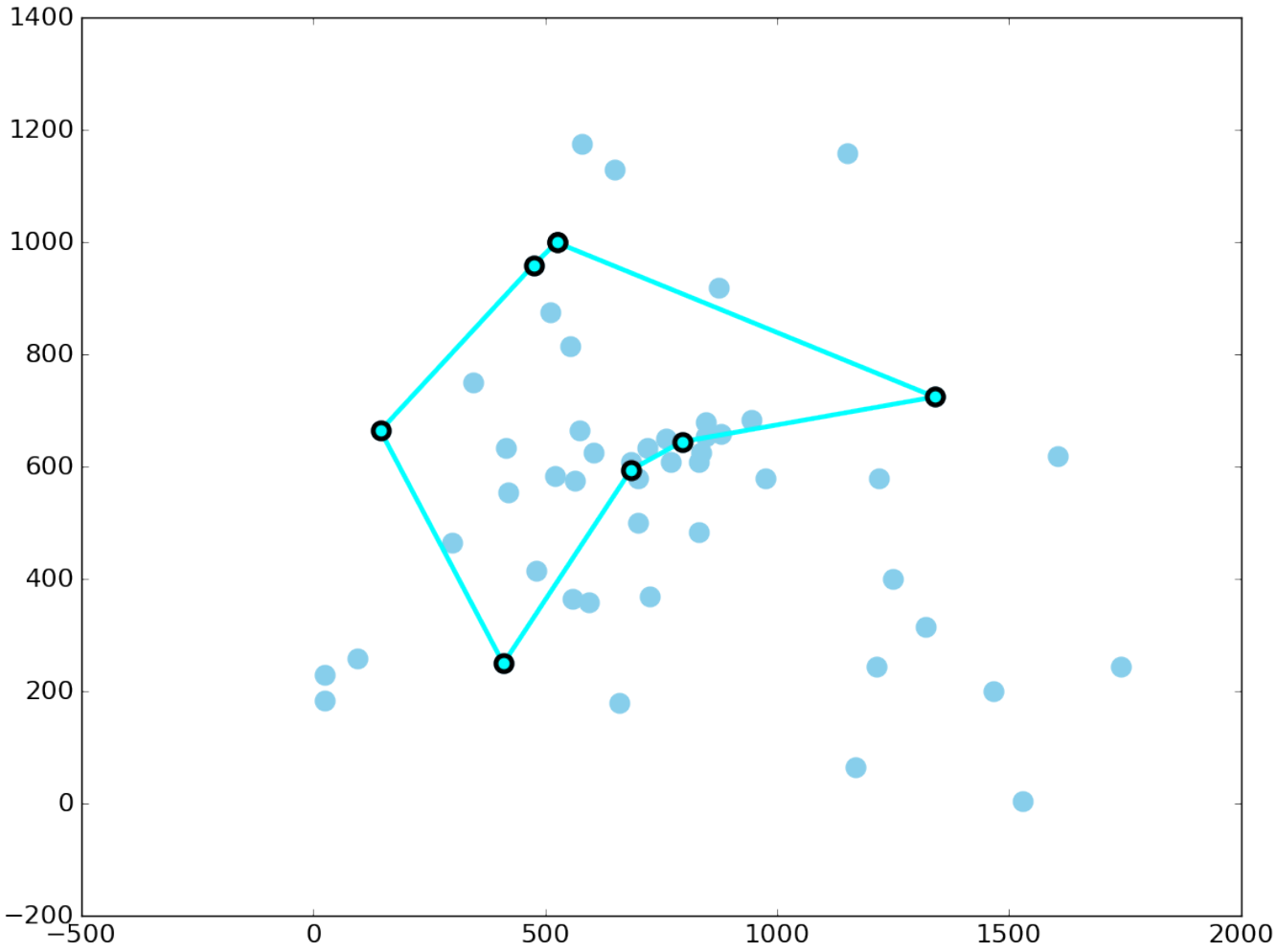
animation of the quick tour algorithm



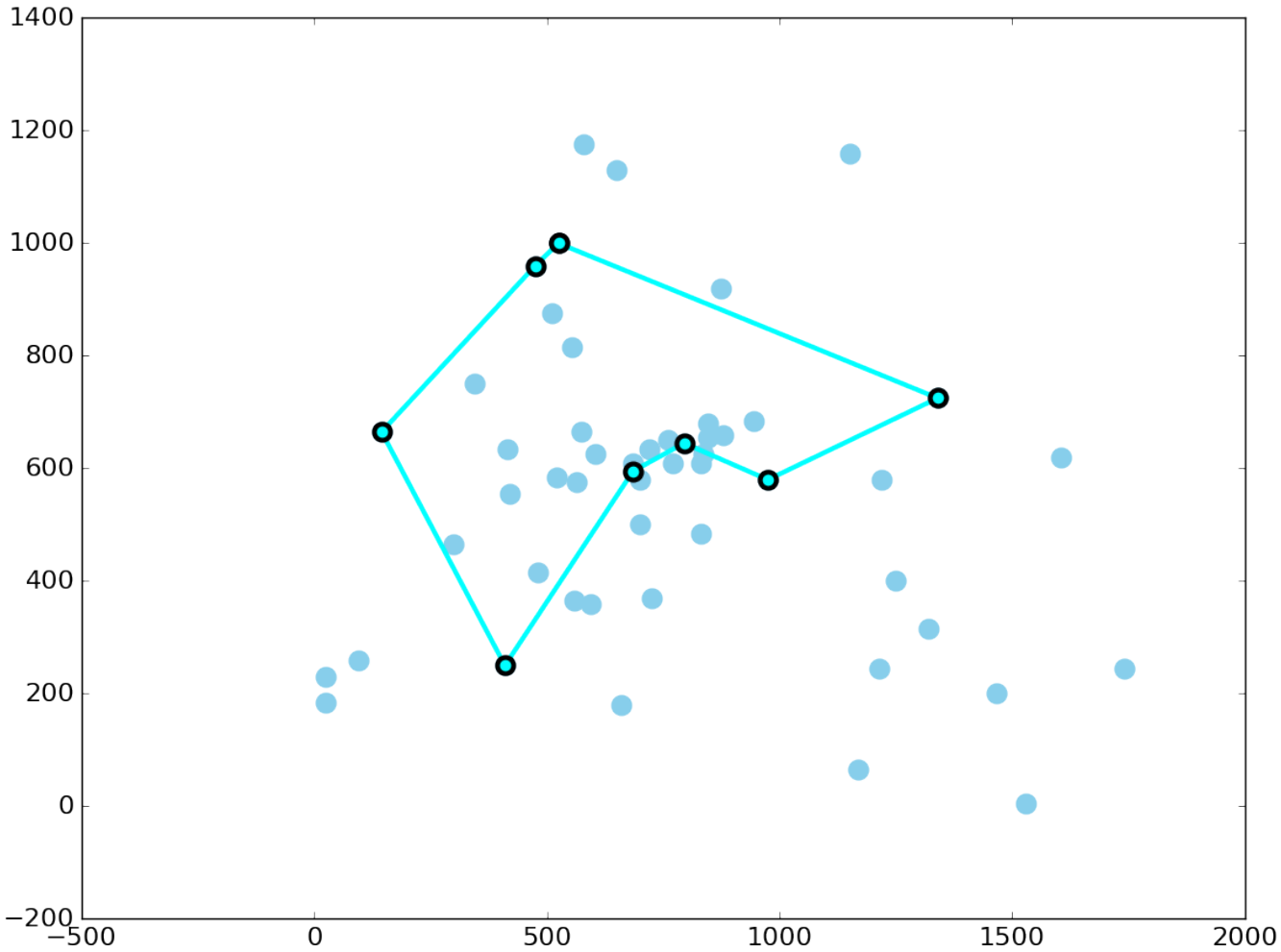
animation of the quick tour algorithm



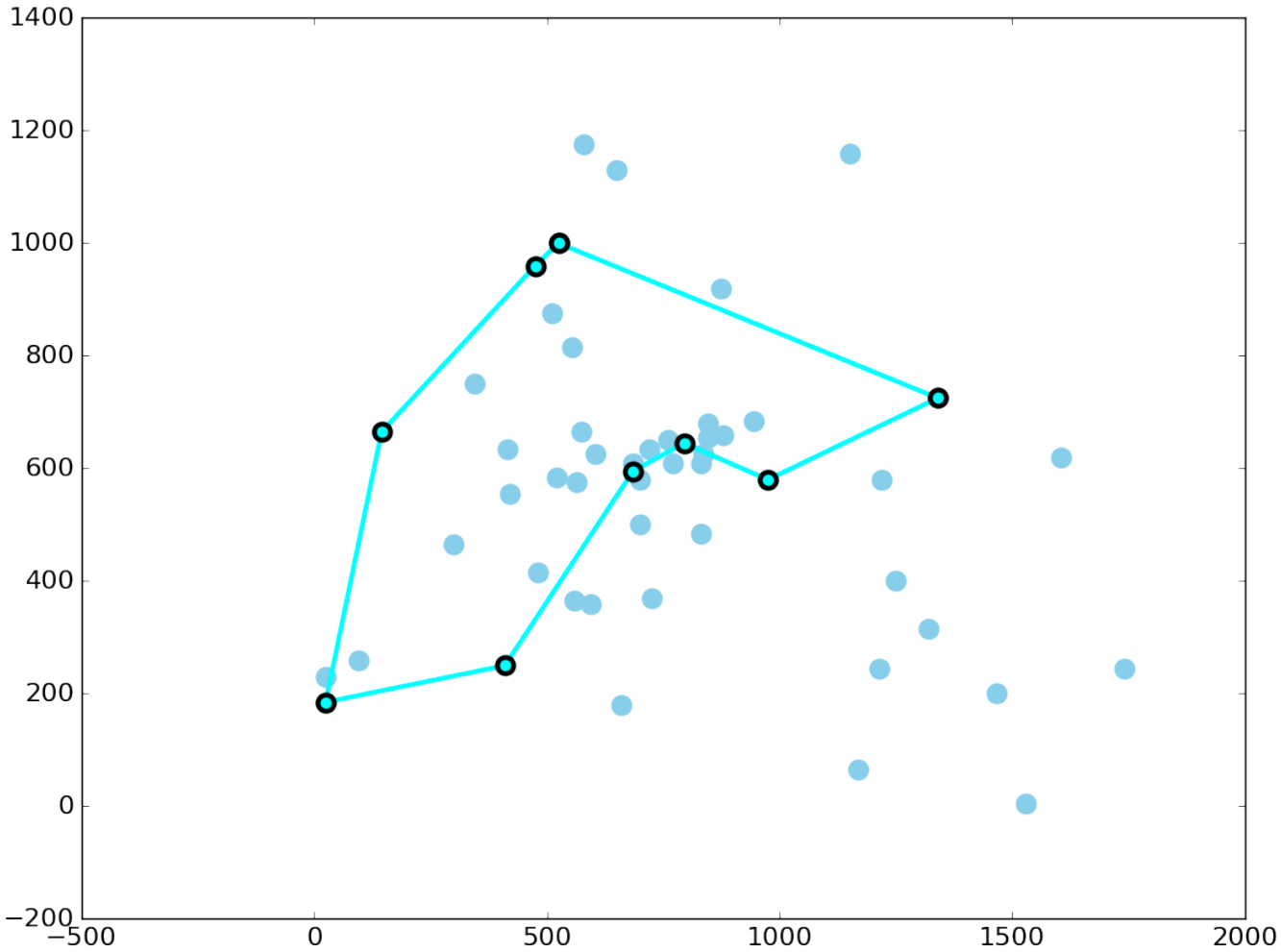
animation of the quick tour algorithm



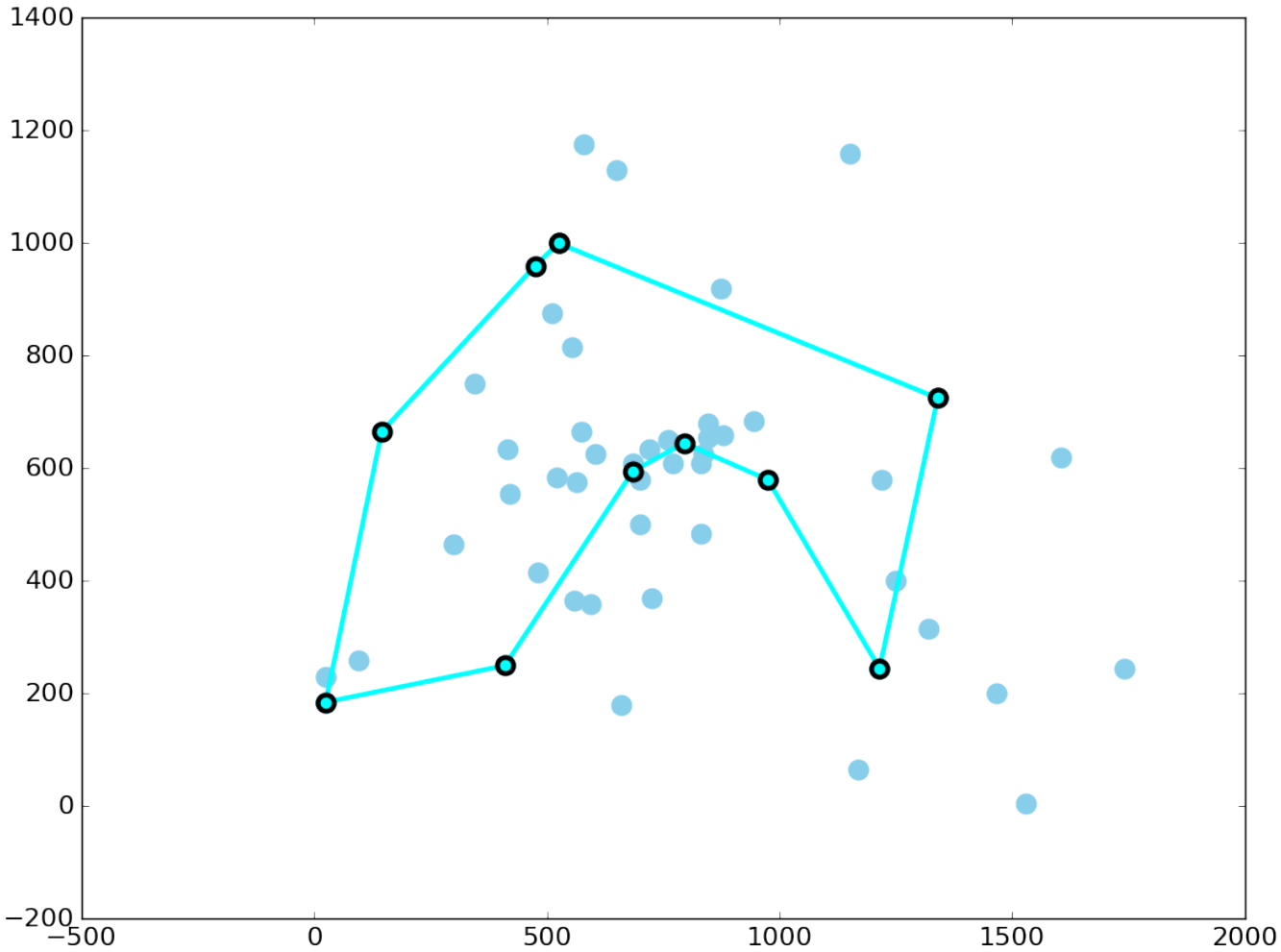
animation of the quick tour algorithm



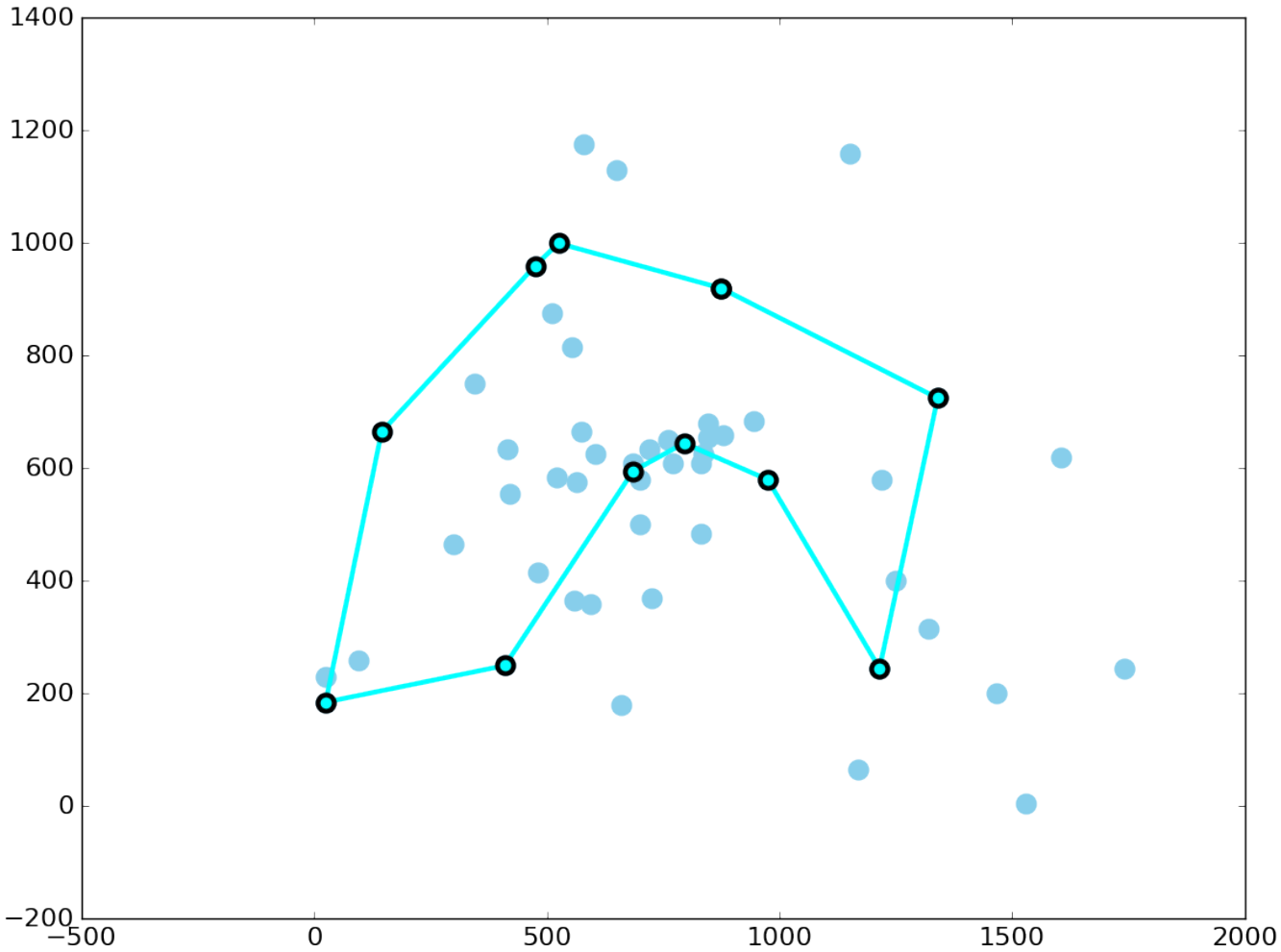
animation of the quick tour algorithm



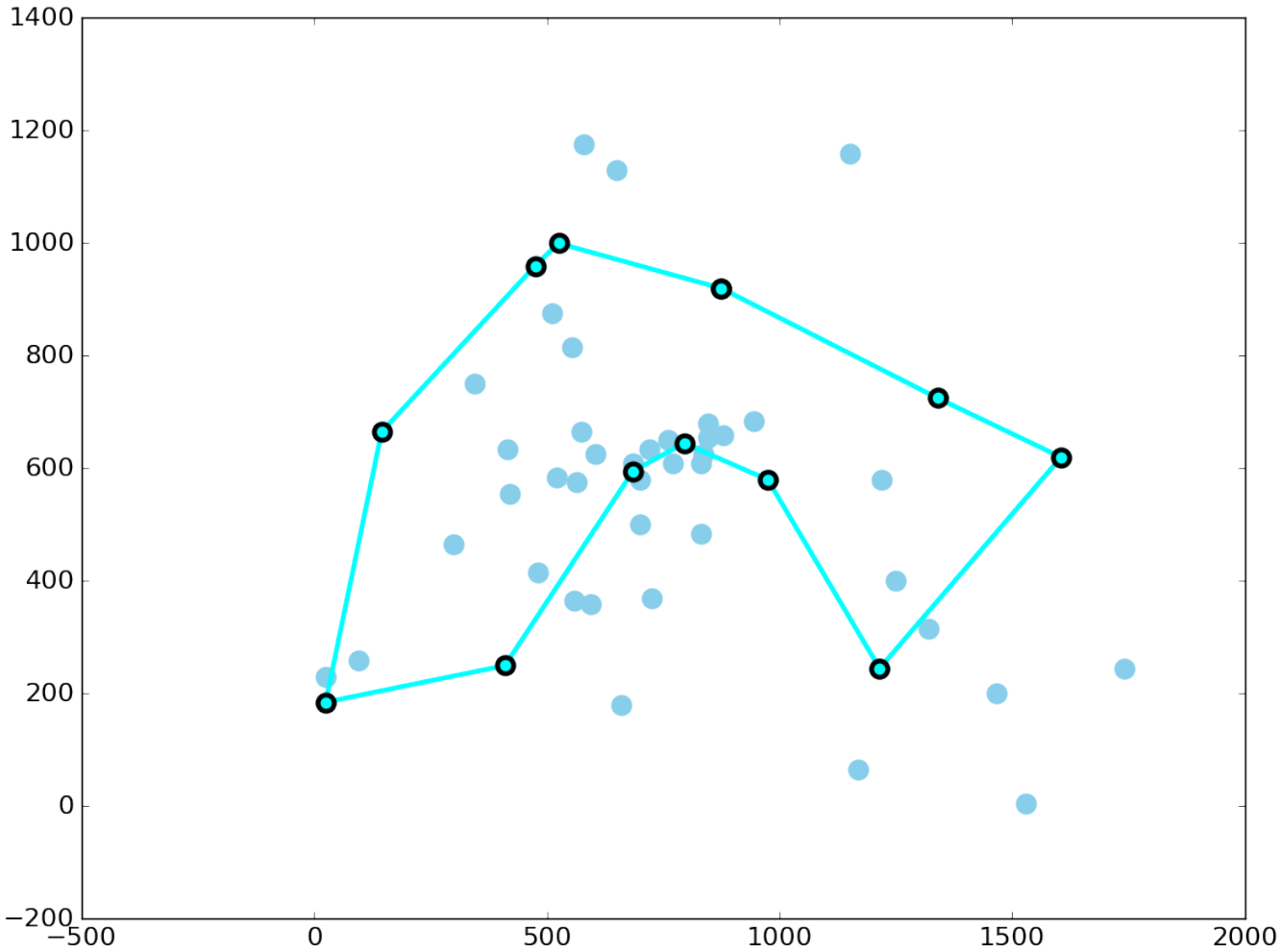
animation of the quick tour algorithm



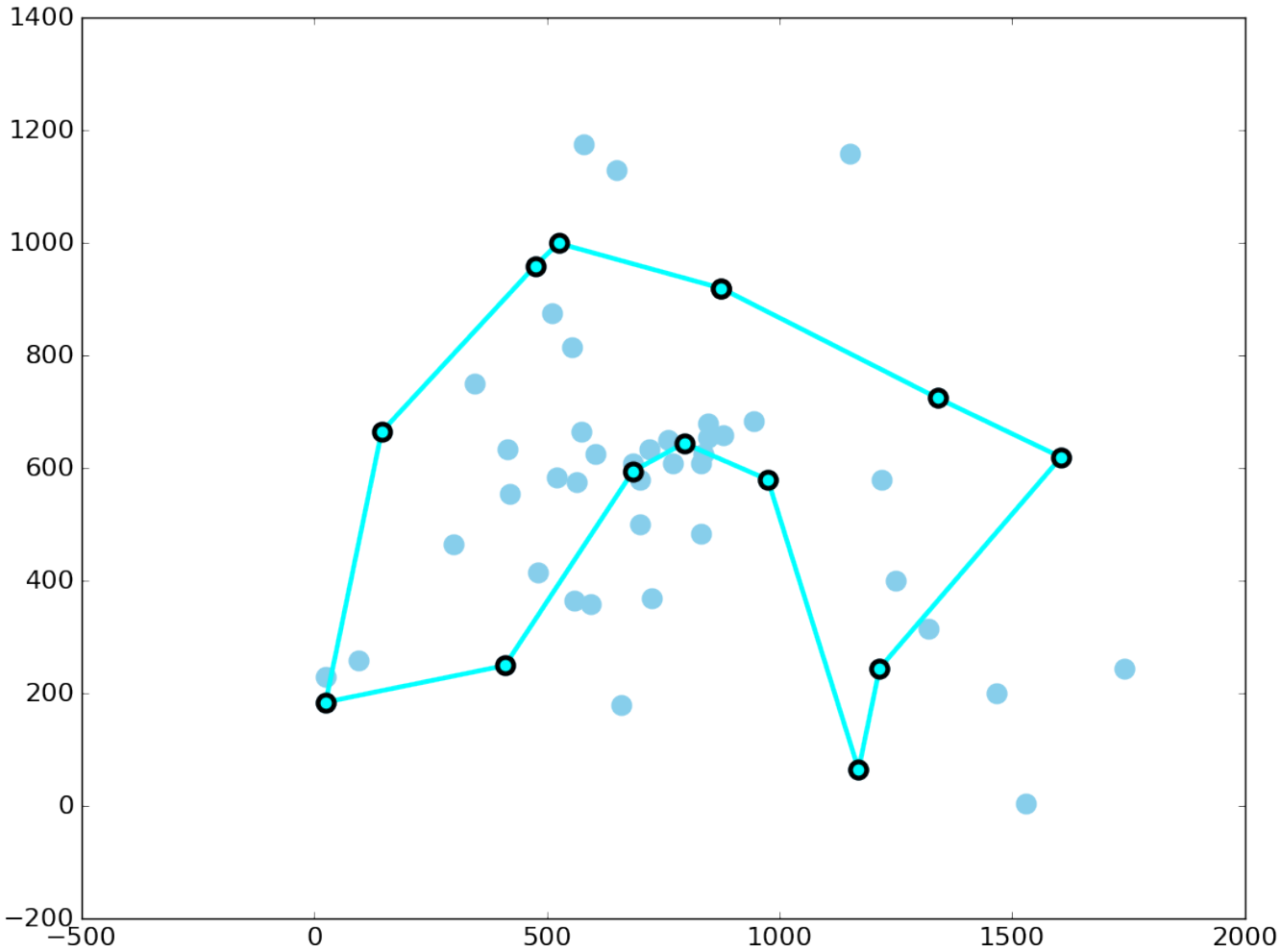
animation of the quick tour algorithm



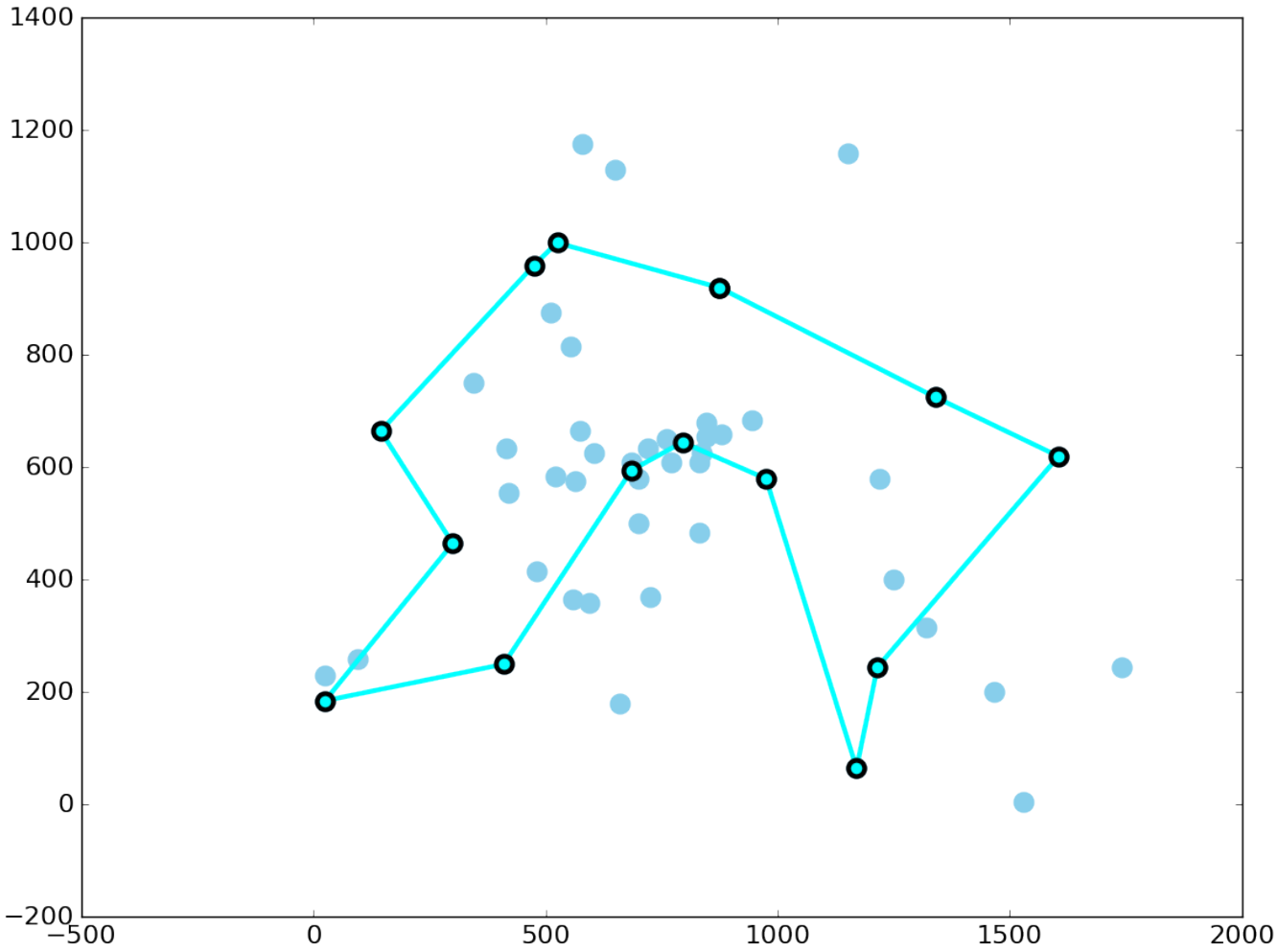
animation of the quick tour algorithm



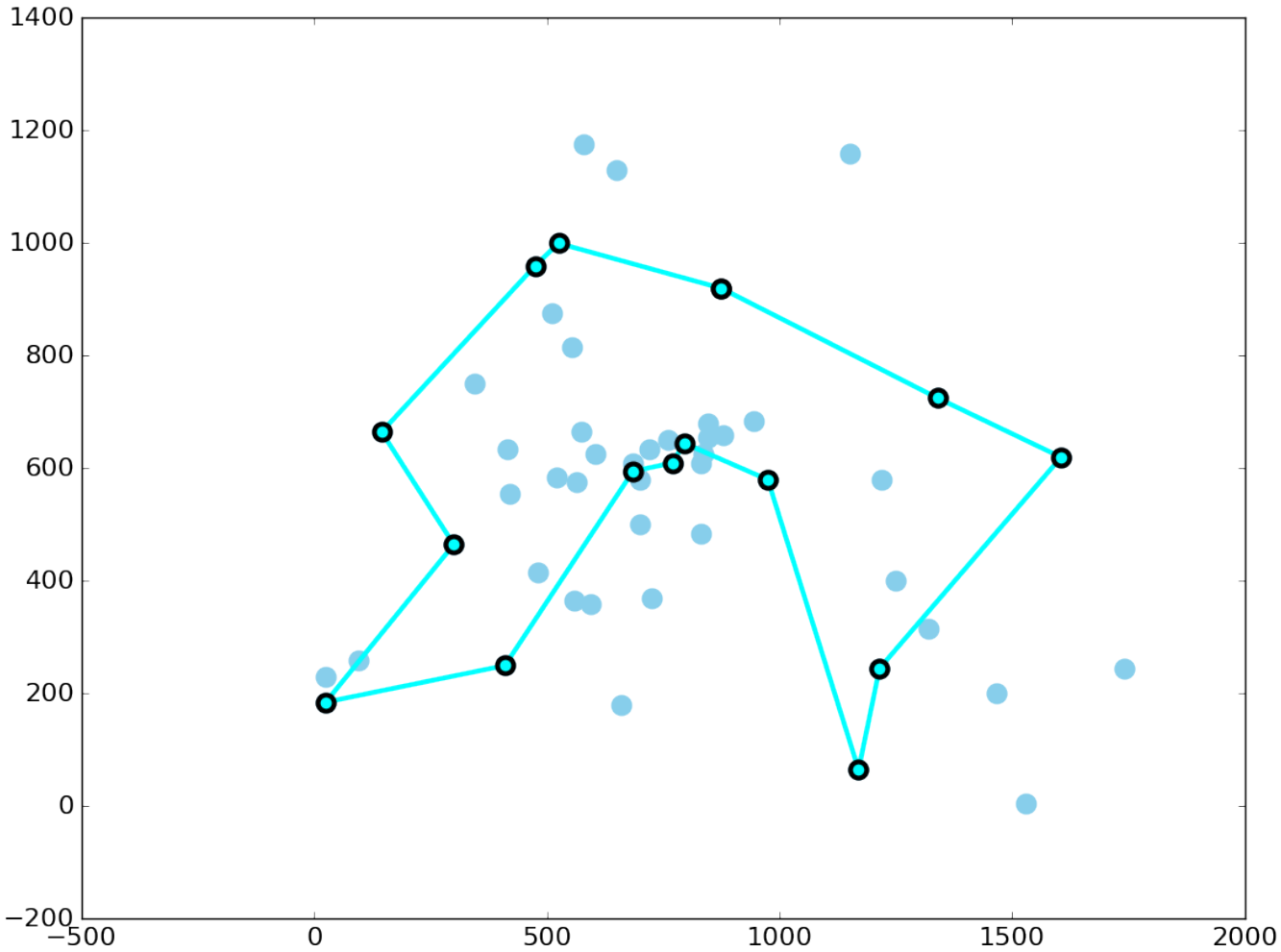
animation of the quick tour algorithm



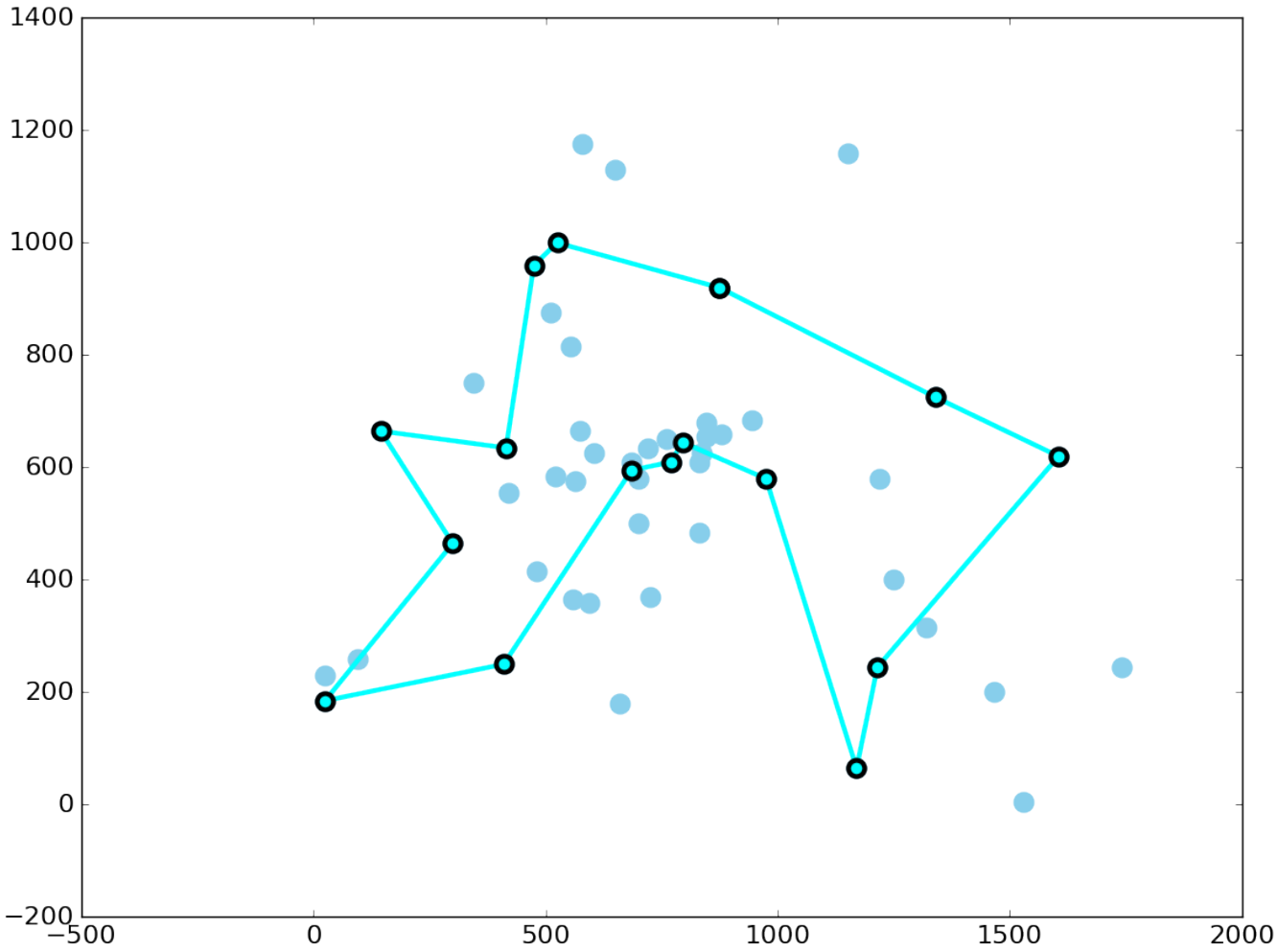
animation of the quick tour algorithm



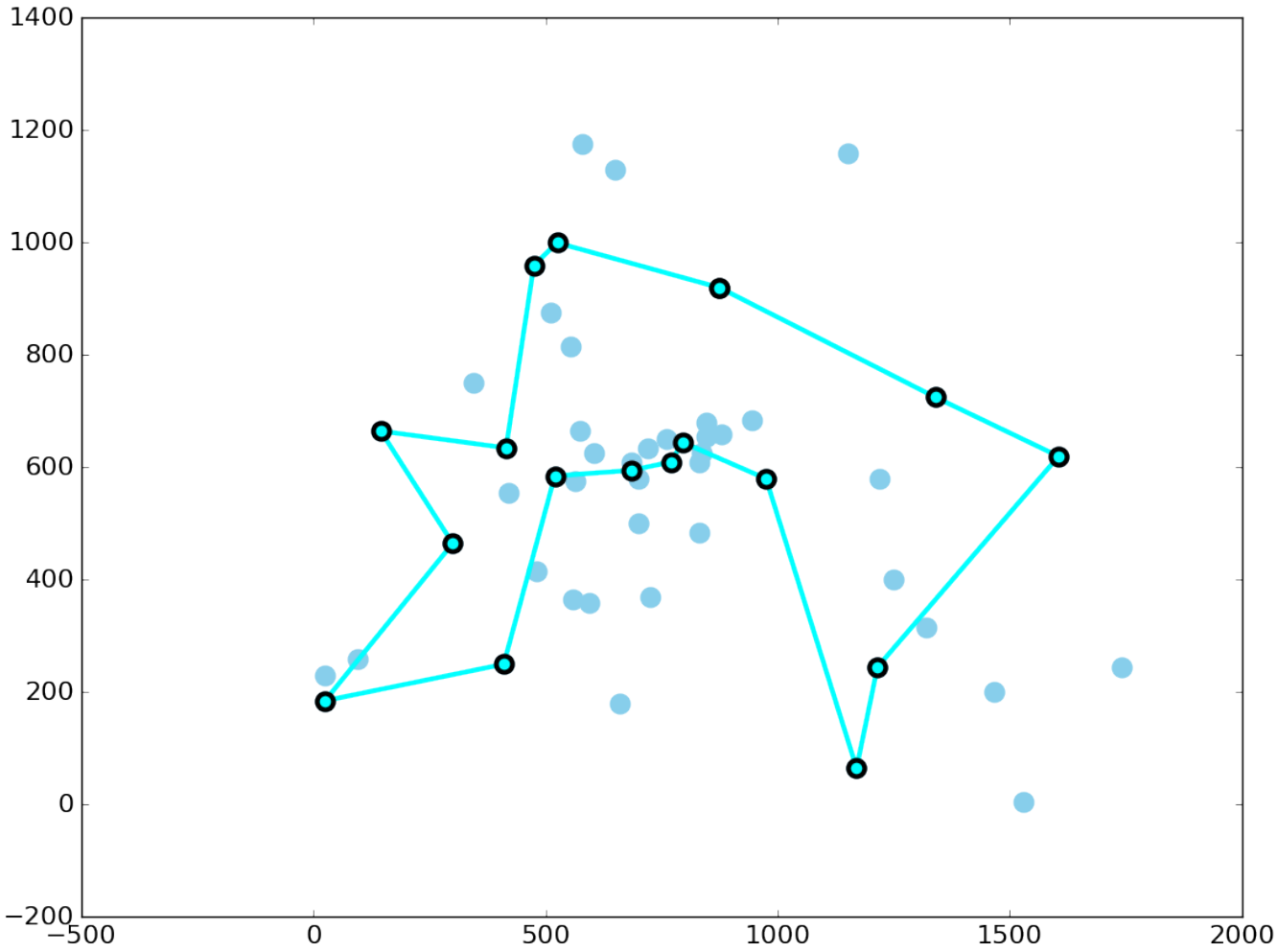
animation of the quick tour algorithm



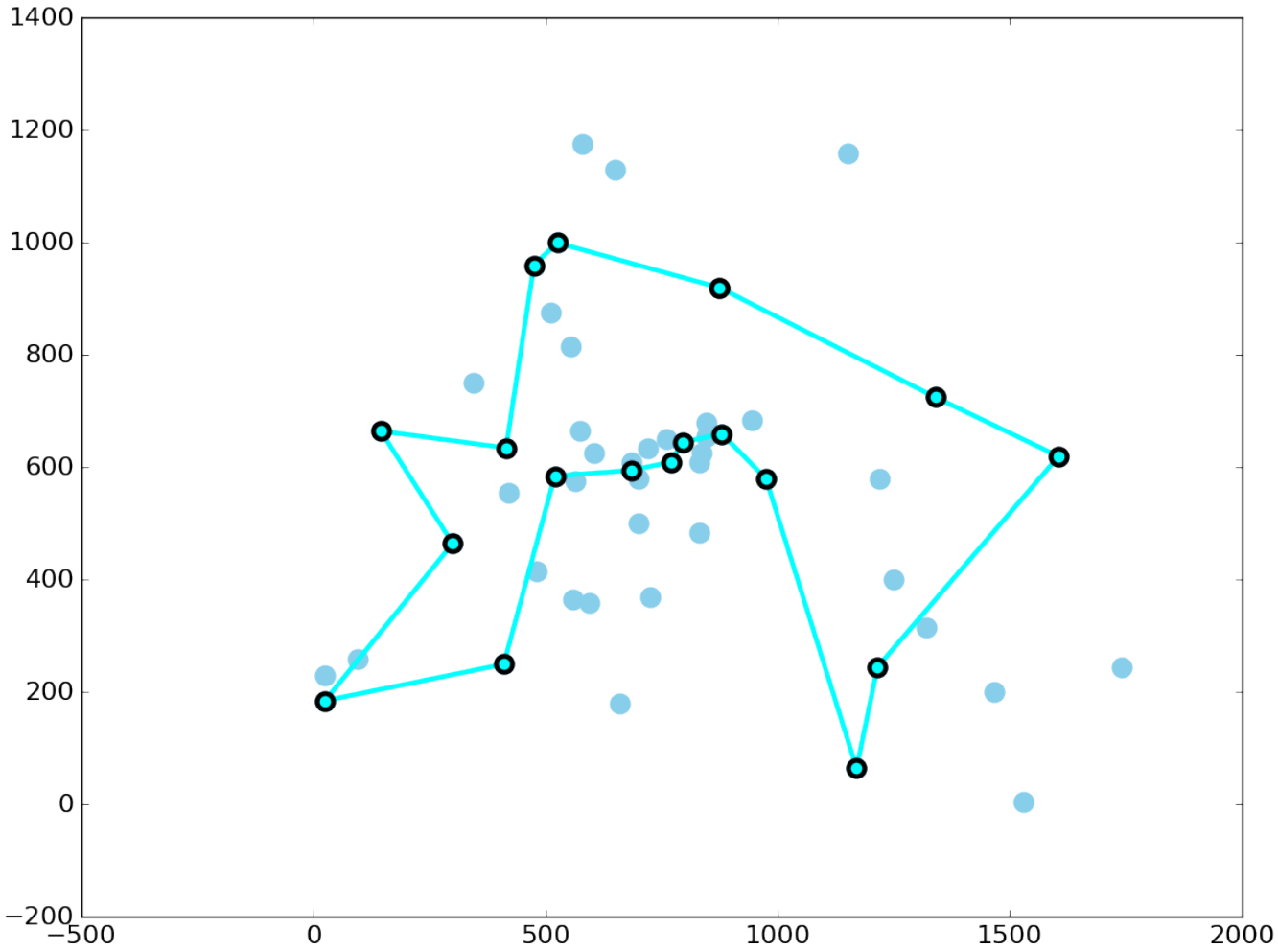
animation of the quick tour algorithm



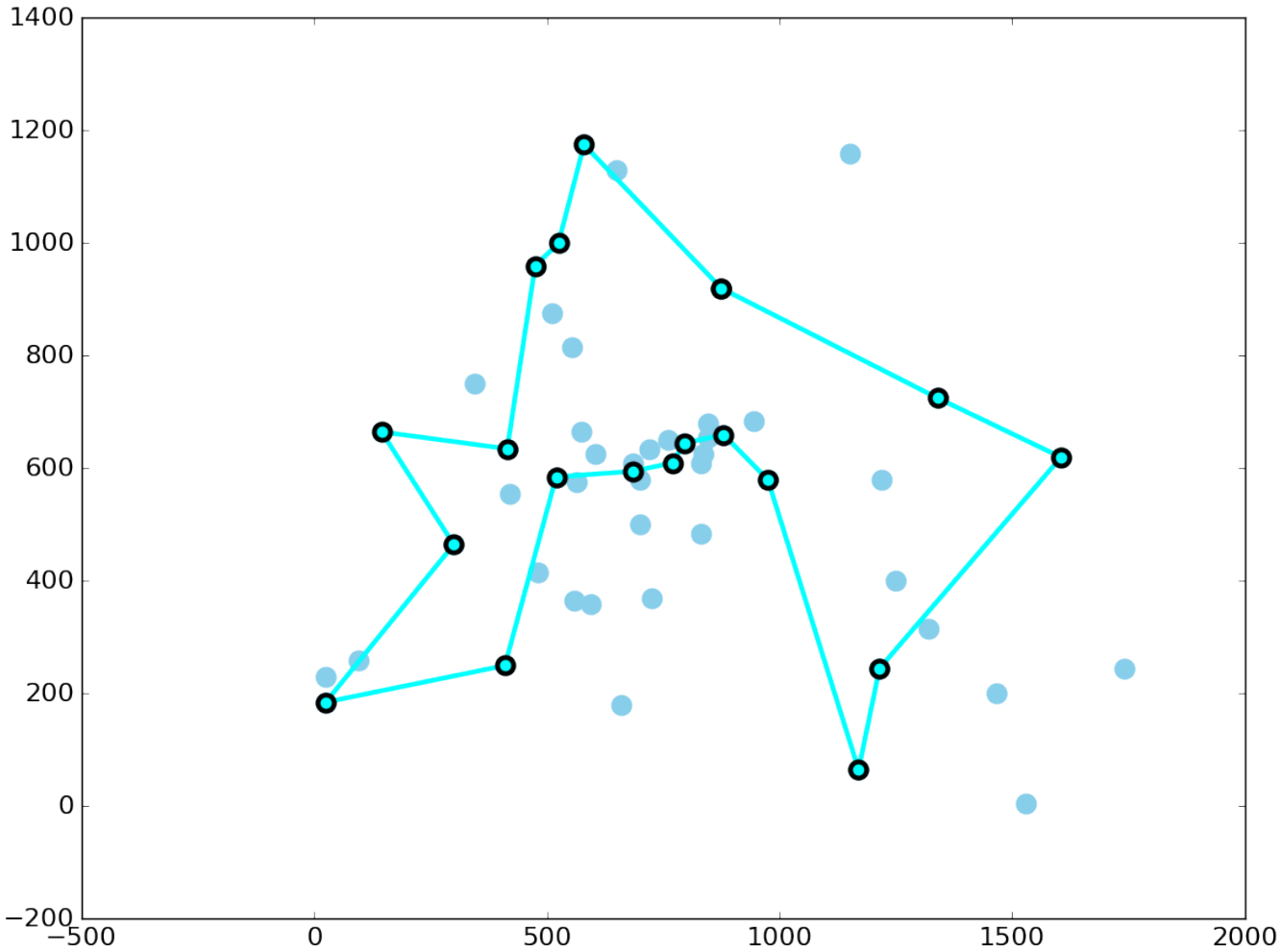
animation of the quick tour algorithm



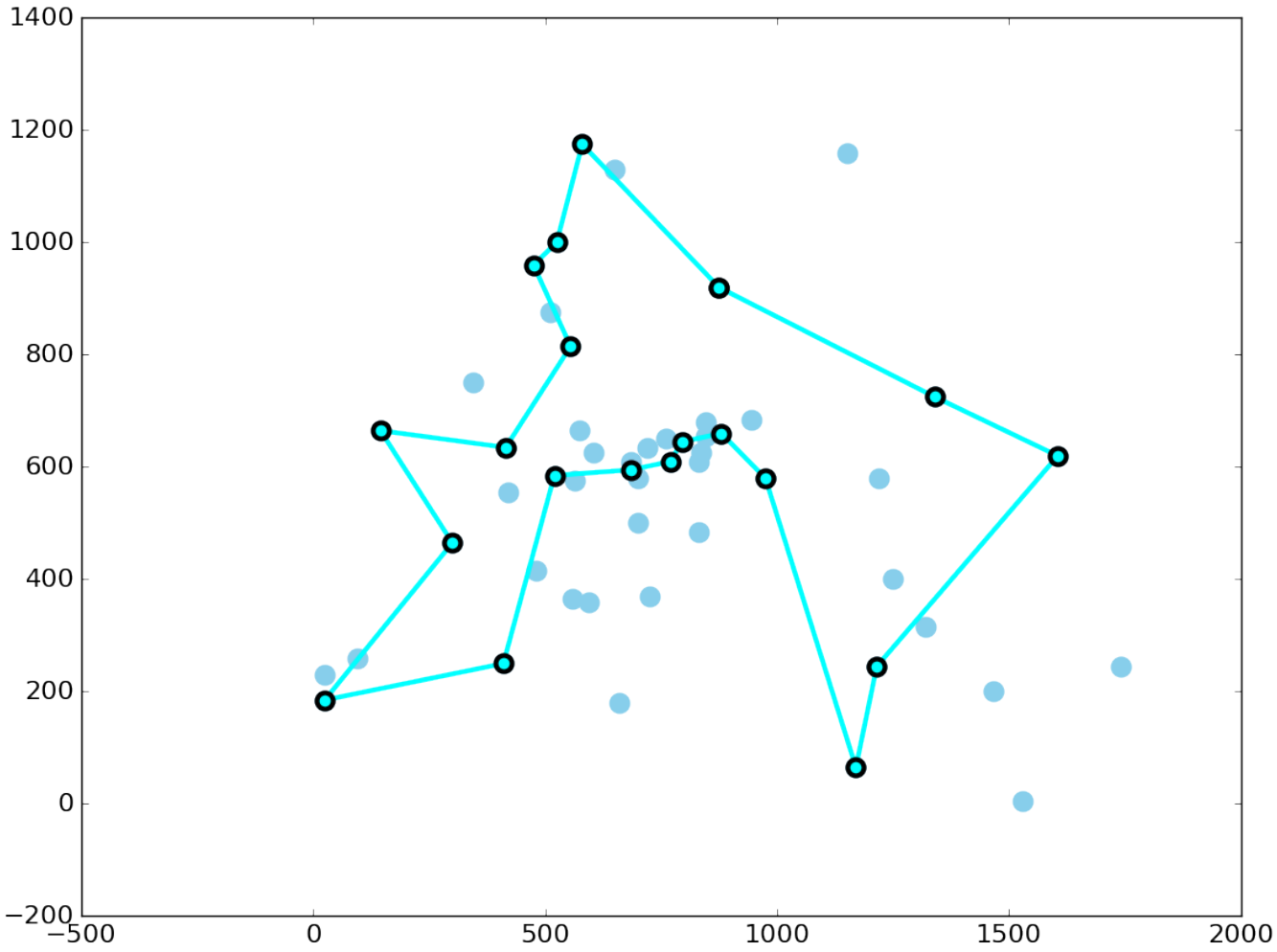
animation of the quick tour algorithm



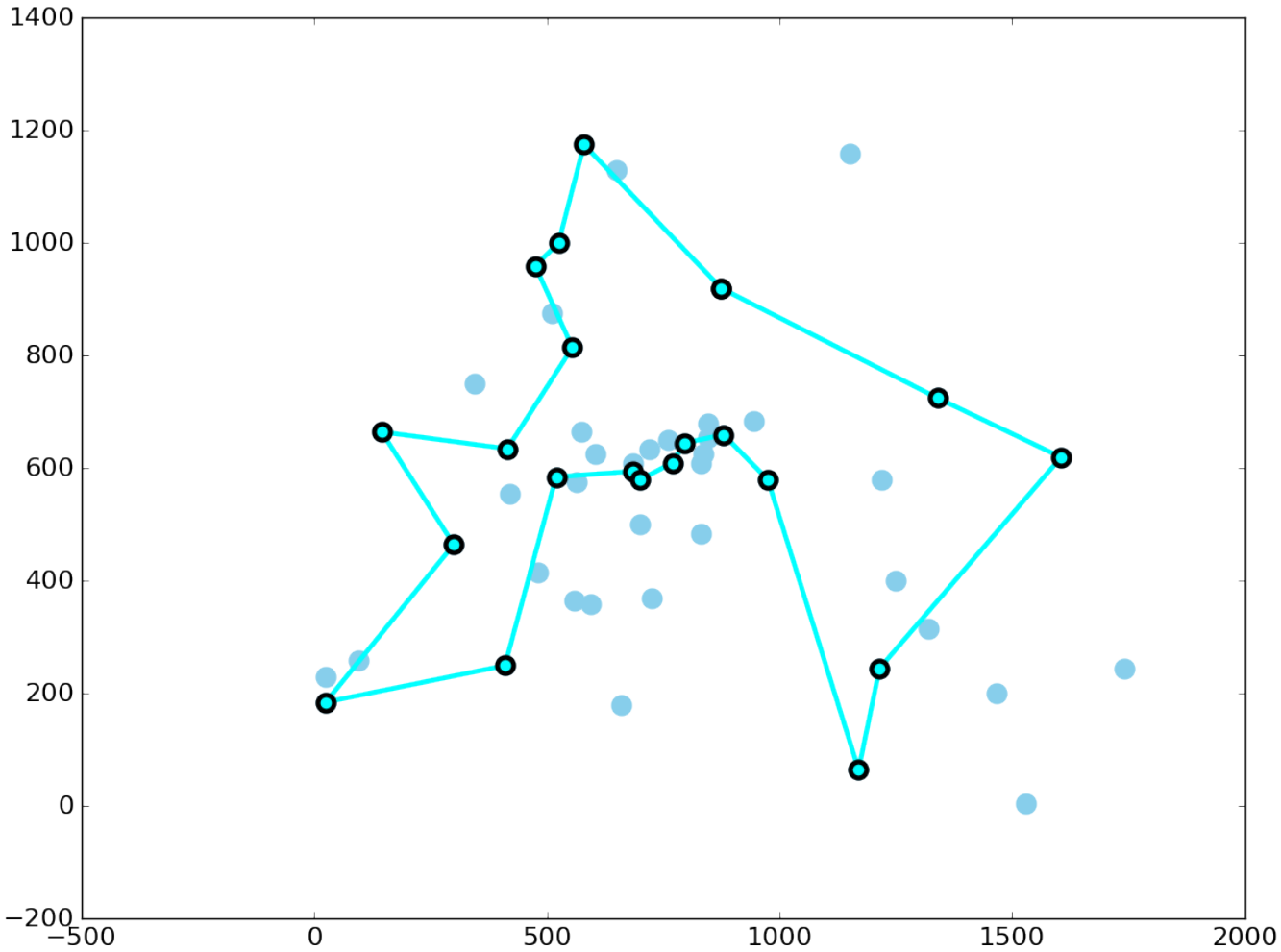
animation of the quick tour algorithm



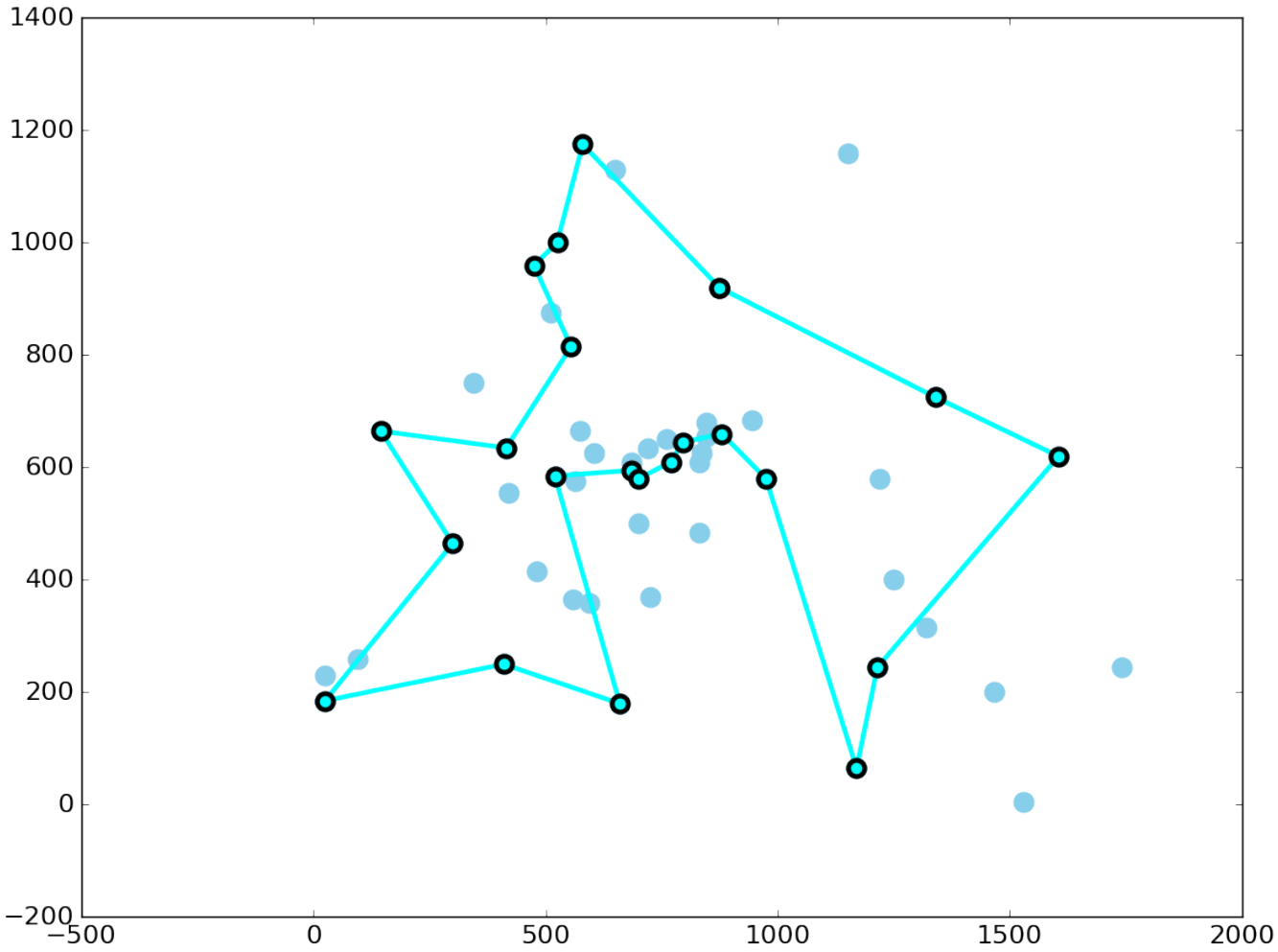
animation of the quick tour algorithm



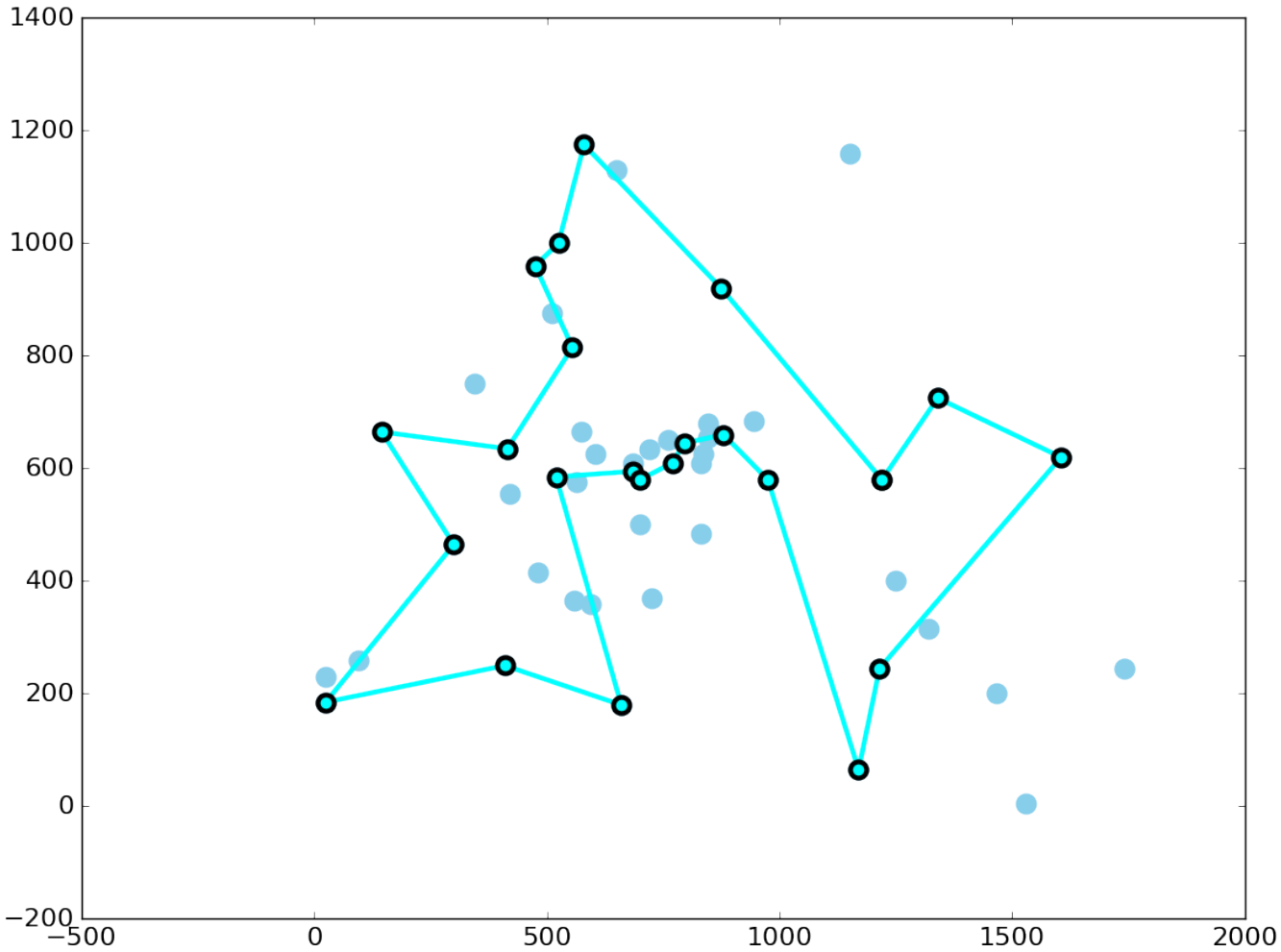
animation of the quick tour algorithm



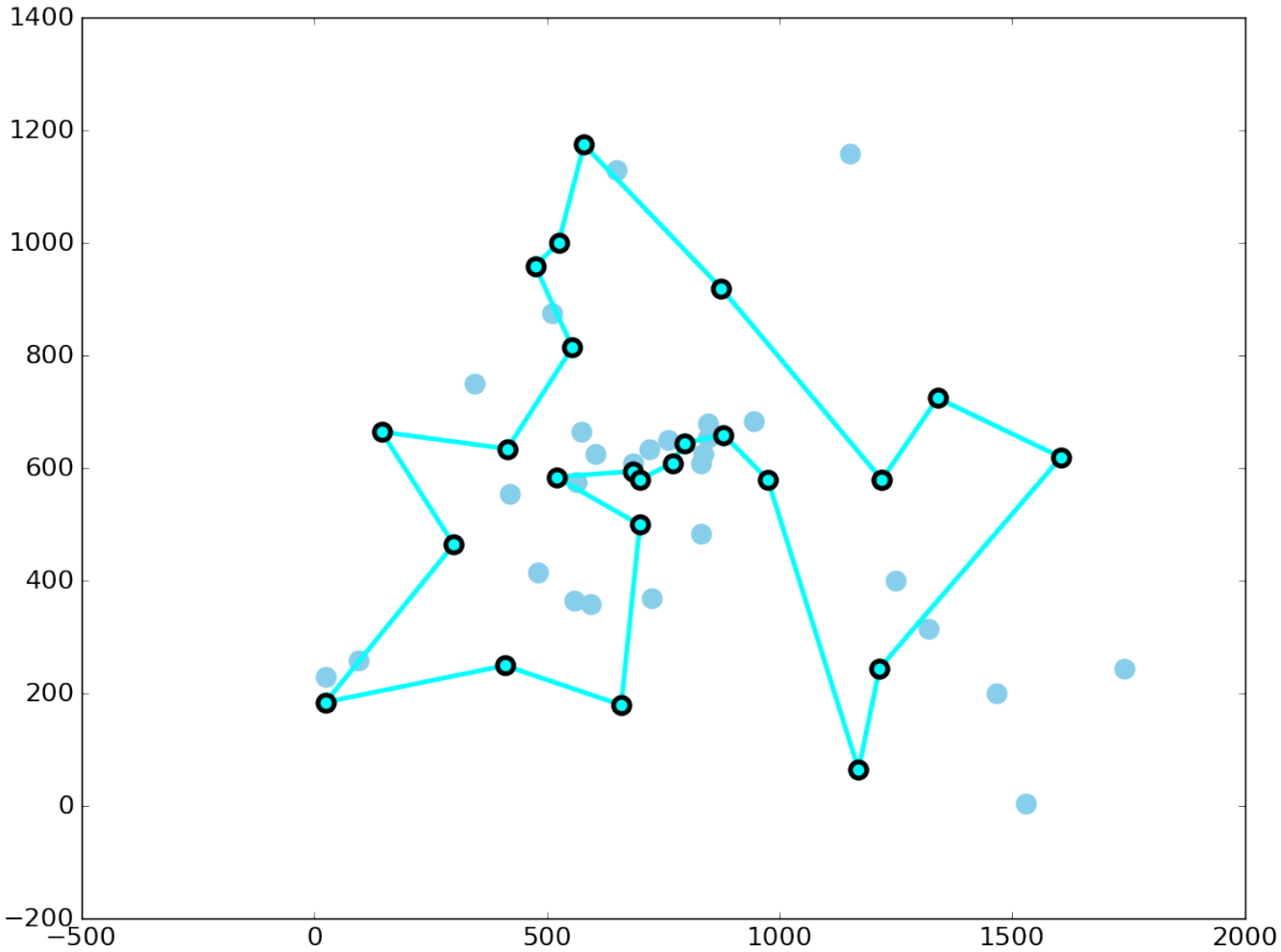
animation of the quick tour algorithm



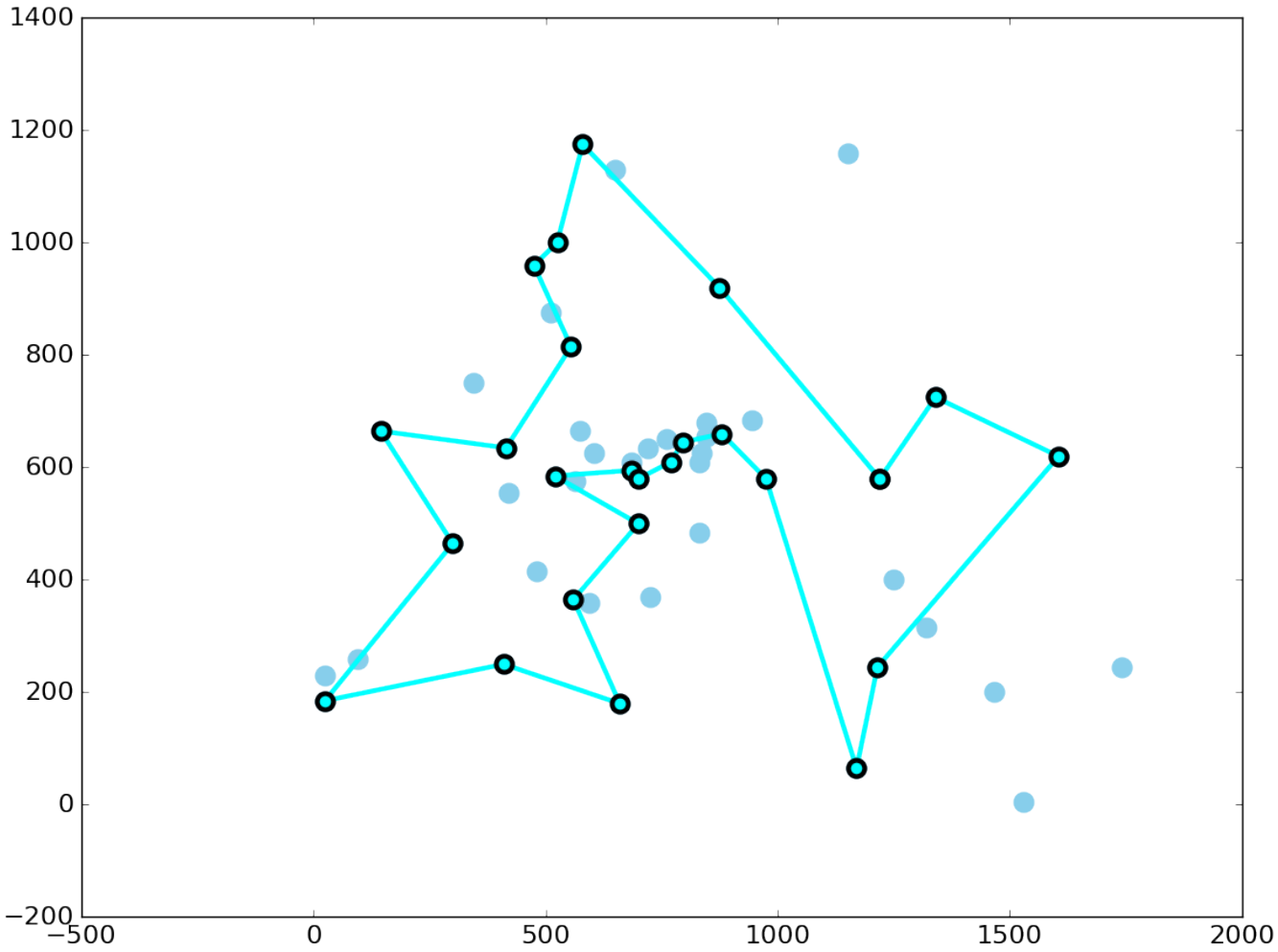
animation of the quick tour algorithm



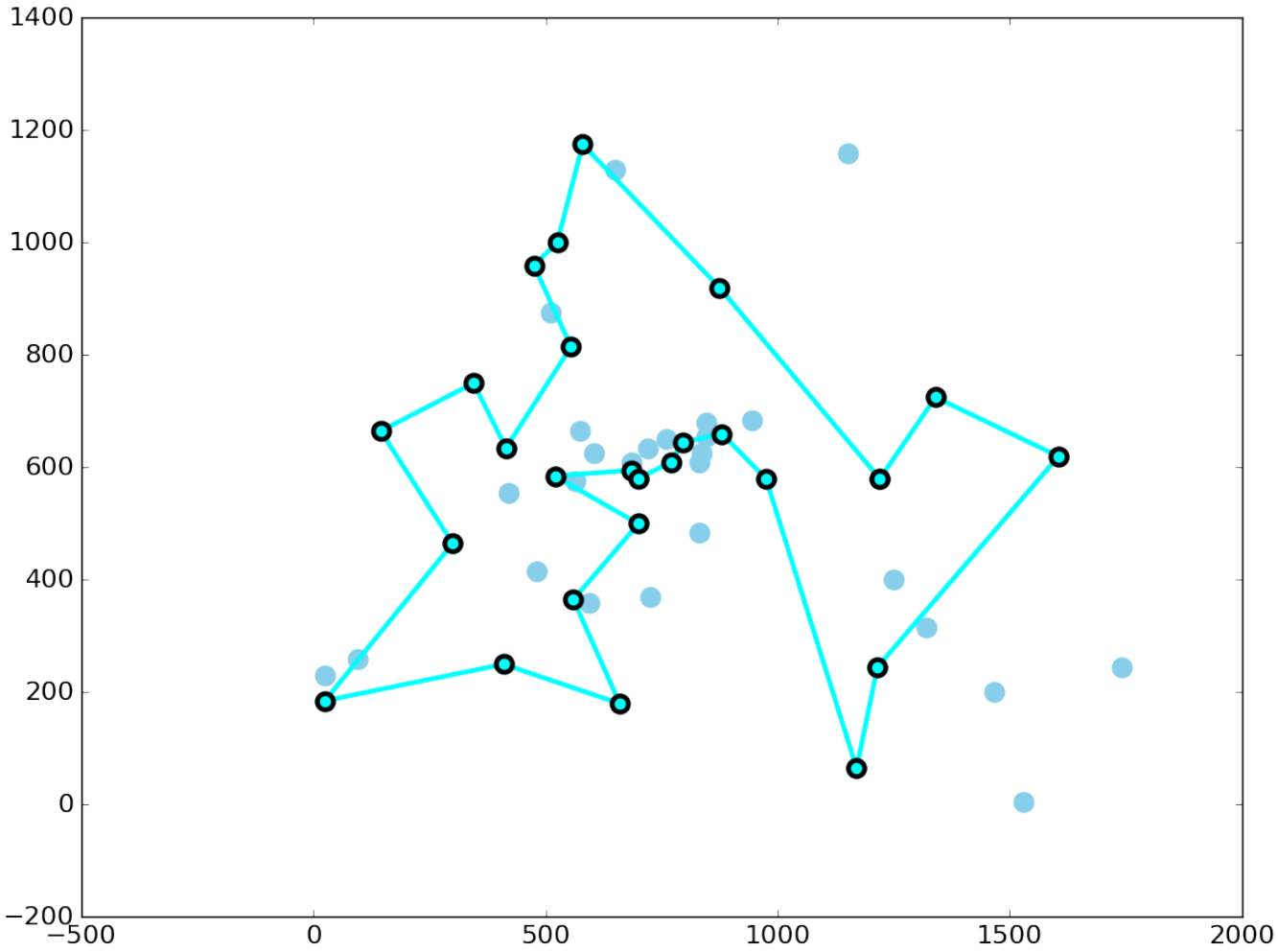
animation of the quick tour algorithm



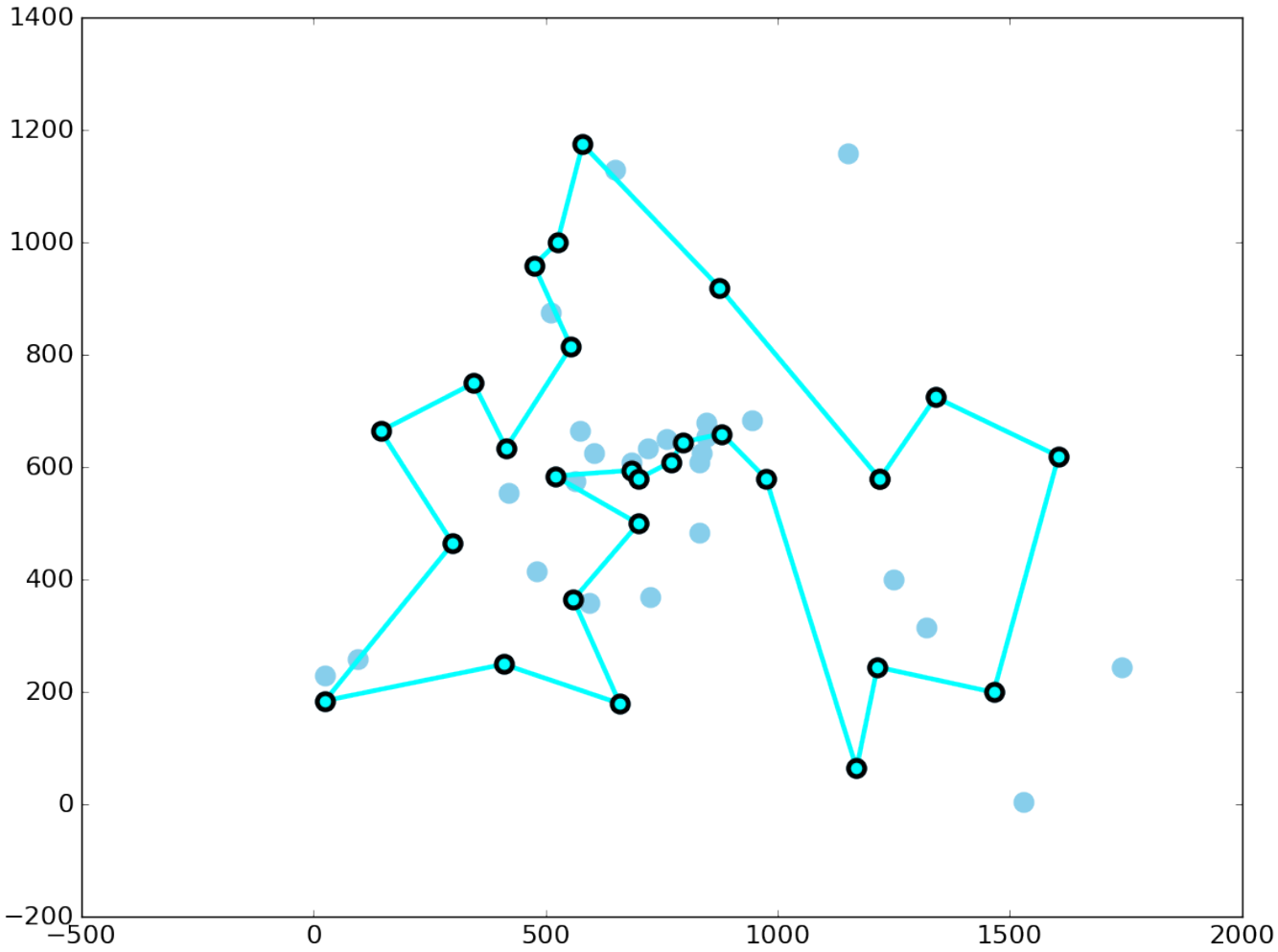
animation of the quick tour algorithm



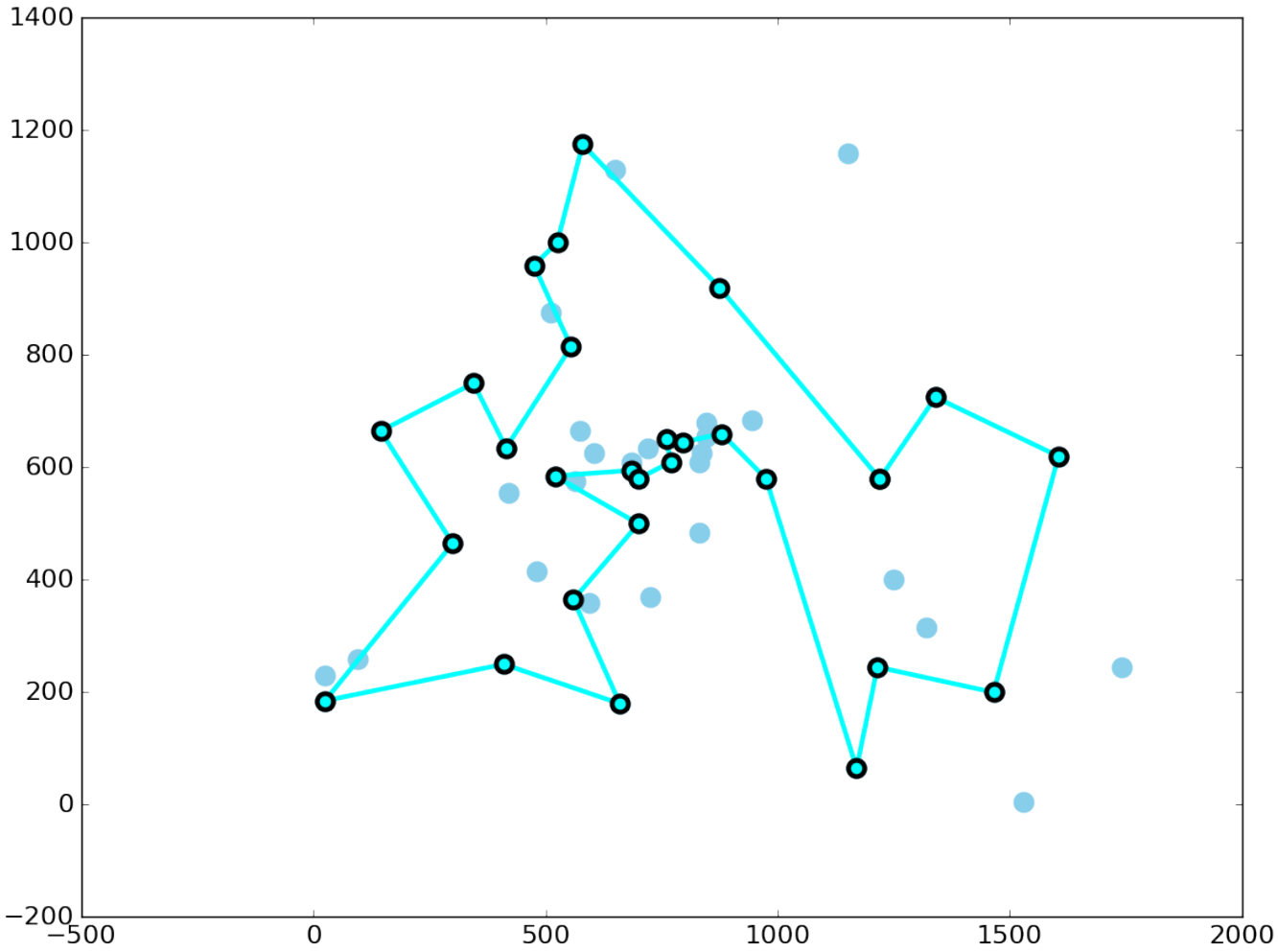
animation of the quick tour algorithm



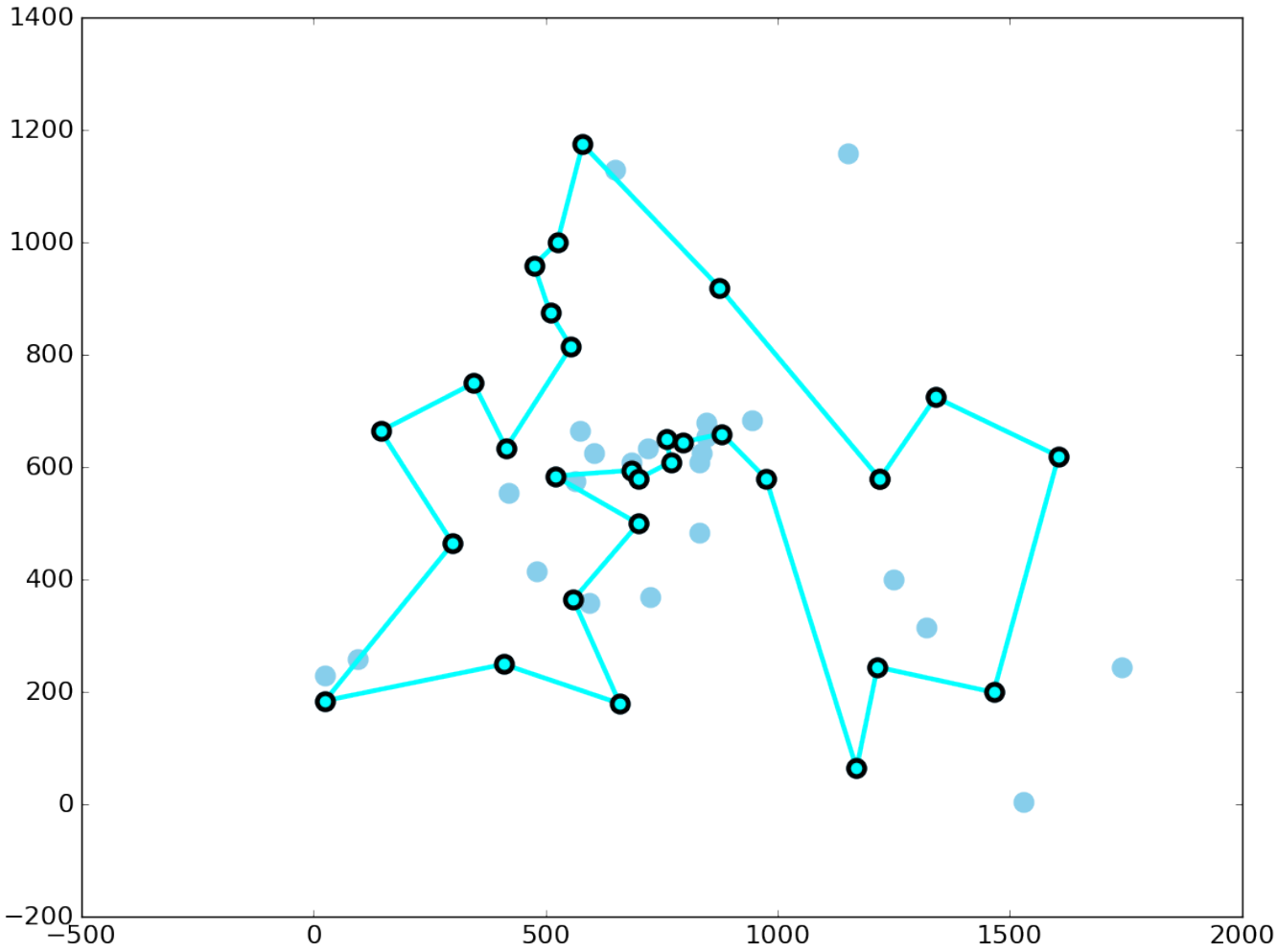
animation of the quick tour algorithm



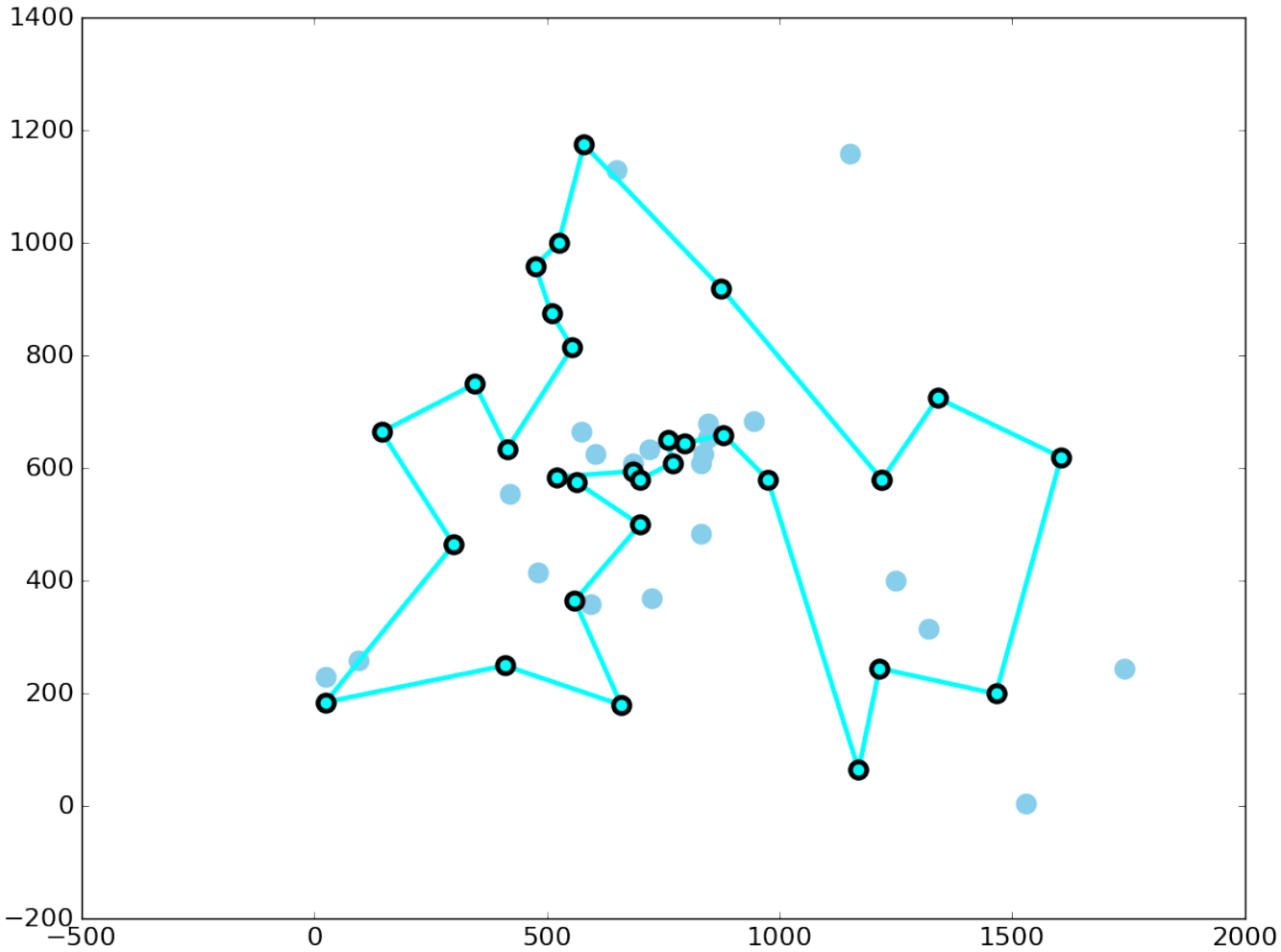
animation of the quick tour algorithm



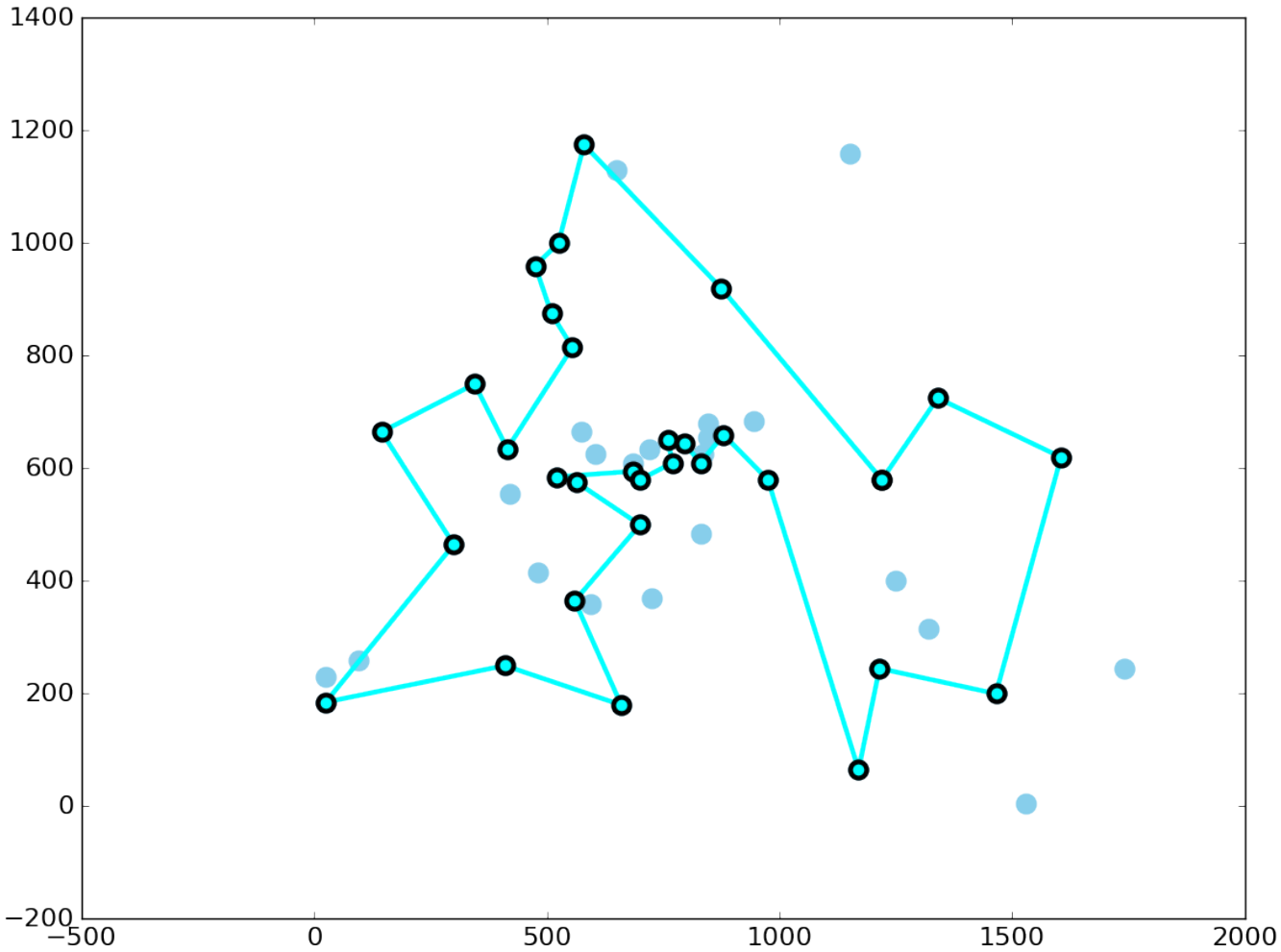
animation of the quick tour algorithm



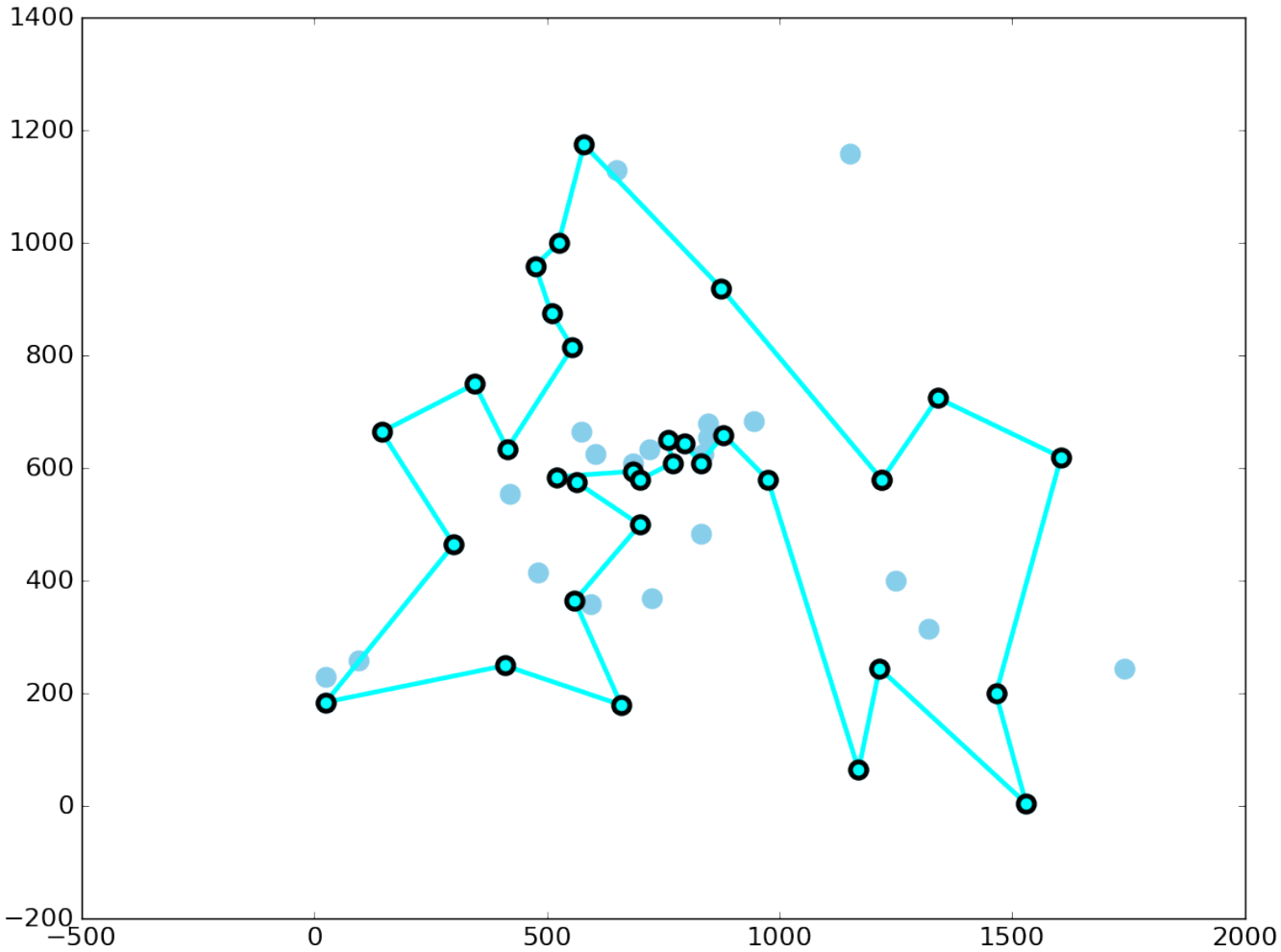
animation of the quick tour algorithm



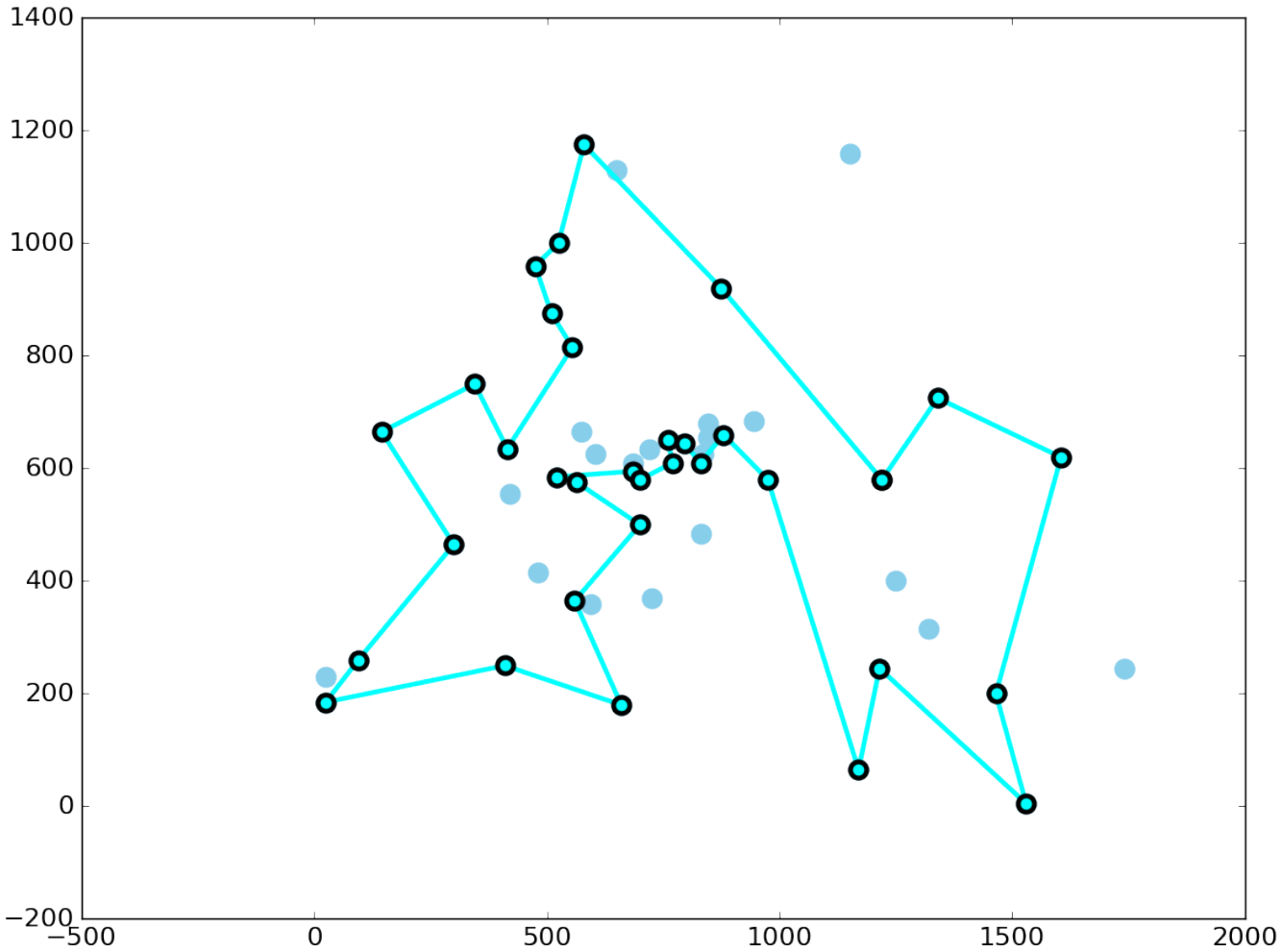
animation of the quick tour algorithm



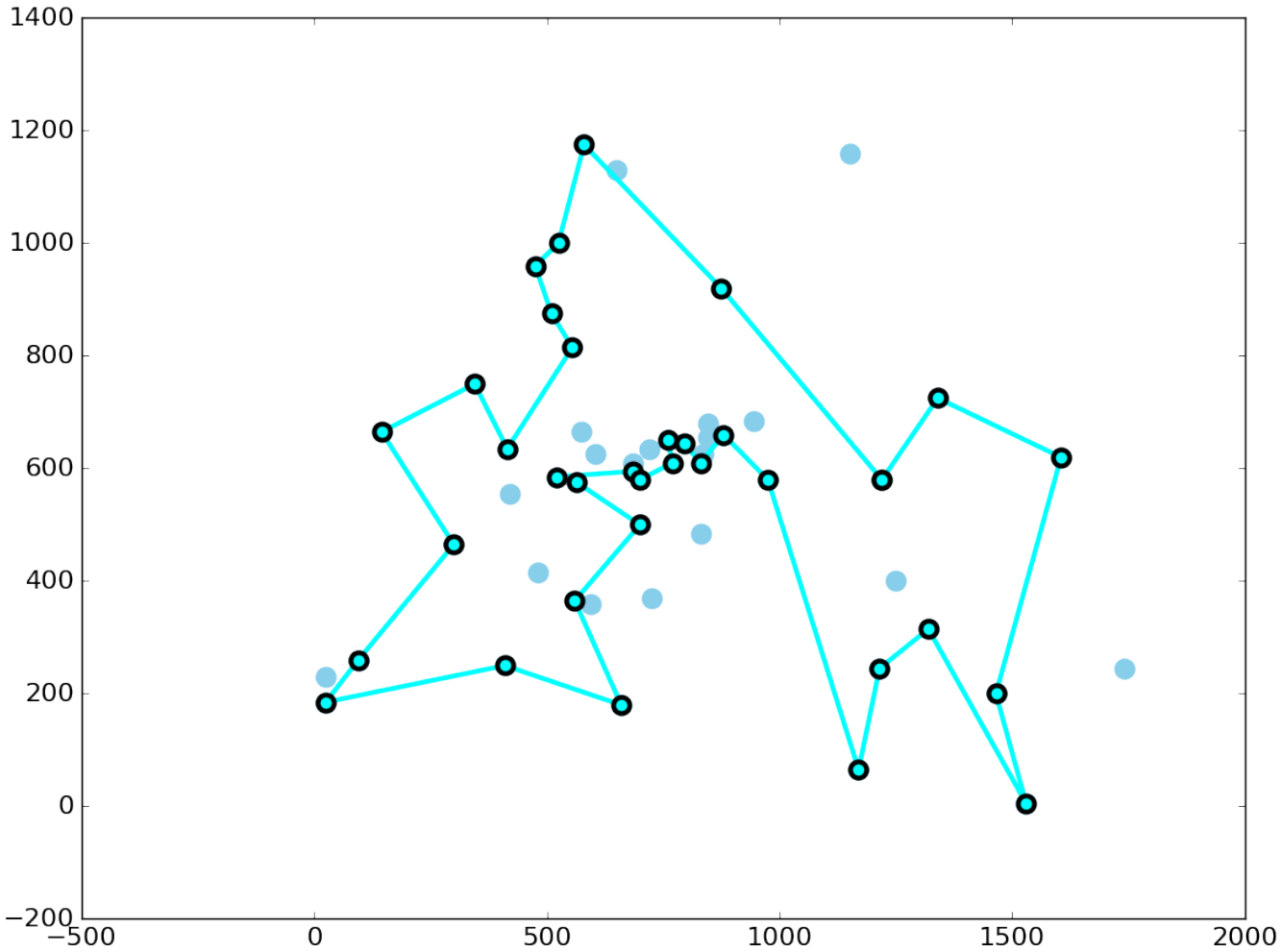
animation of the quick tour algorithm



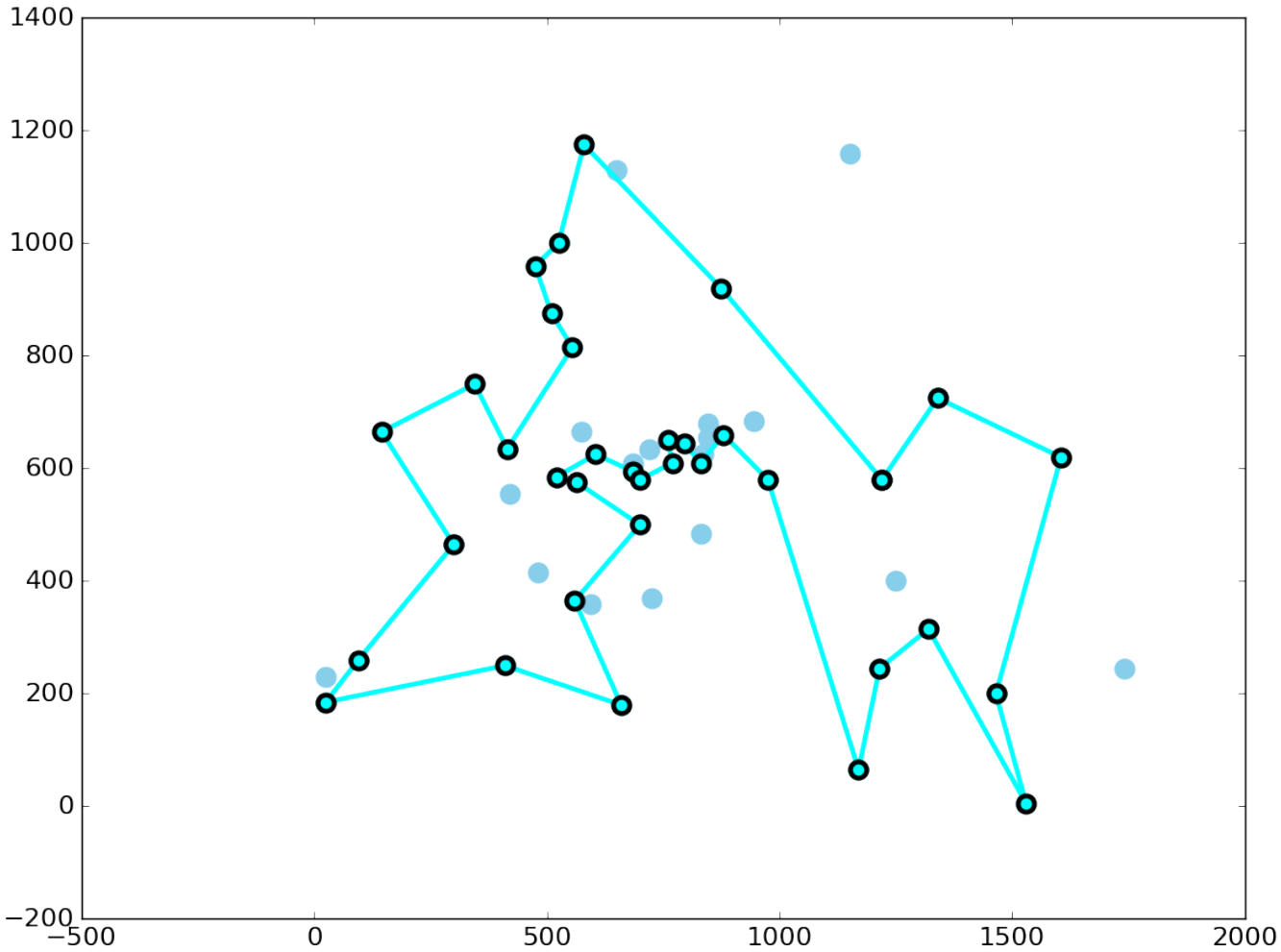
animation of the quick tour algorithm



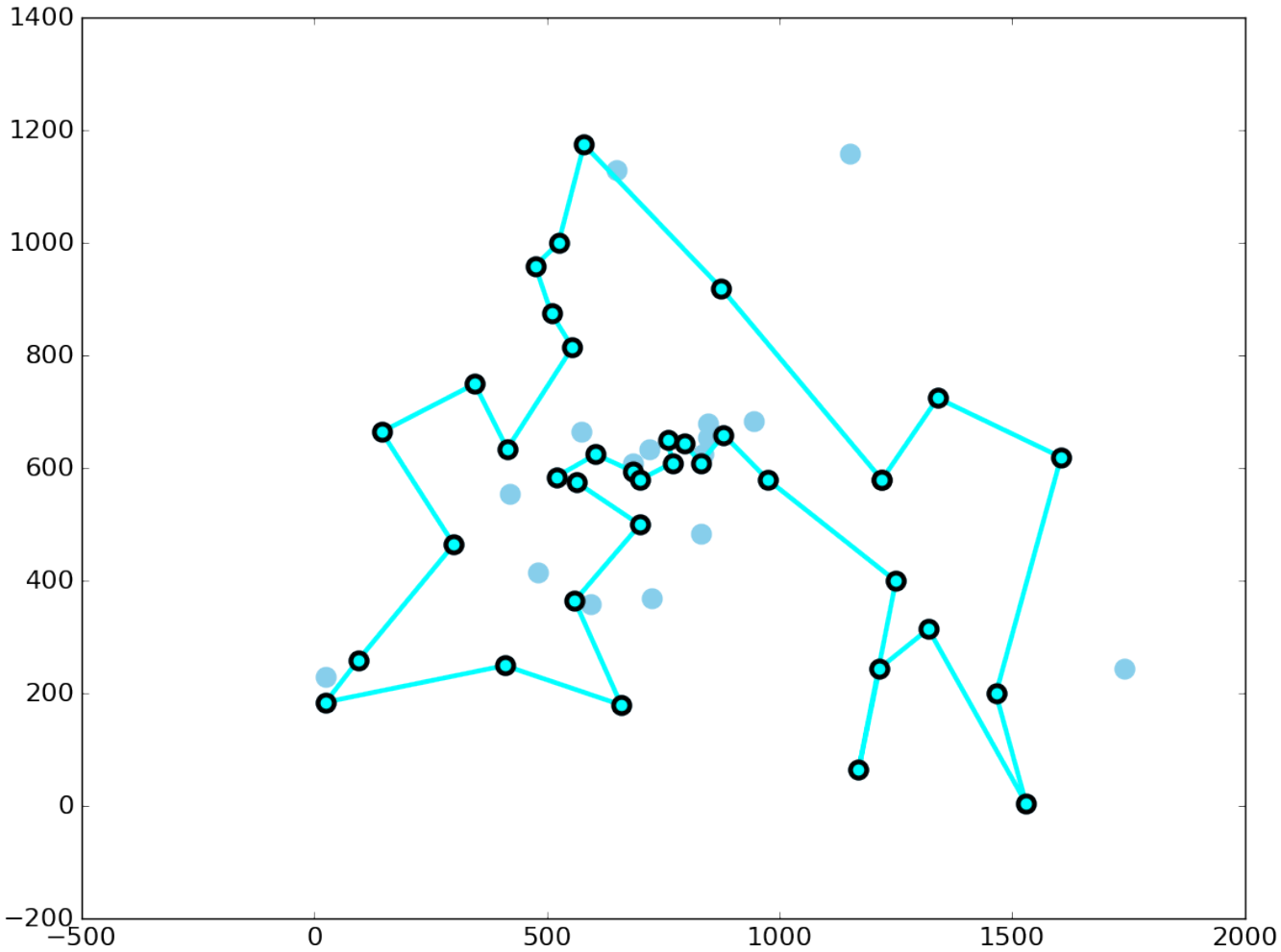
animation of the quick tour algorithm



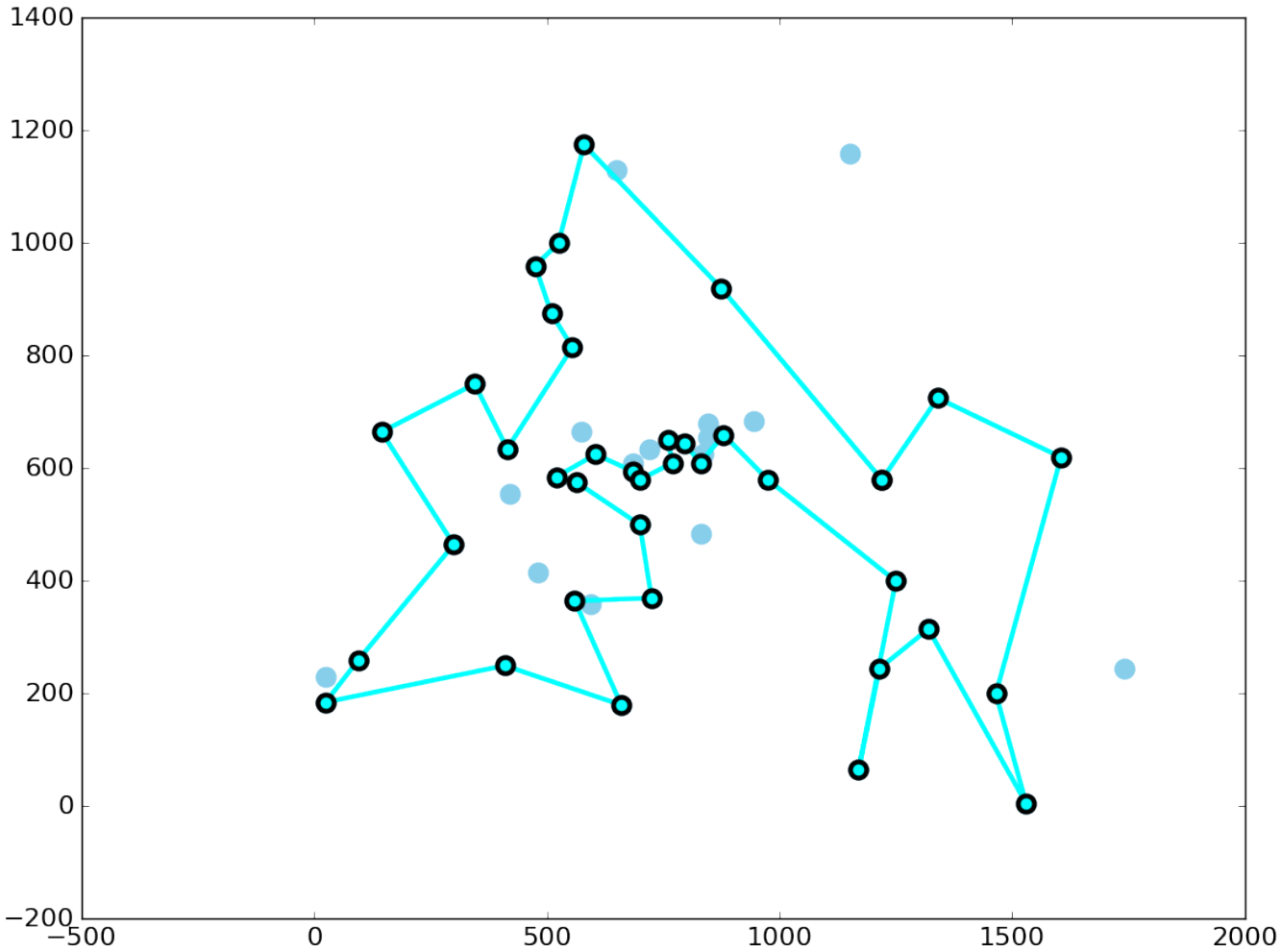
animation of the quick tour algorithm



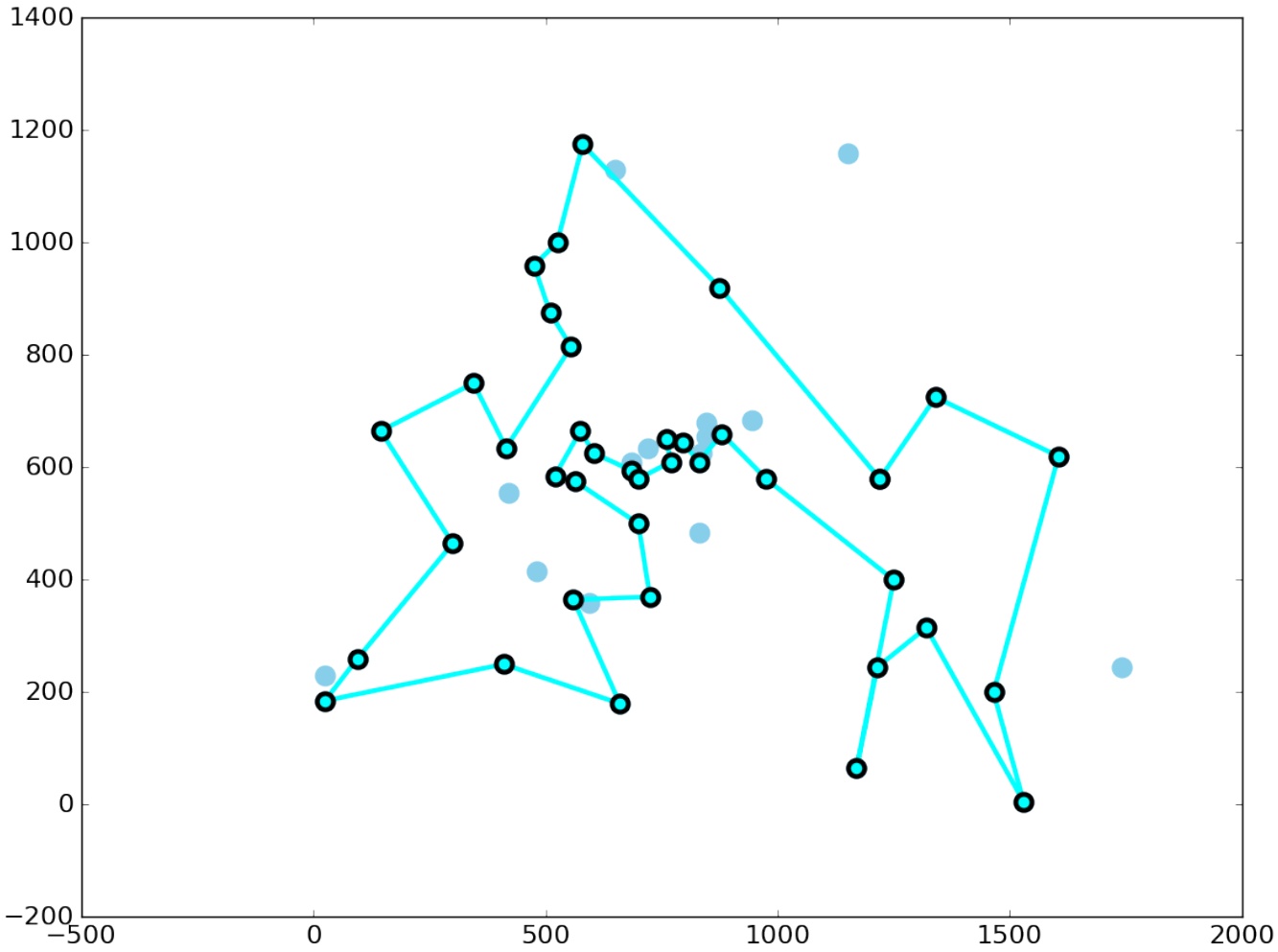
animation of the quick tour algorithm



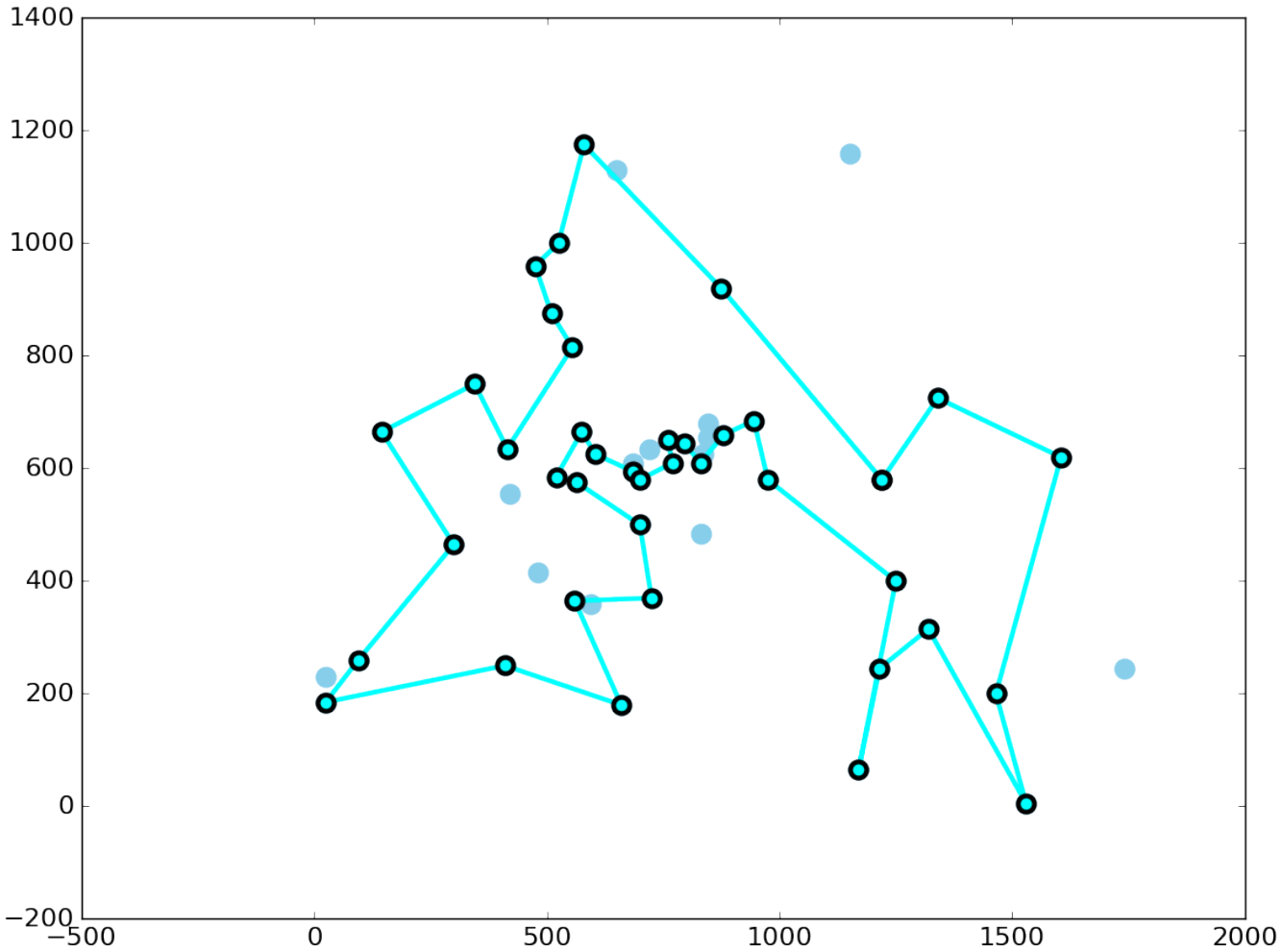
animation of the quick tour algorithm



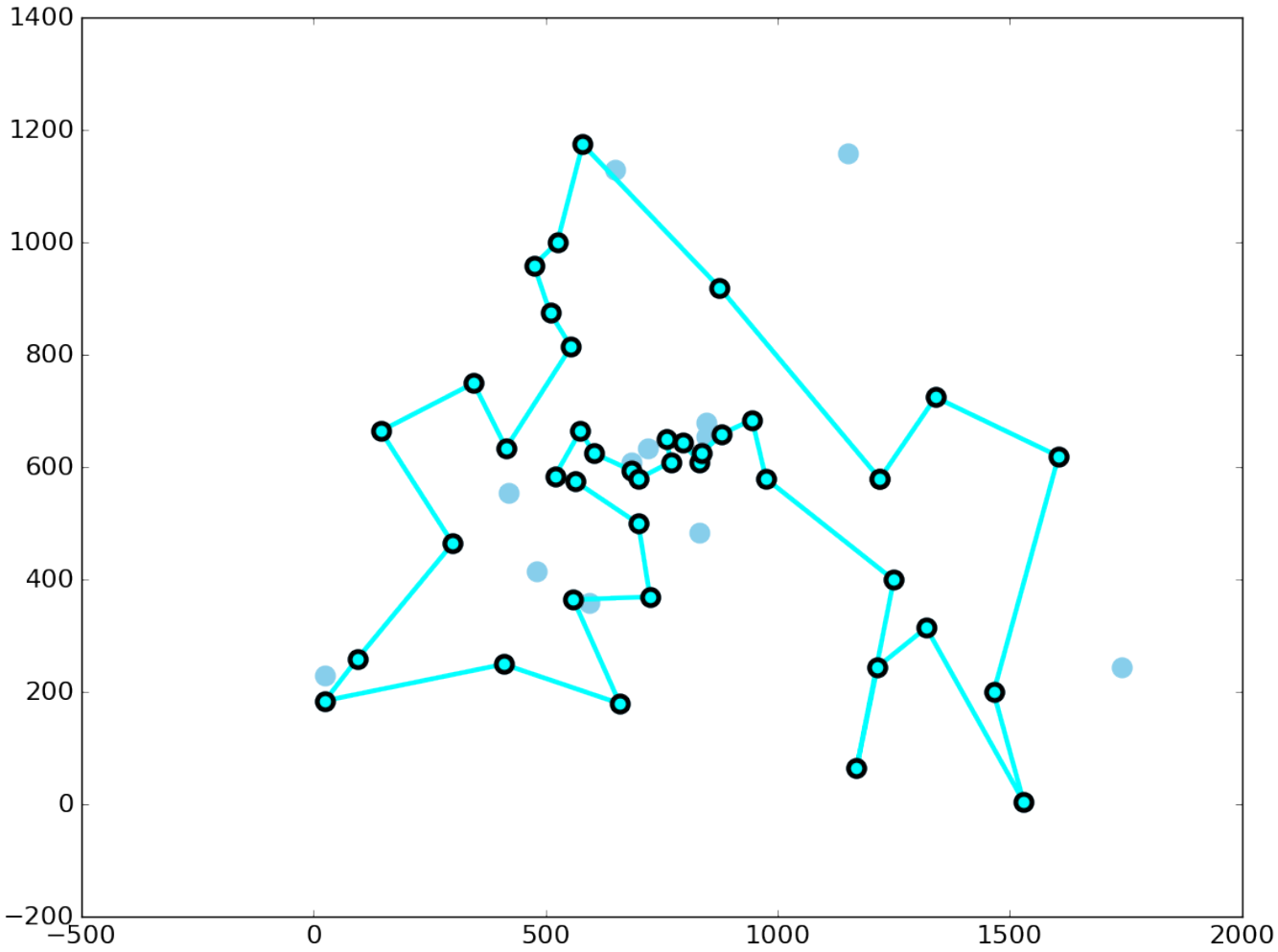
animation of the quick tour algorithm



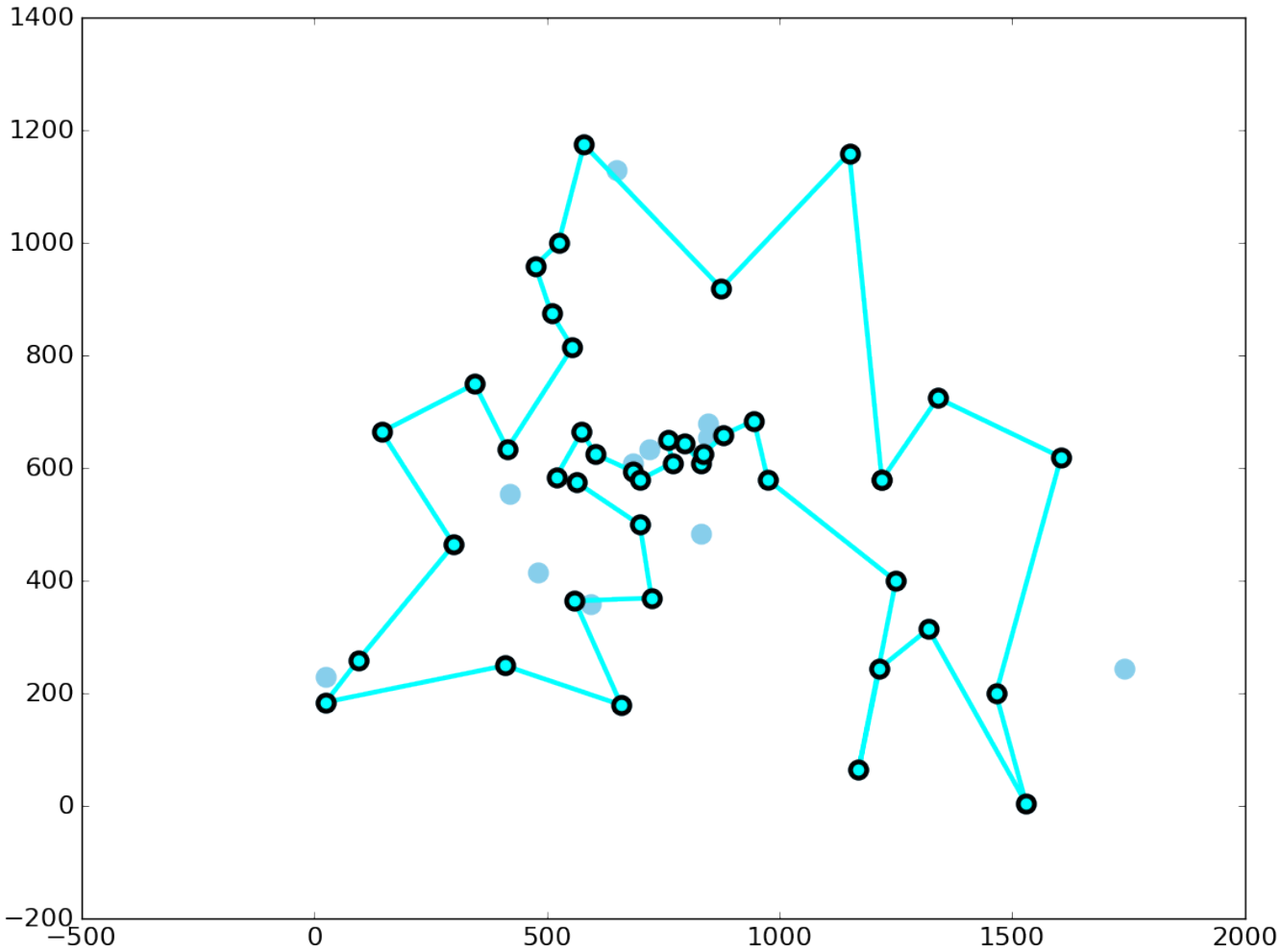
animation of the quick tour algorithm



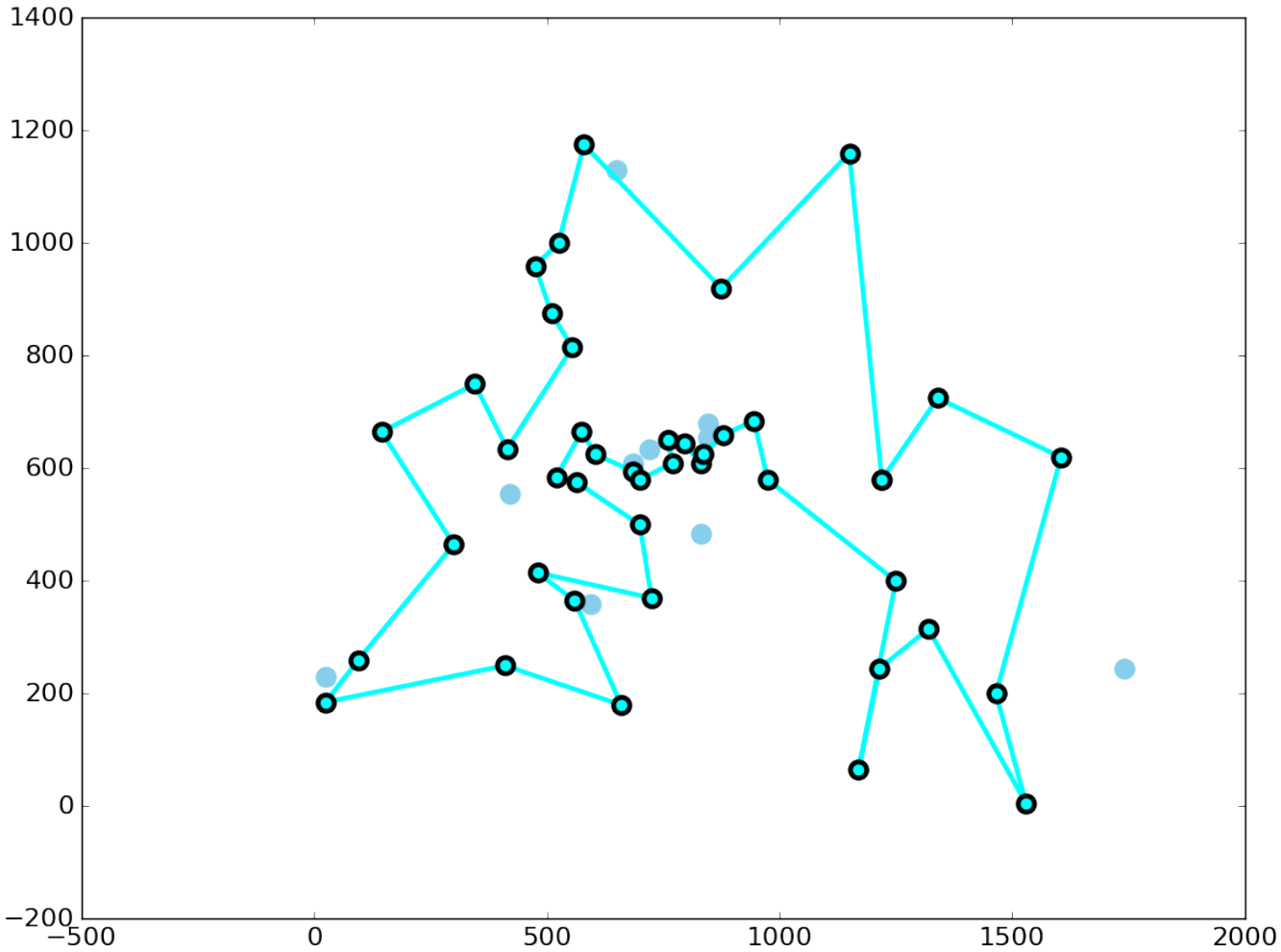
animation of the quick tour algorithm



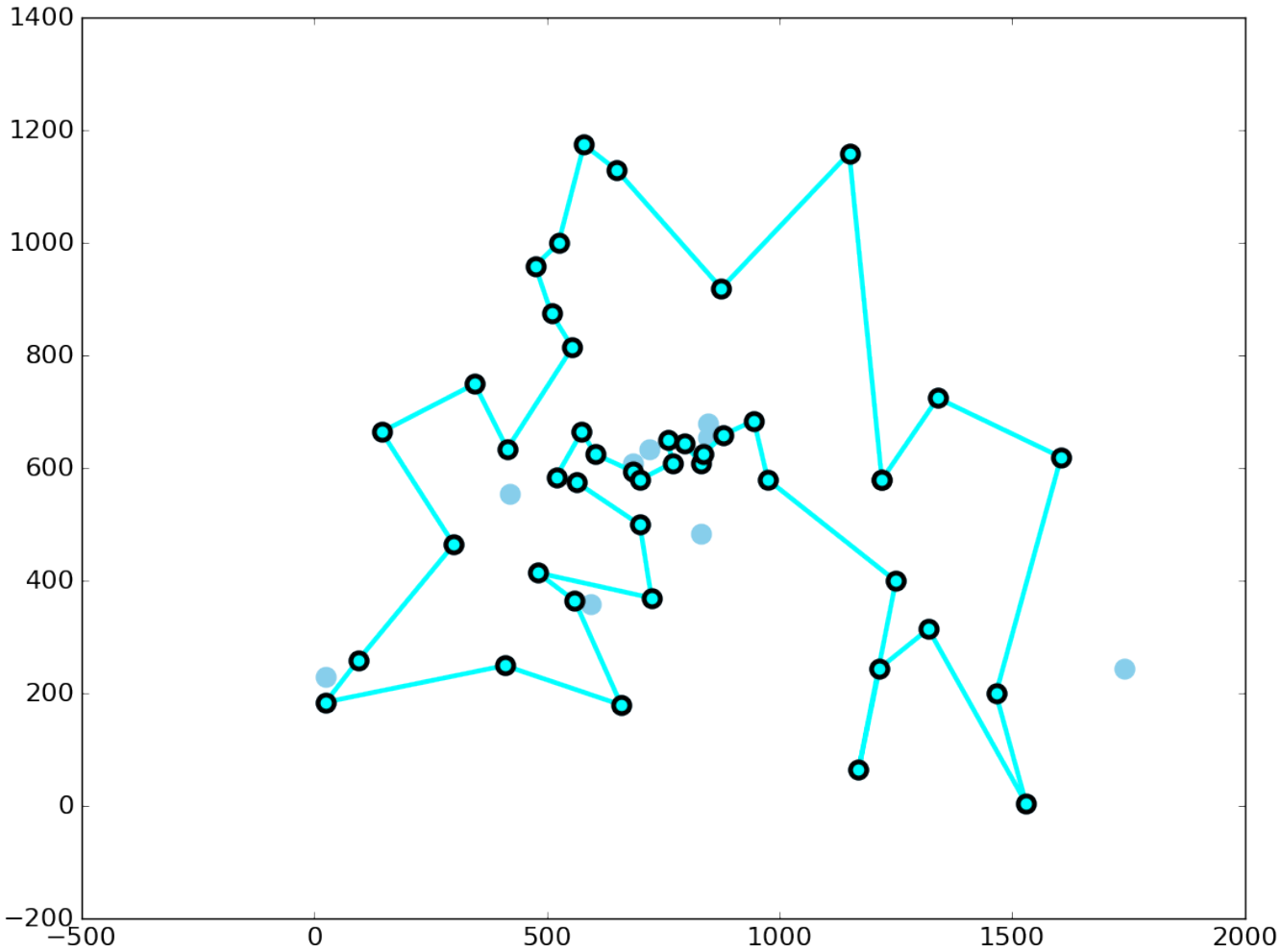
animation of the quick tour algorithm



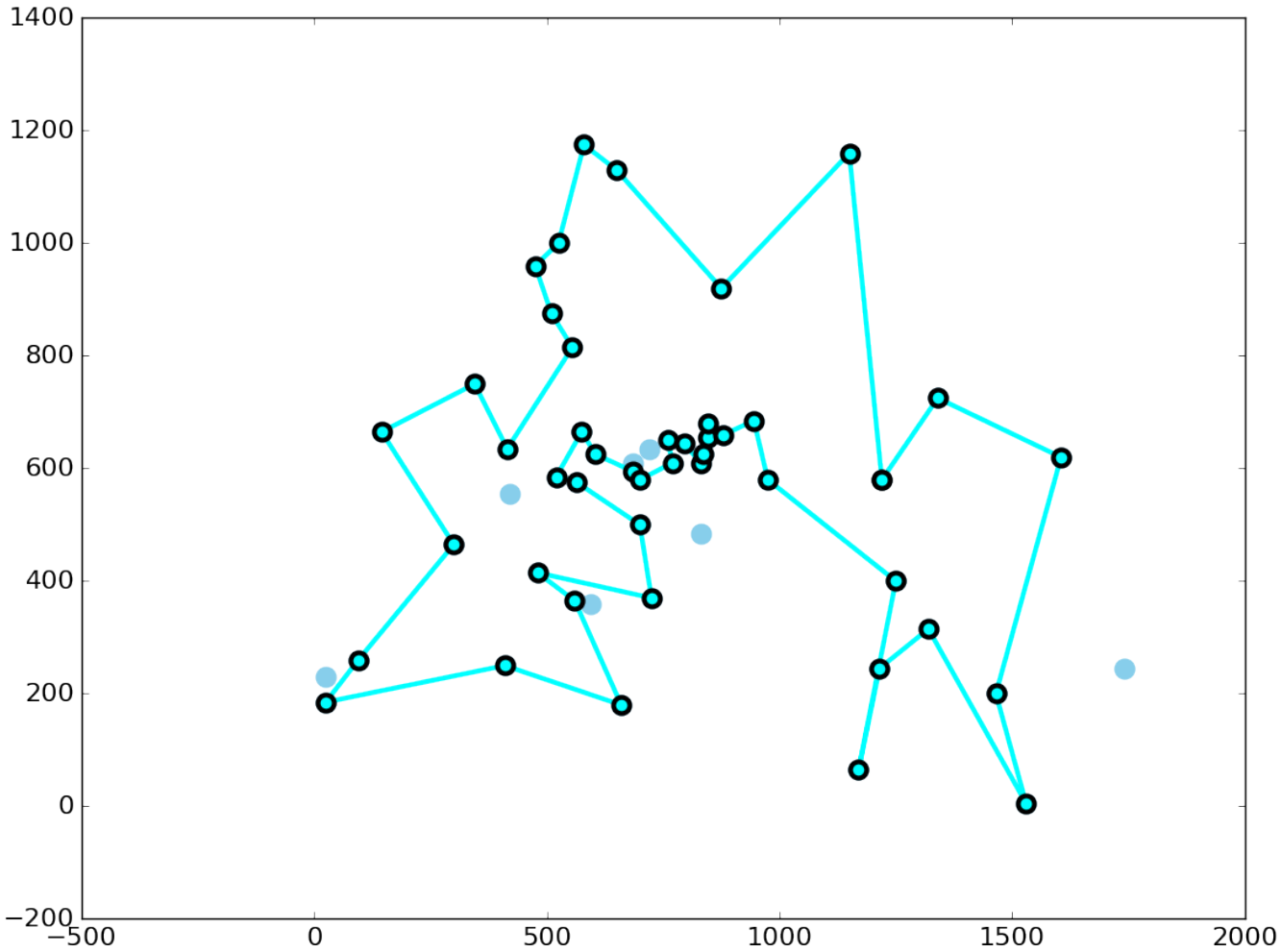
animation of the quick tour algorithm



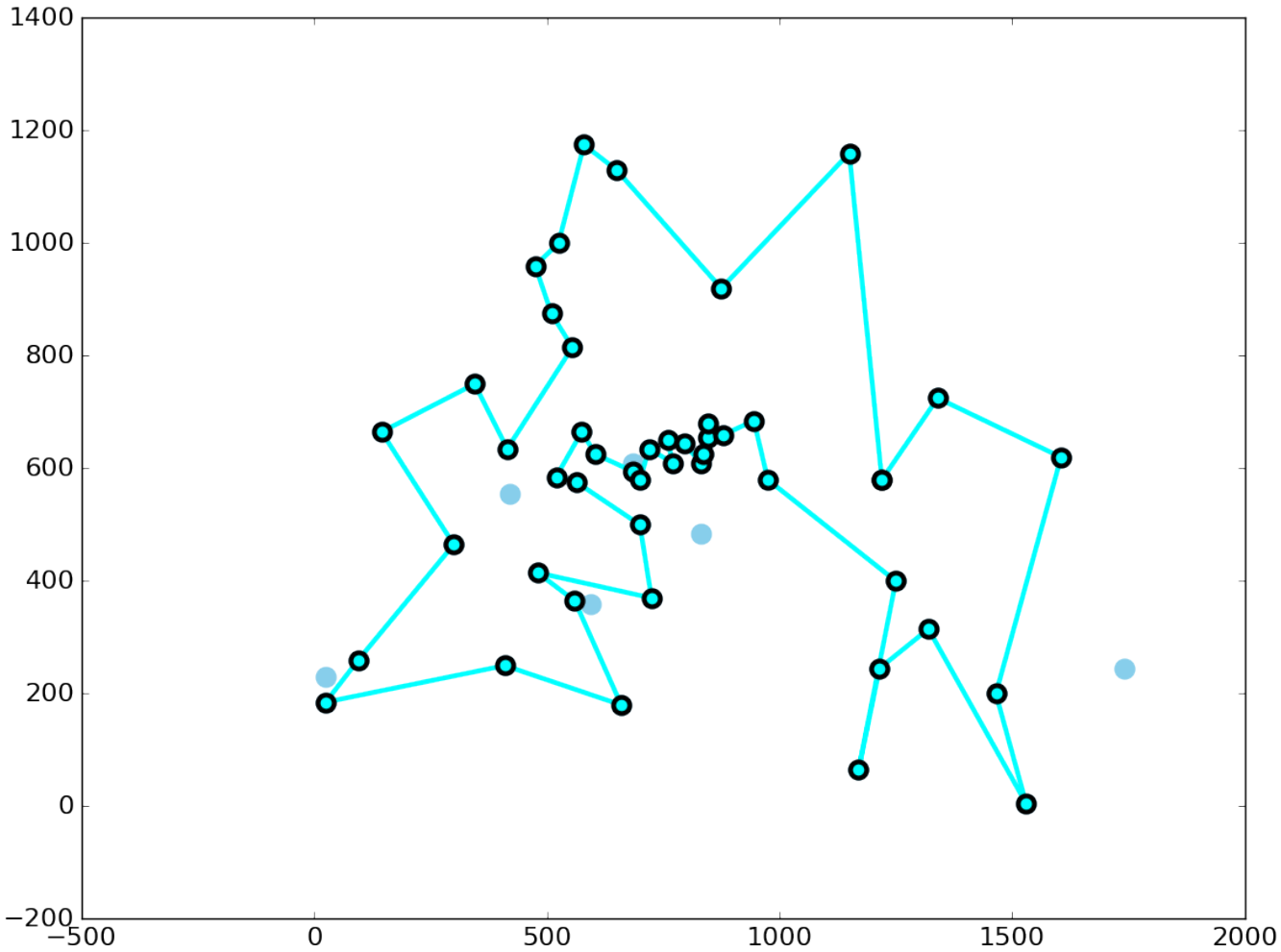
animation of the quick tour algorithm



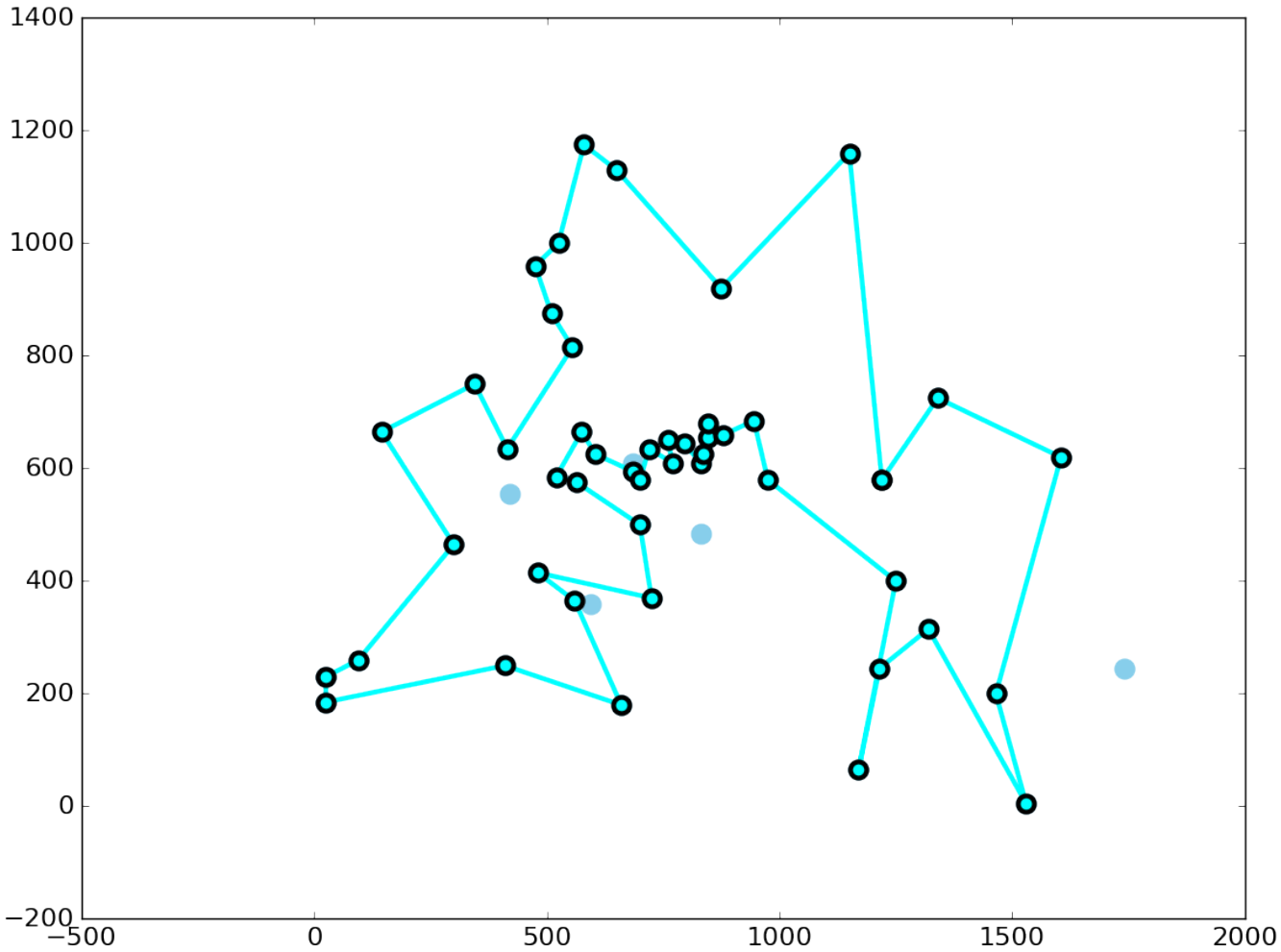
animation of the quick tour algorithm



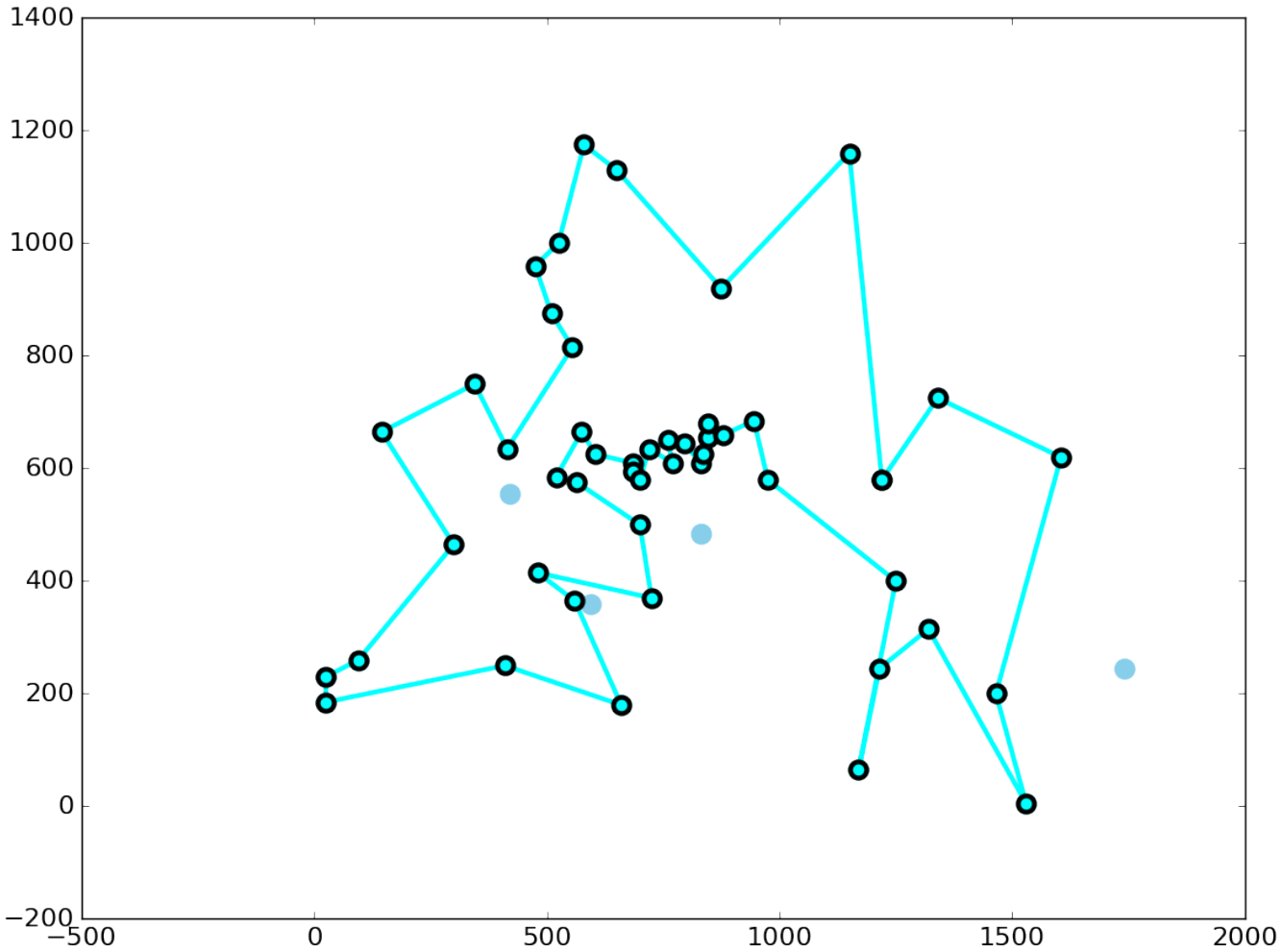
animation of the quick tour algorithm



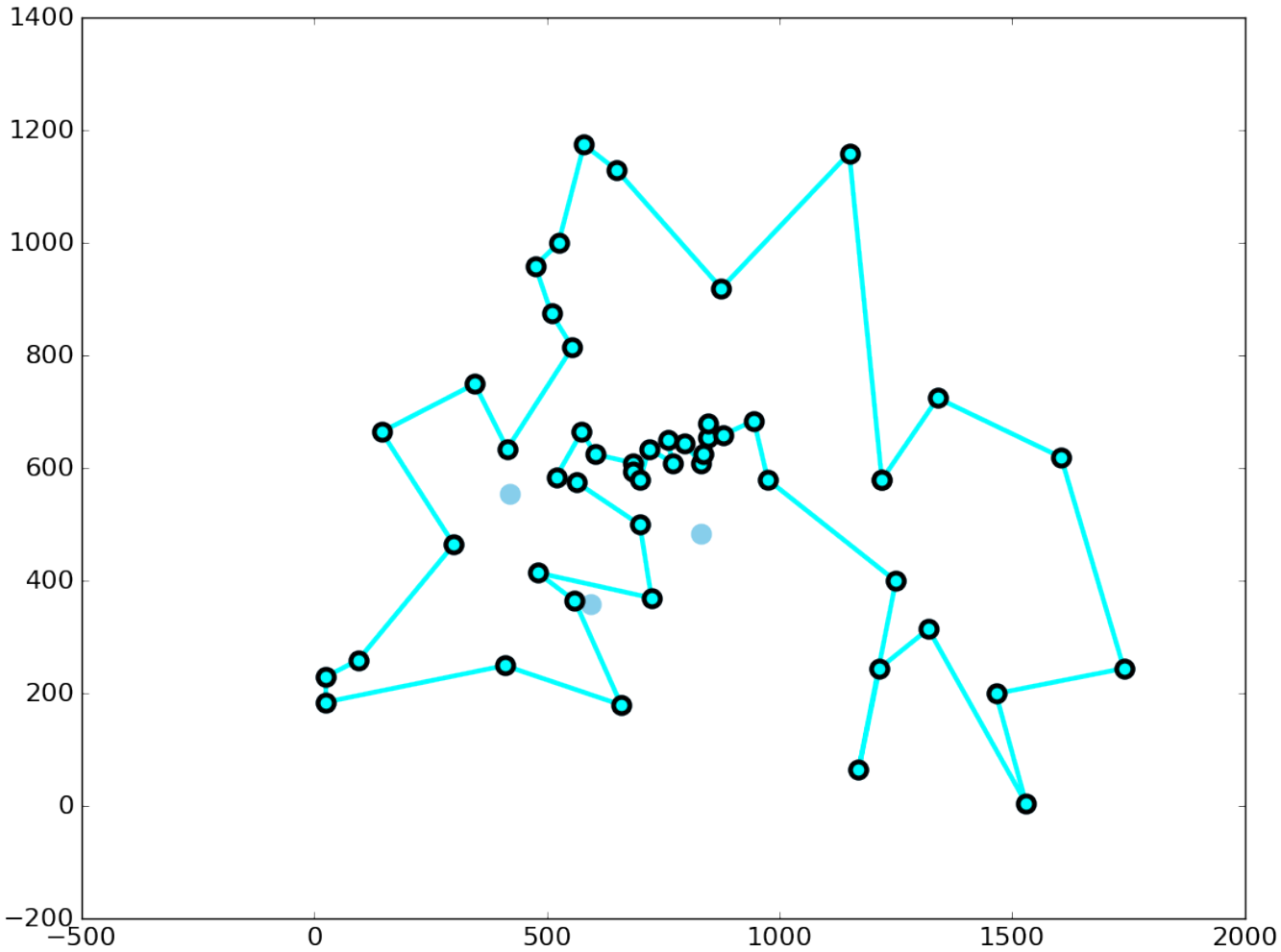
animation of the quick tour algorithm



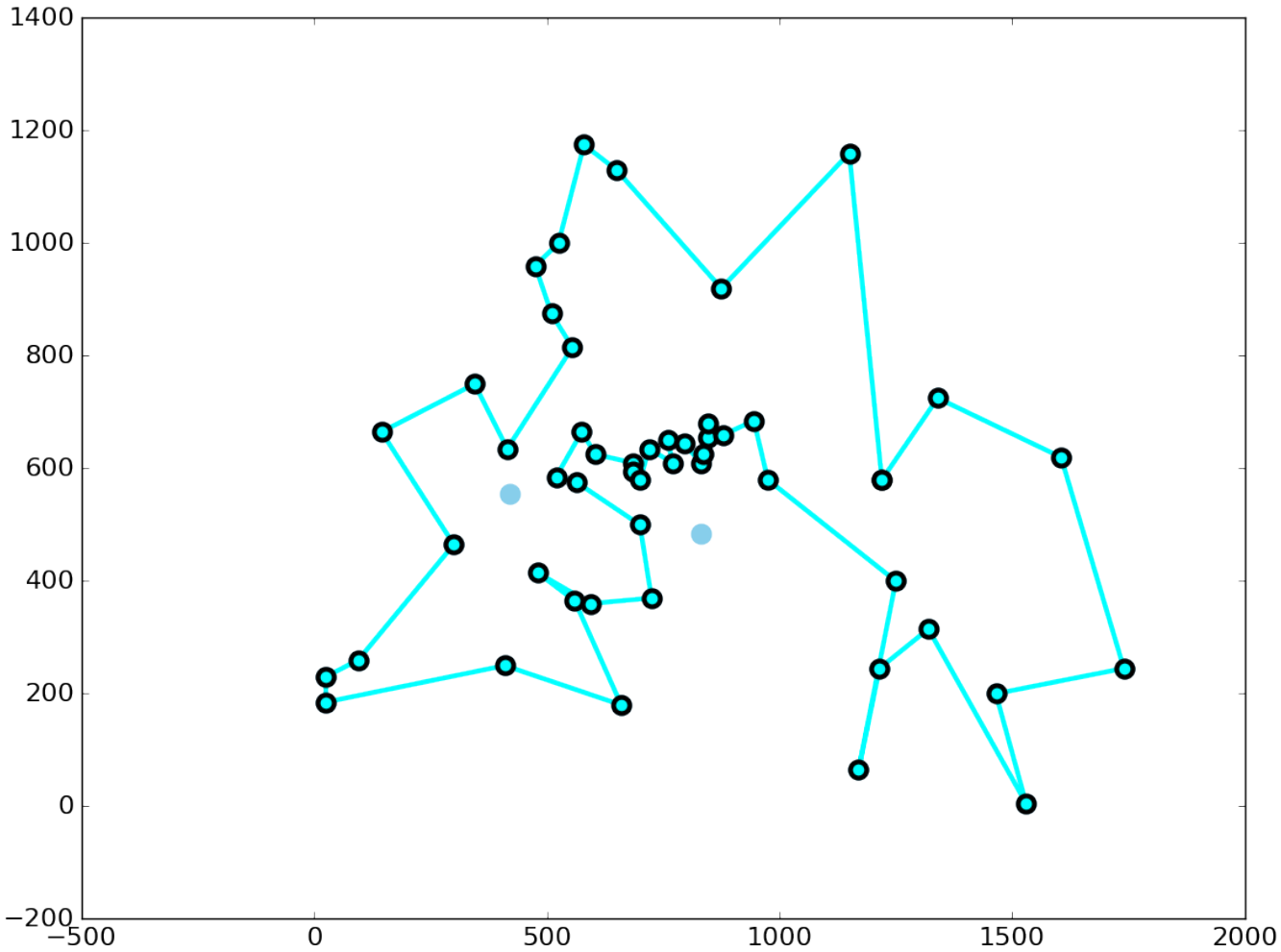
animation of the quick tour algorithm



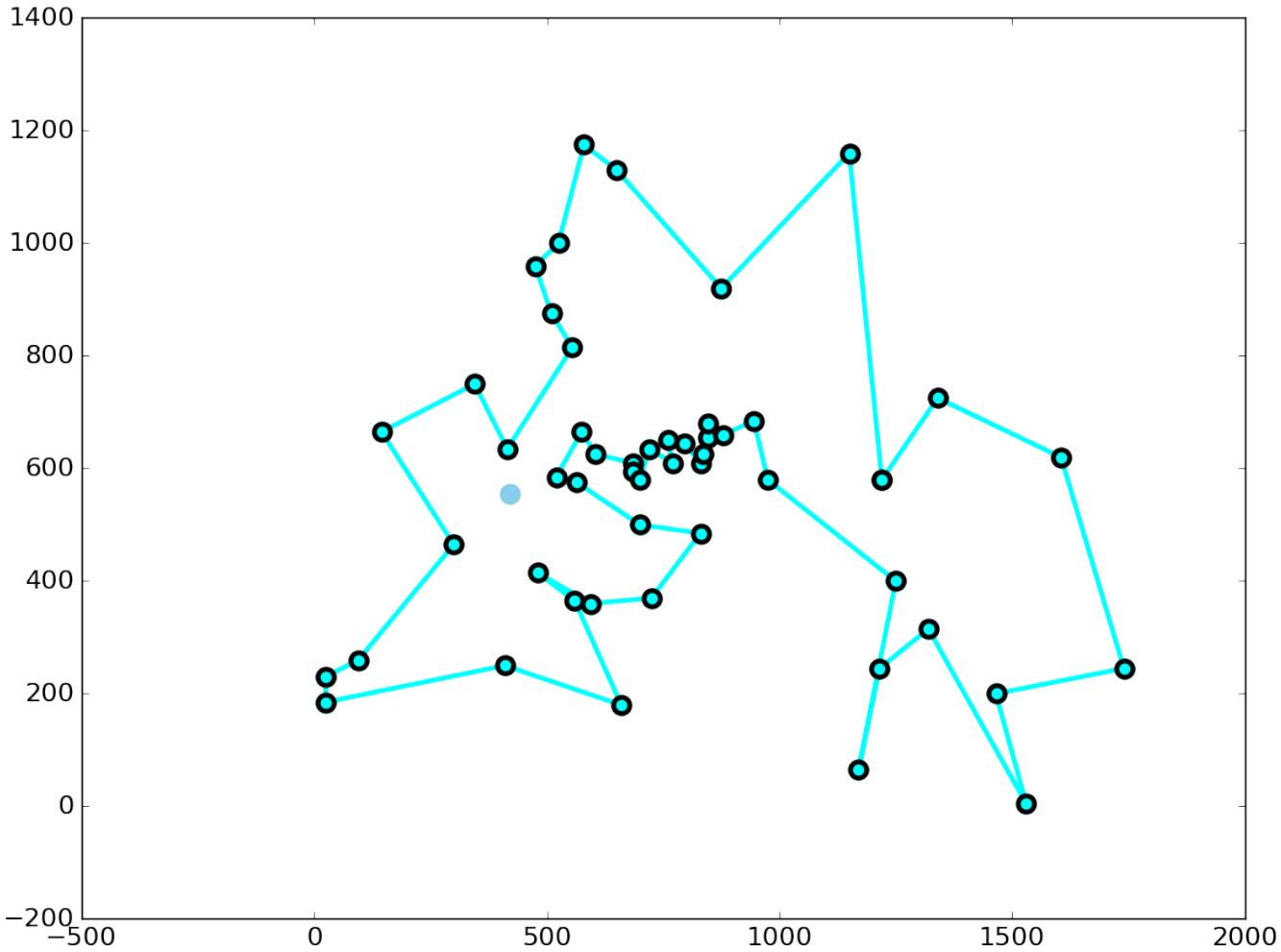
animation of the quick tour algorithm



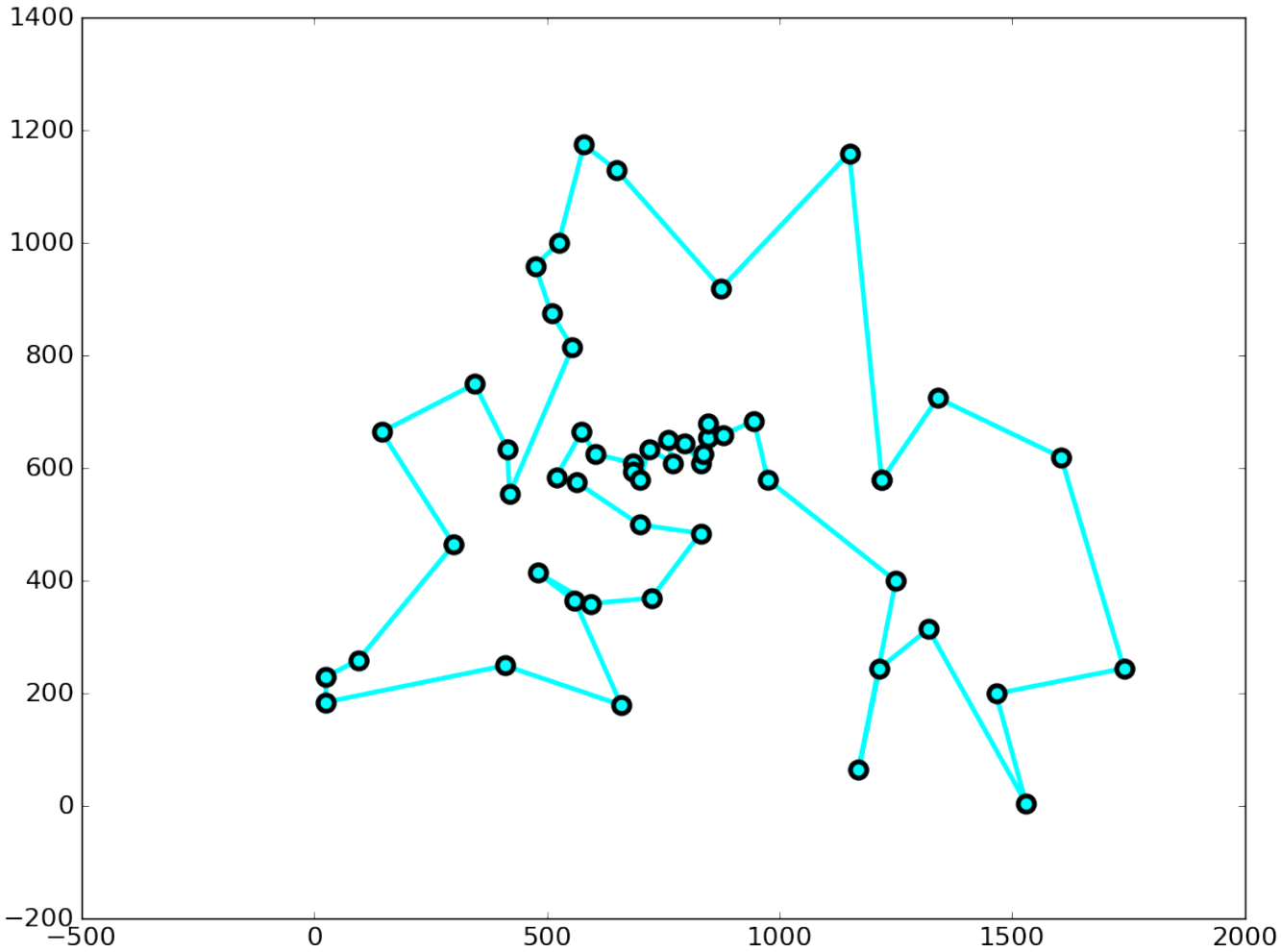
animation of the quick tour algorithm



animation of the quick tour algorithm



animation of the quick tour algorithm



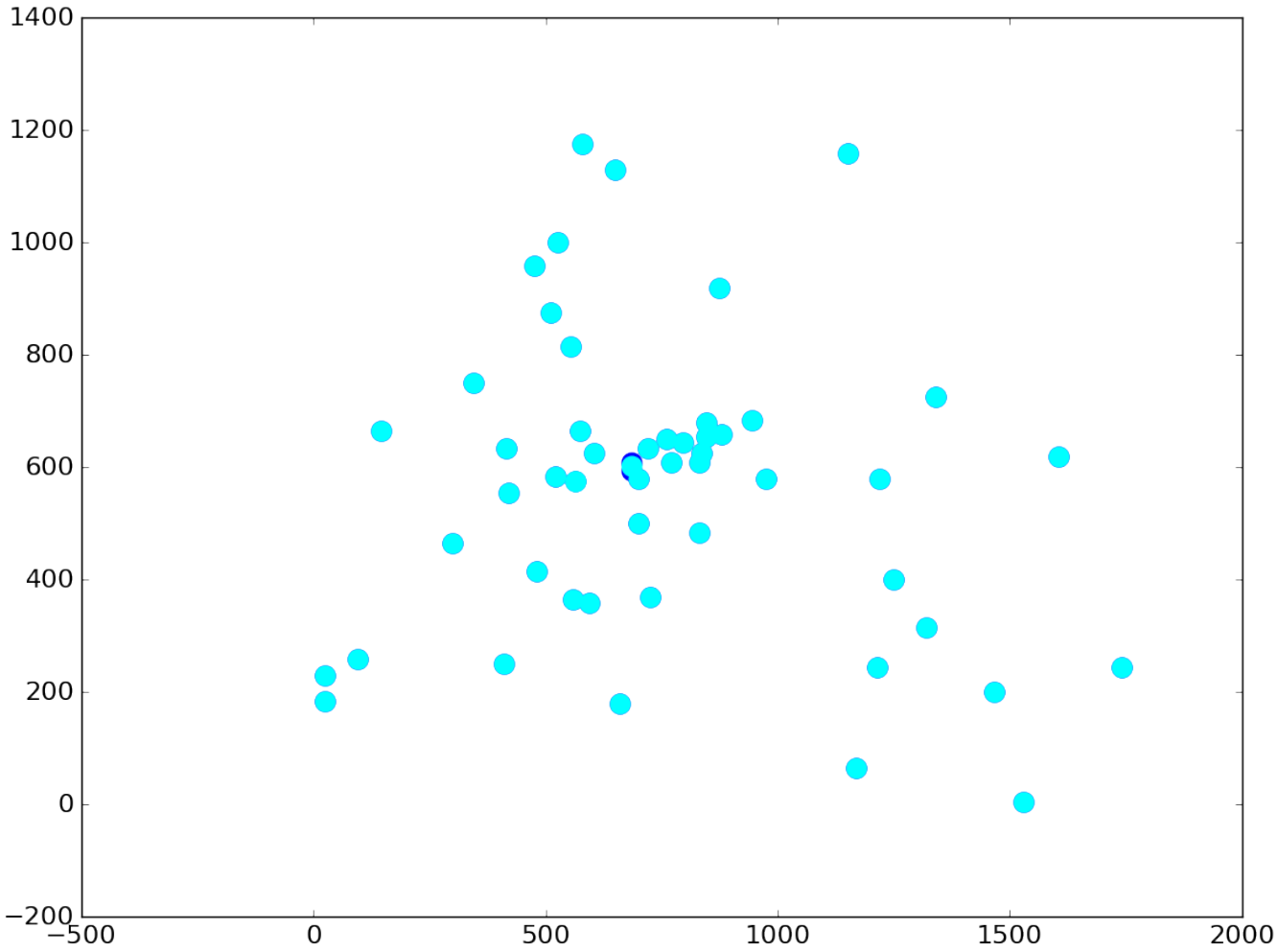
How does the pair-center algorithm work?

The pair-center tour algorithm is a contribution of my own for this course (still haven't found it on the internet). It is a deterministic algorithm:

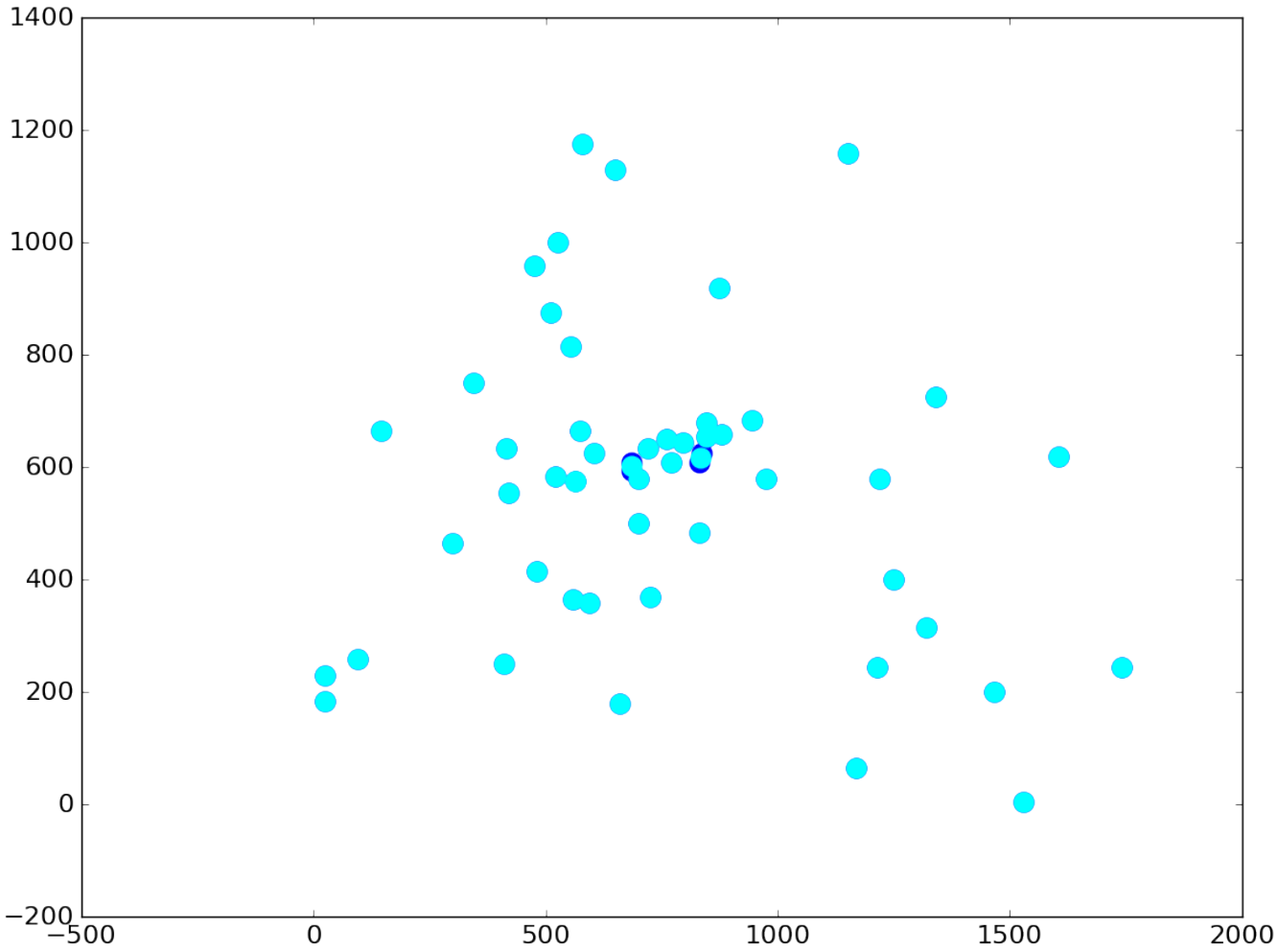
- with a bottom-up construction build a binary tree by replacing the/a closest pair of points by their center
- with a top-down construction build the tour by inserting the corresponding pairs in the best possible way

The runtime is in $\mathcal{O}(n^2)$, I guess (implemented is $\mathcal{O}(n^3)$ and a $\mathcal{O}(n^2)$ version with more sophisticated data structures).
Can you prove a worst case bound for the tour length? Recently I've improved to $\mathcal{O}(n\mathrm{polylog}(n))$ with practical runtime in the order of $n \log n$ and error below 5%.

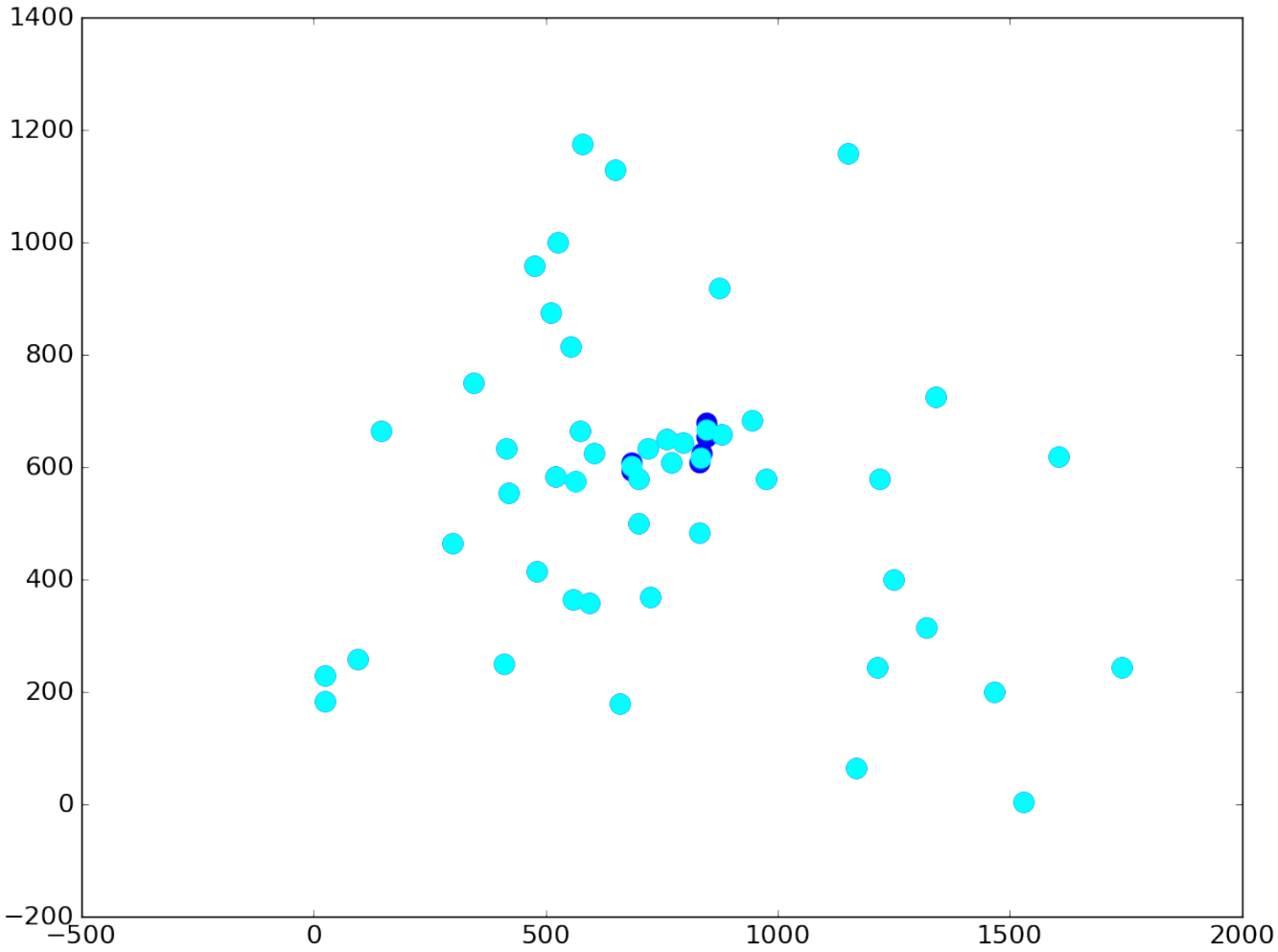
animation of the pair-center tour algorithm



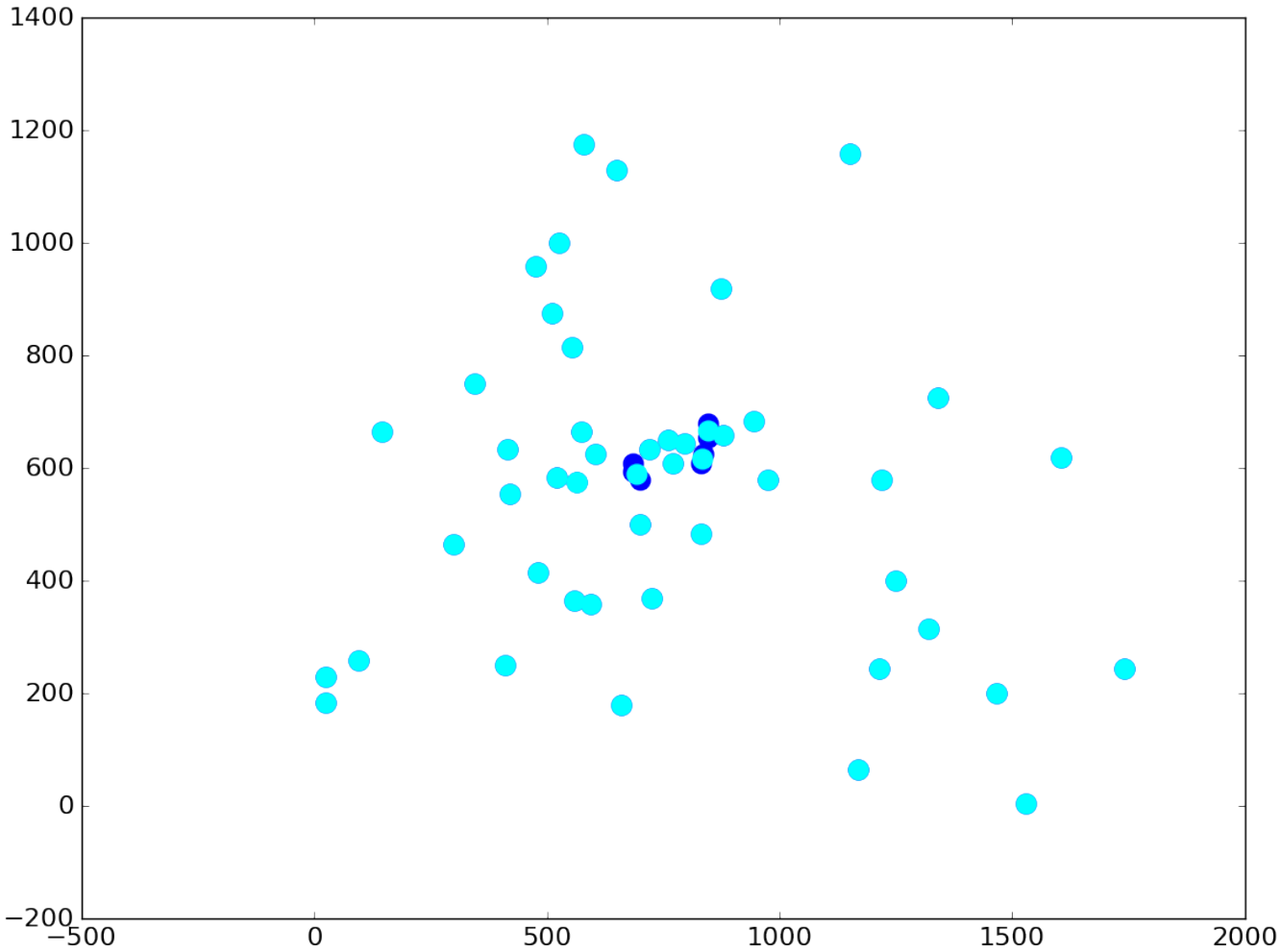
animation of the pair-center tour algorithm



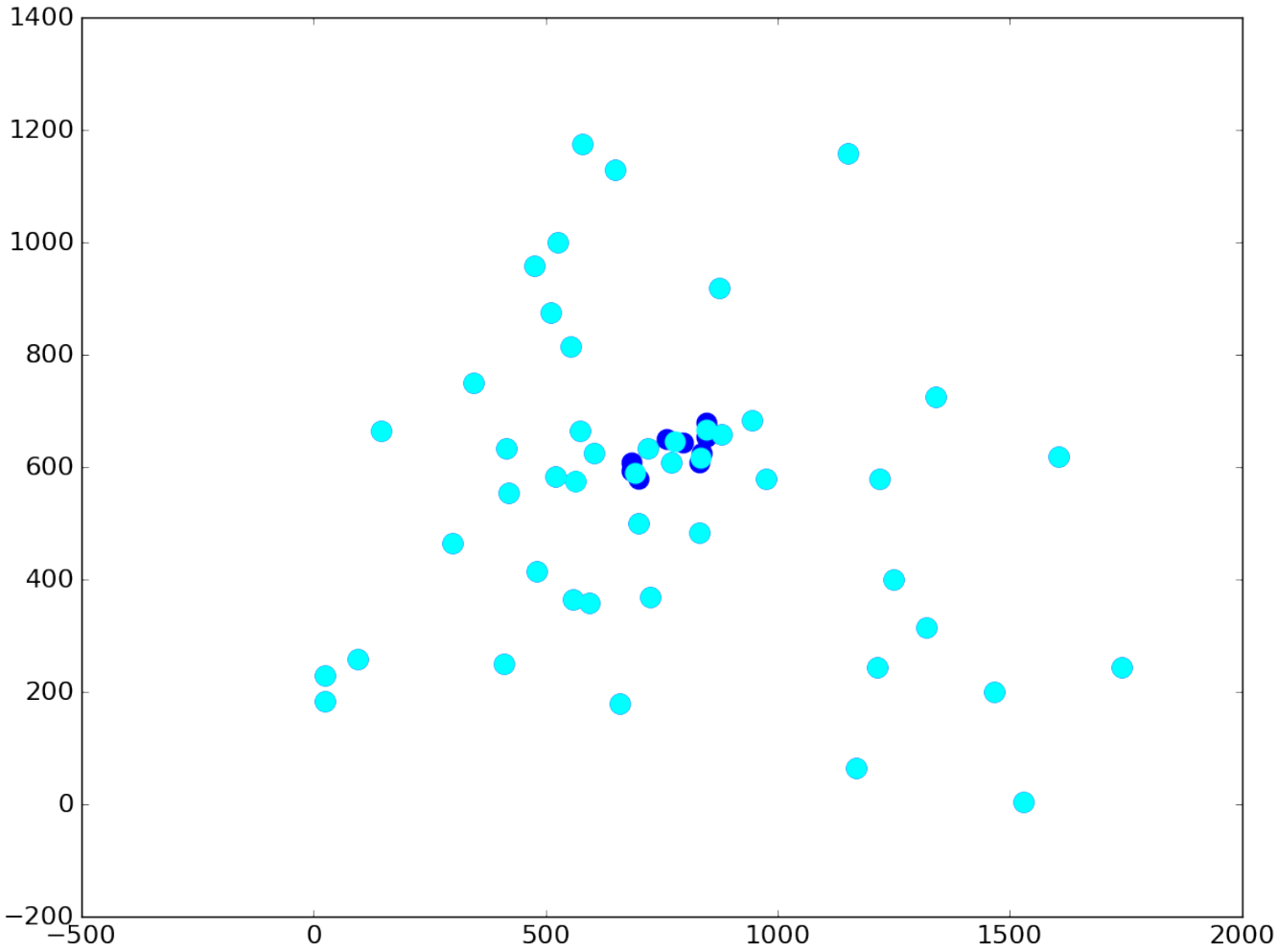
animation of the pair-center tour algorithm



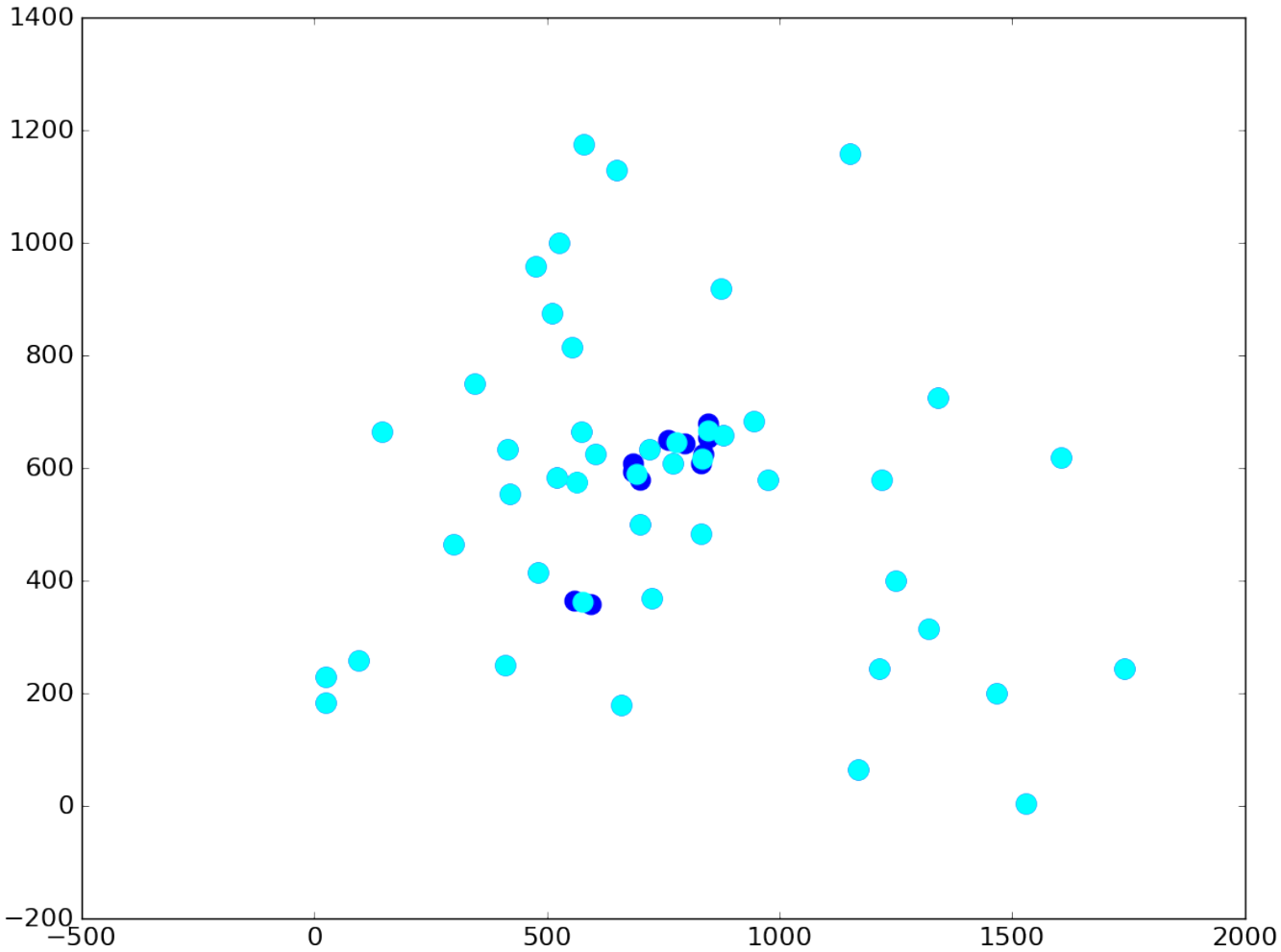
animation of the pair-center tour algorithm



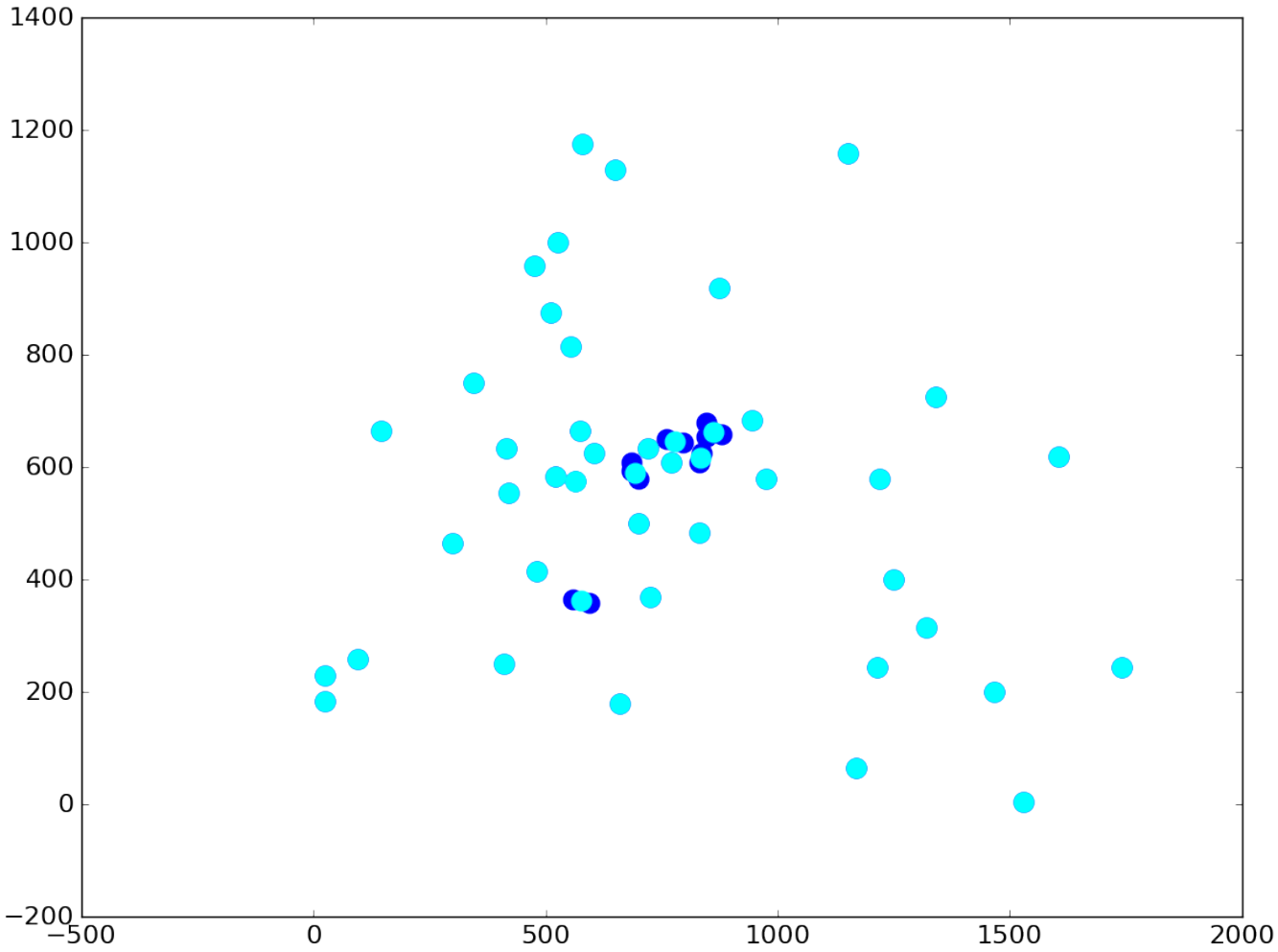
animation of the pair-center tour algorithm



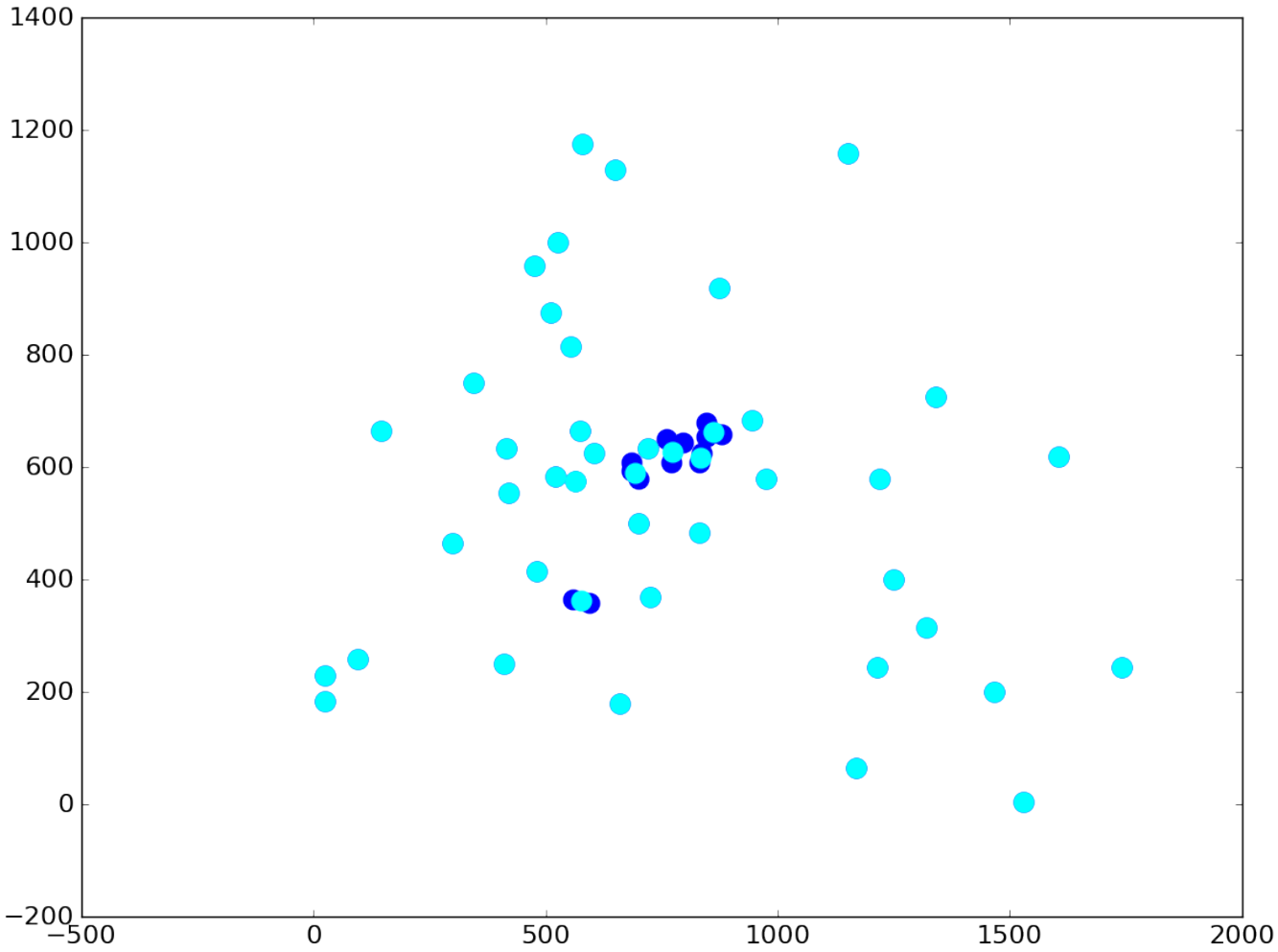
animation of the pair-center tour algorithm



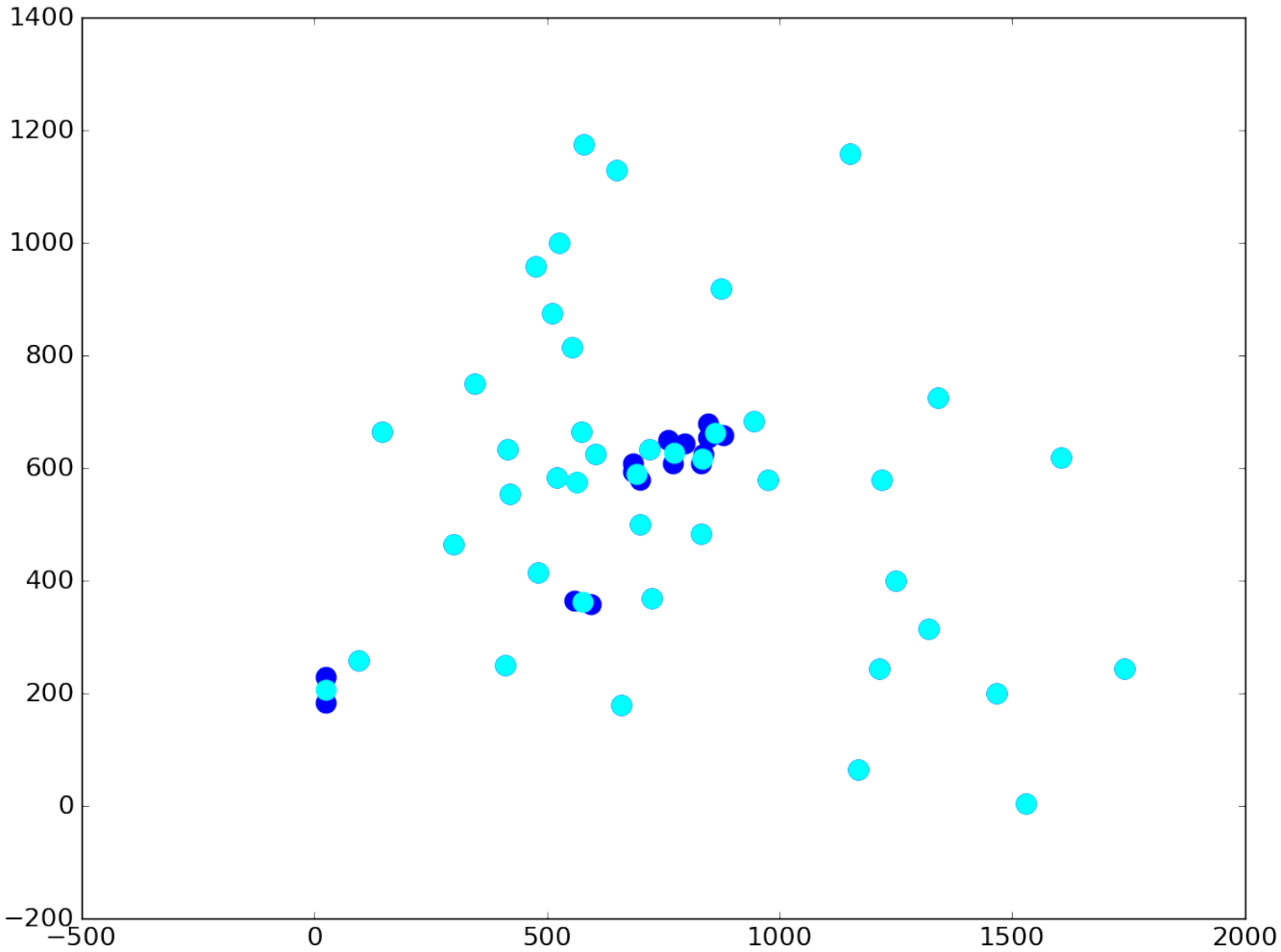
animation of the pair-center tour algorithm



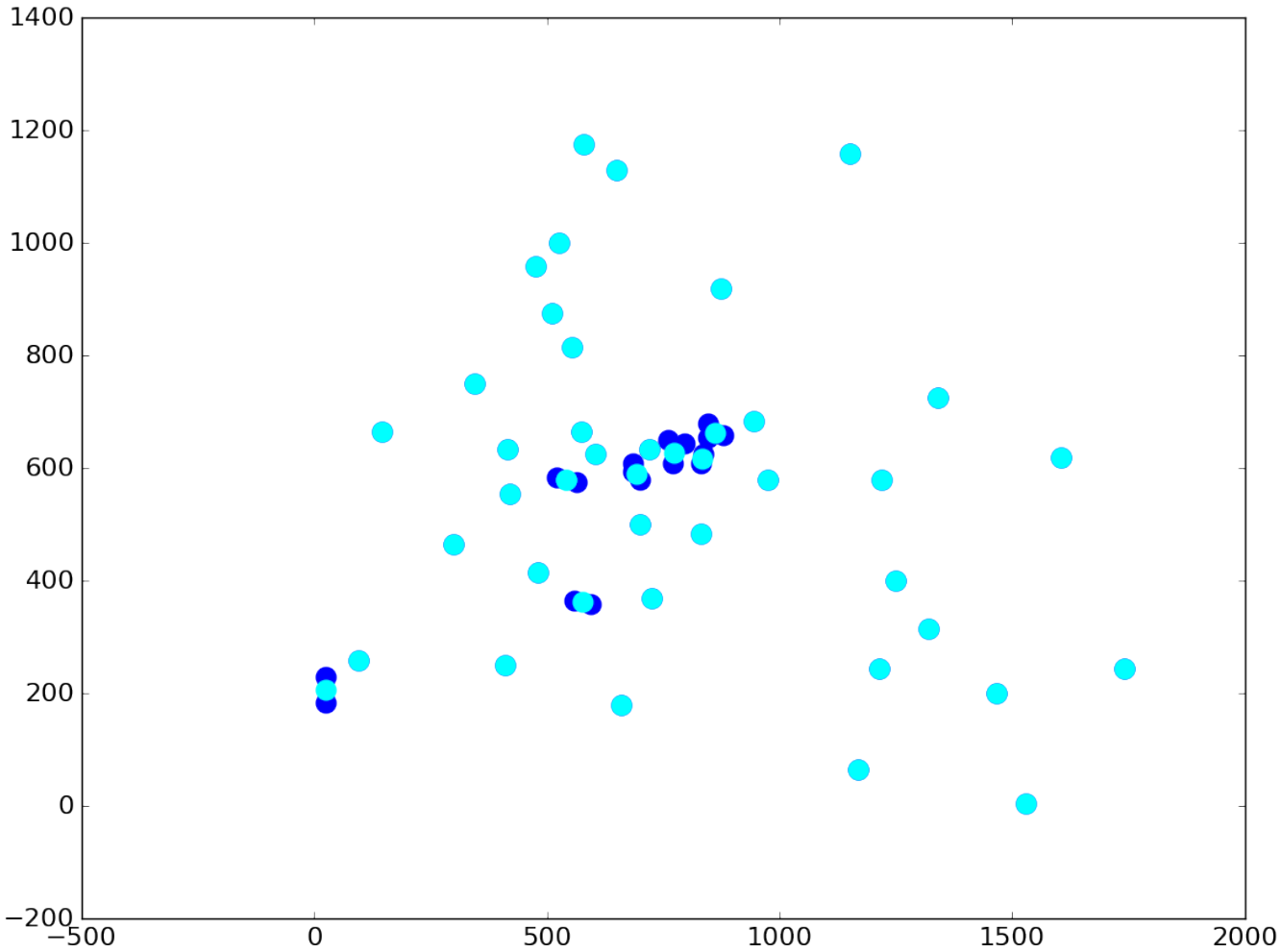
animation of the pair-center tour algorithm



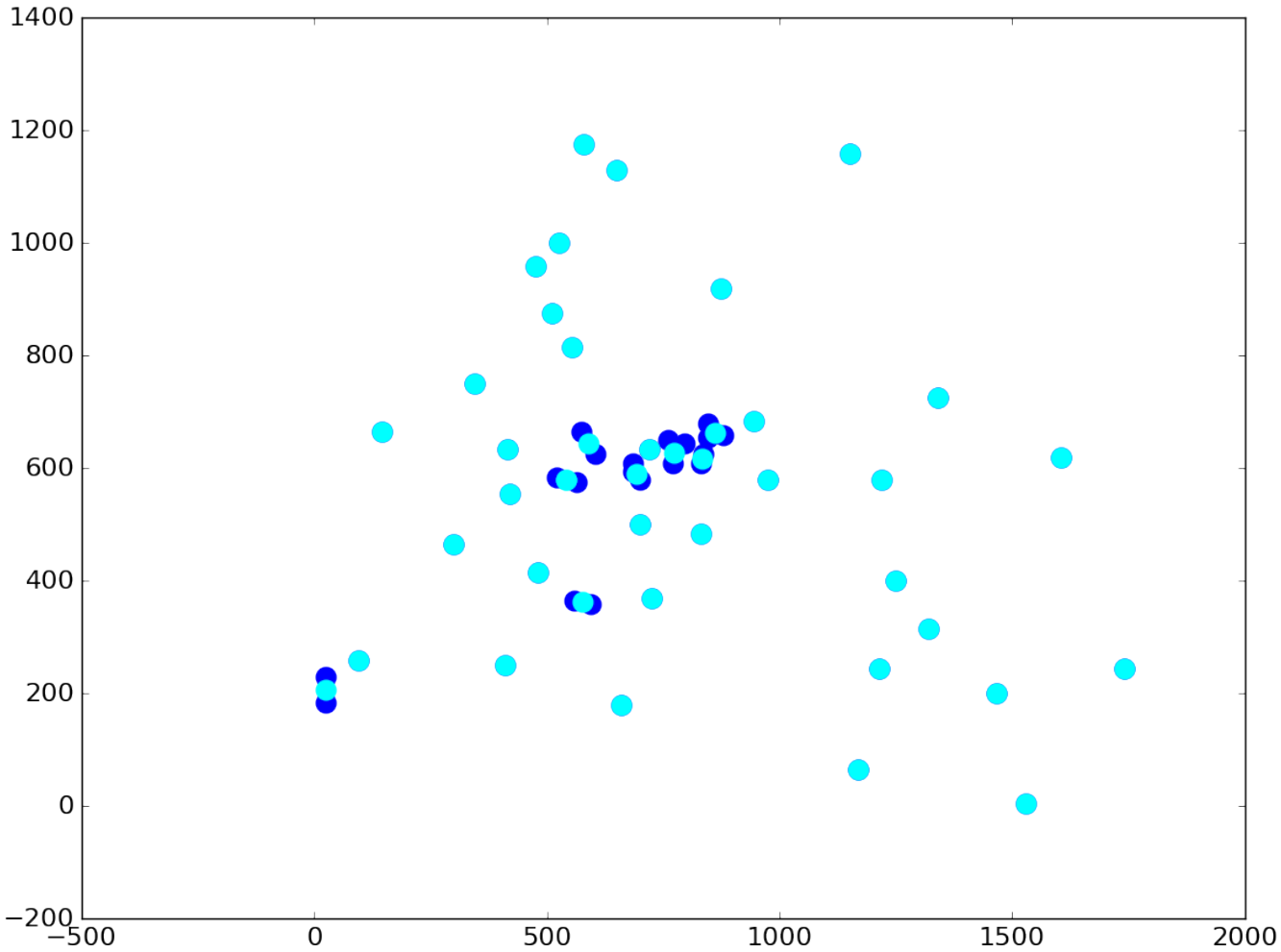
animation of the pair-center tour algorithm



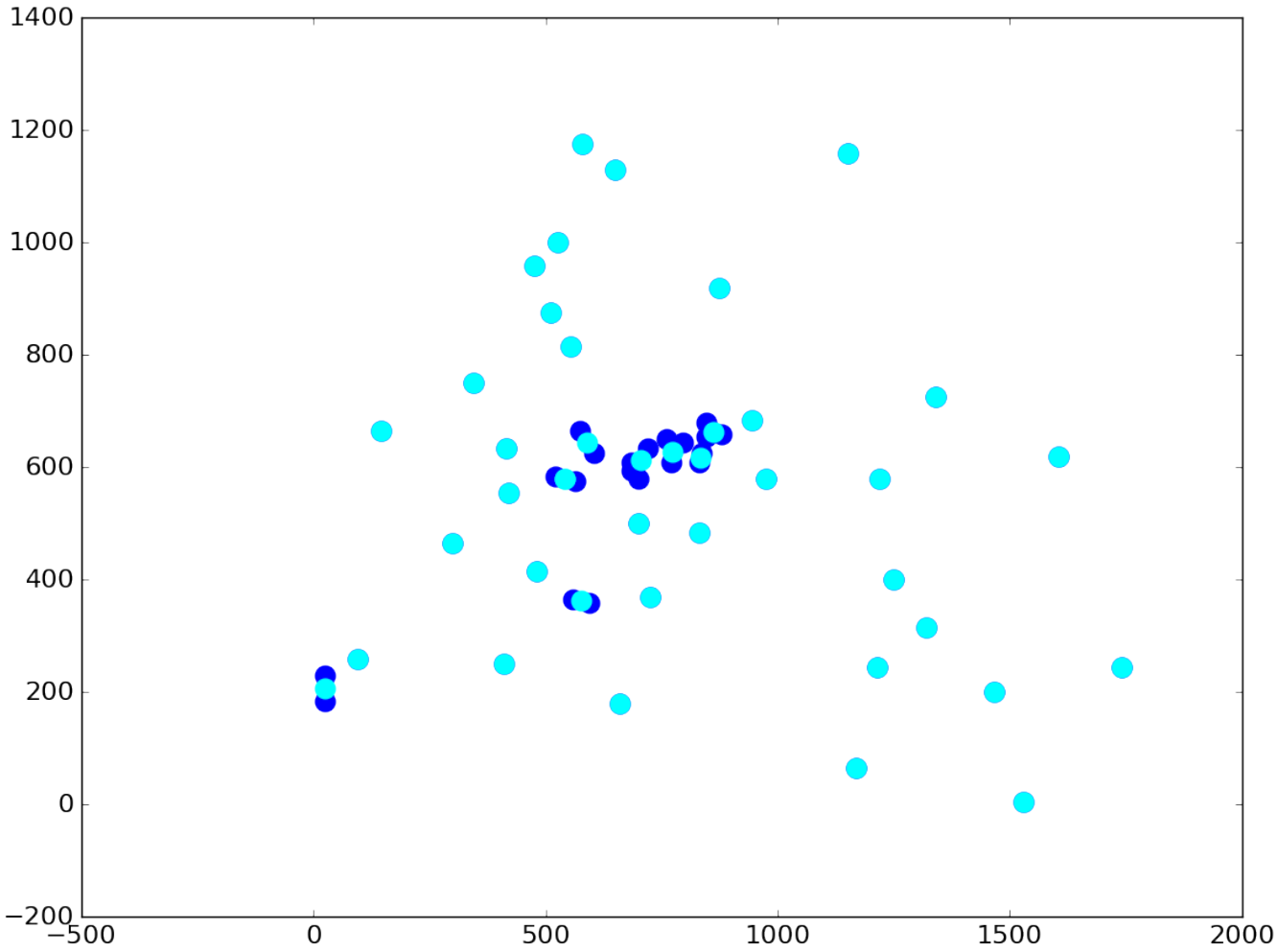
animation of the pair-center tour algorithm



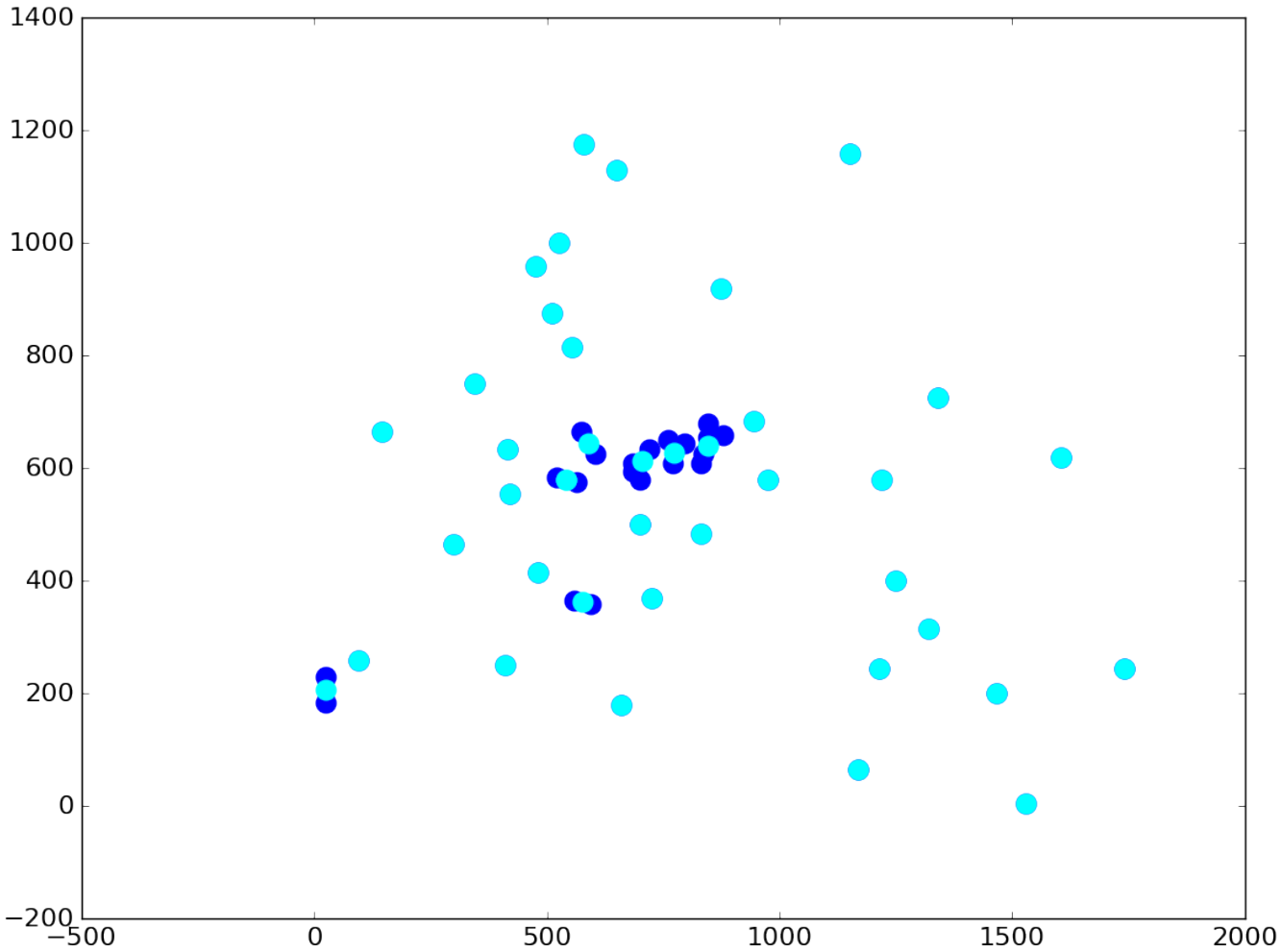
animation of the pair-center tour algorithm



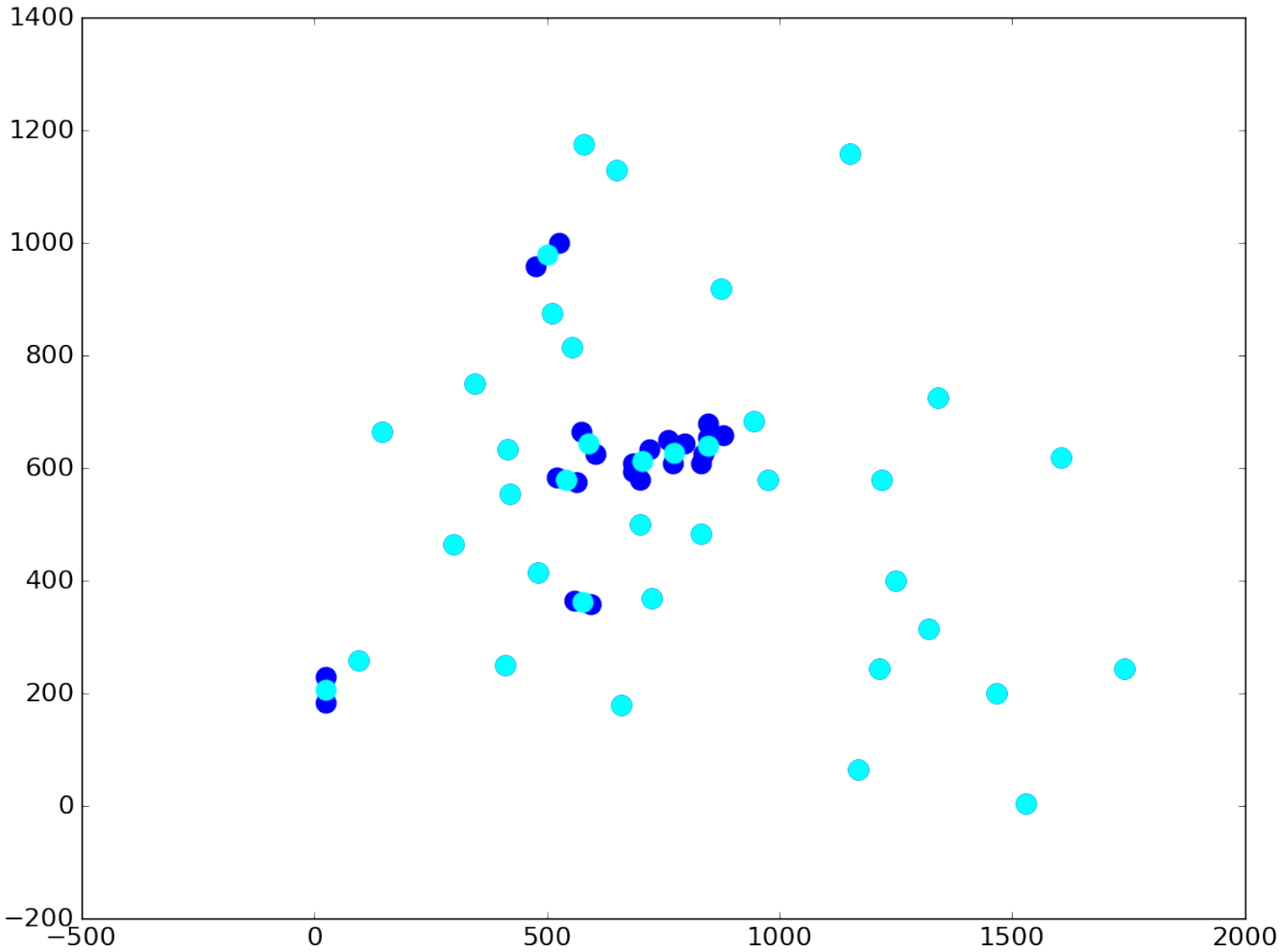
animation of the pair-center tour algorithm



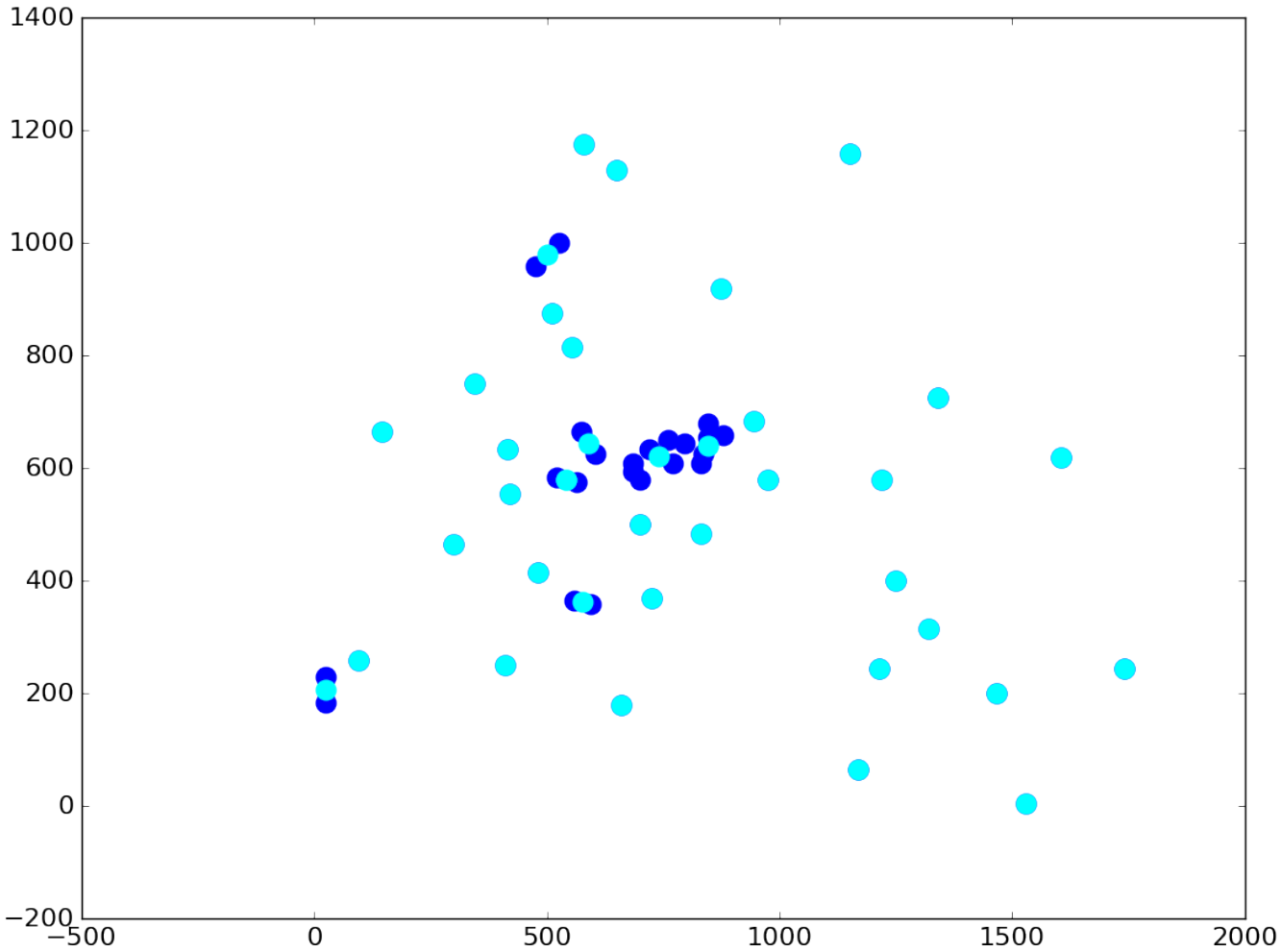
animation of the pair-center tour algorithm



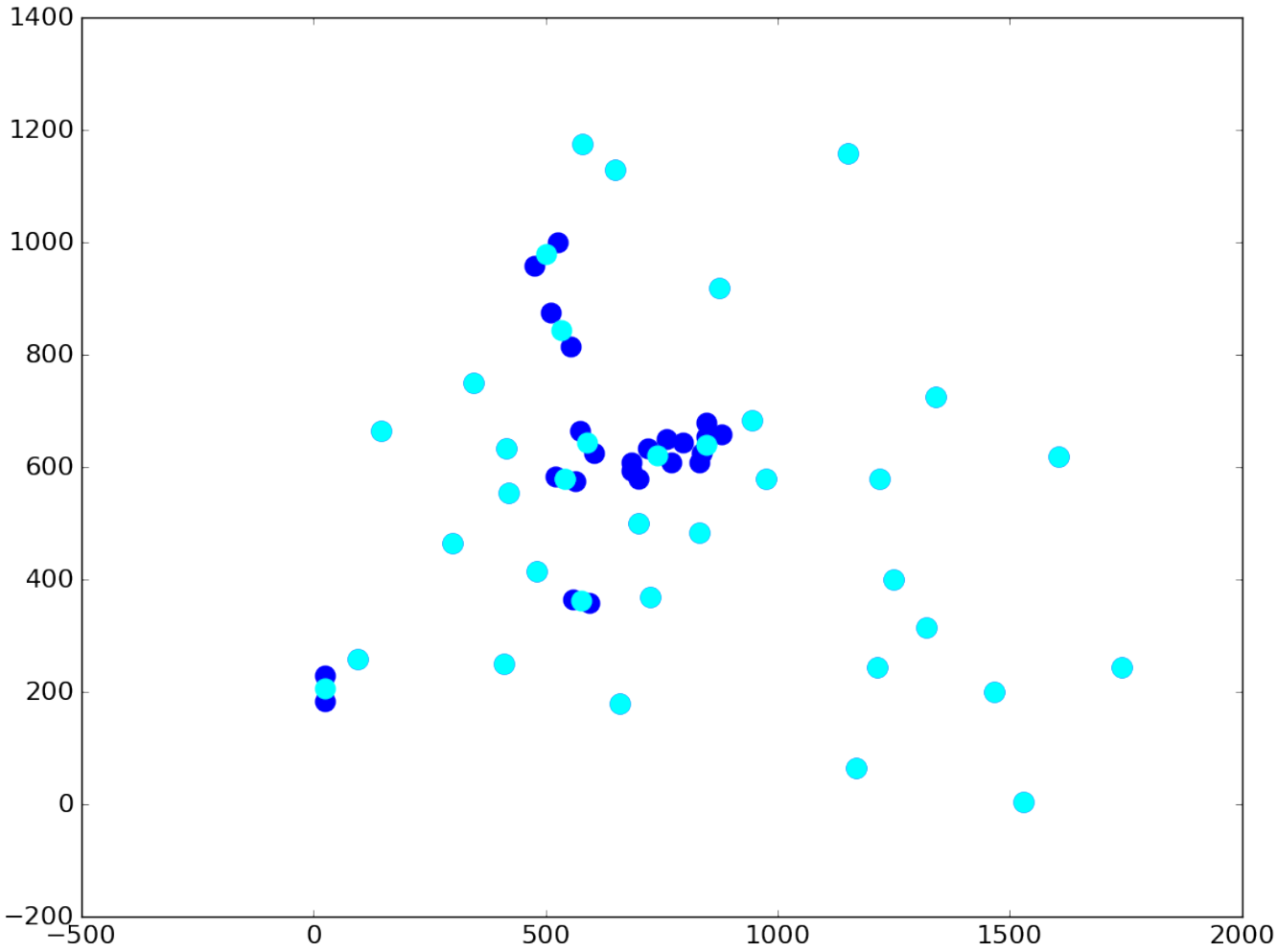
animation of the pair-center tour algorithm



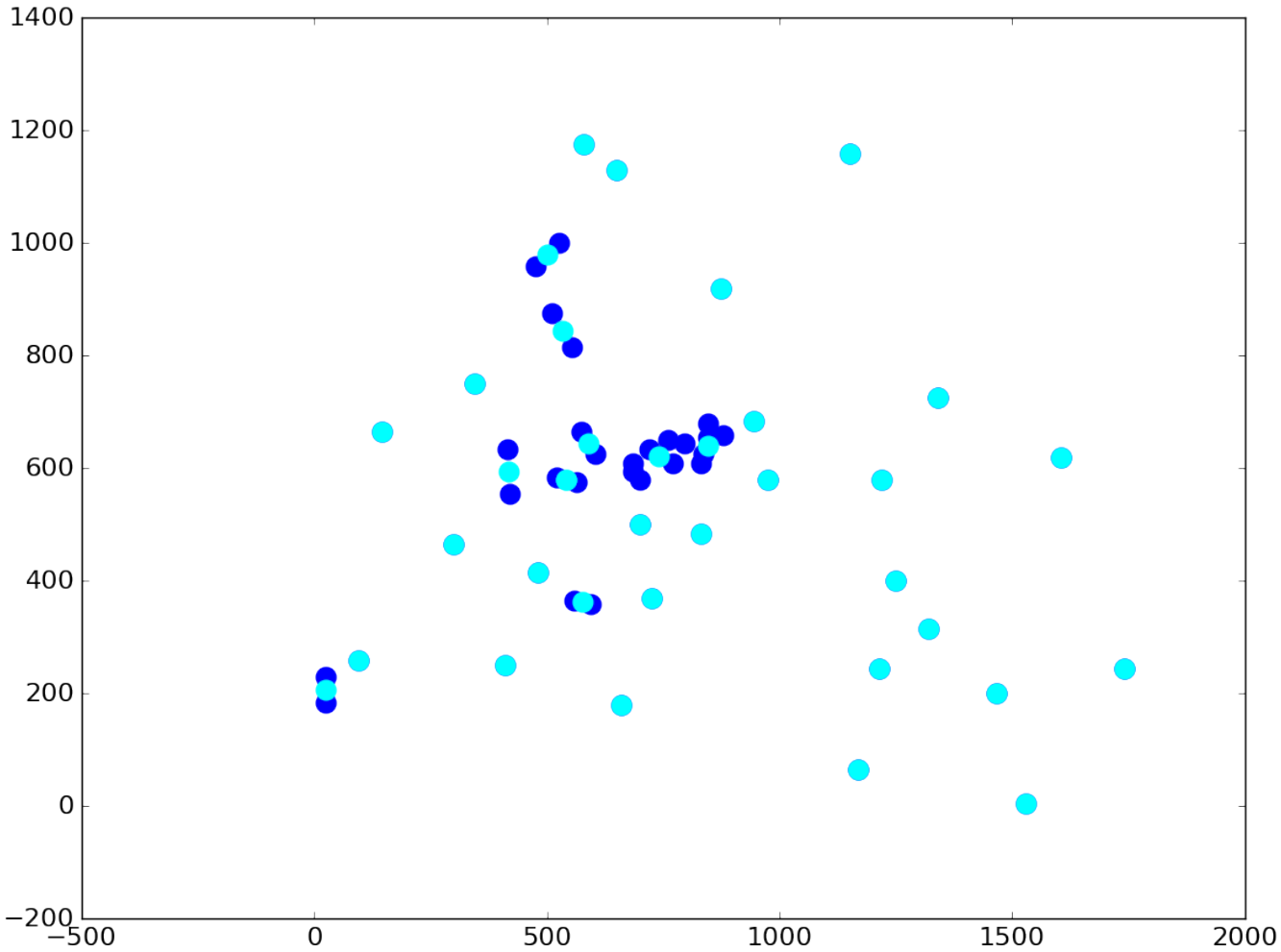
animation of the pair-center tour algorithm



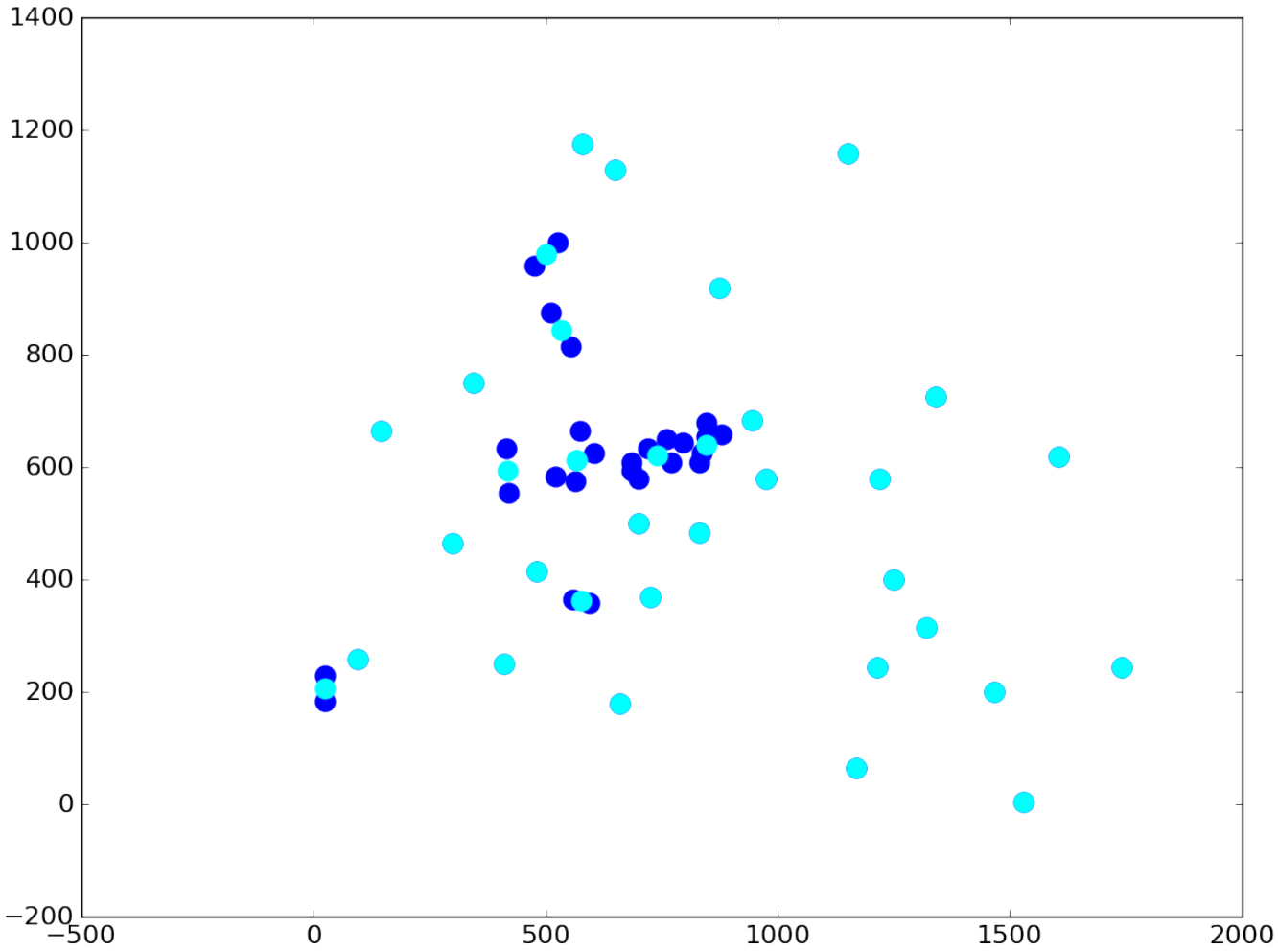
animation of the pair-center tour algorithm



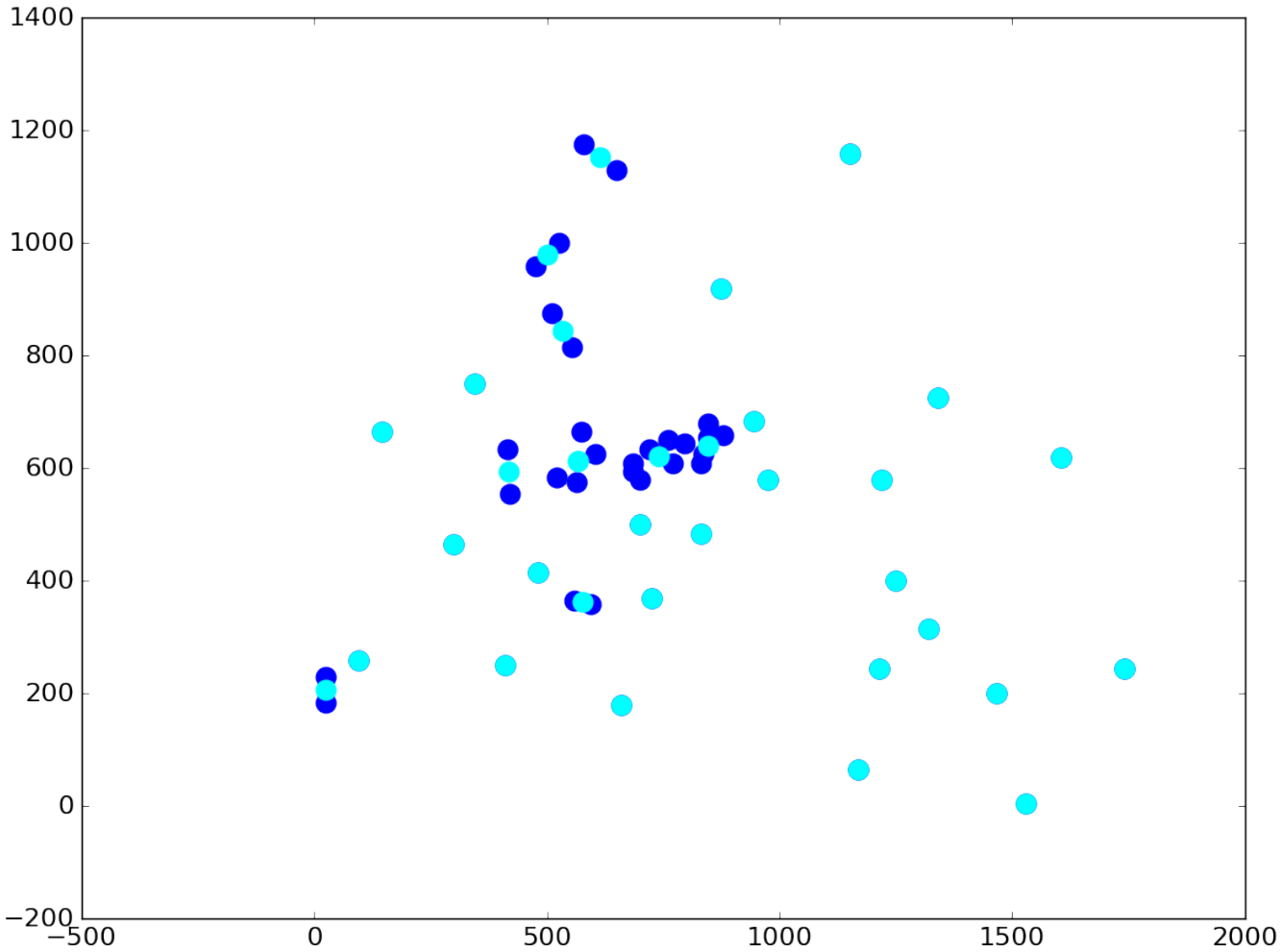
animation of the pair-center tour algorithm



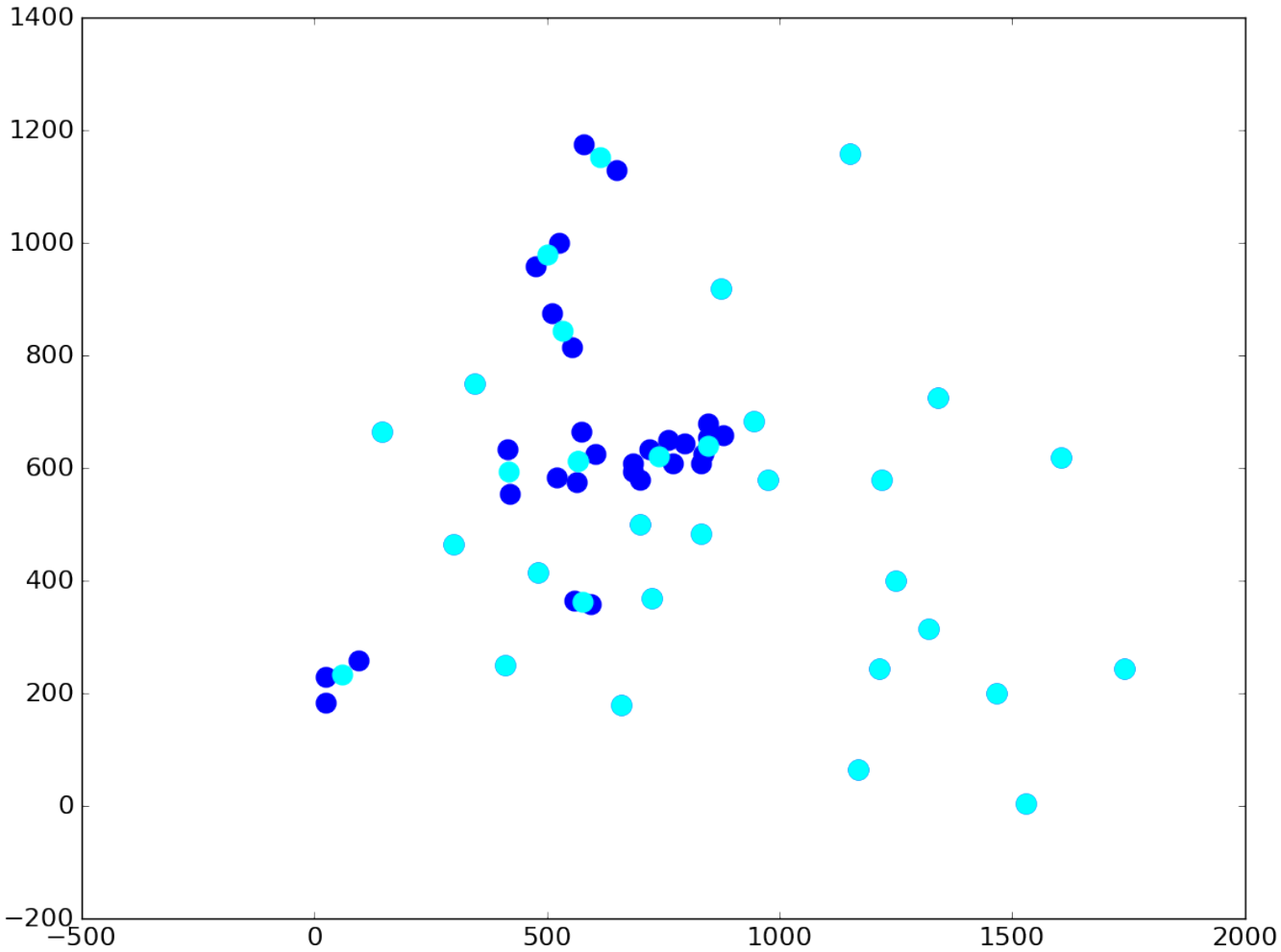
animation of the pair-center tour algorithm



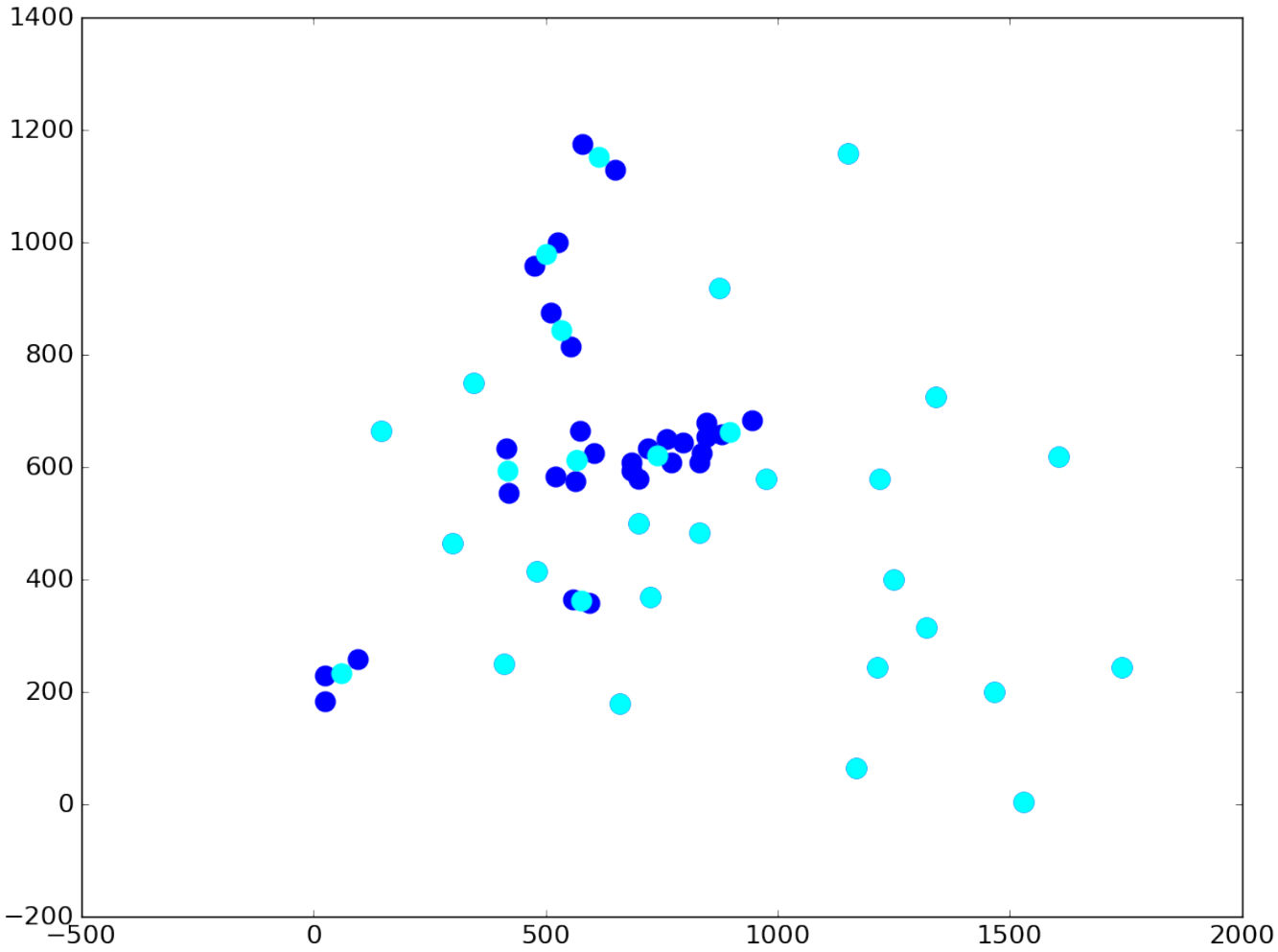
animation of the pair-center tour algorithm



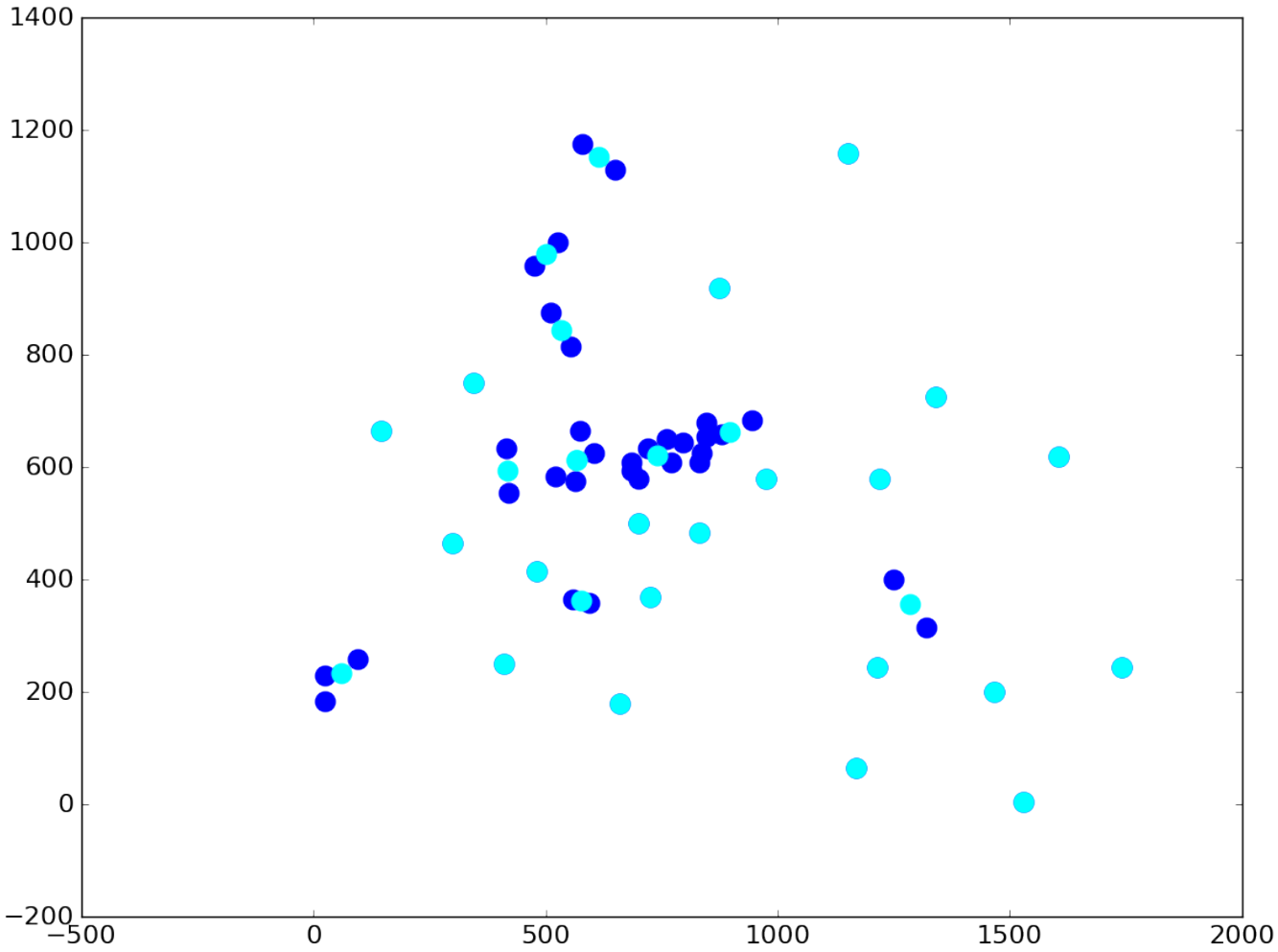
animation of the pair-center tour algorithm



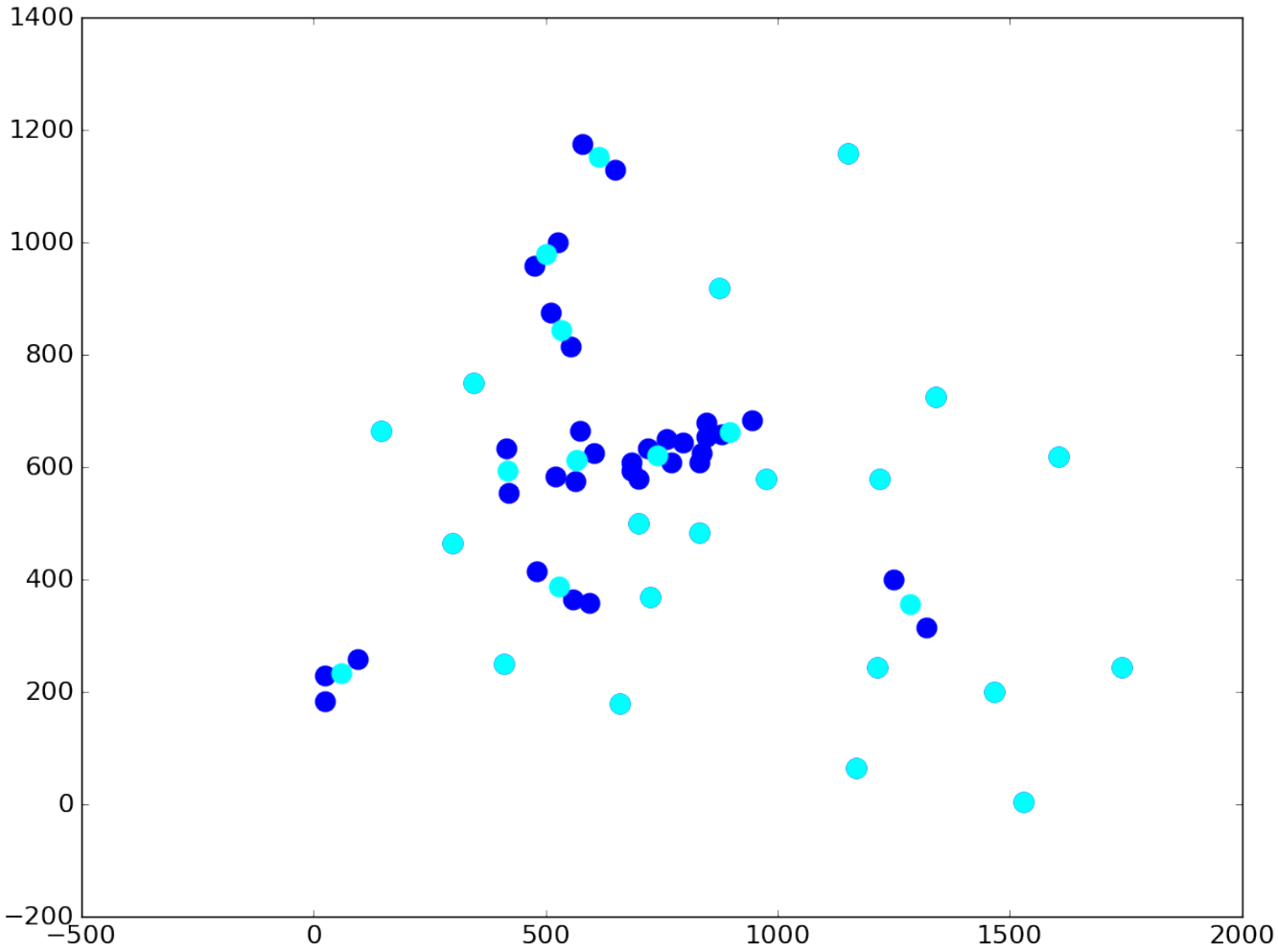
animation of the pair-center tour algorithm



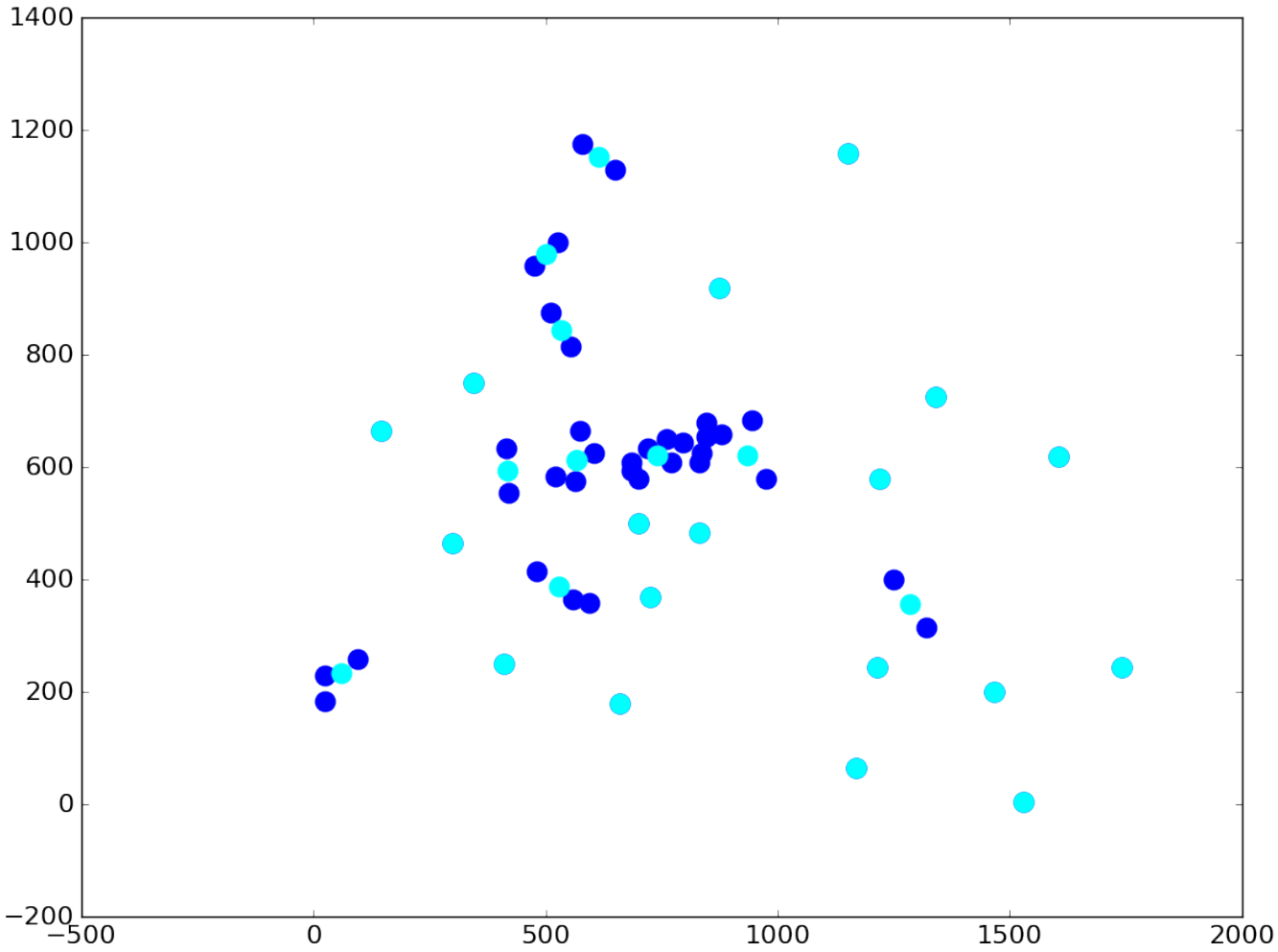
animation of the pair-center tour algorithm



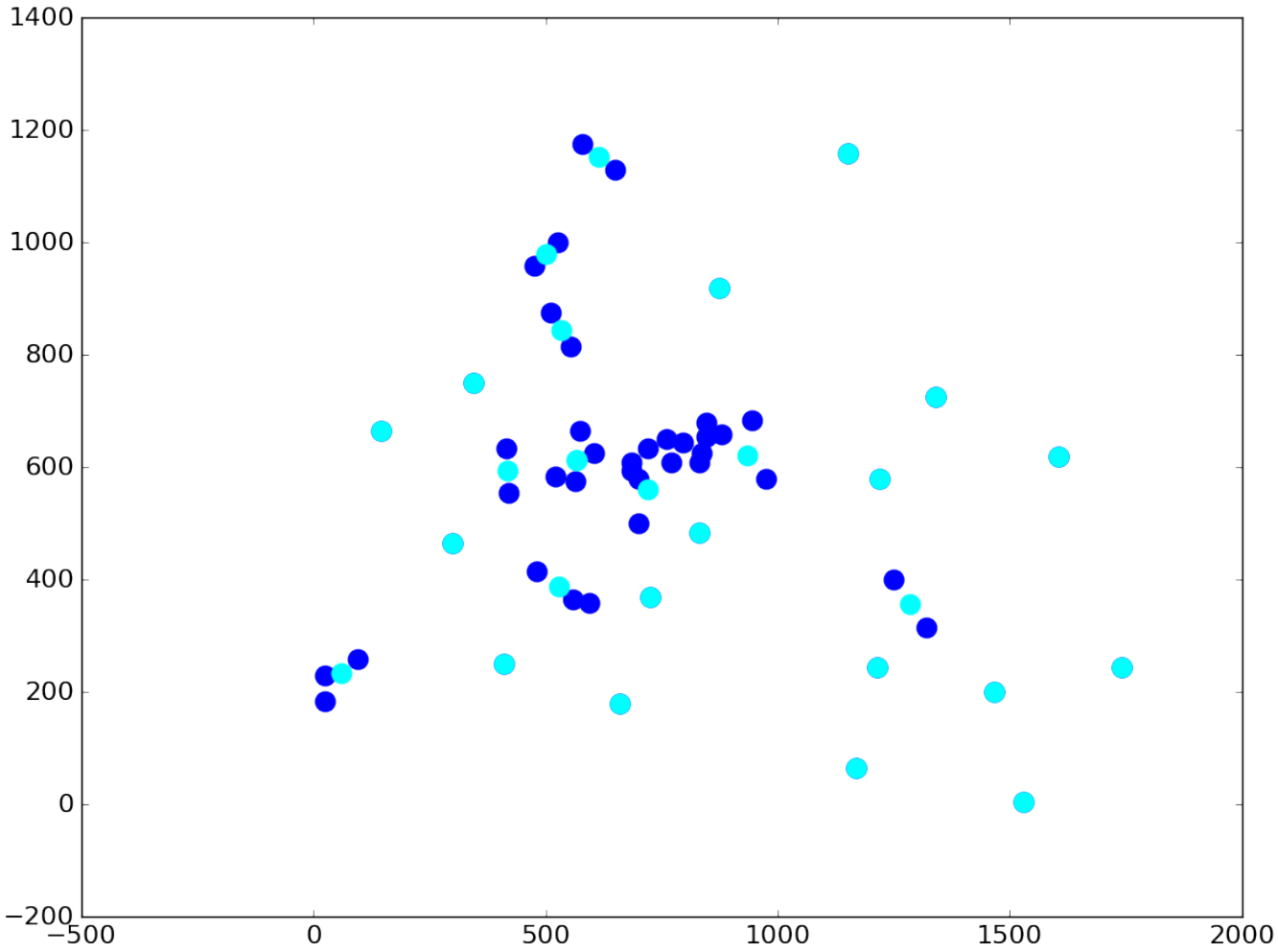
animation of the pair-center tour algorithm



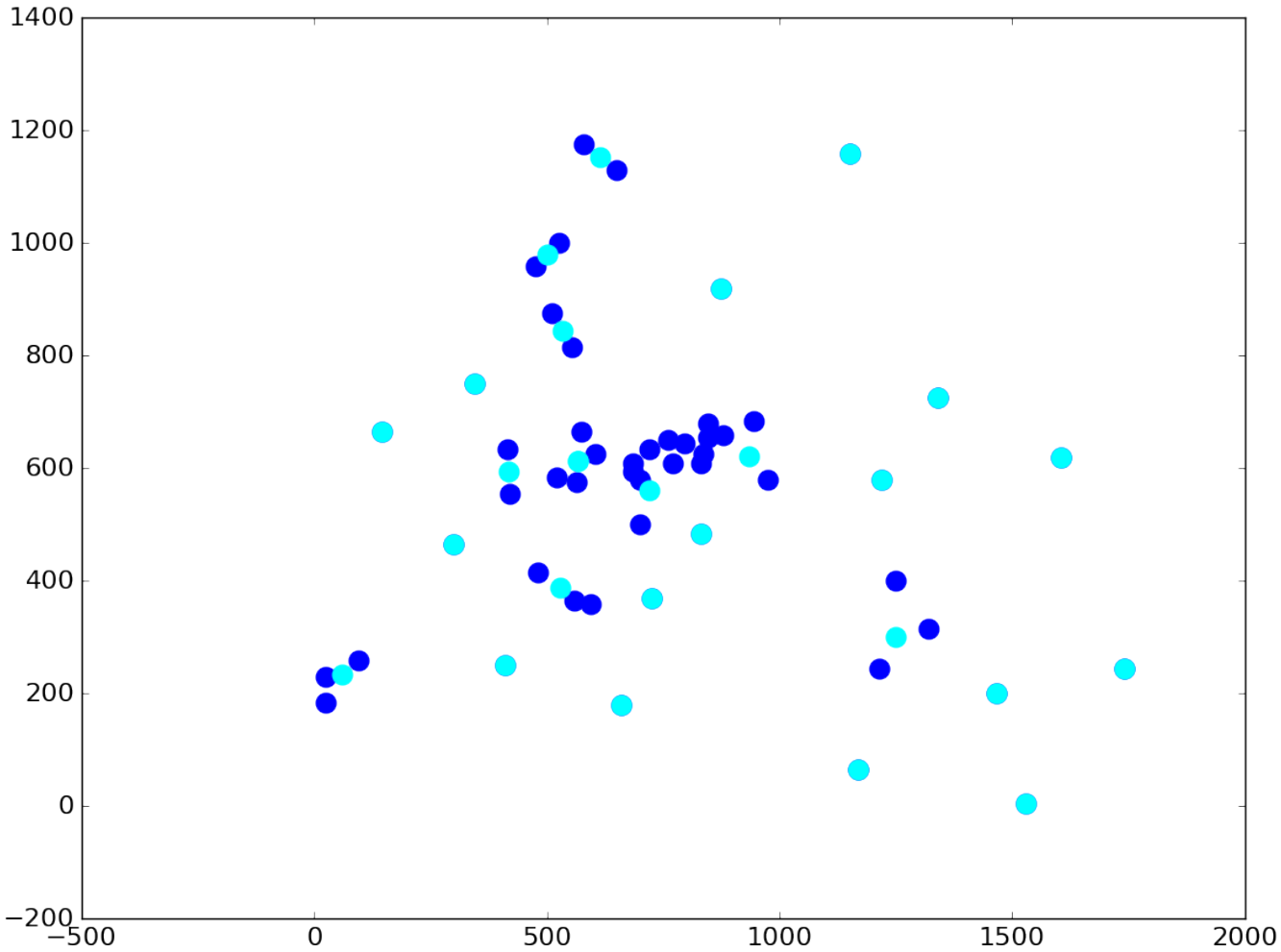
animation of the pair-center tour algorithm



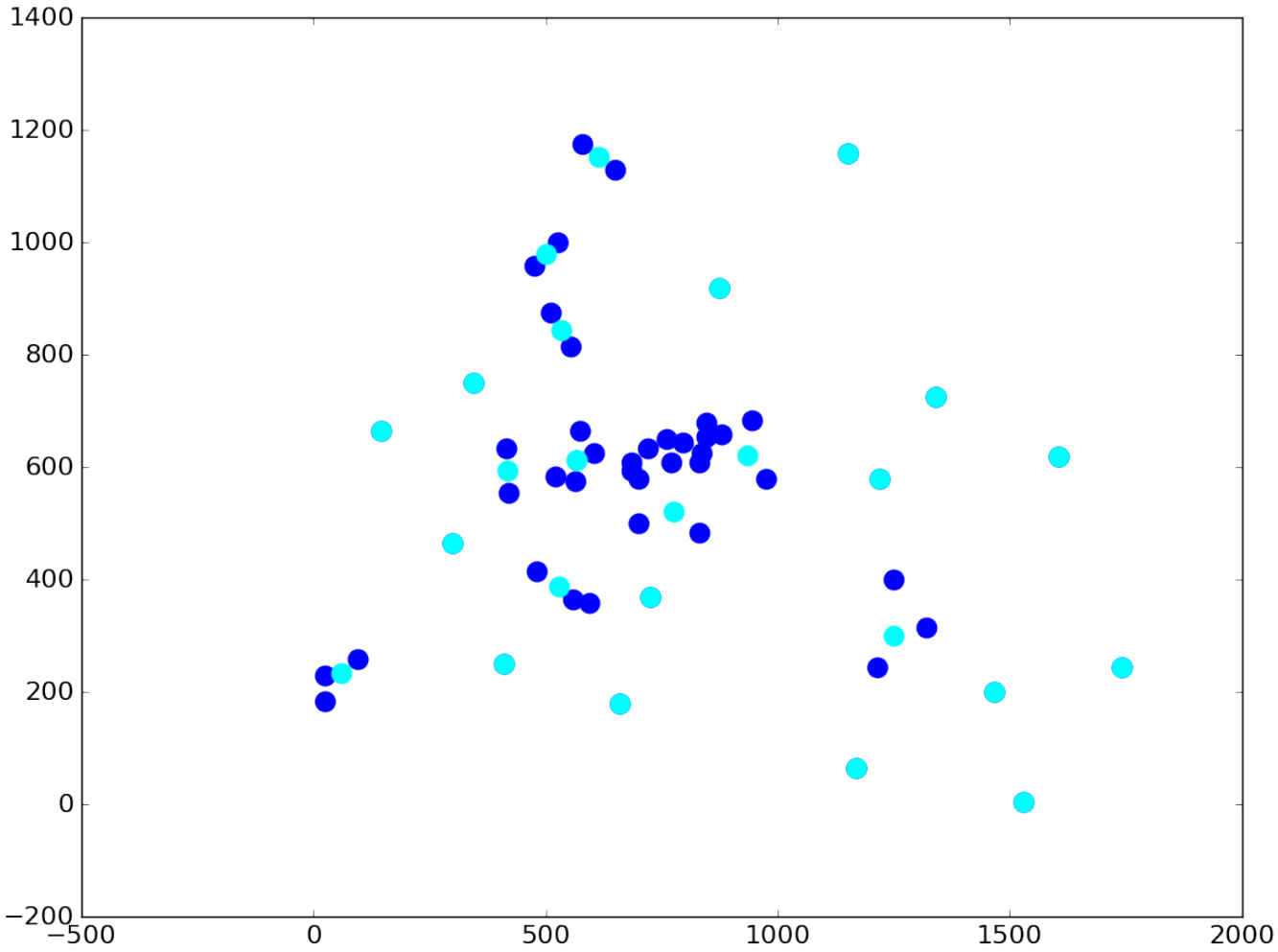
animation of the pair-center tour algorithm



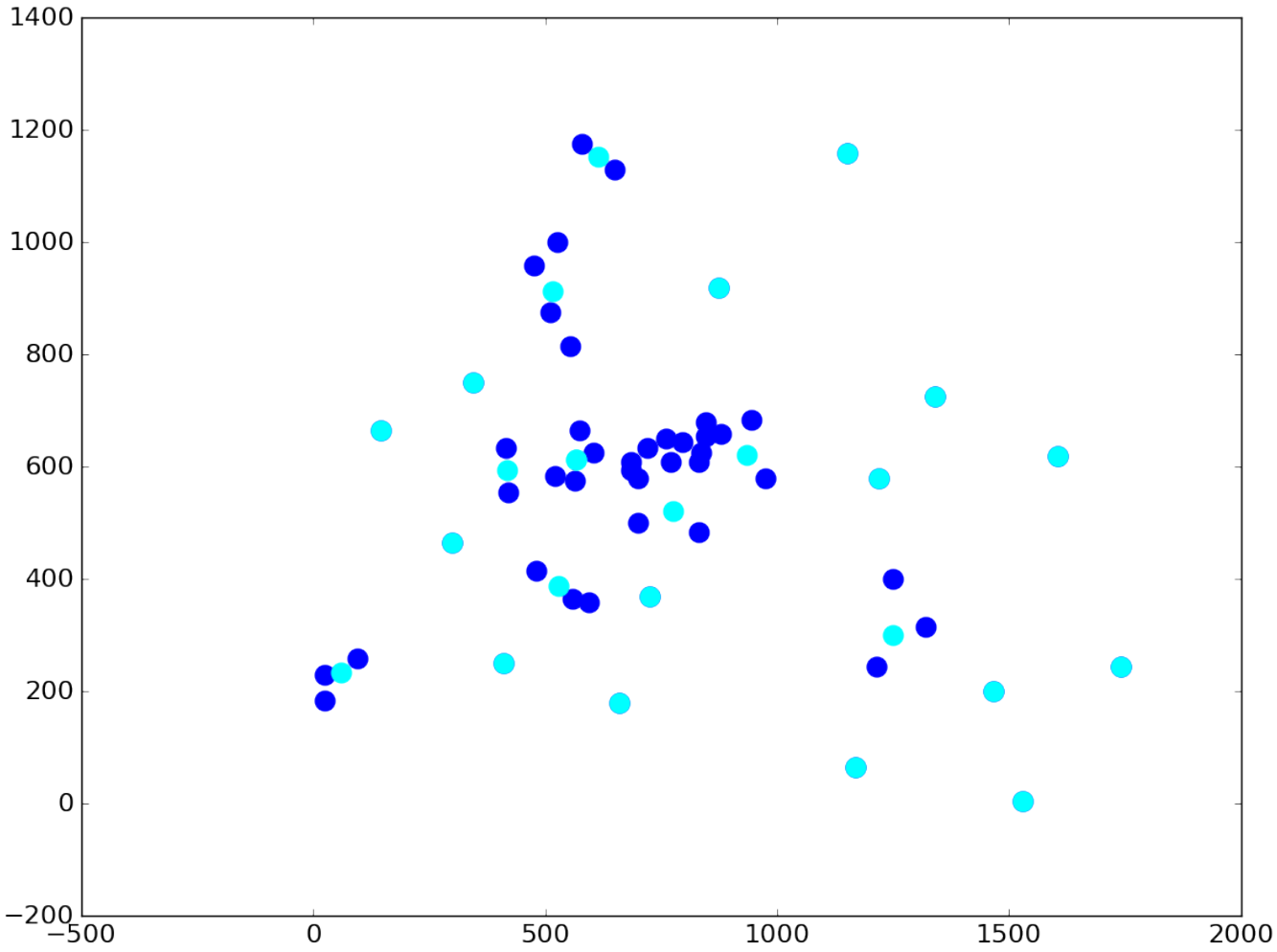
animation of the pair-center tour algorithm



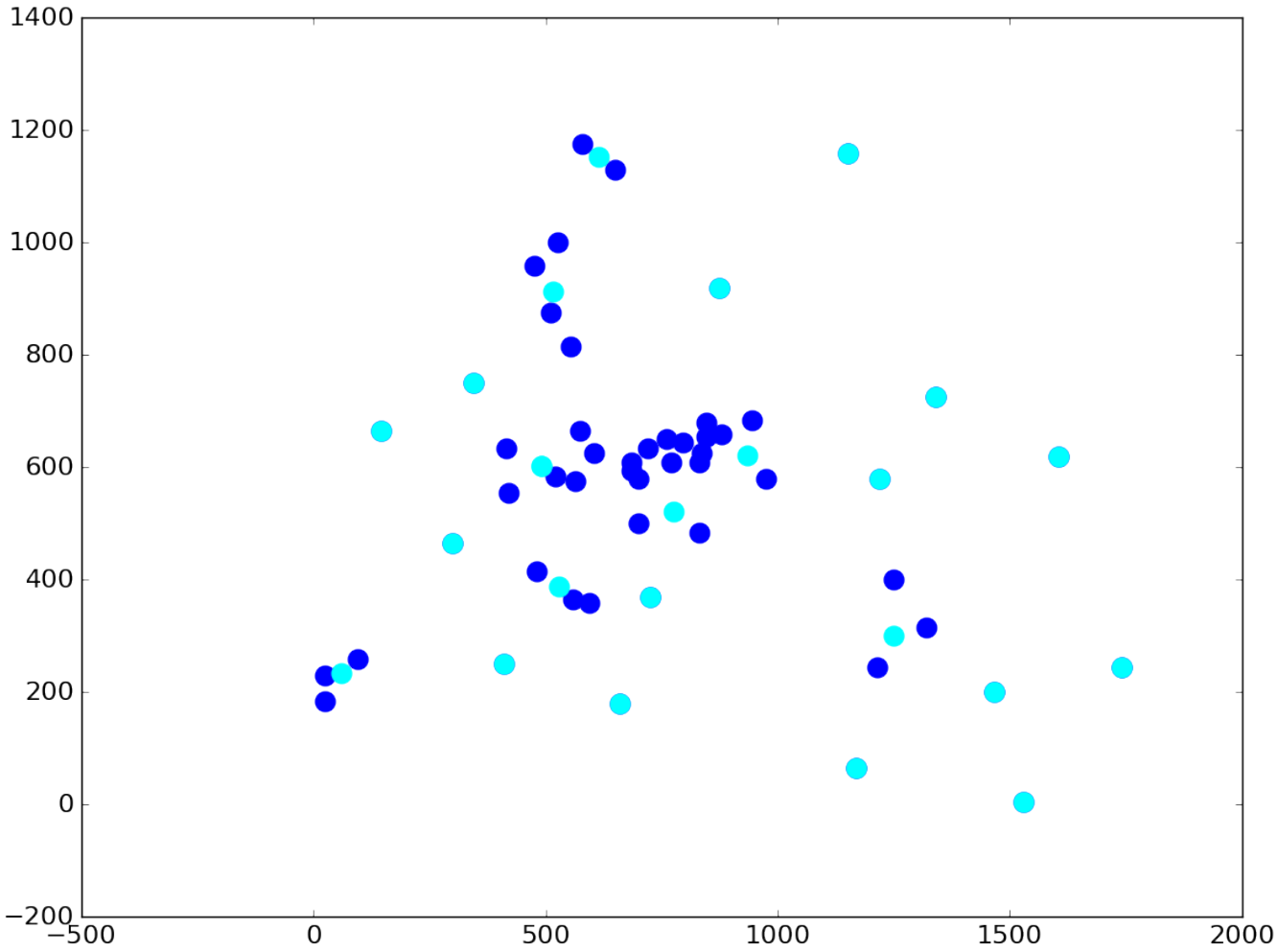
animation of the pair-center tour algorithm



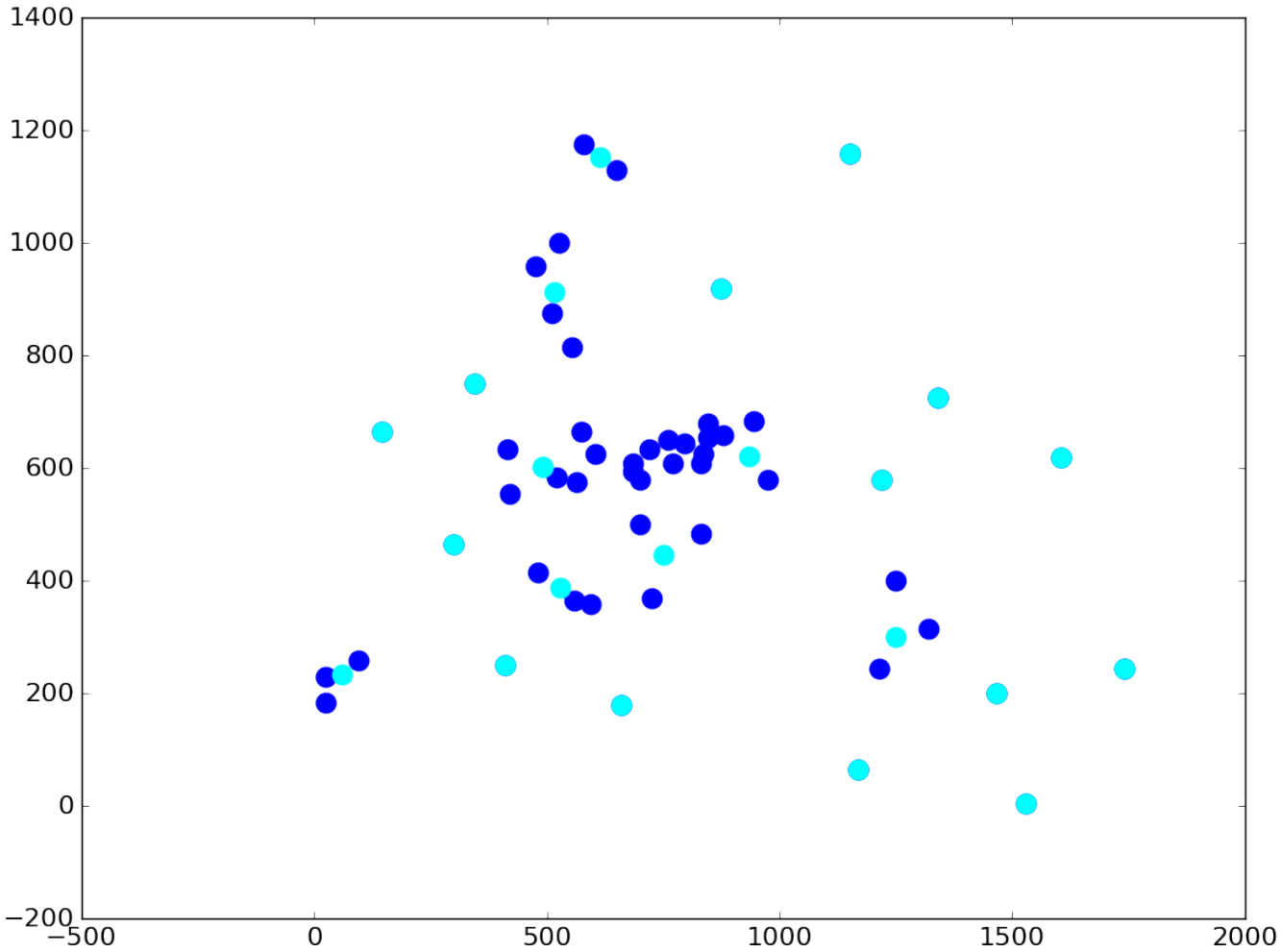
animation of the pair-center tour algorithm



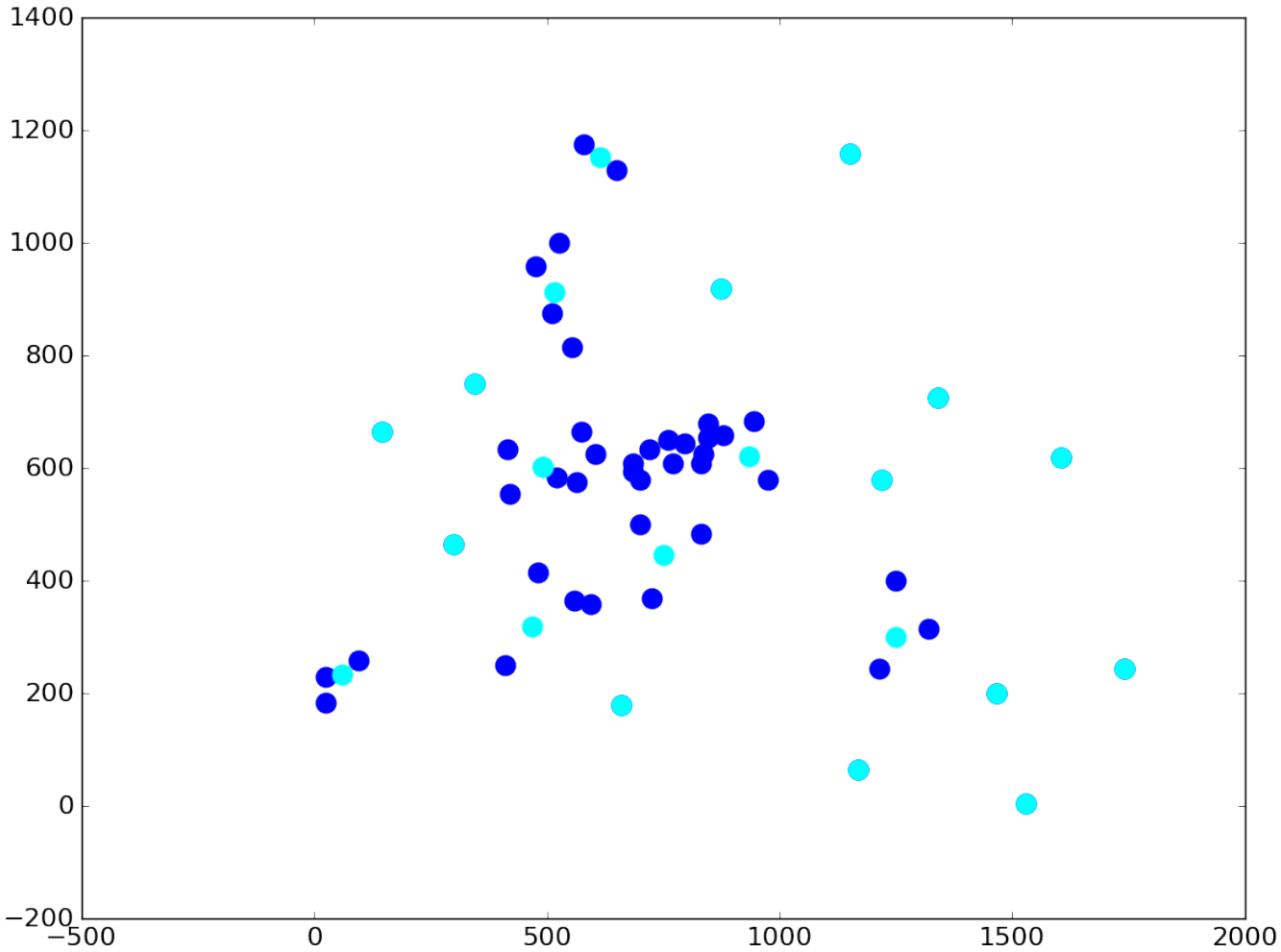
animation of the pair-center tour algorithm



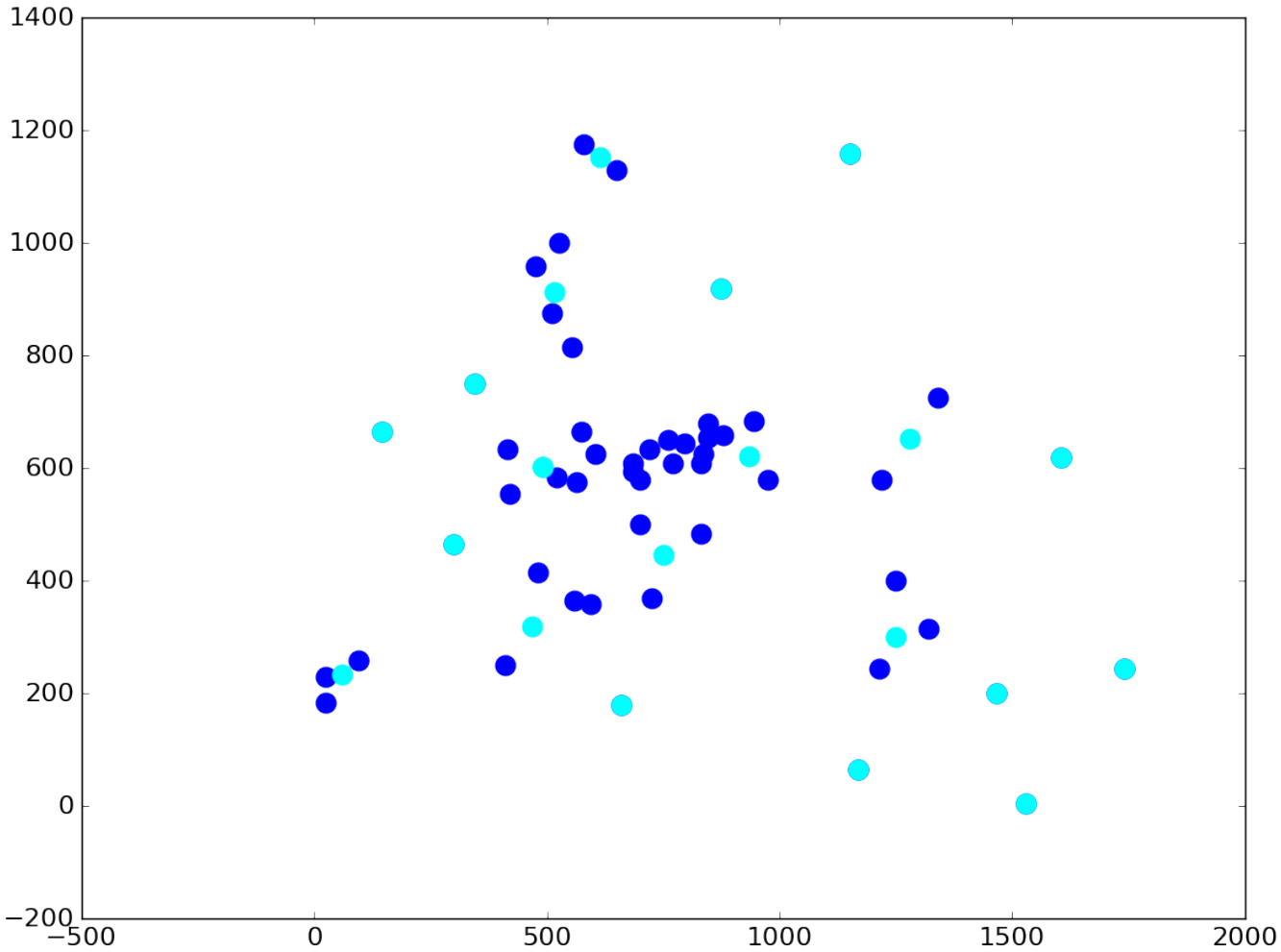
animation of the pair-center tour algorithm



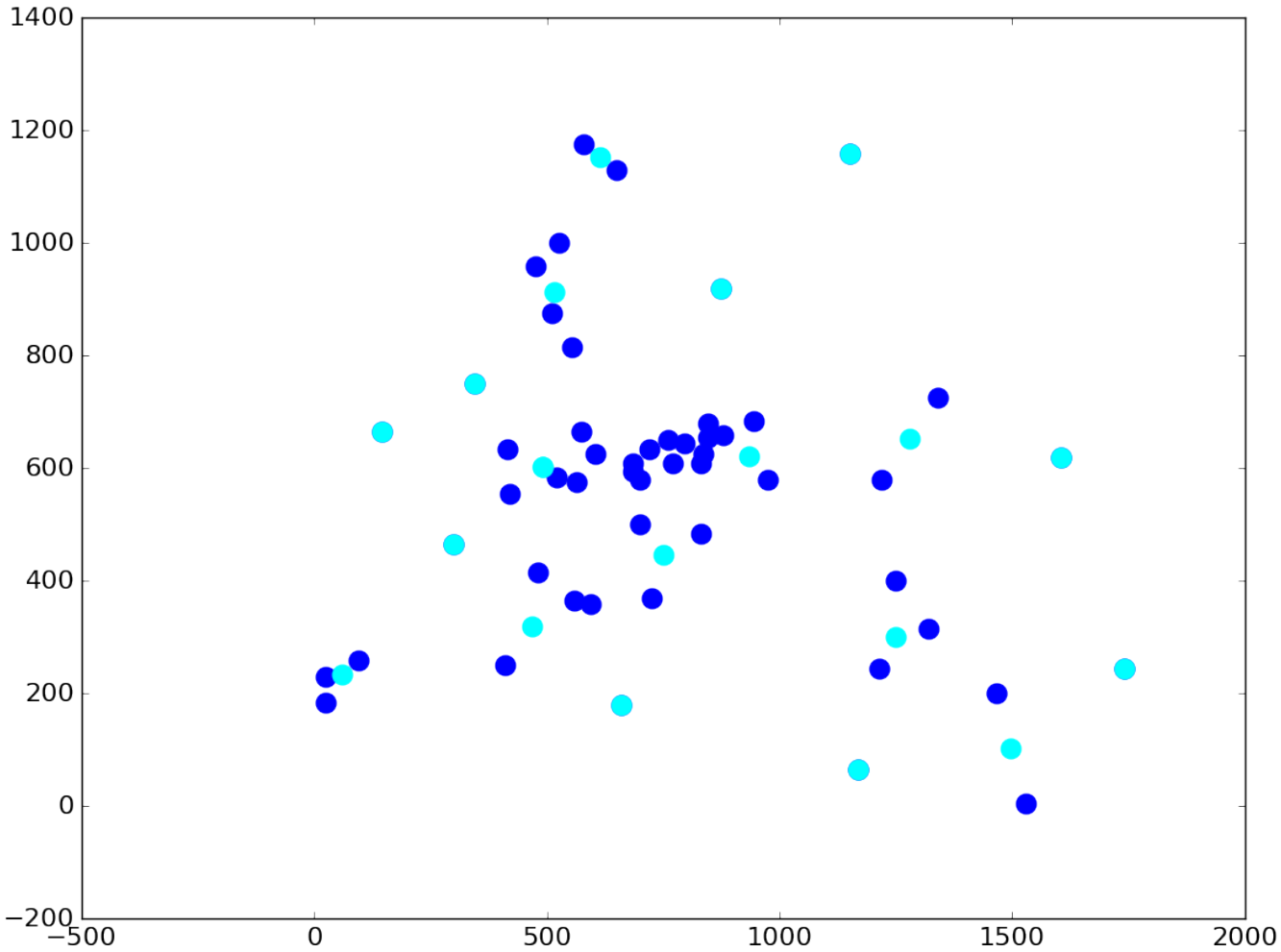
animation of the pair-center tour algorithm



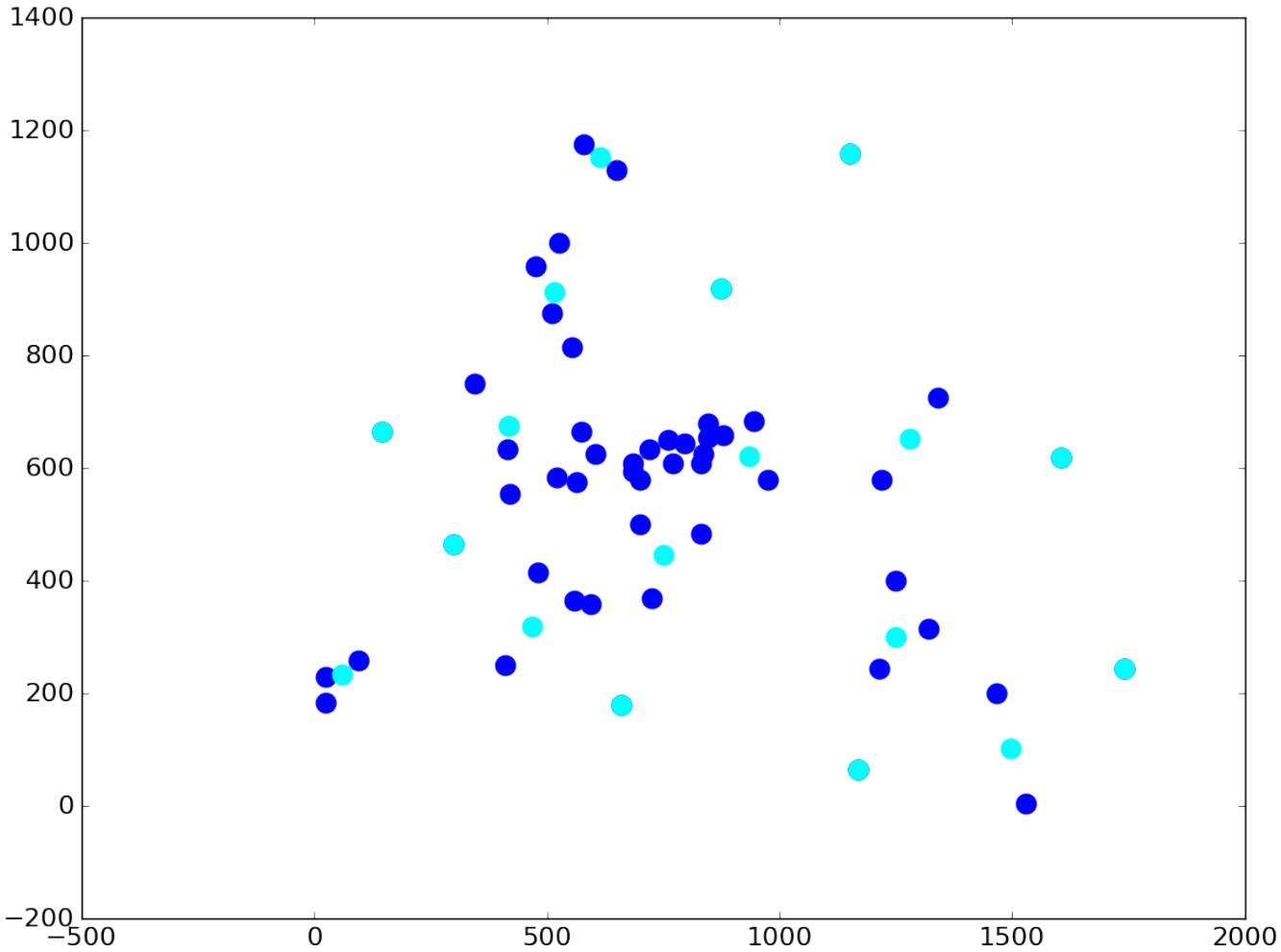
animation of the pair-center tour algorithm



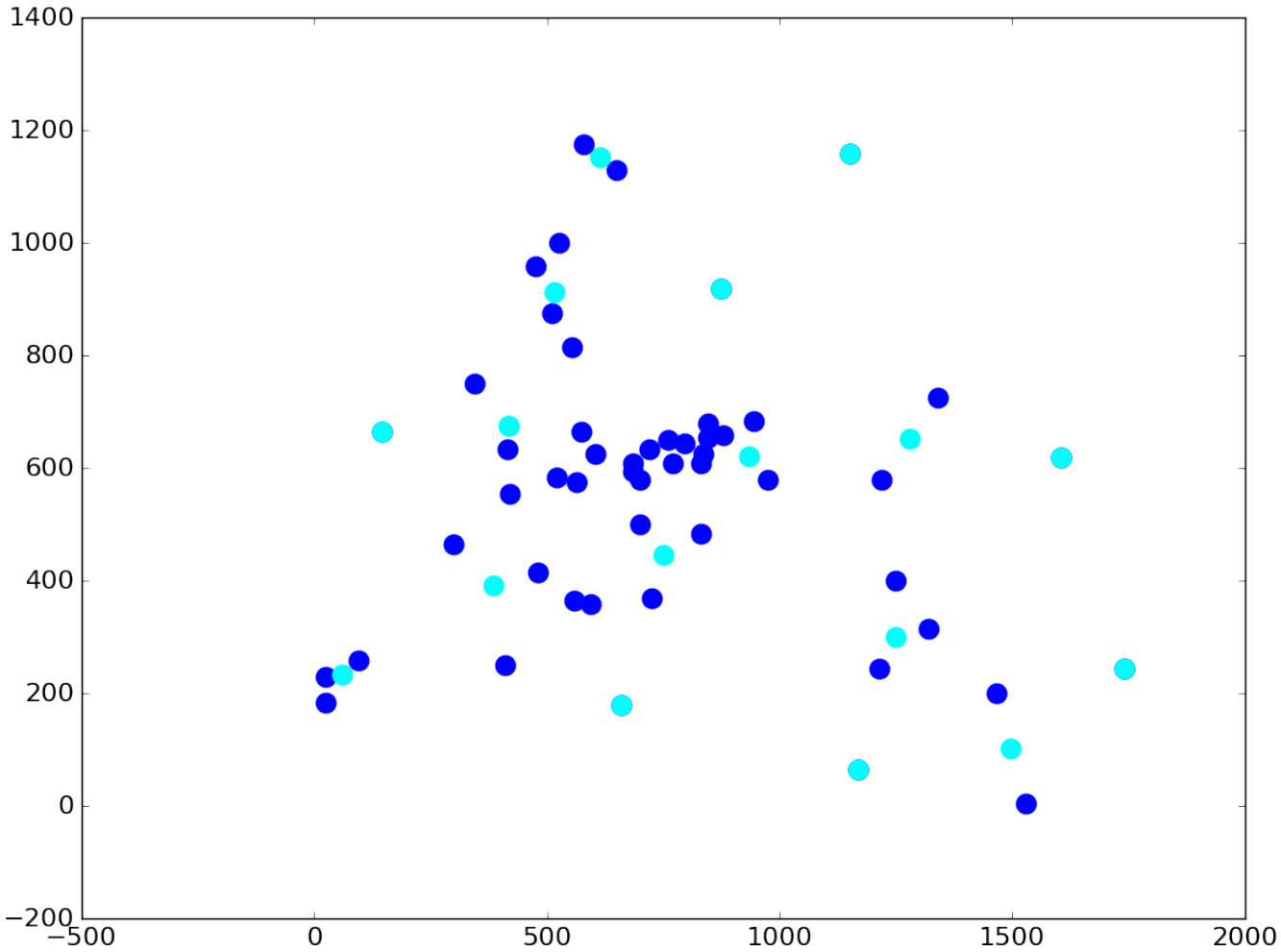
animation of the pair-center tour algorithm



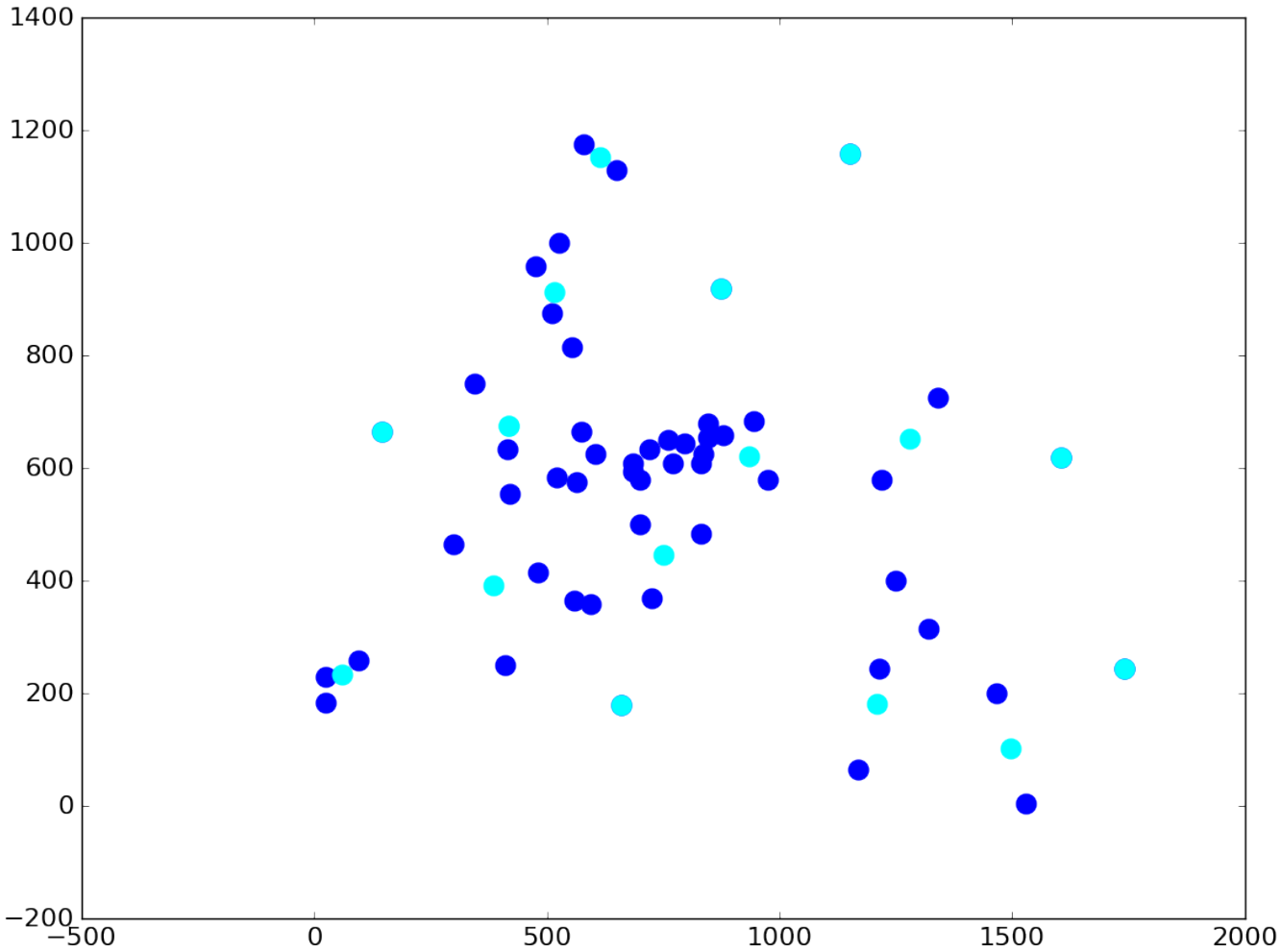
animation of the pair-center tour algorithm



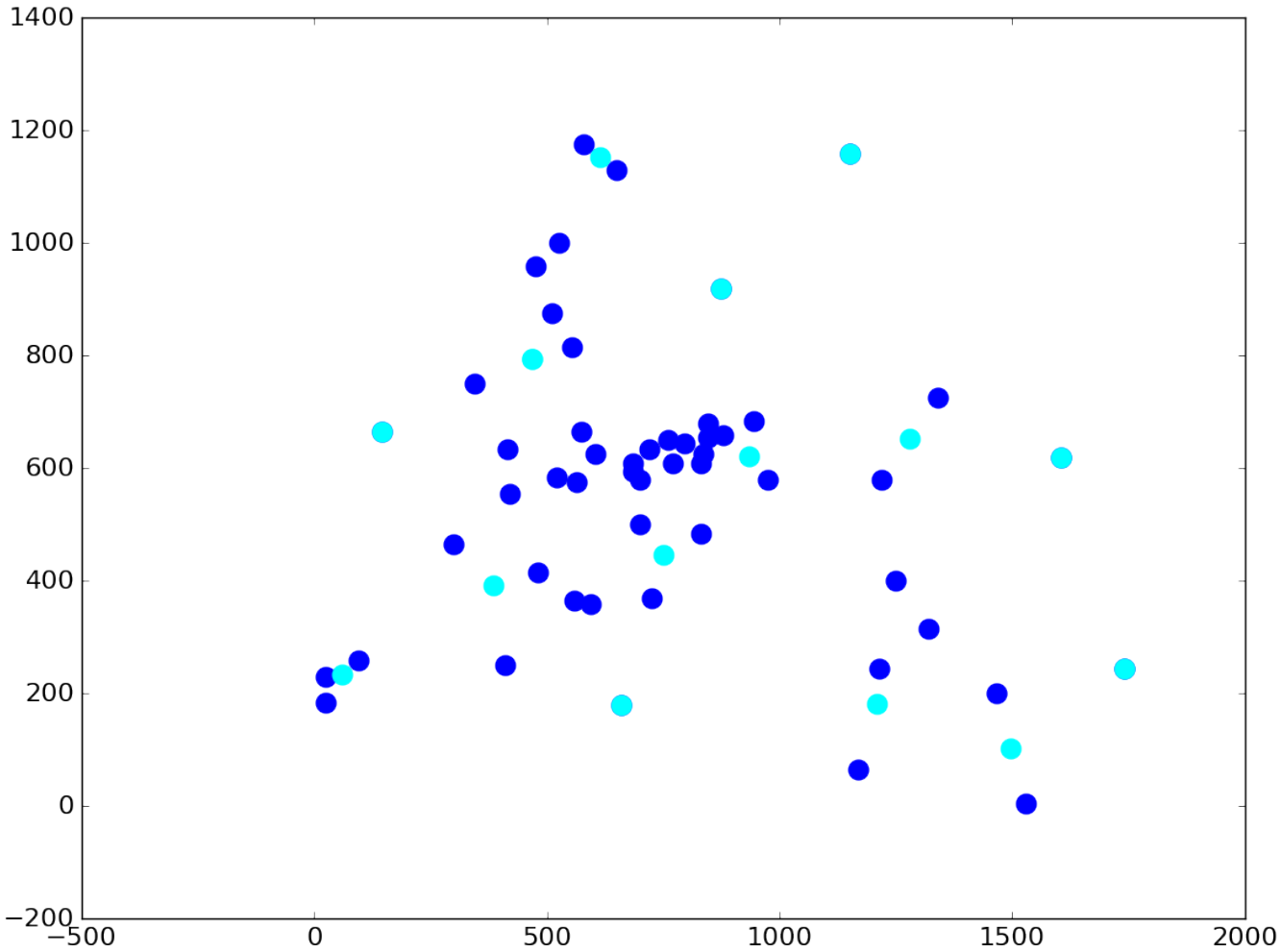
animation of the pair-center tour algorithm



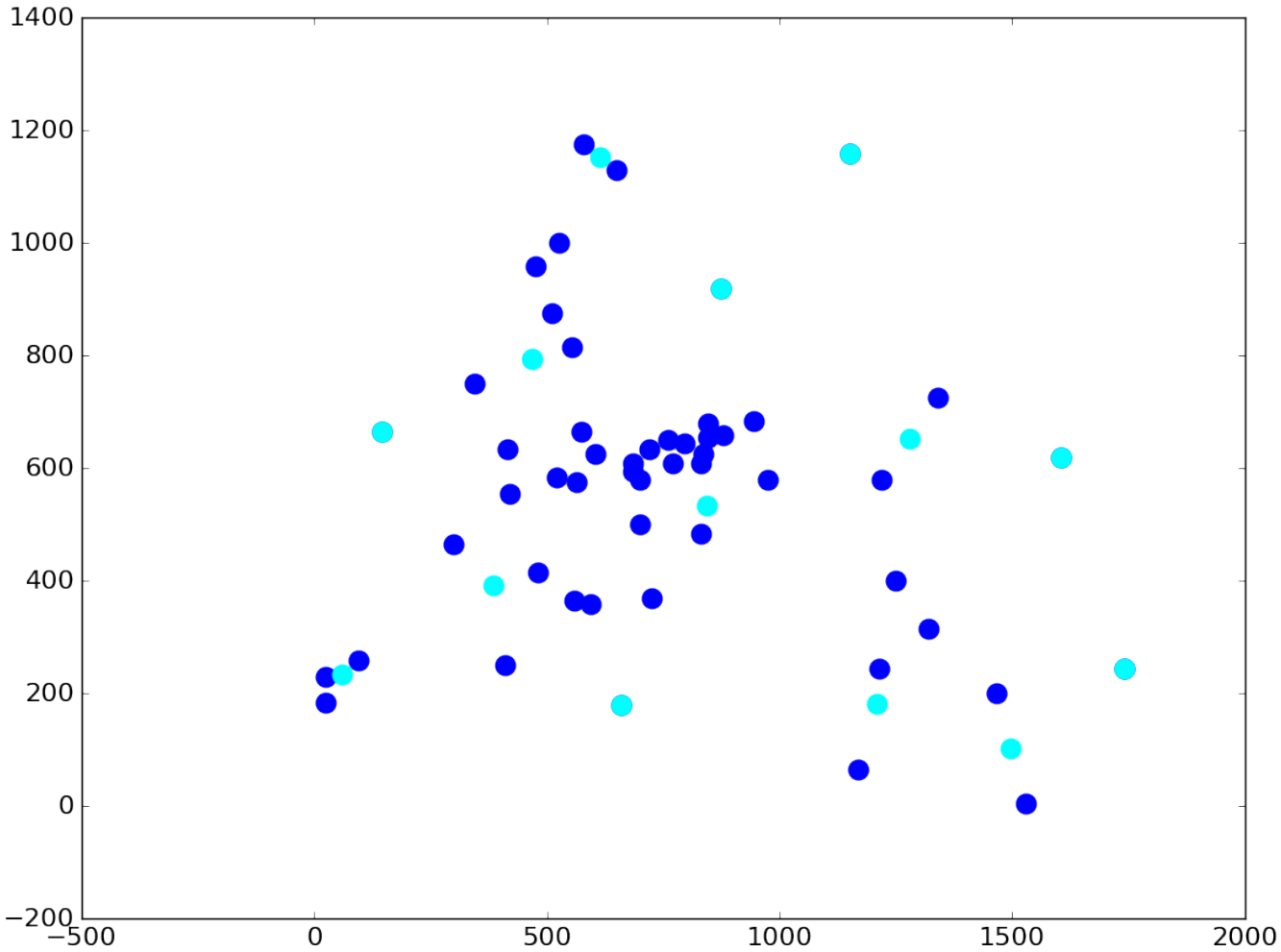
animation of the pair-center tour algorithm



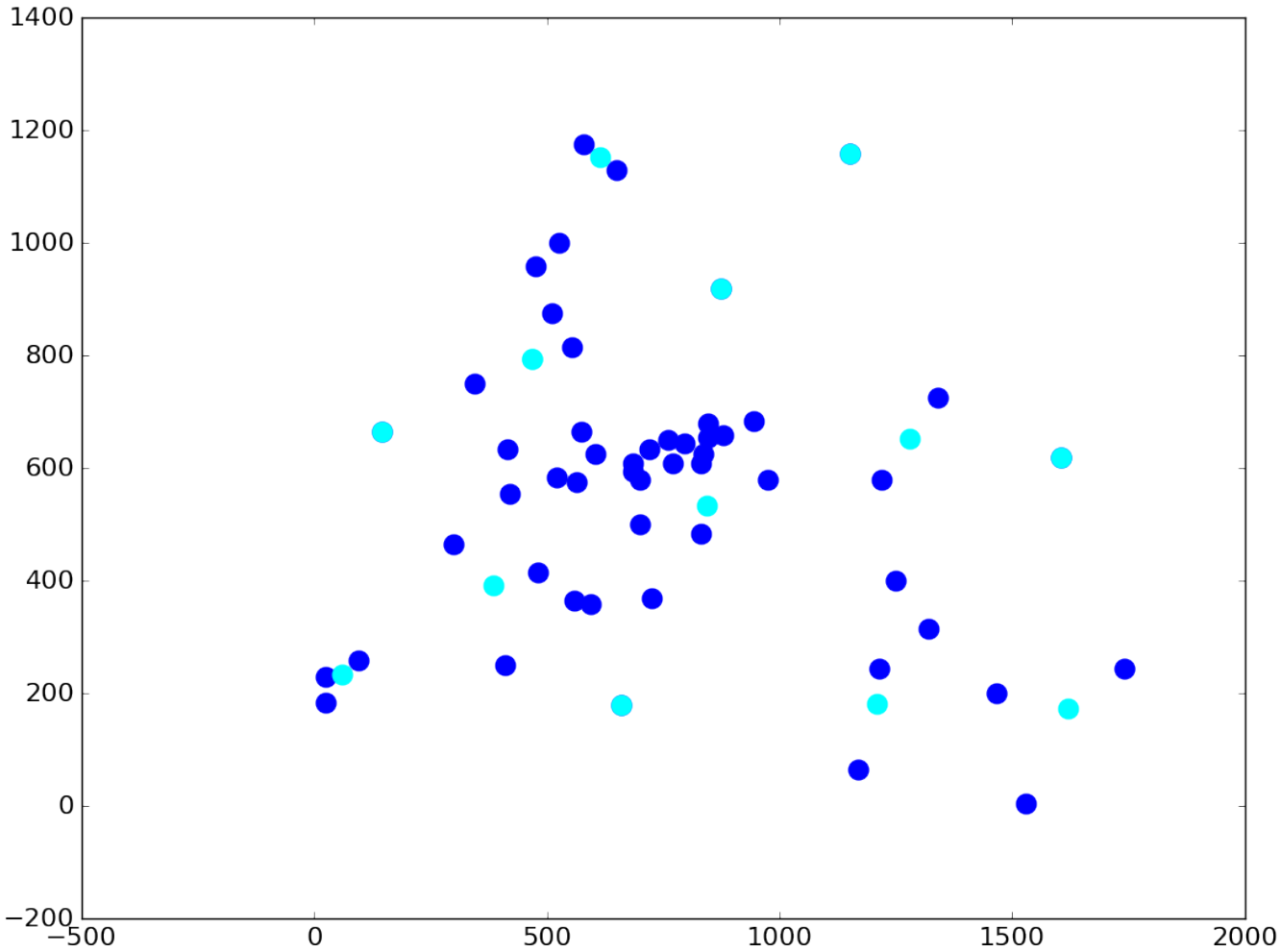
animation of the pair-center tour algorithm



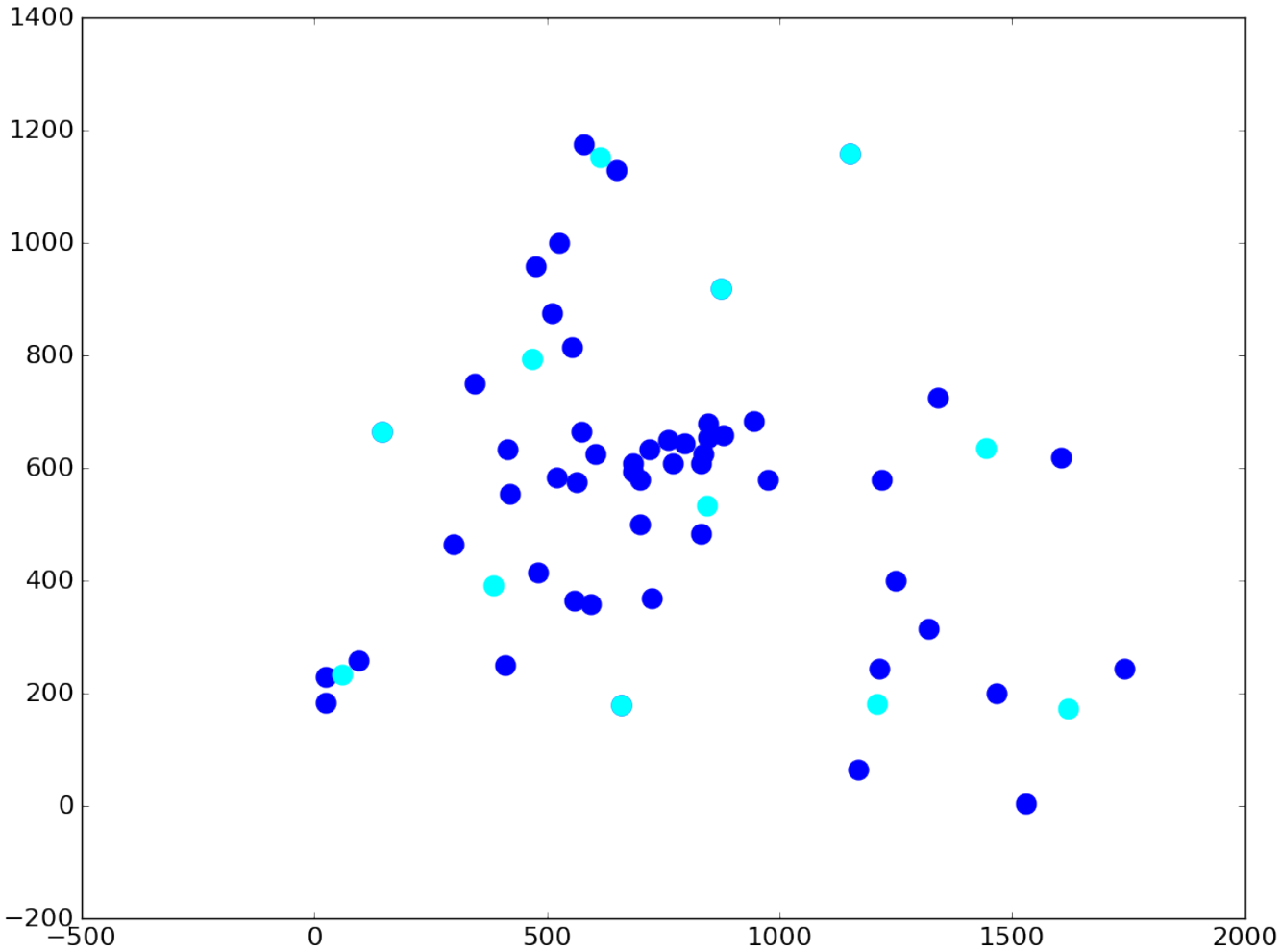
animation of the pair-center tour algorithm



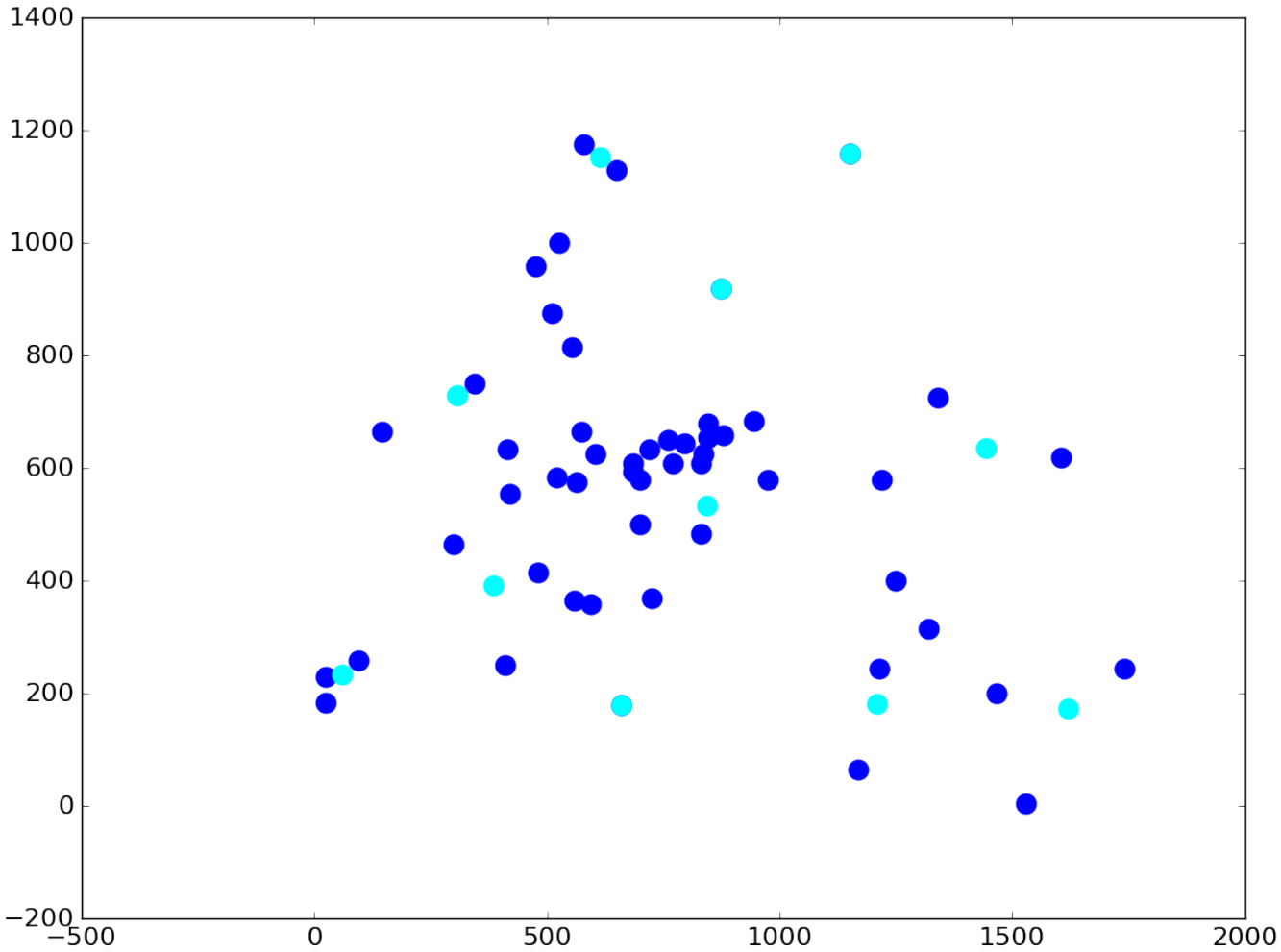
animation of the pair-center tour algorithm



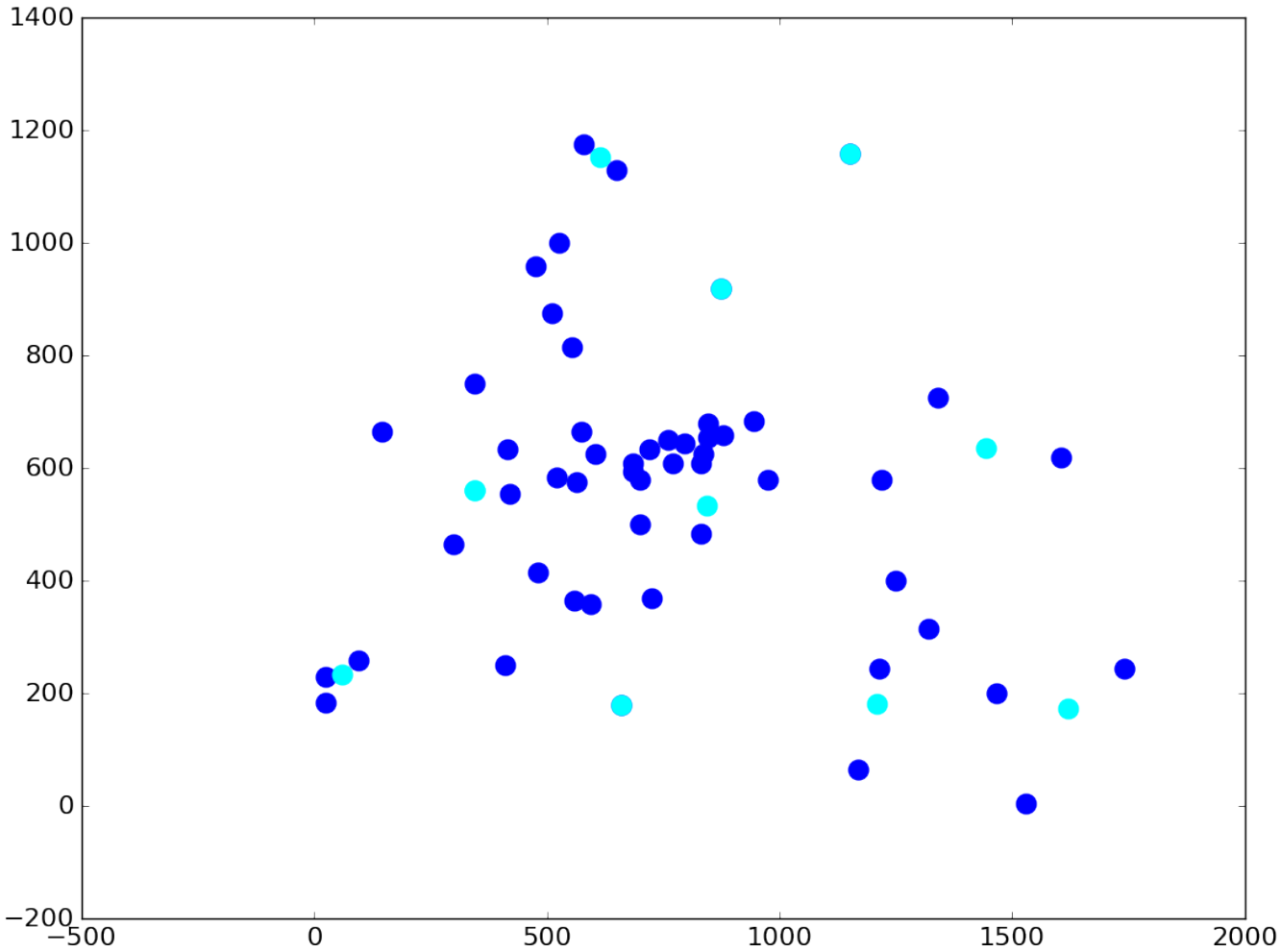
animation of the pair-center tour algorithm



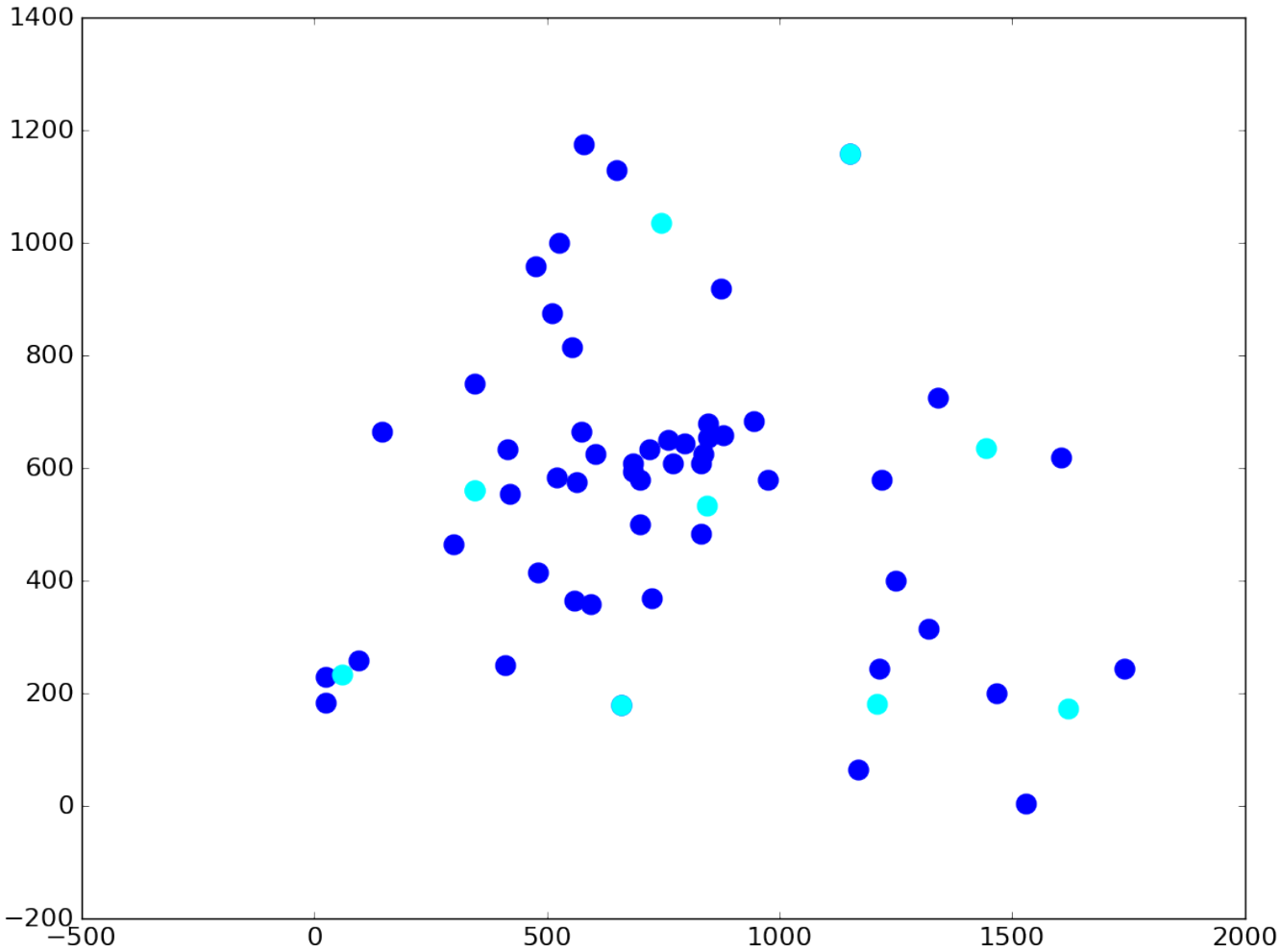
animation of the pair-center tour algorithm



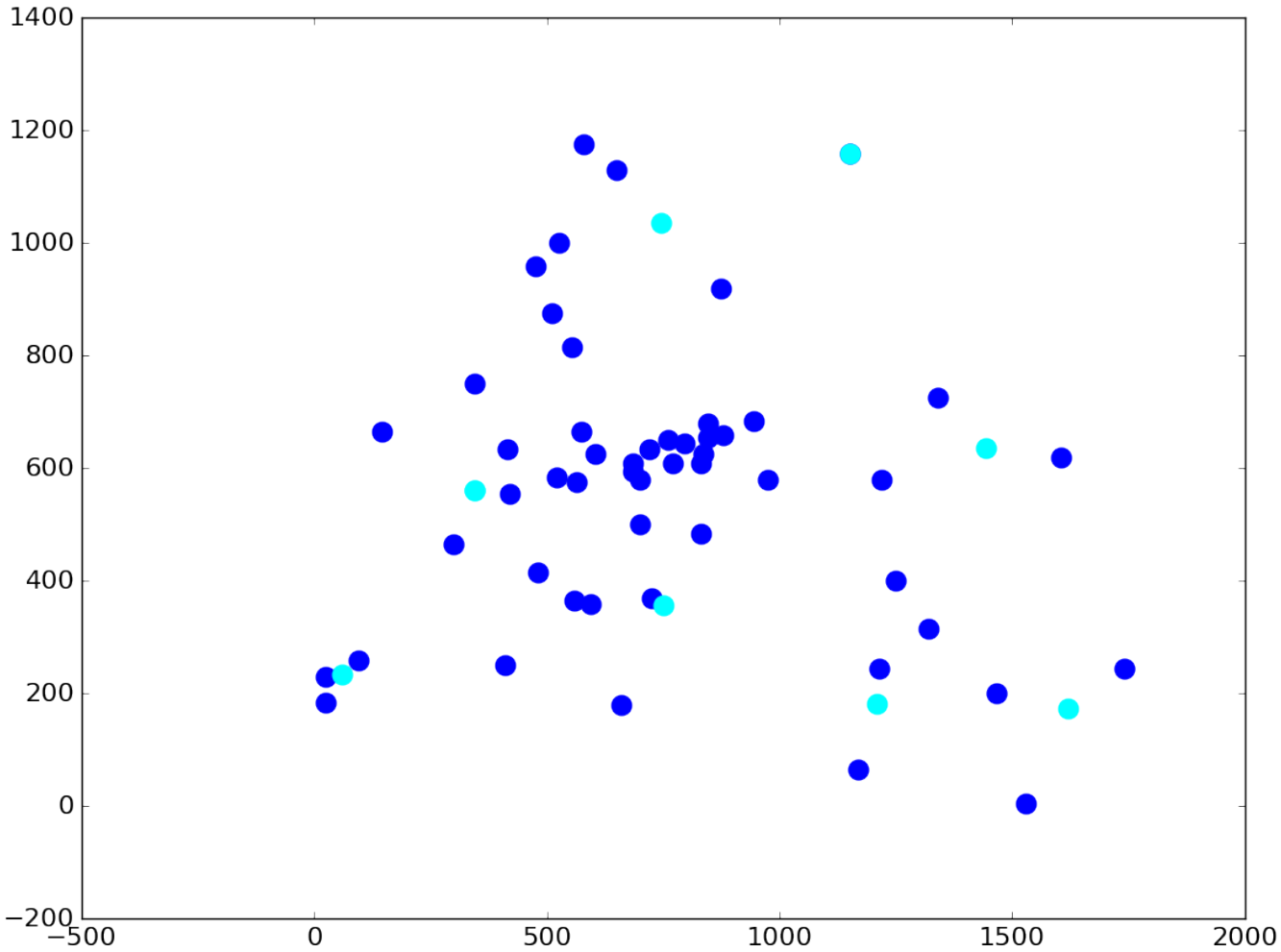
animation of the pair-center tour algorithm



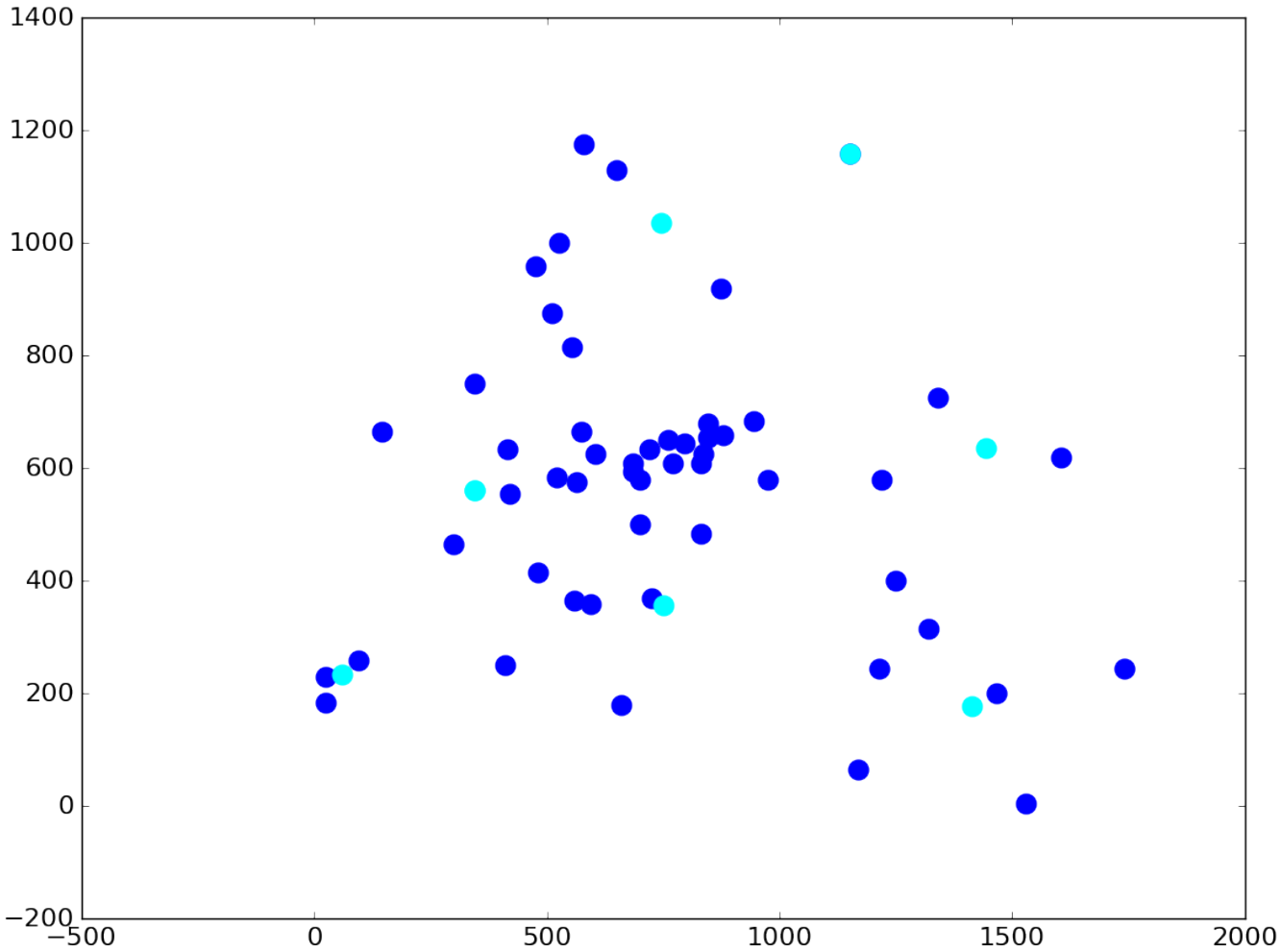
animation of the pair-center tour algorithm



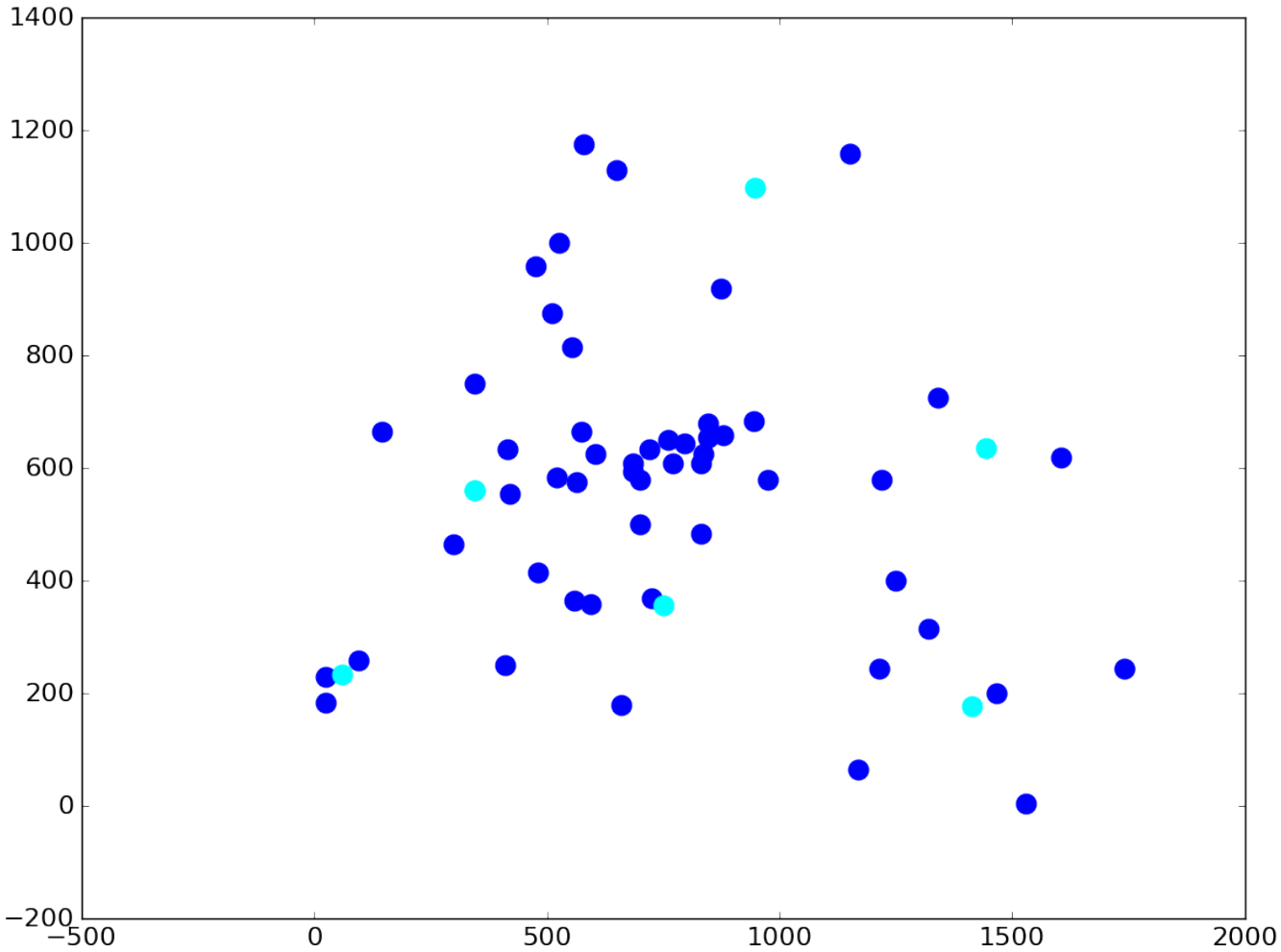
animation of the pair-center tour algorithm



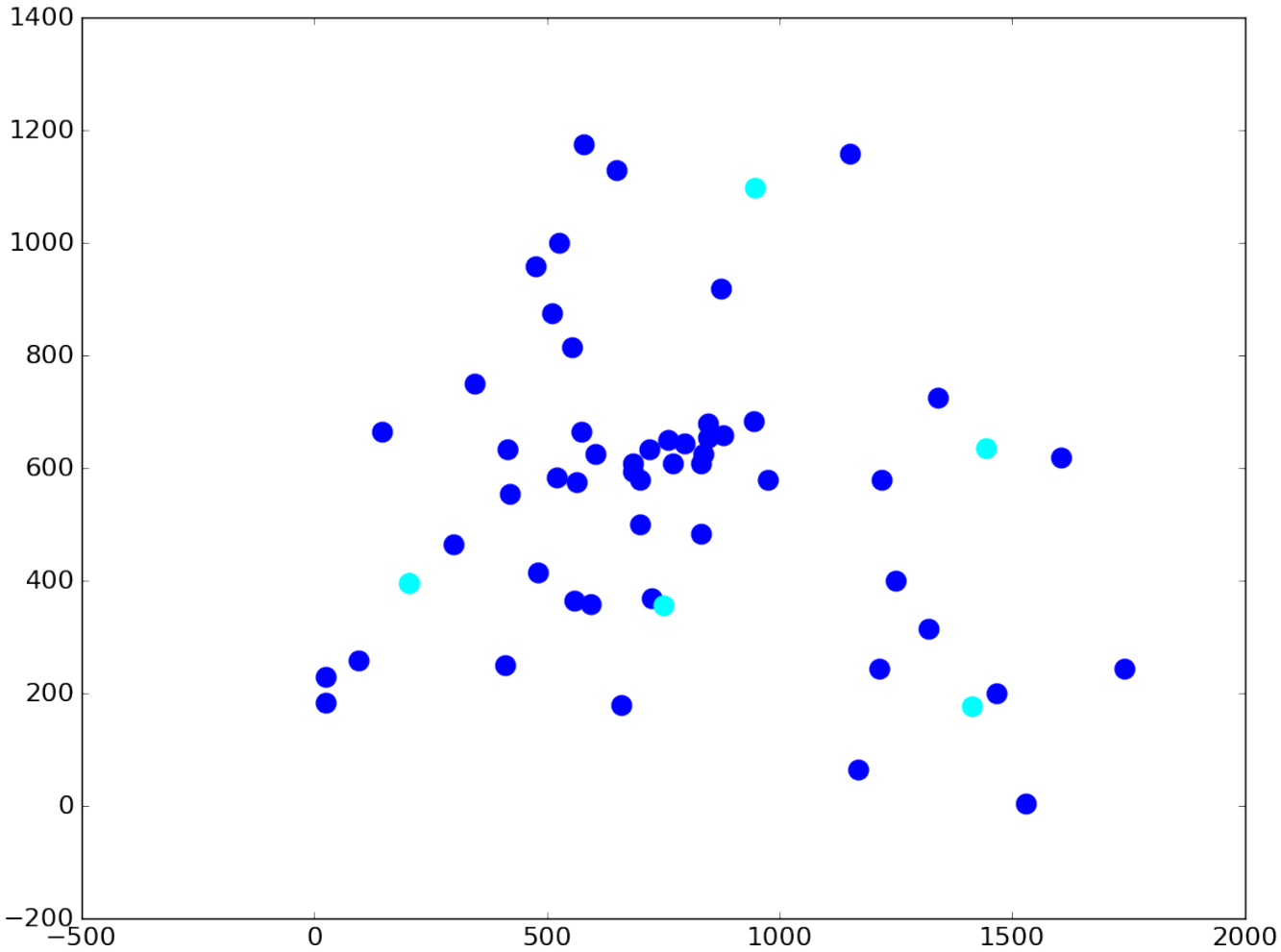
animation of the pair-center tour algorithm



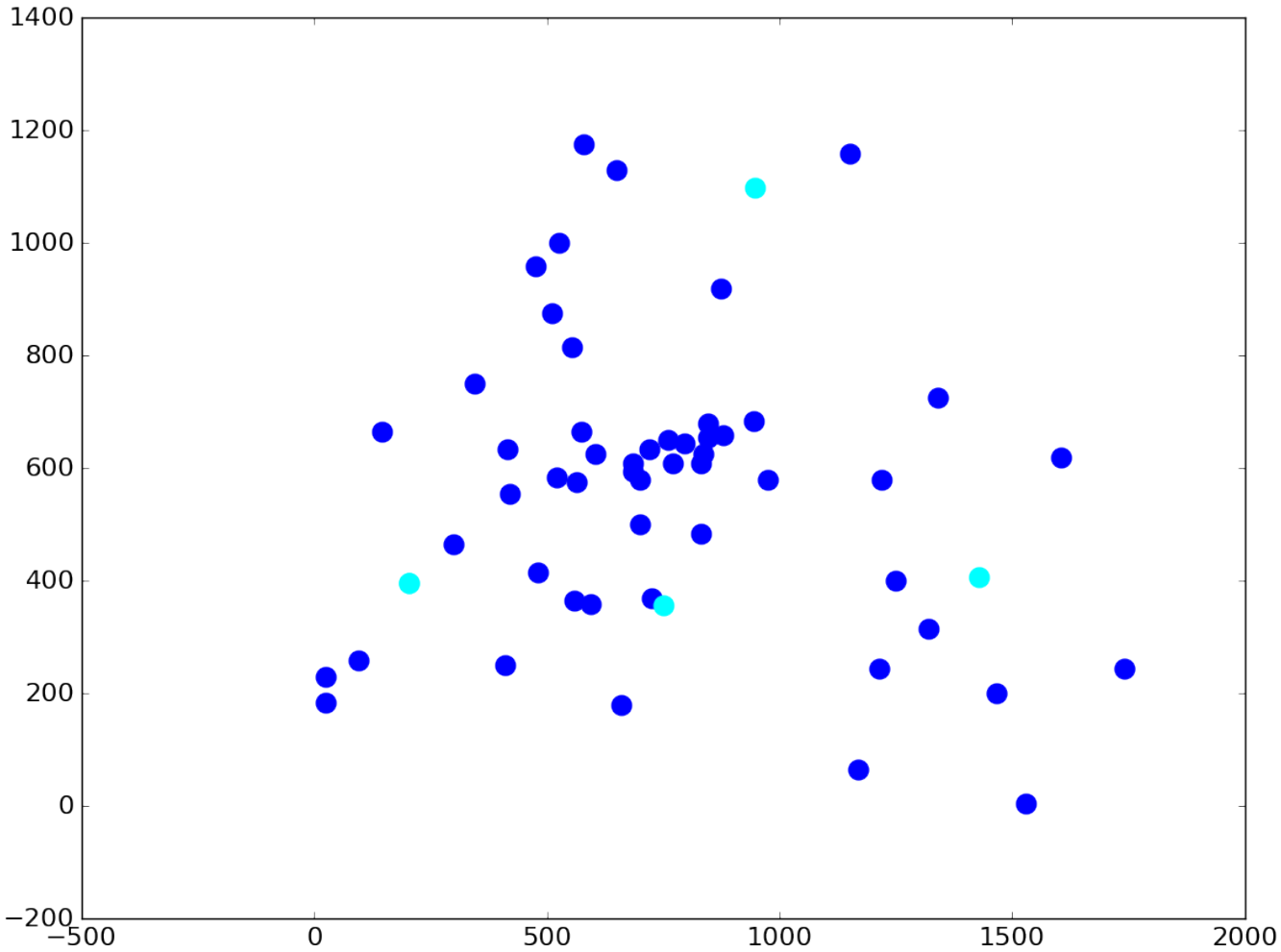
animation of the pair-center tour algorithm



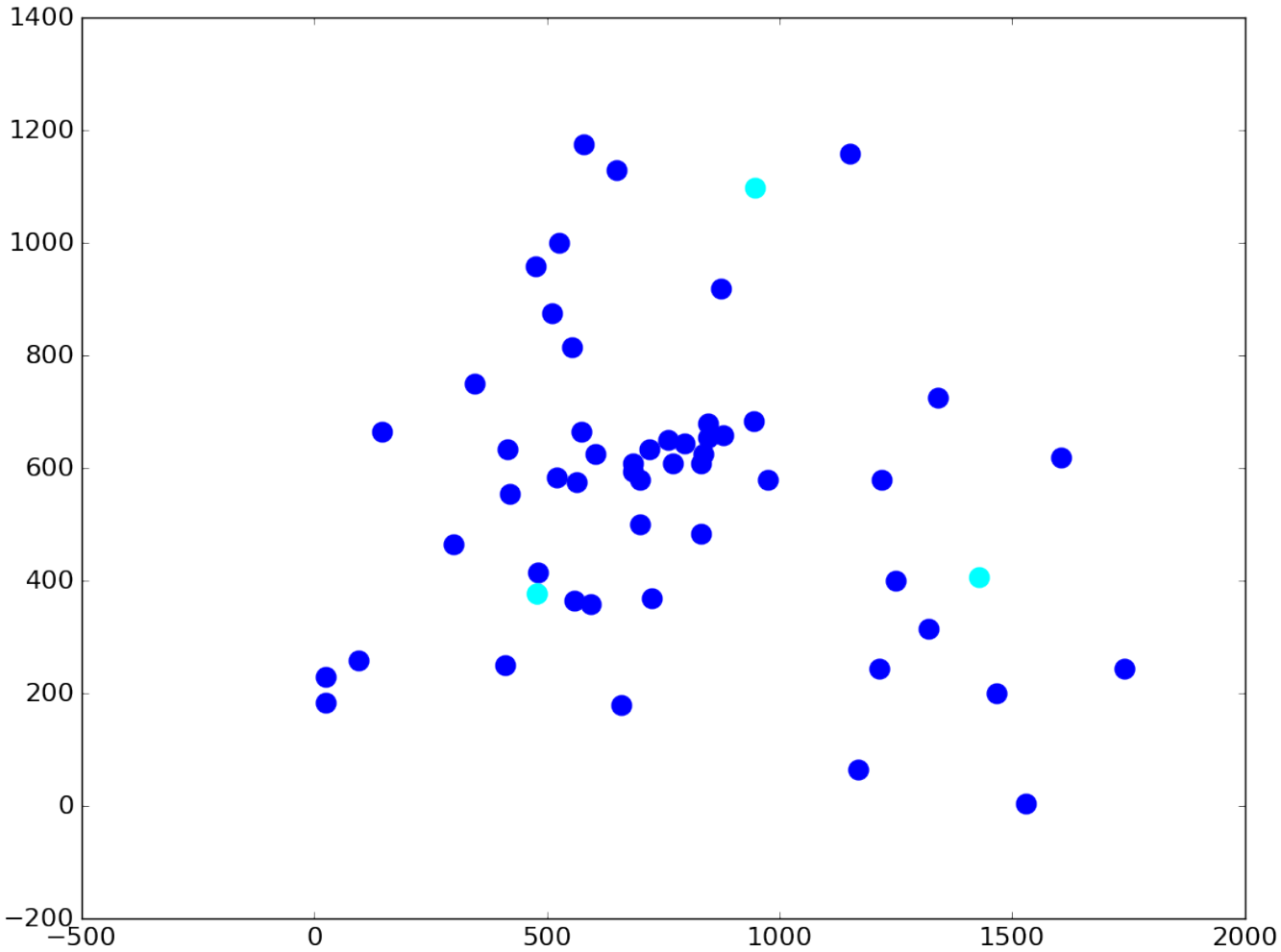
animation of the pair-center tour algorithm



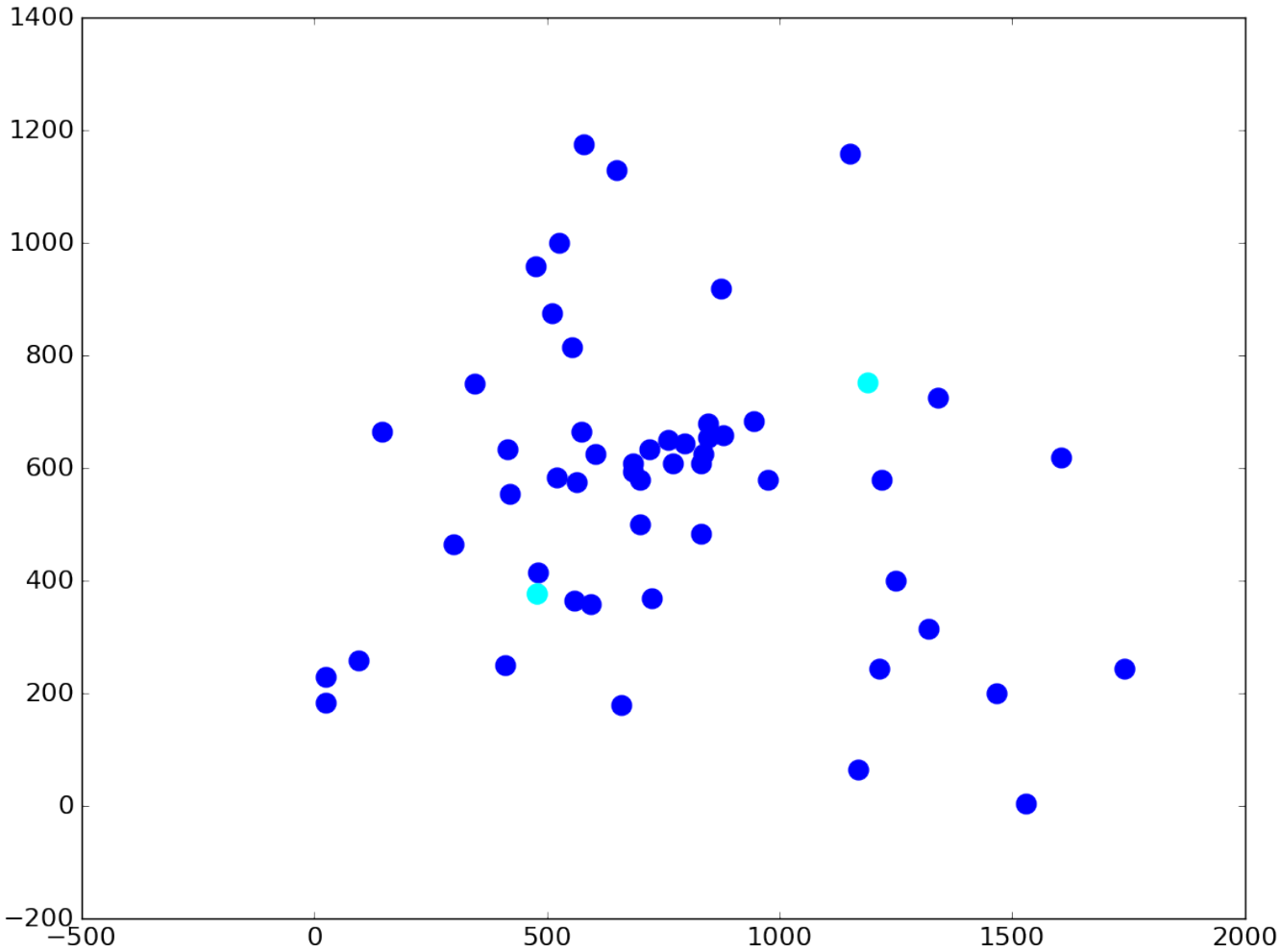
animation of the pair-center tour algorithm



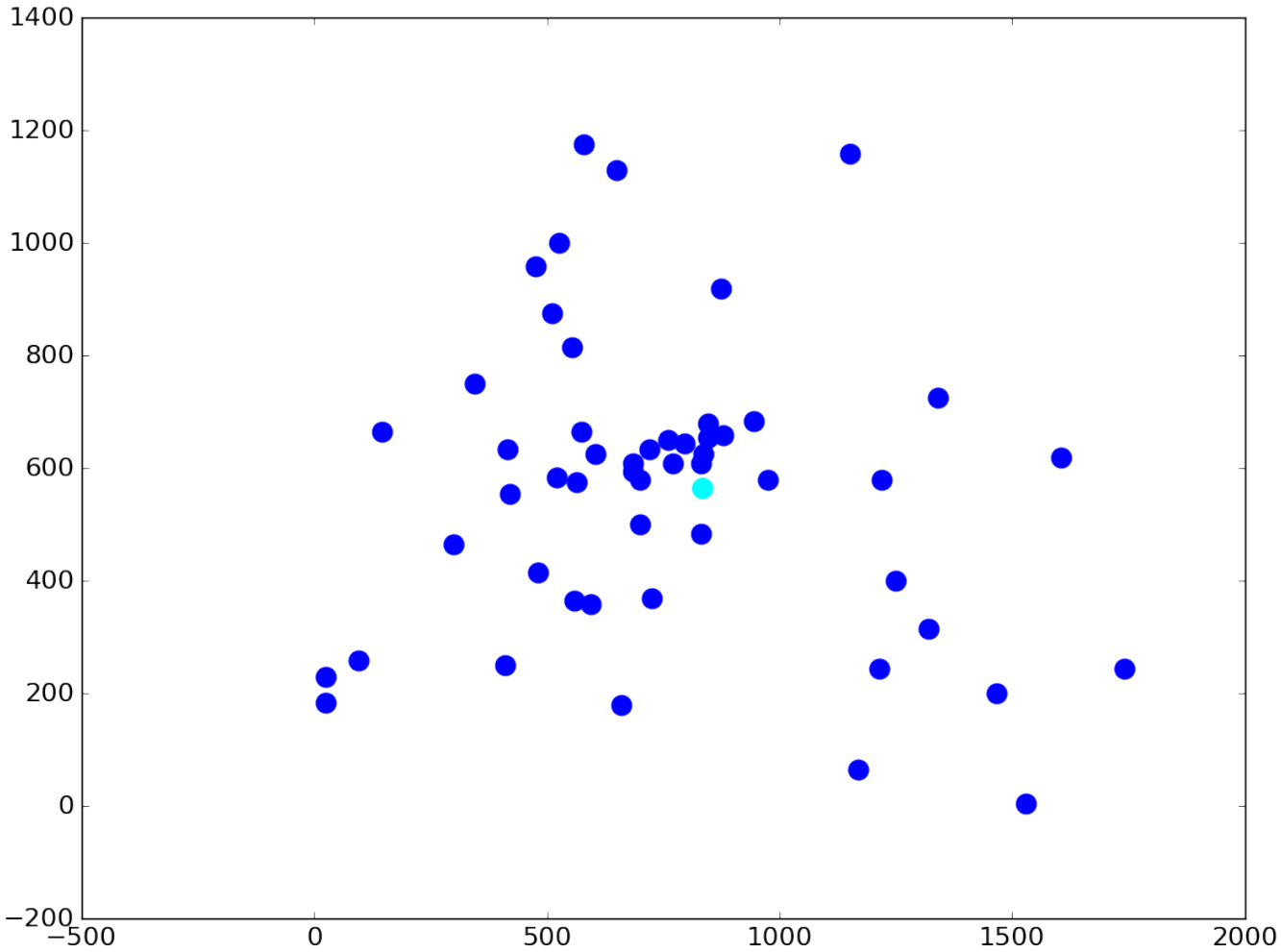
animation of the pair-center tour algorithm



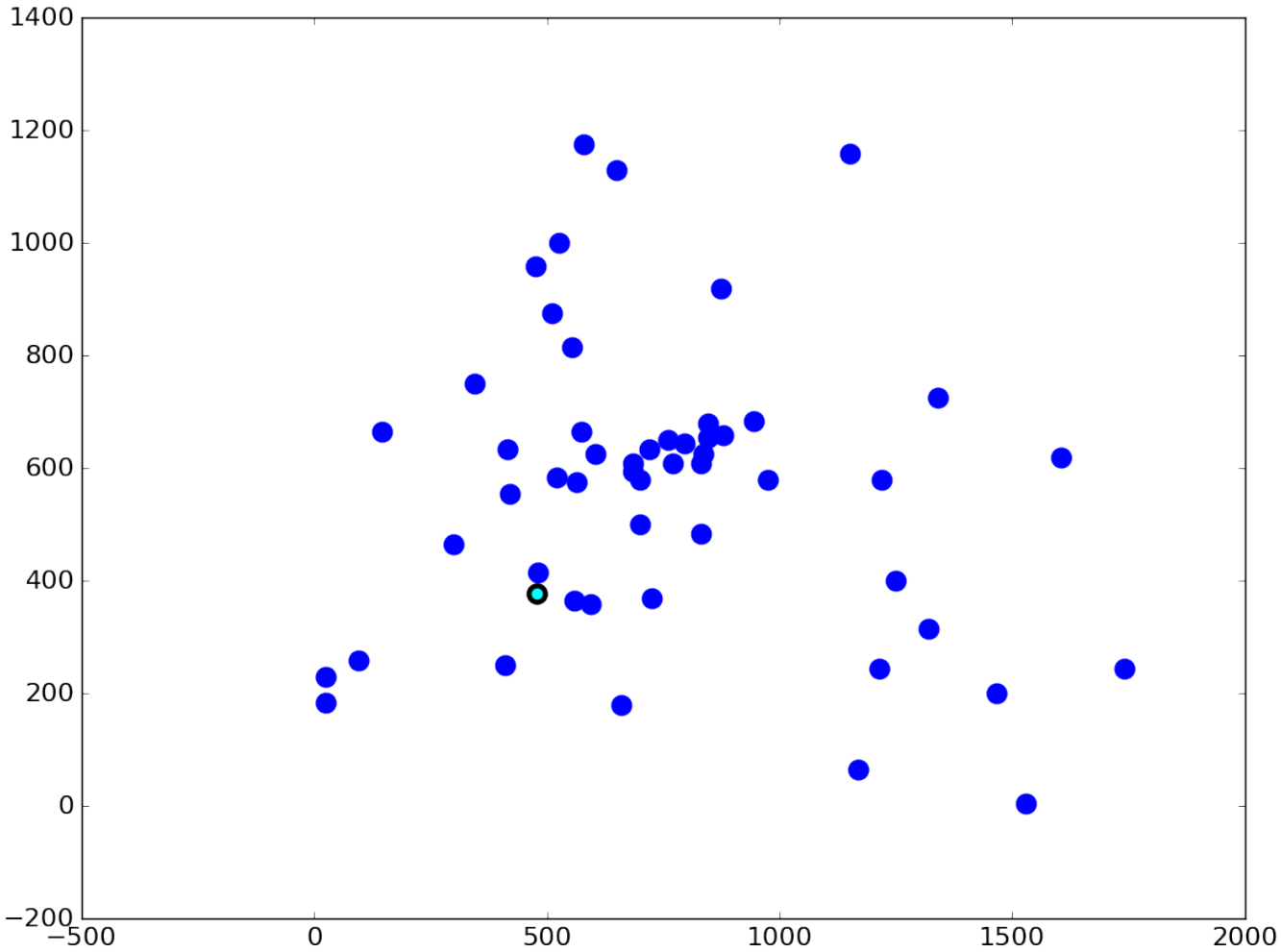
animation of the pair-center tour algorithm



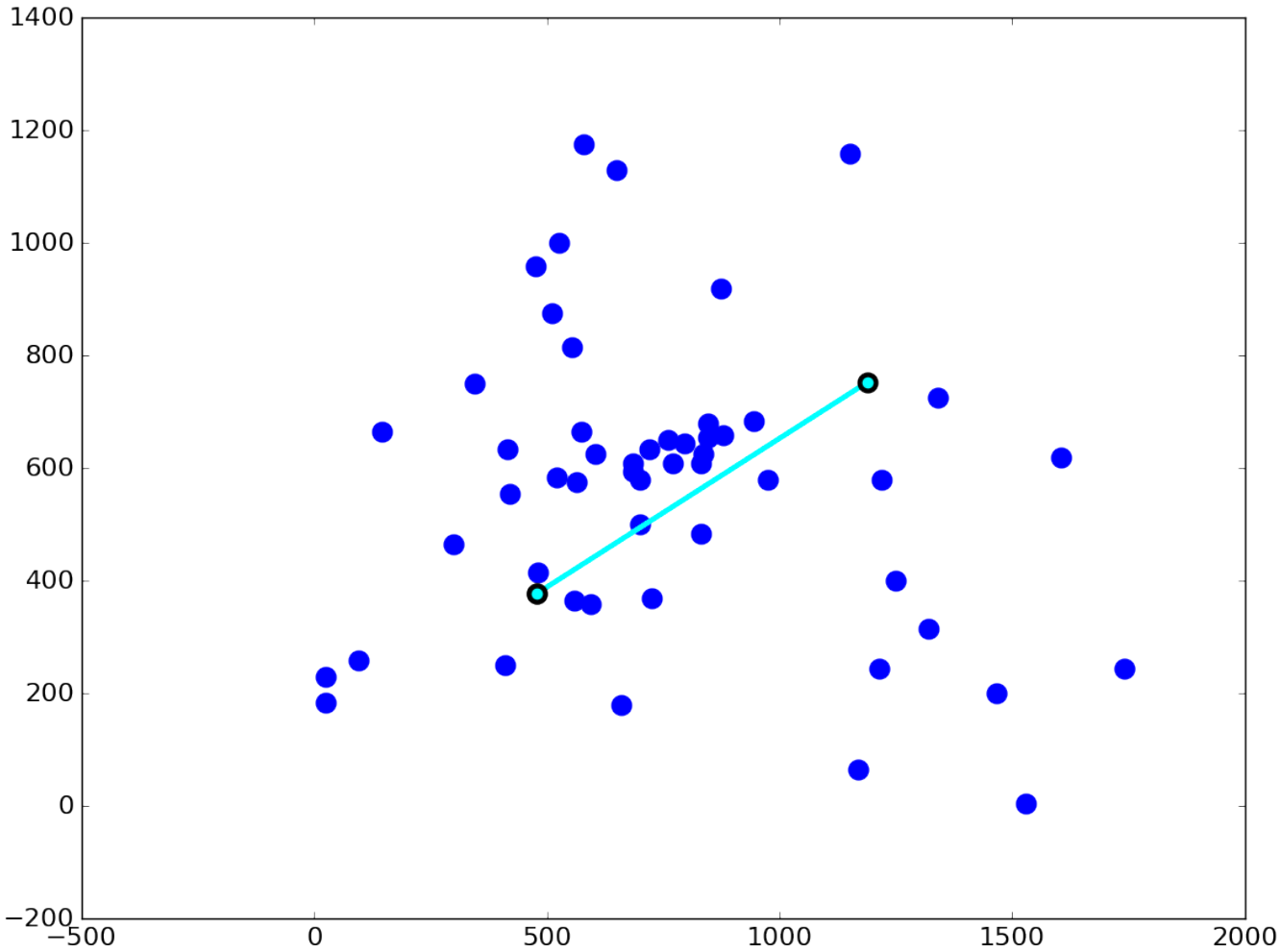
animation of the pair-center tour algorithm



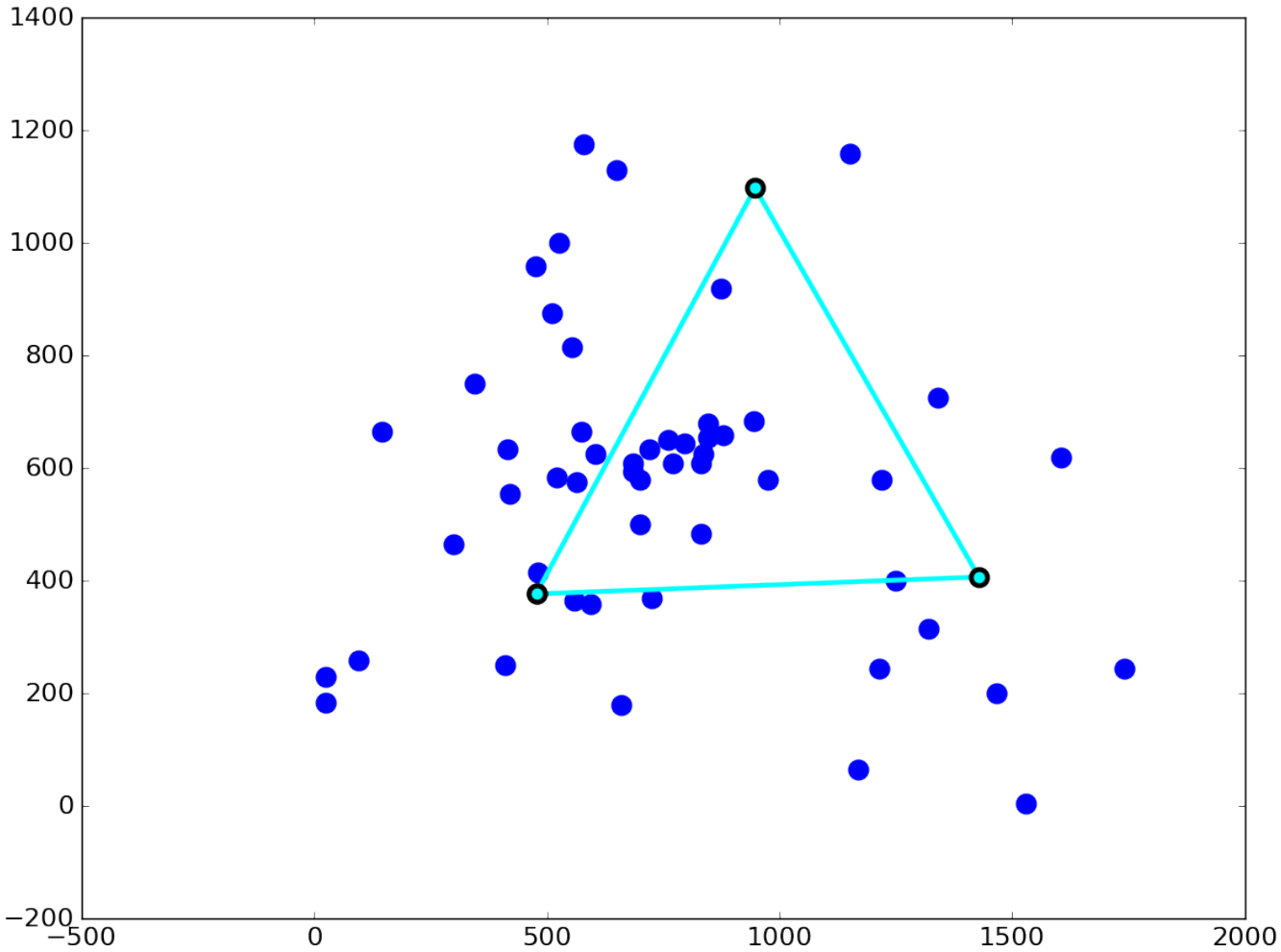
animation of the pair-center tour algorithm



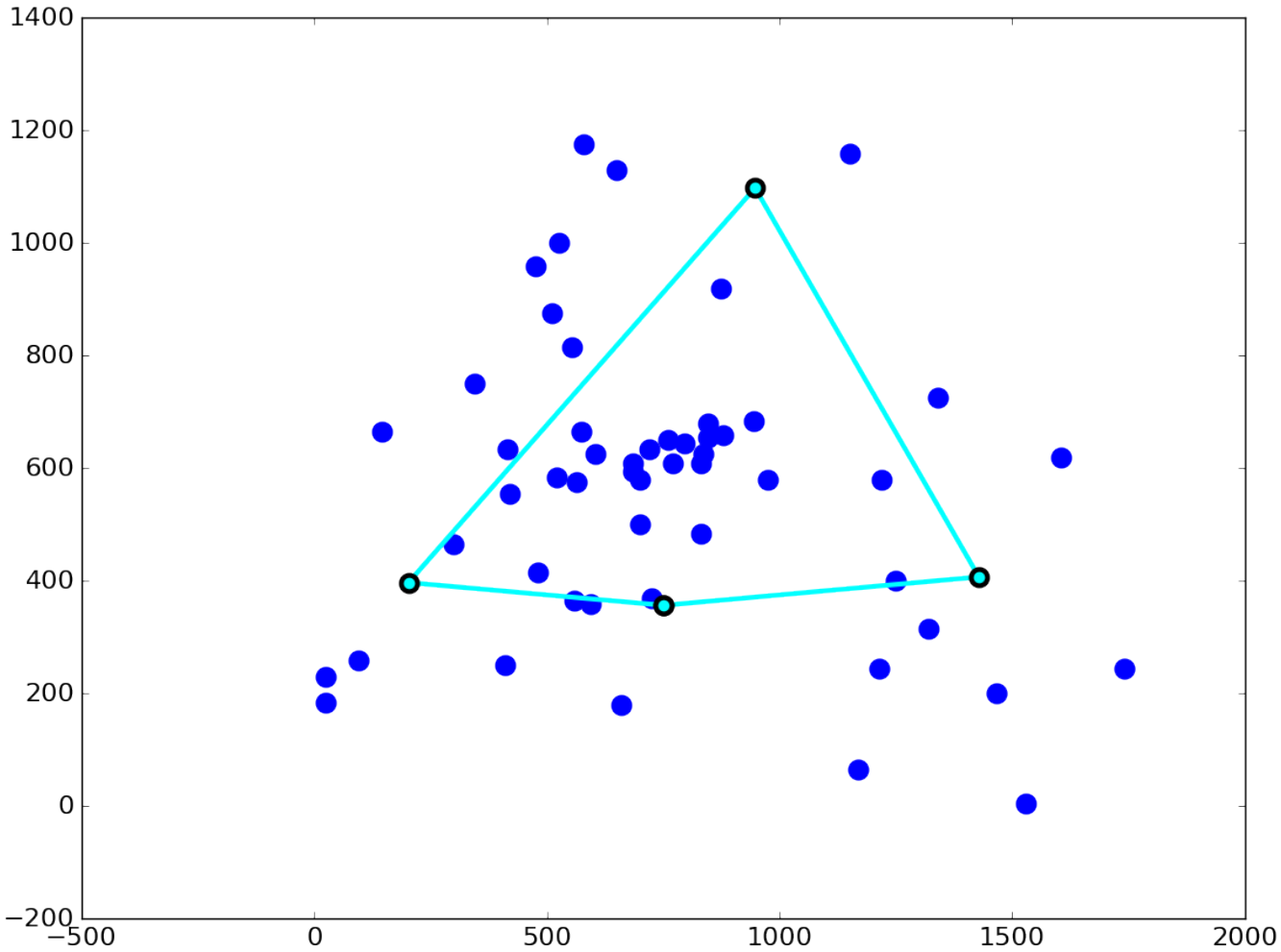
animation of the pair-center tour algorithm



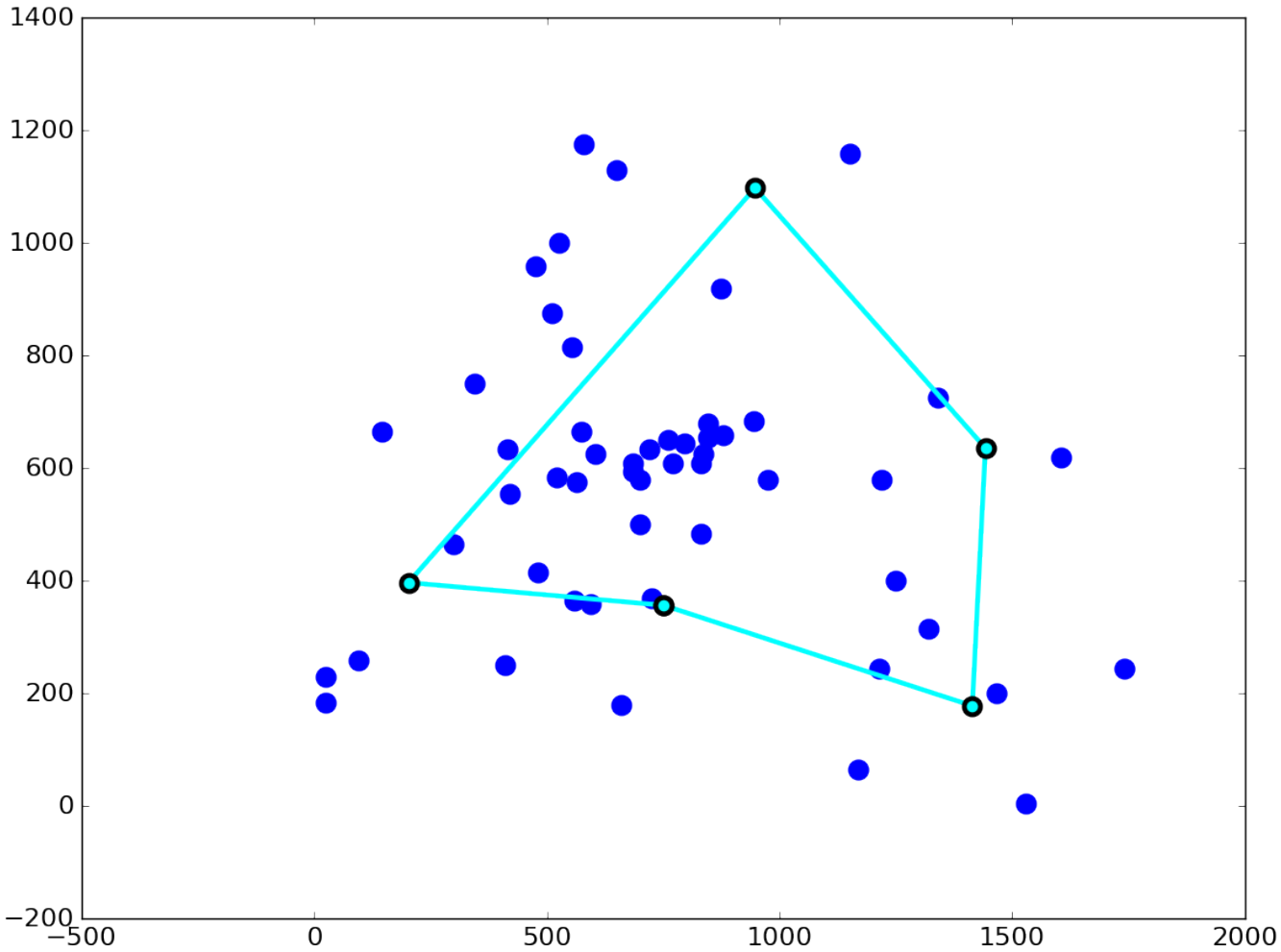
animation of the pair-center tour algorithm



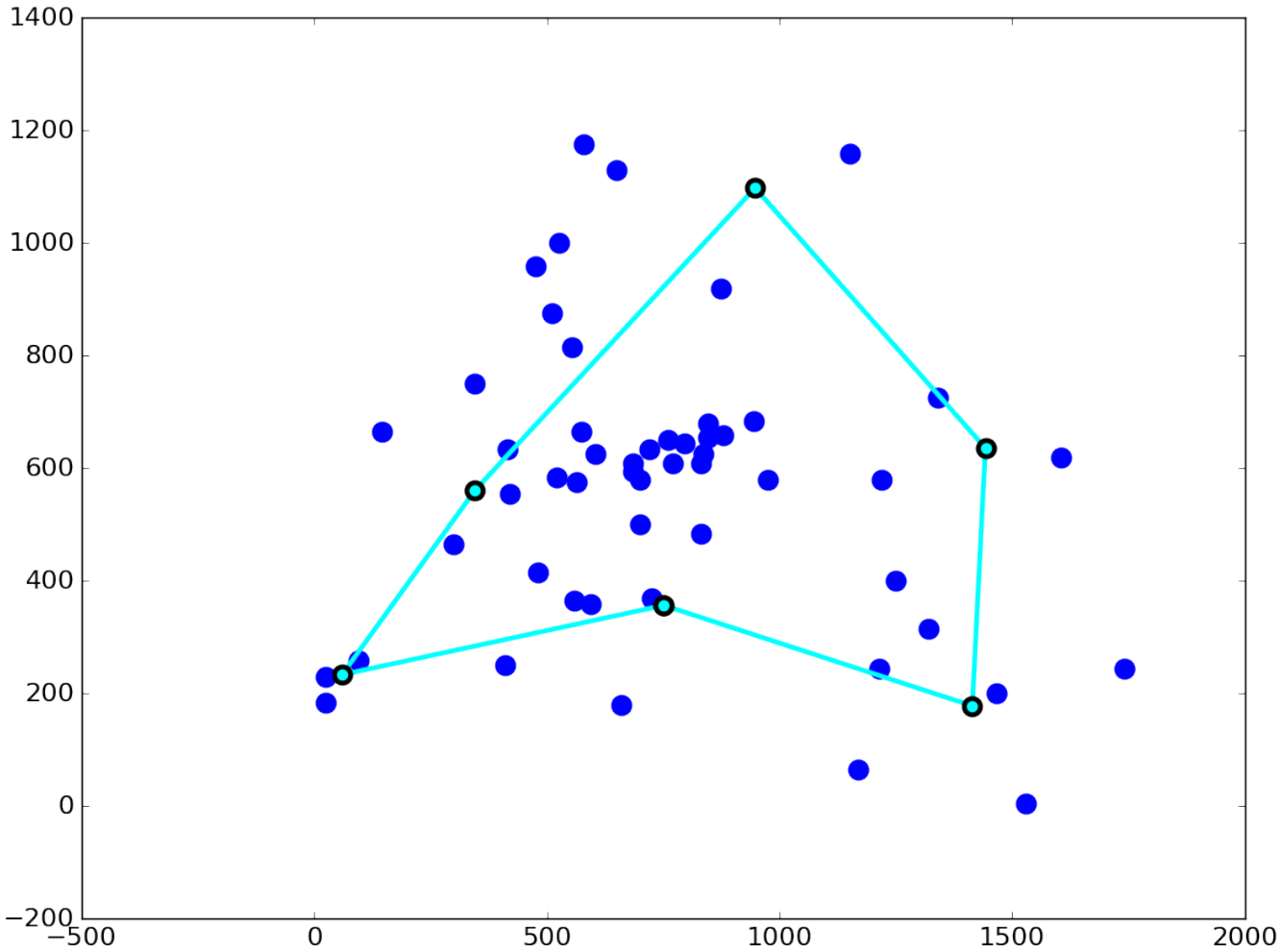
animation of the pair-center tour algorithm



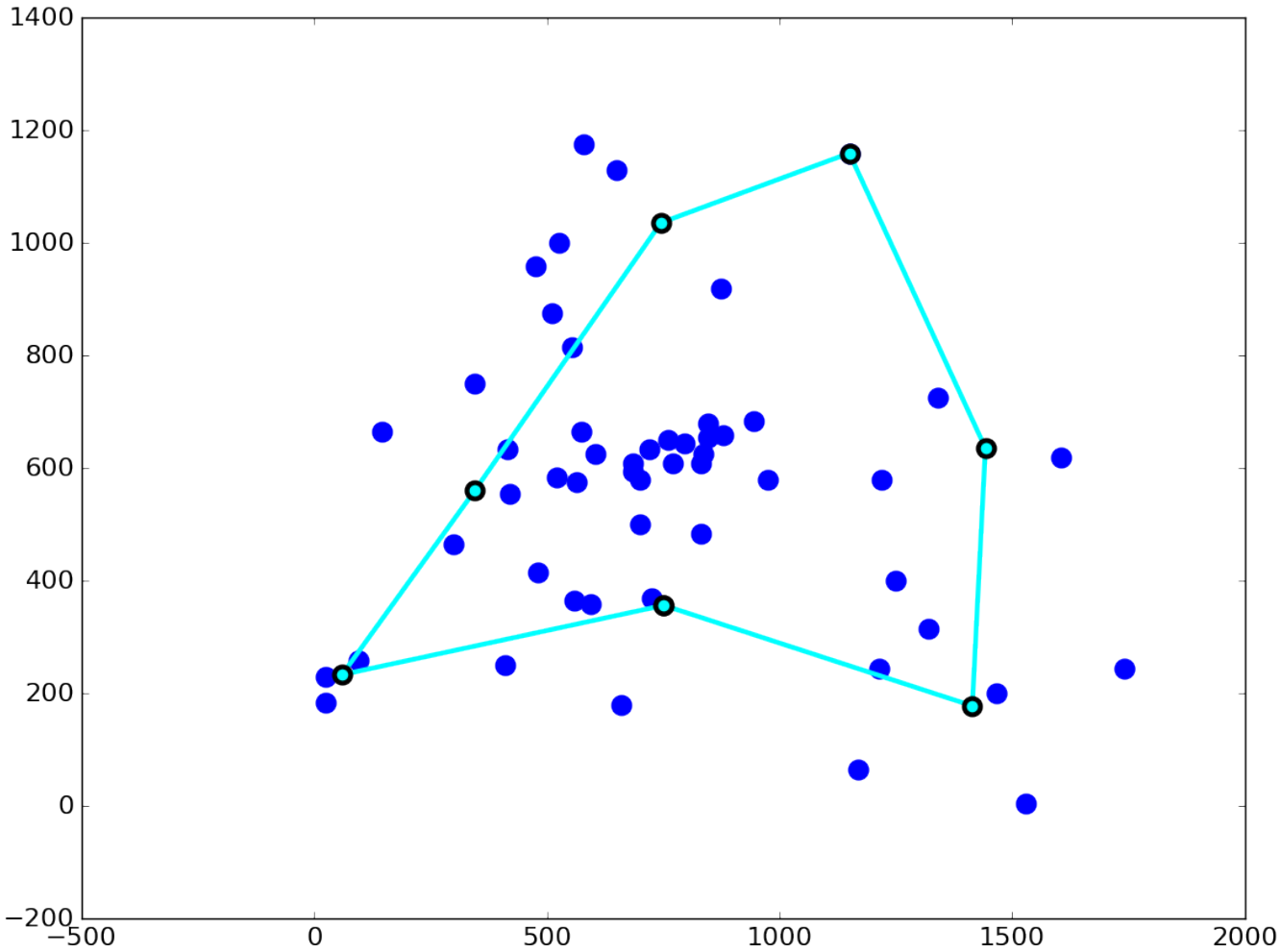
animation of the pair-center tour algorithm



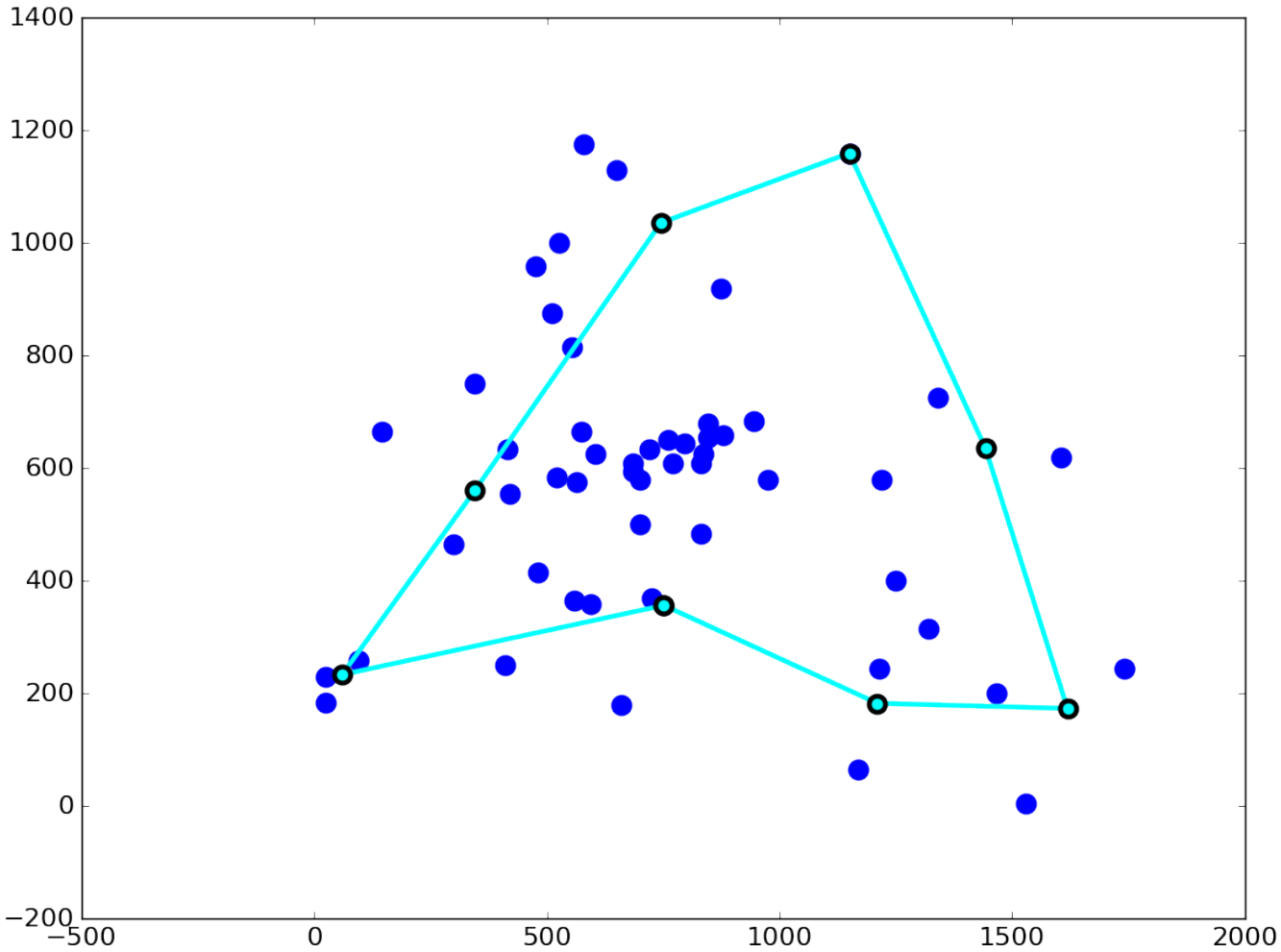
animation of the pair-center tour algorithm



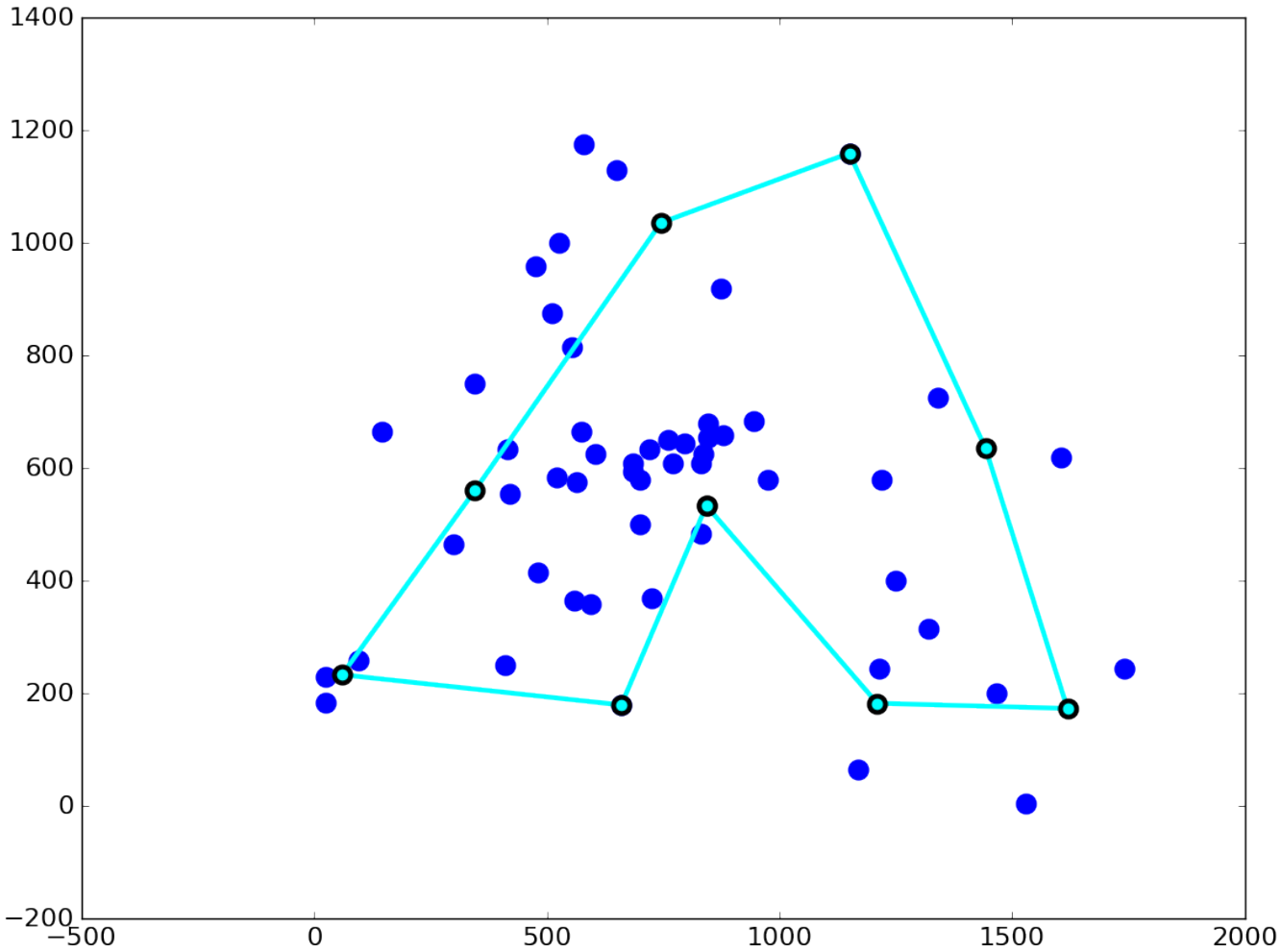
animation of the pair-center tour algorithm



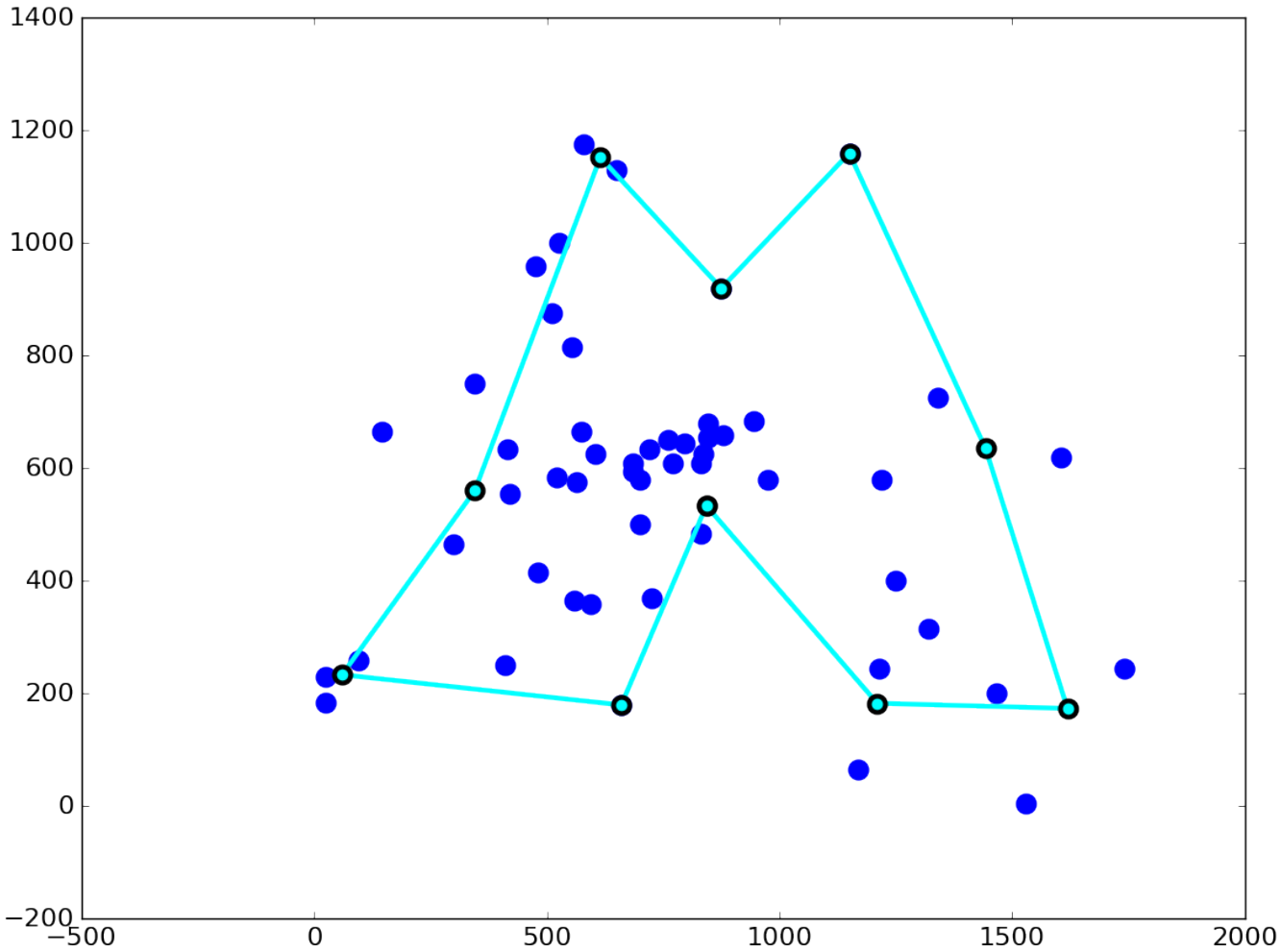
animation of the pair-center tour algorithm



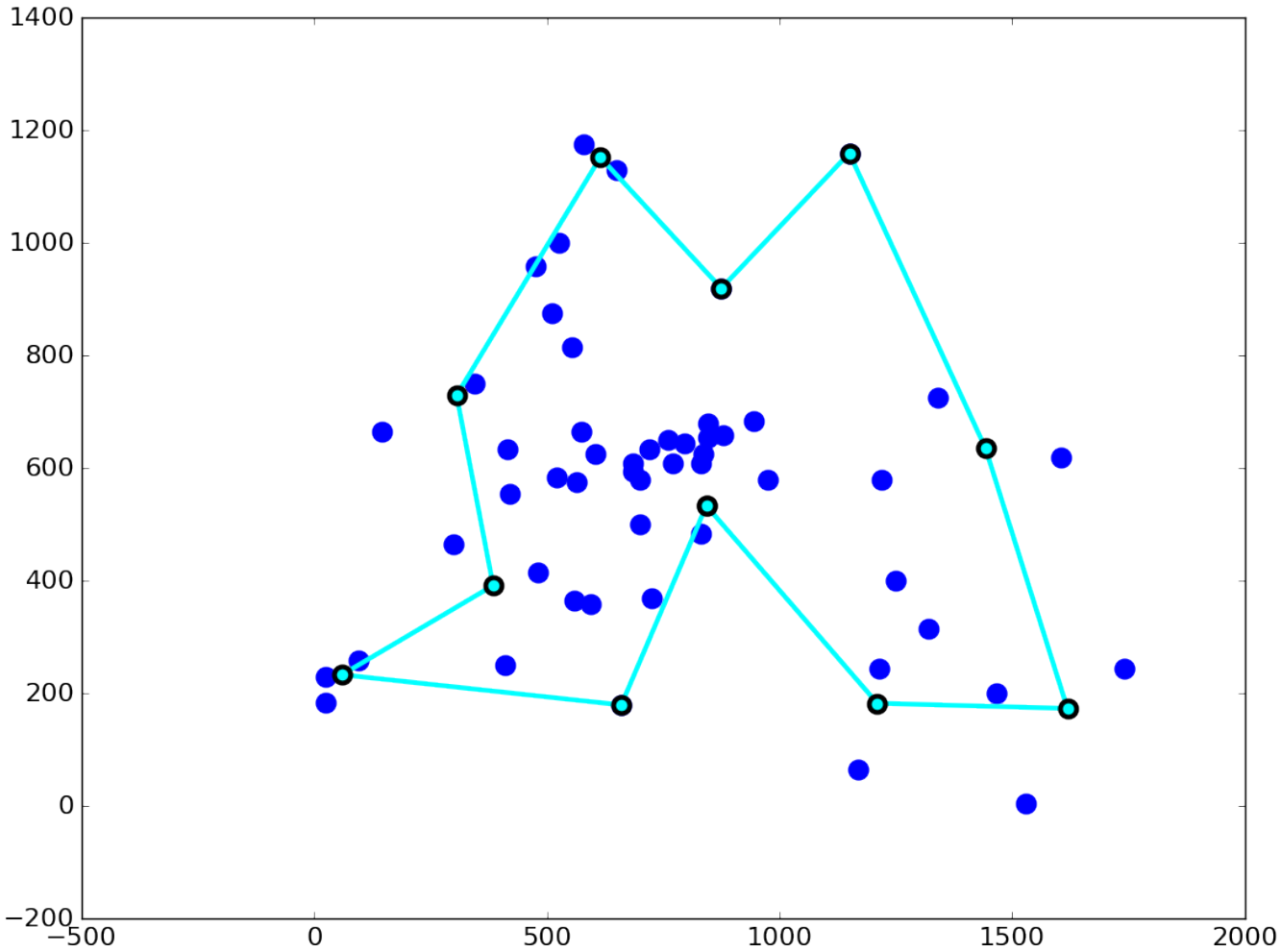
animation of the pair-center tour algorithm



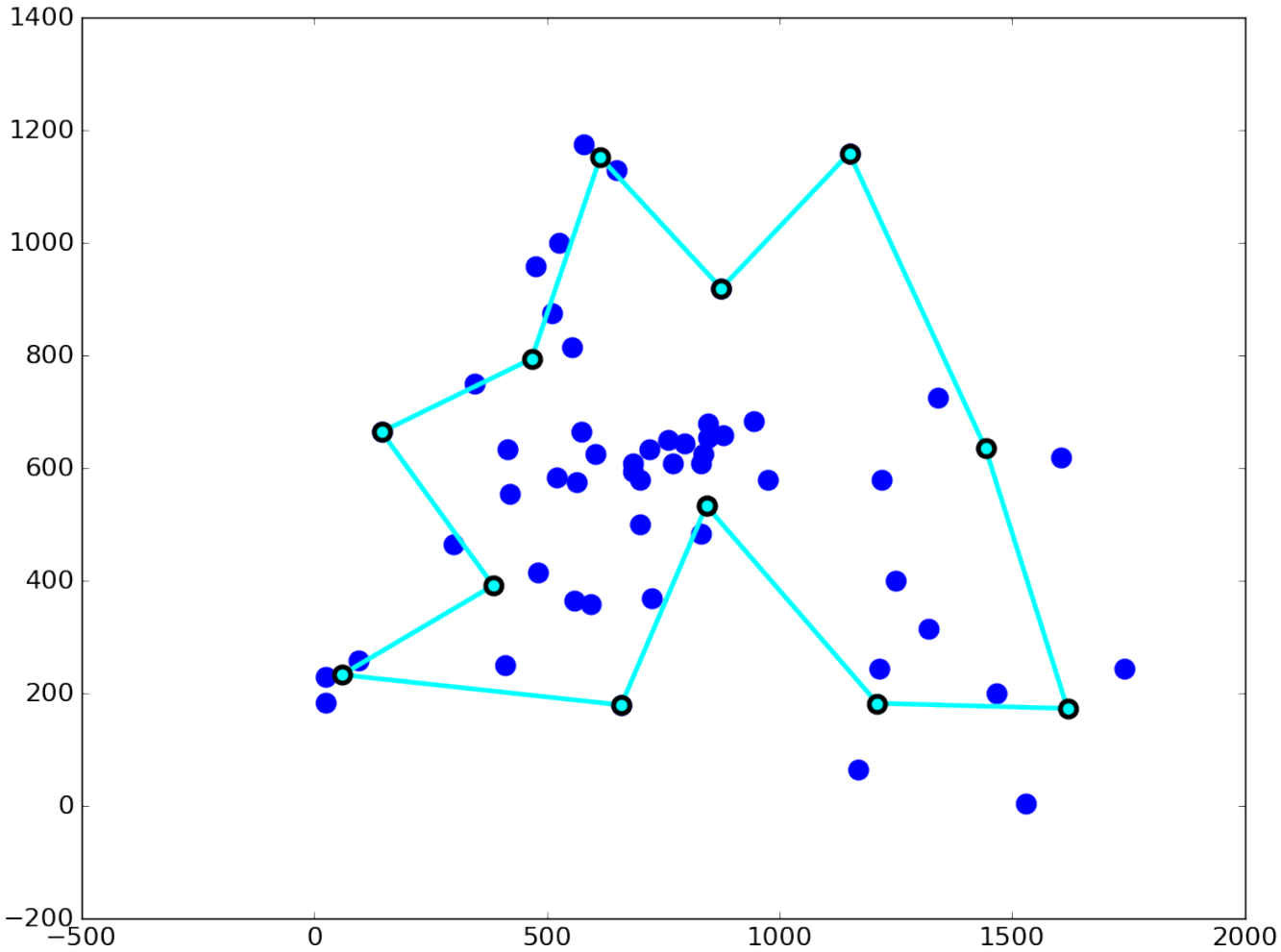
animation of the pair-center tour algorithm



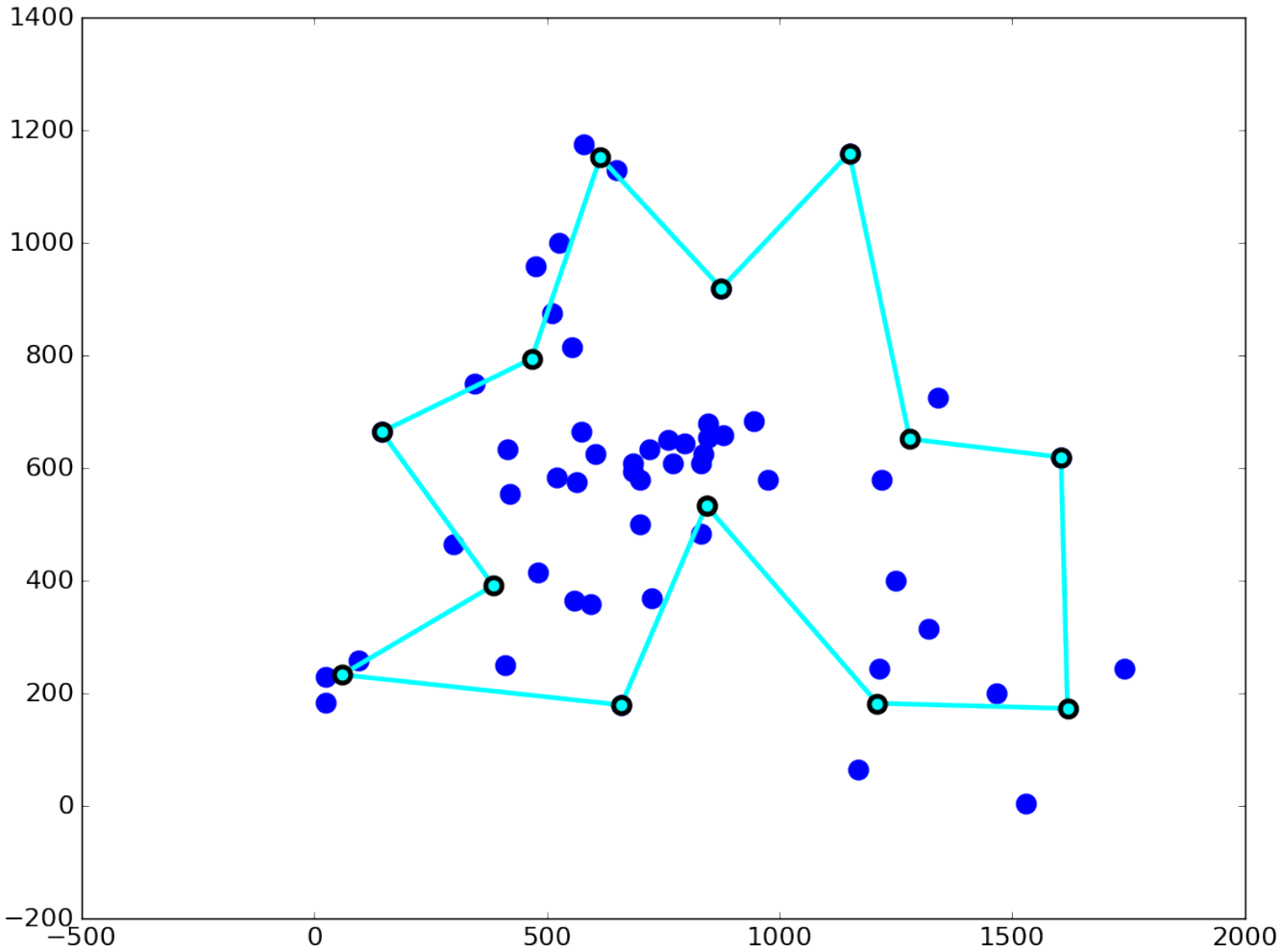
animation of the pair-center tour algorithm



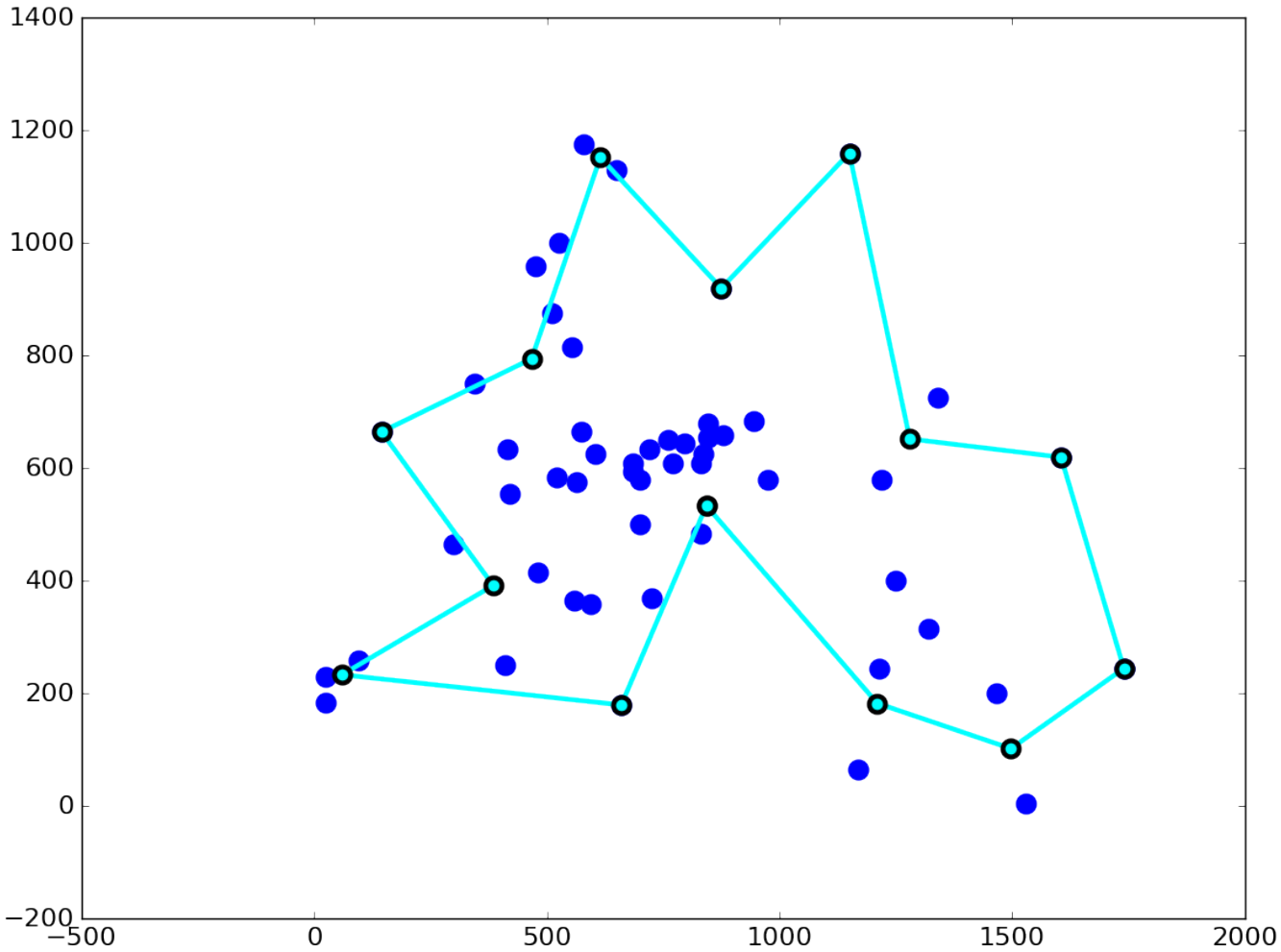
animation of the pair-center tour algorithm



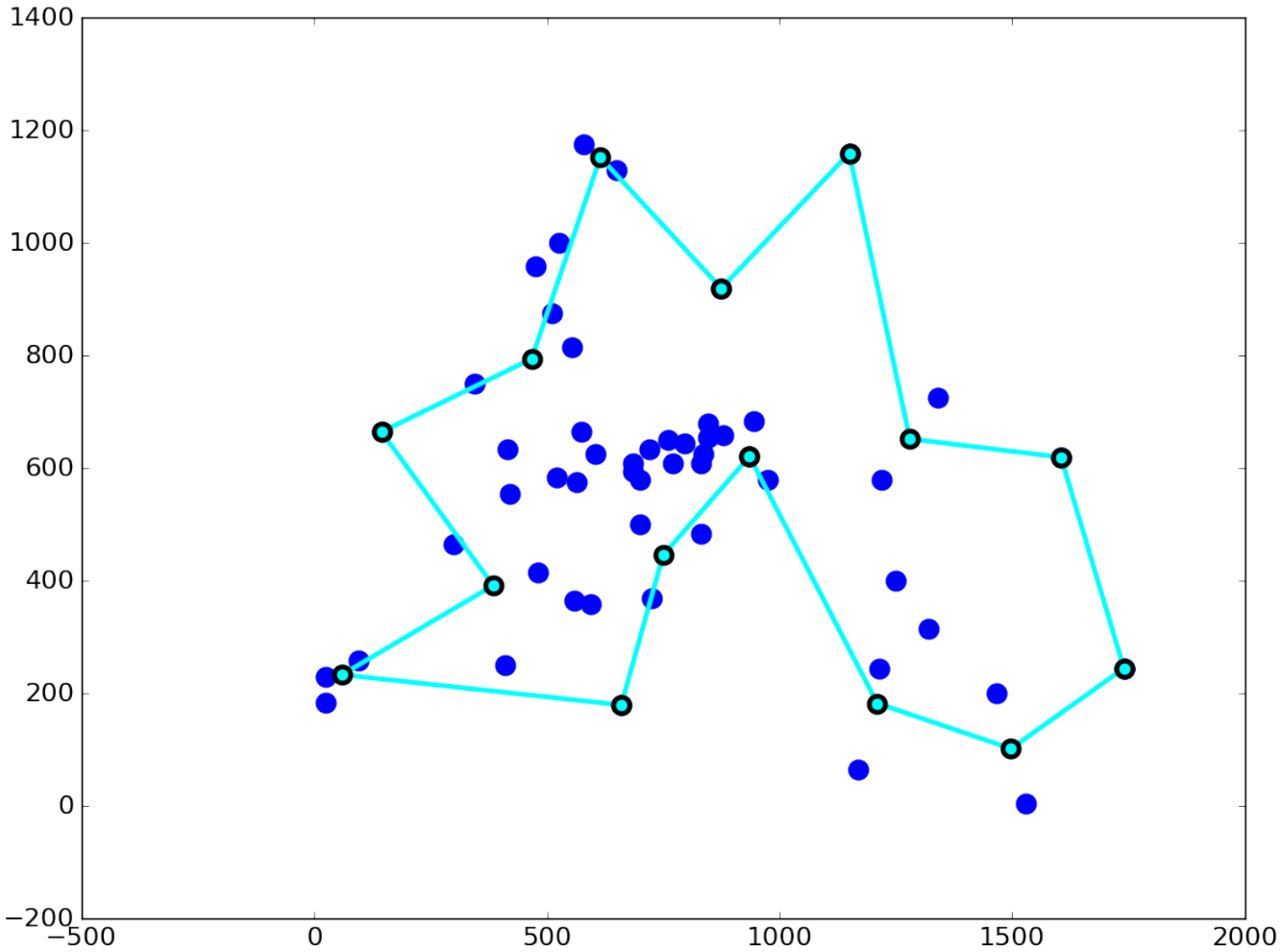
animation of the pair-center tour algorithm



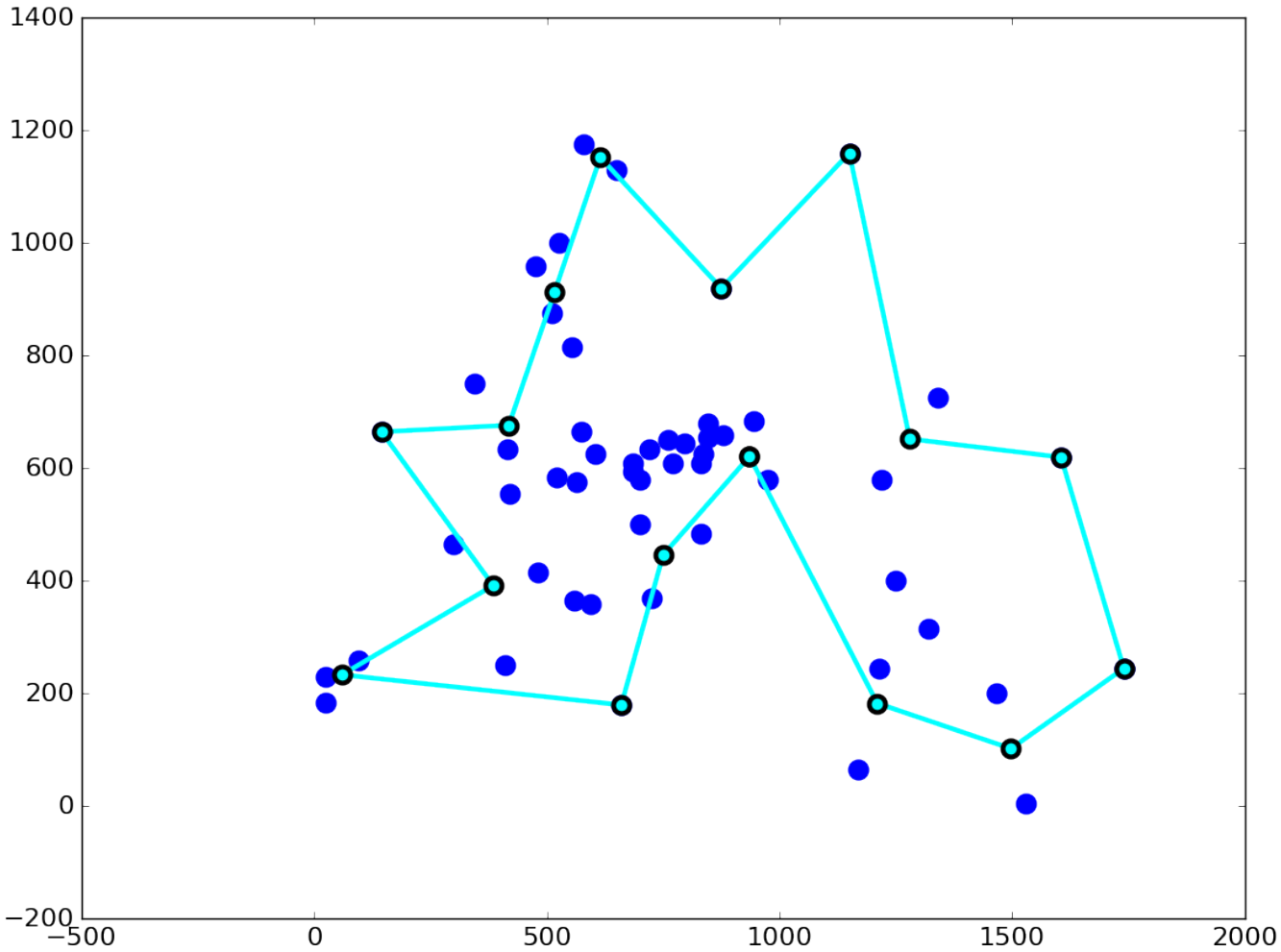
animation of the pair-center tour algorithm



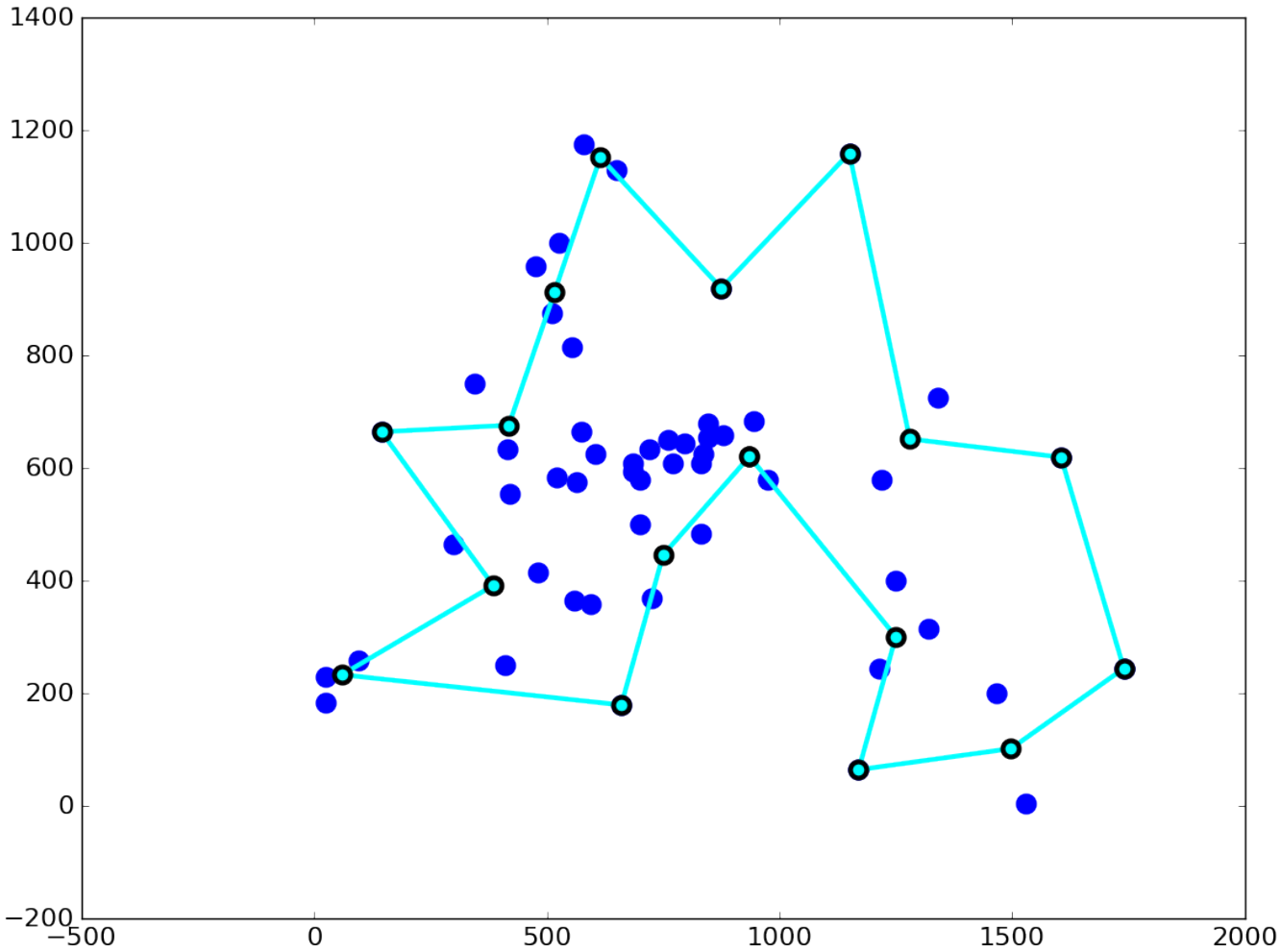
animation of the pair-center tour algorithm



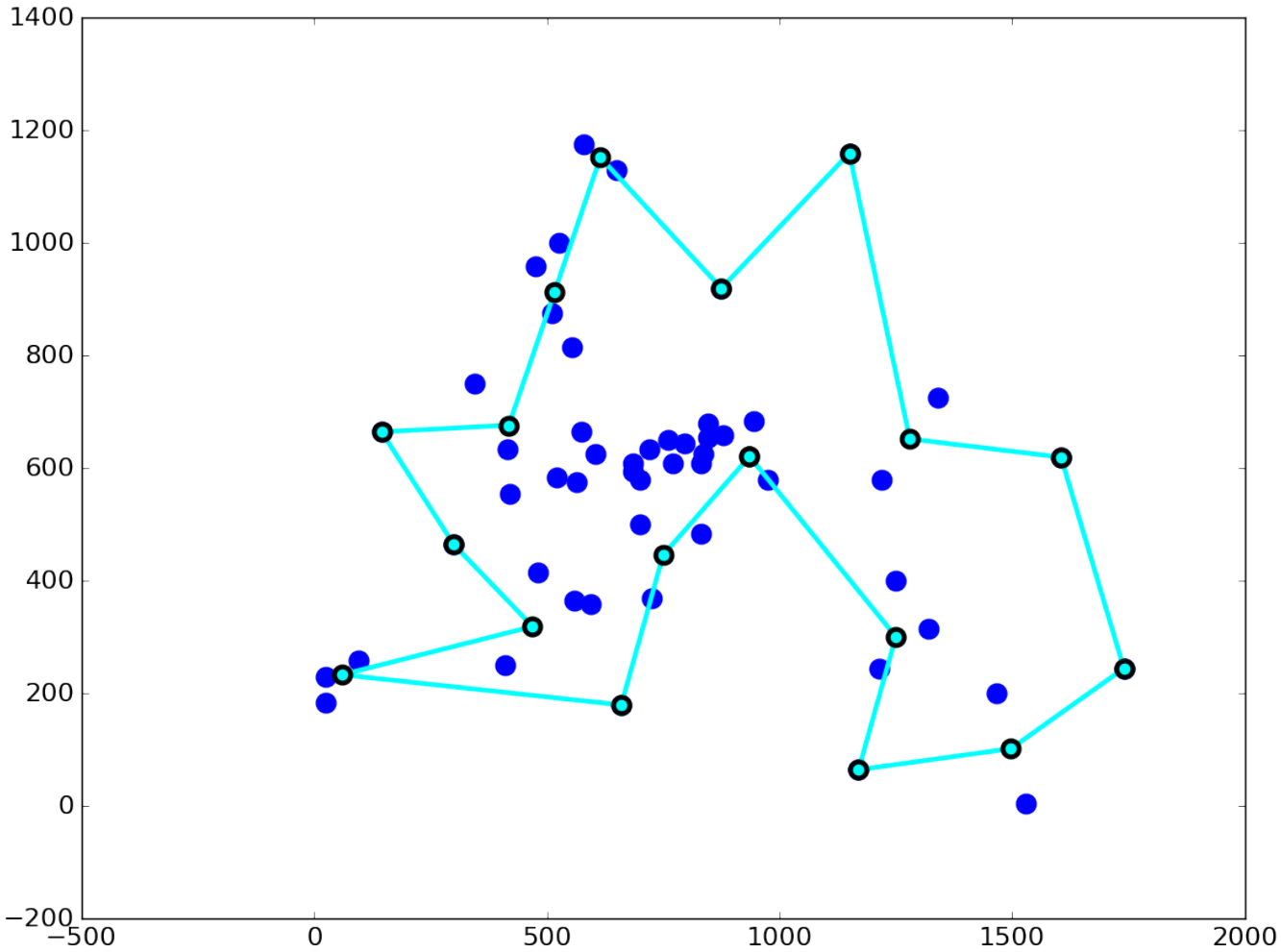
animation of the pair-center tour algorithm



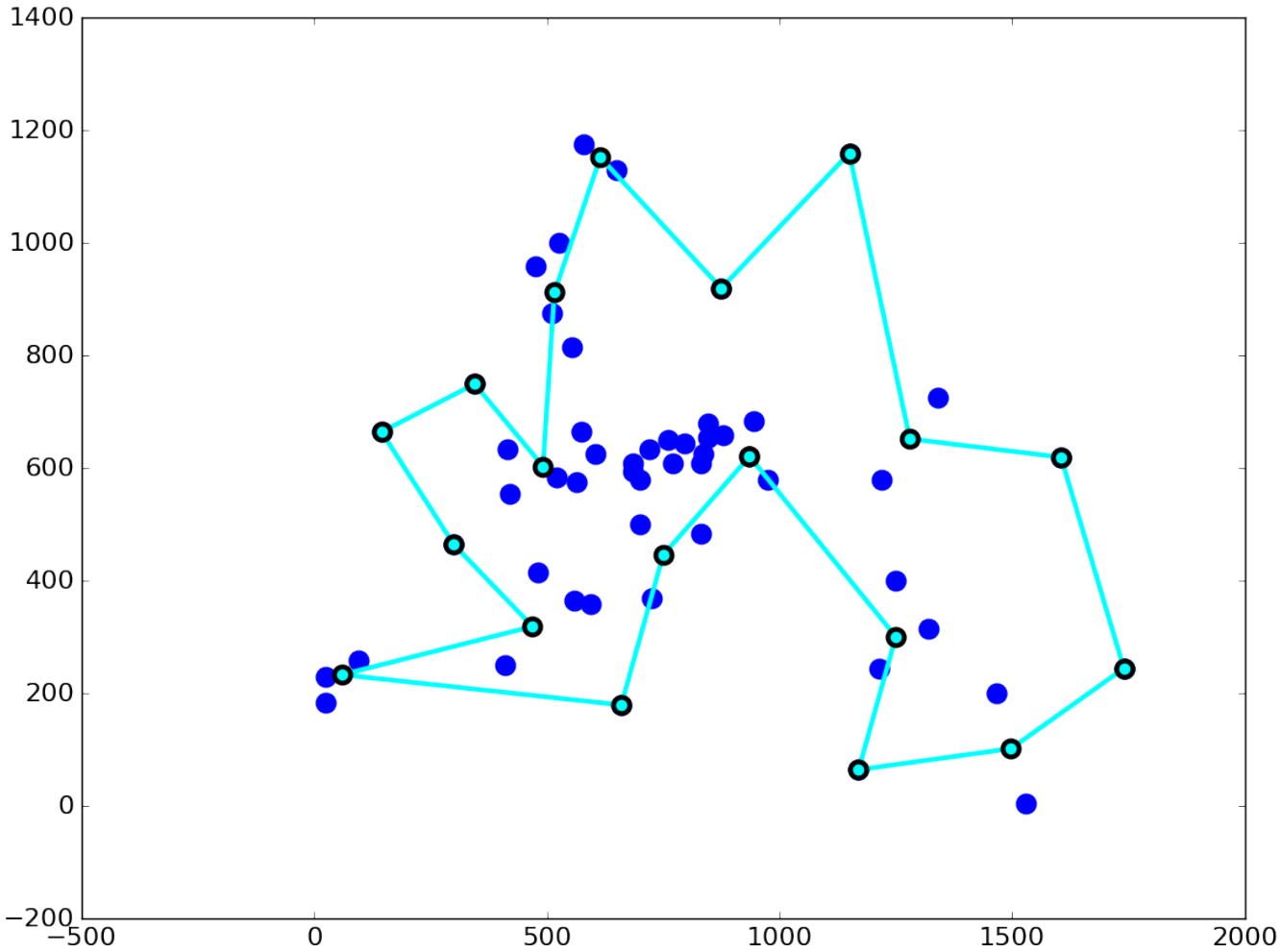
animation of the pair-center tour algorithm



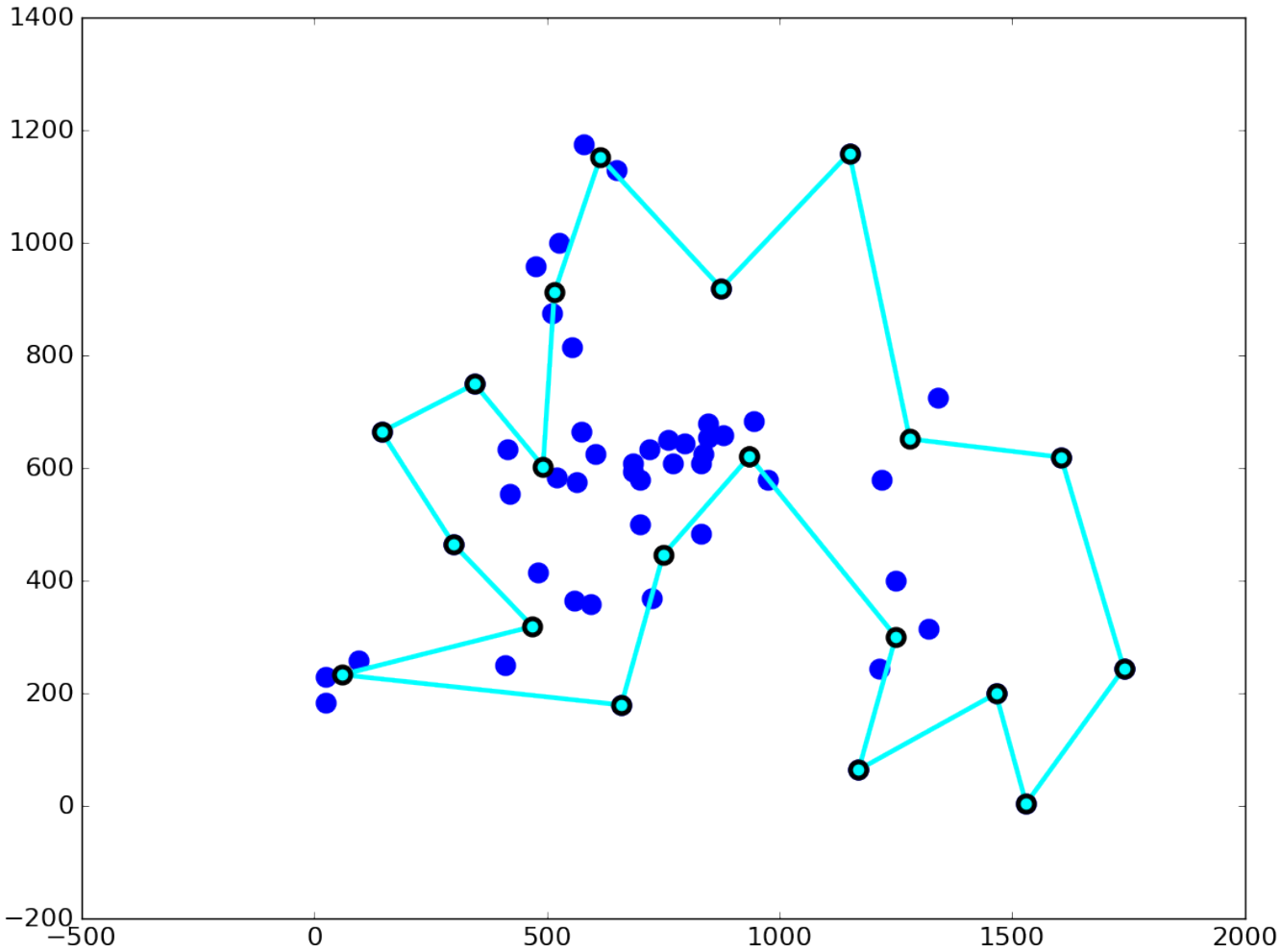
animation of the pair-center tour algorithm



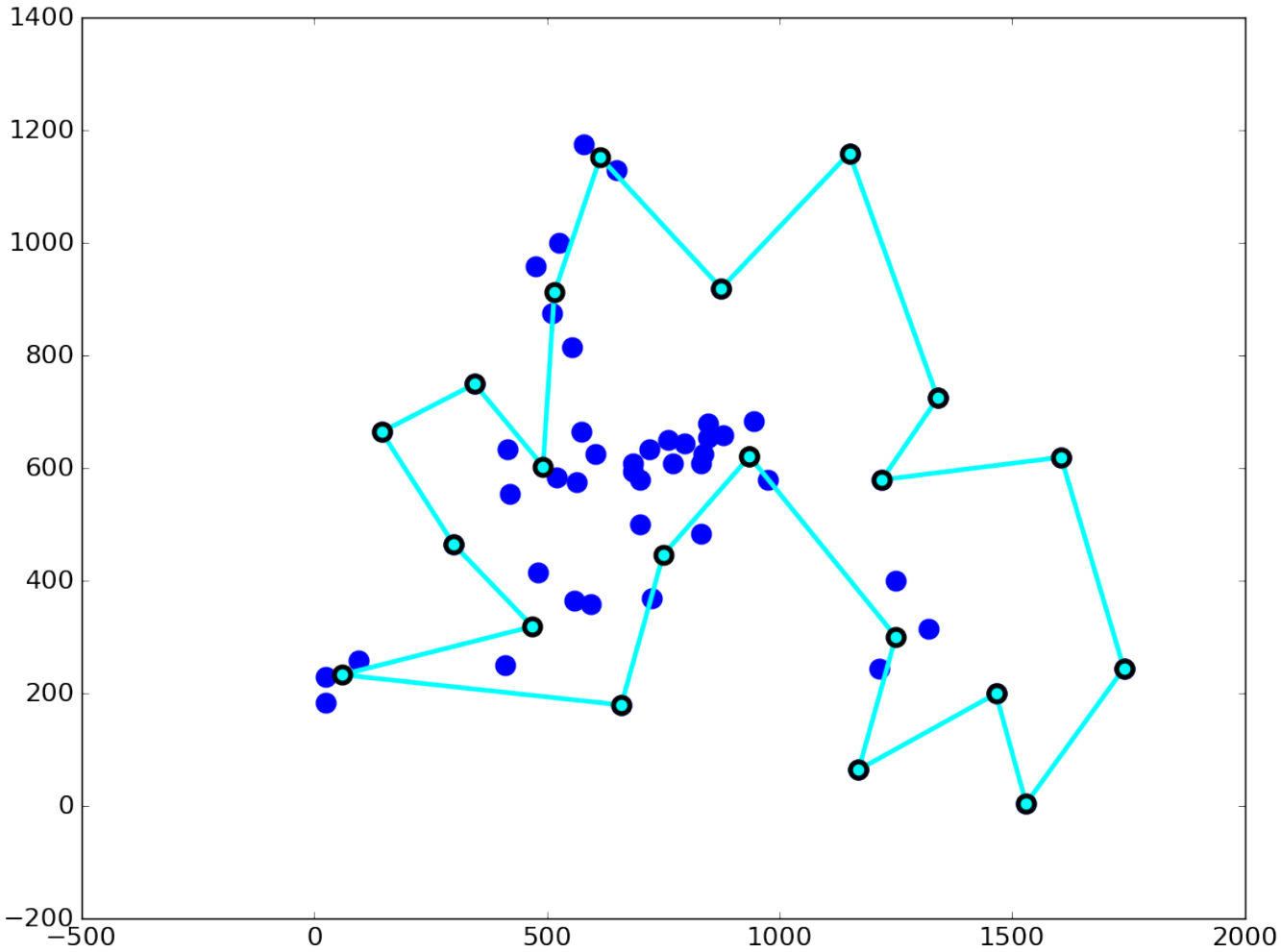
animation of the pair-center tour algorithm



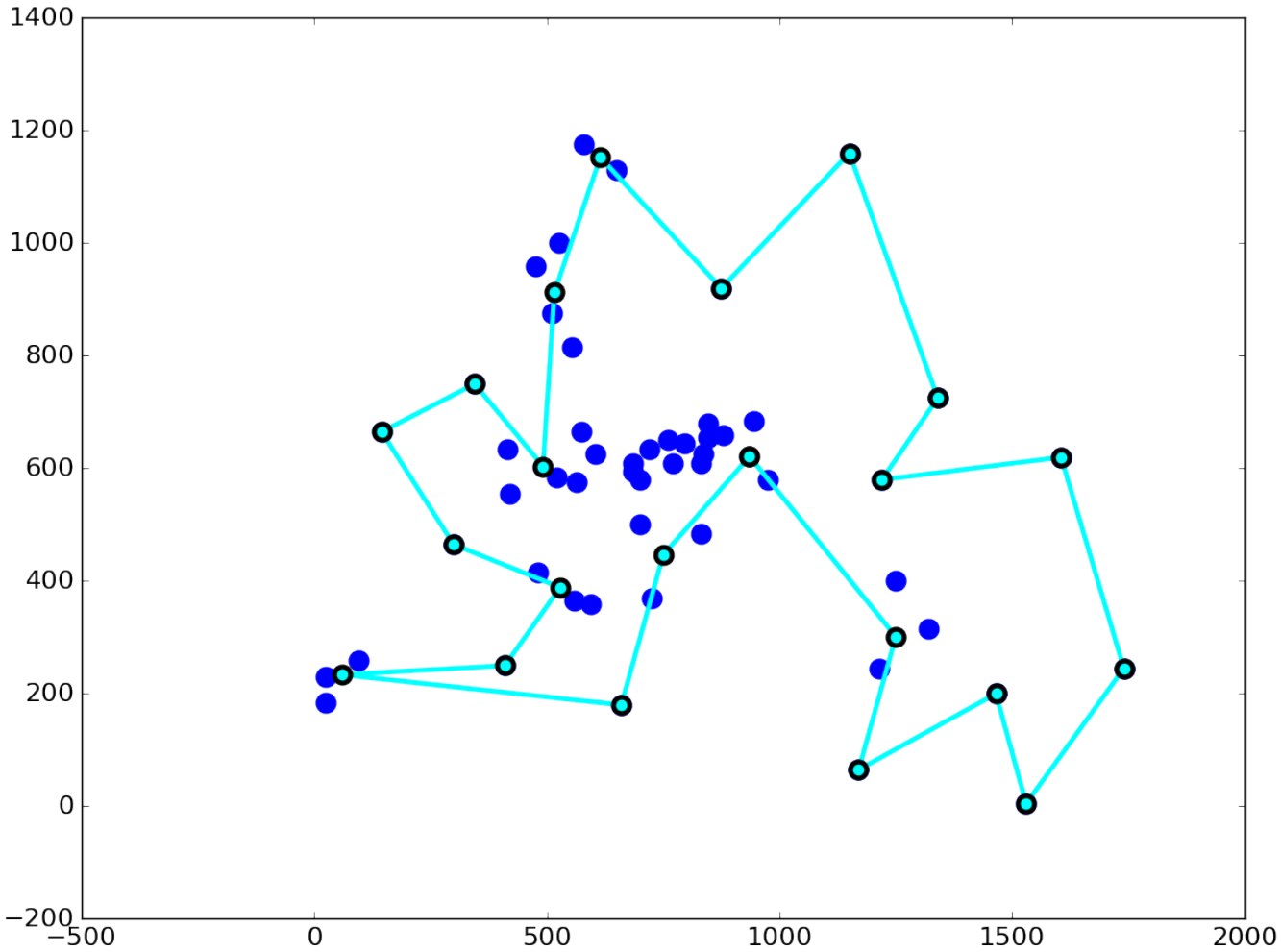
animation of the pair-center tour algorithm



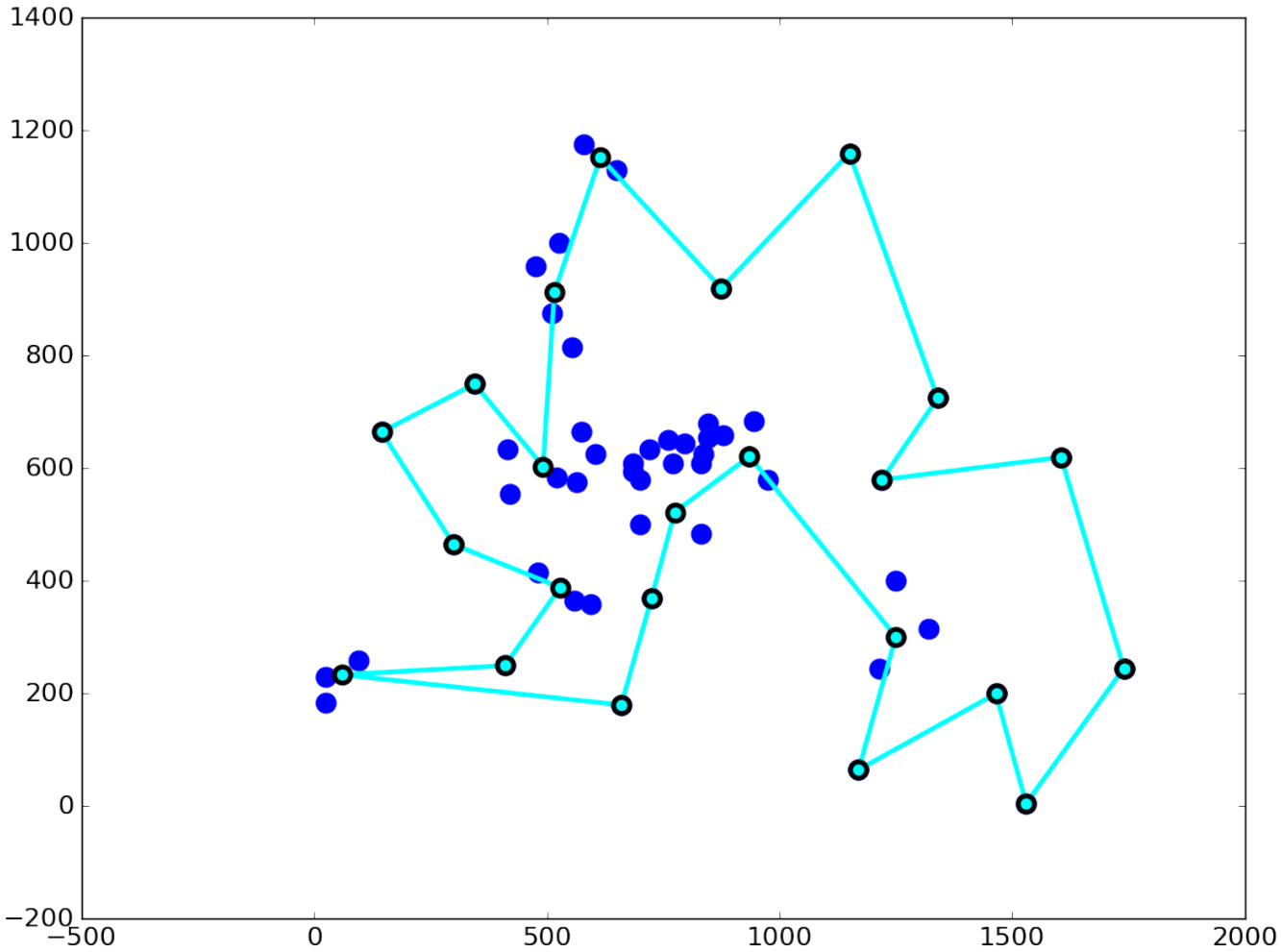
animation of the pair-center tour algorithm



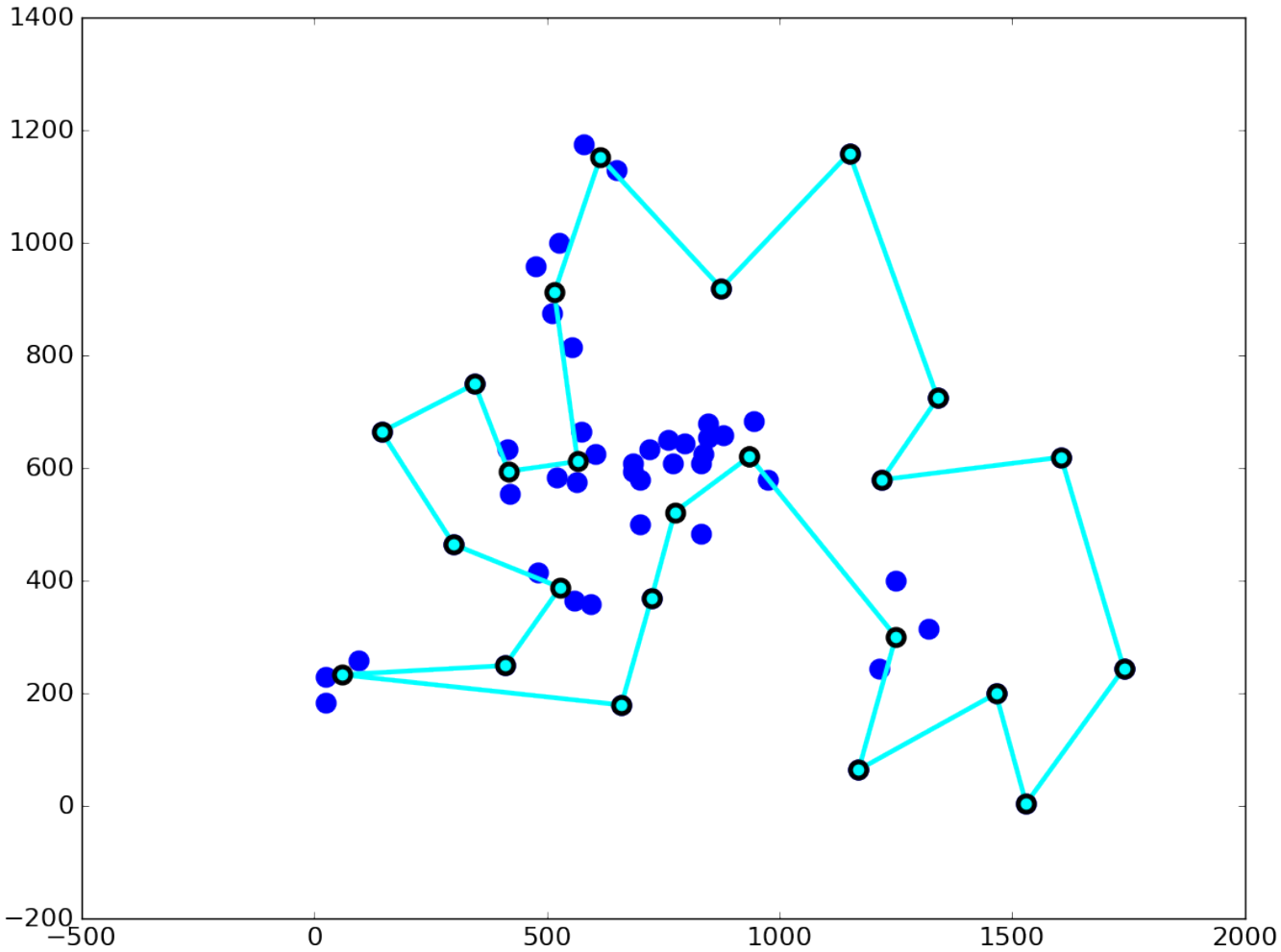
animation of the pair-center tour algorithm



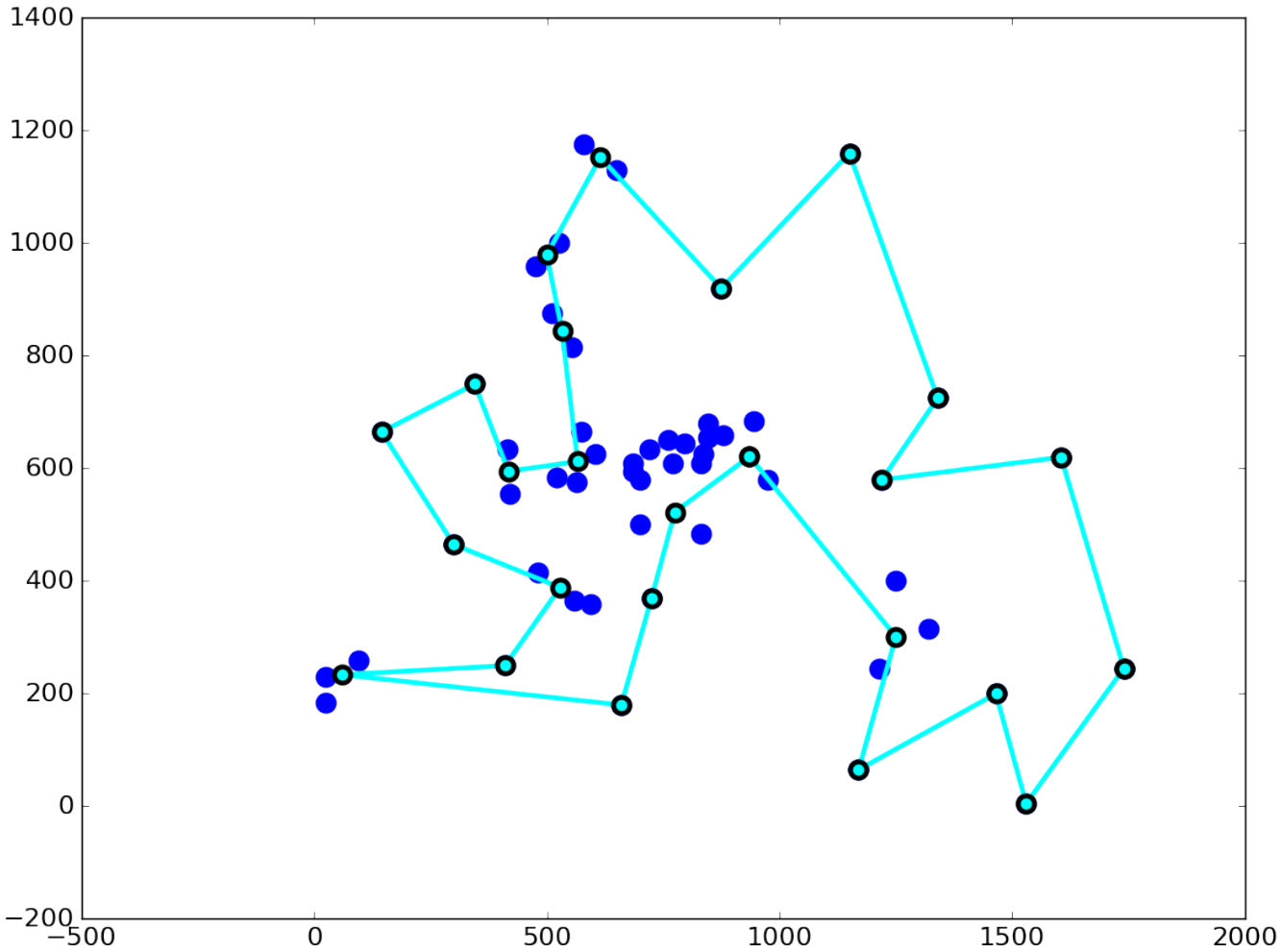
animation of the pair-center tour algorithm



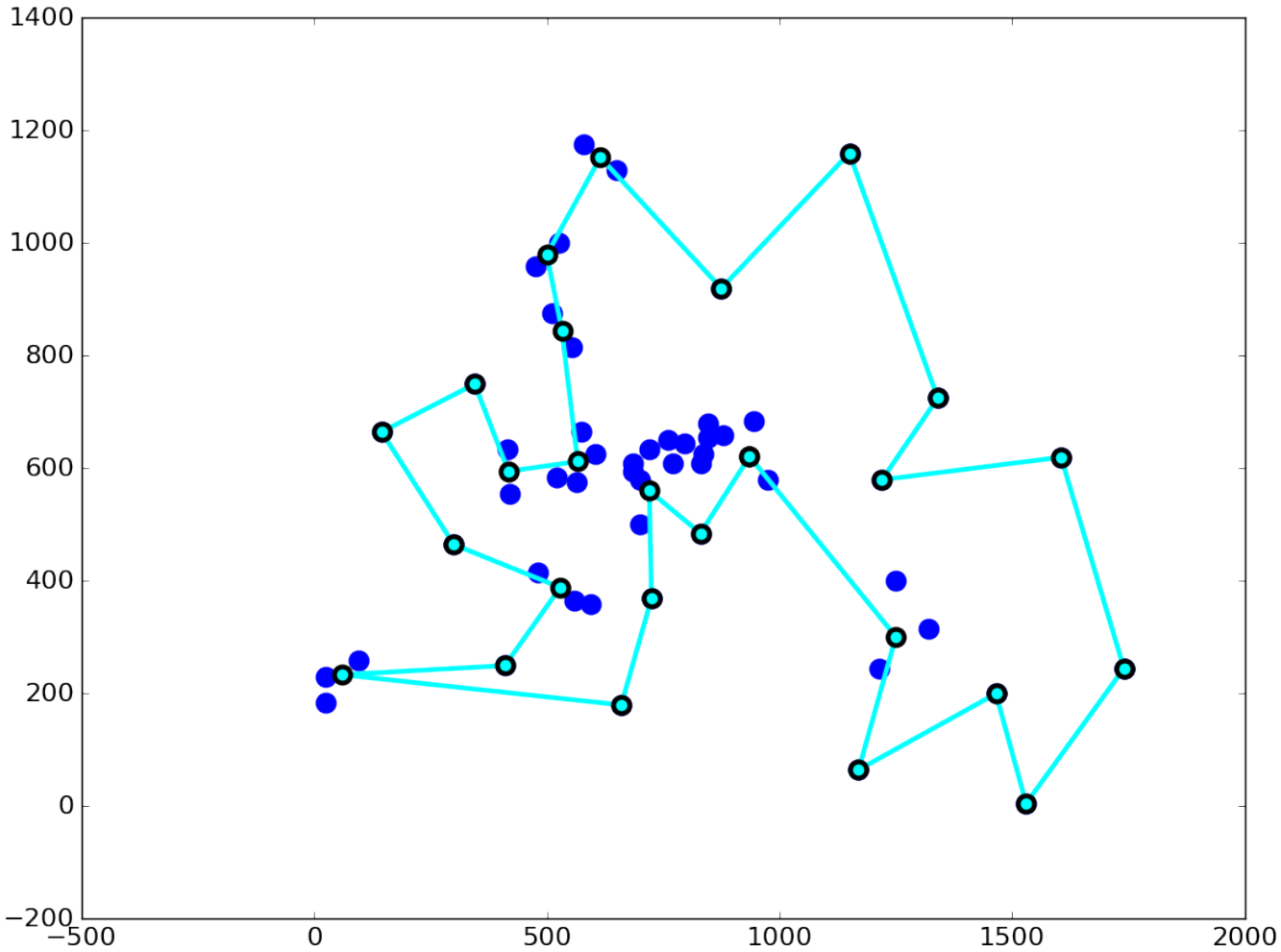
animation of the pair-center tour algorithm



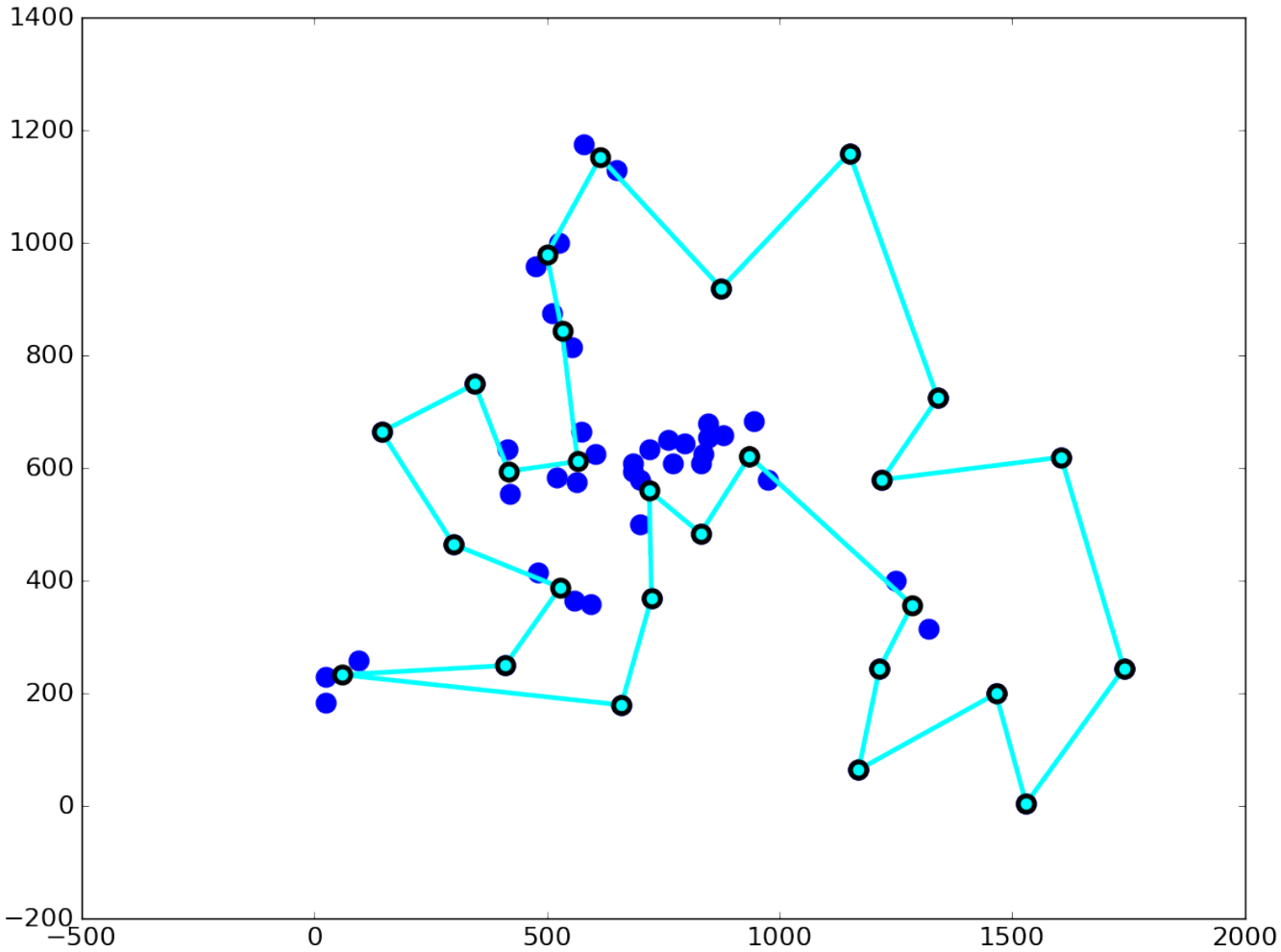
animation of the pair-center tour algorithm



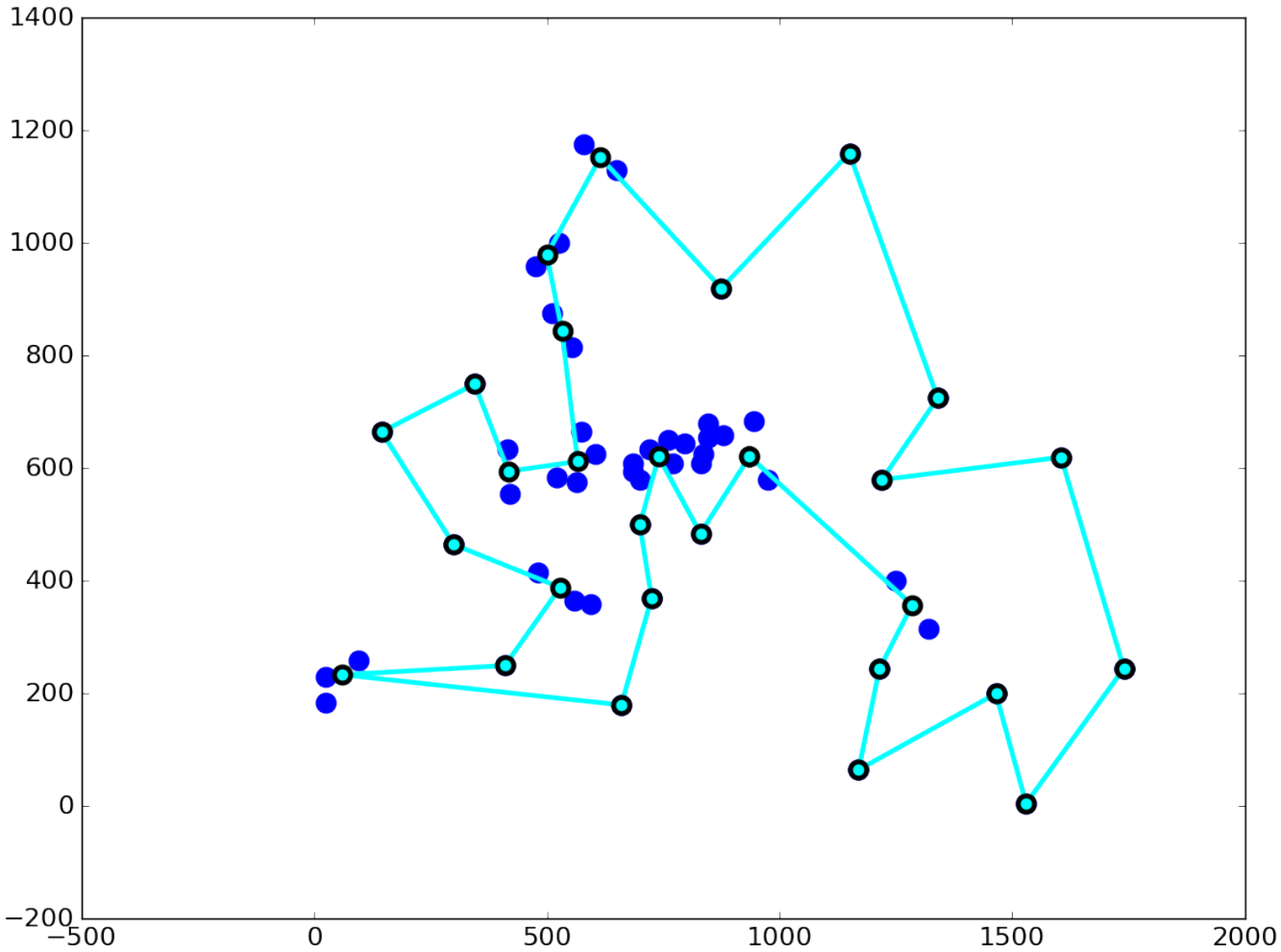
animation of the pair-center tour algorithm



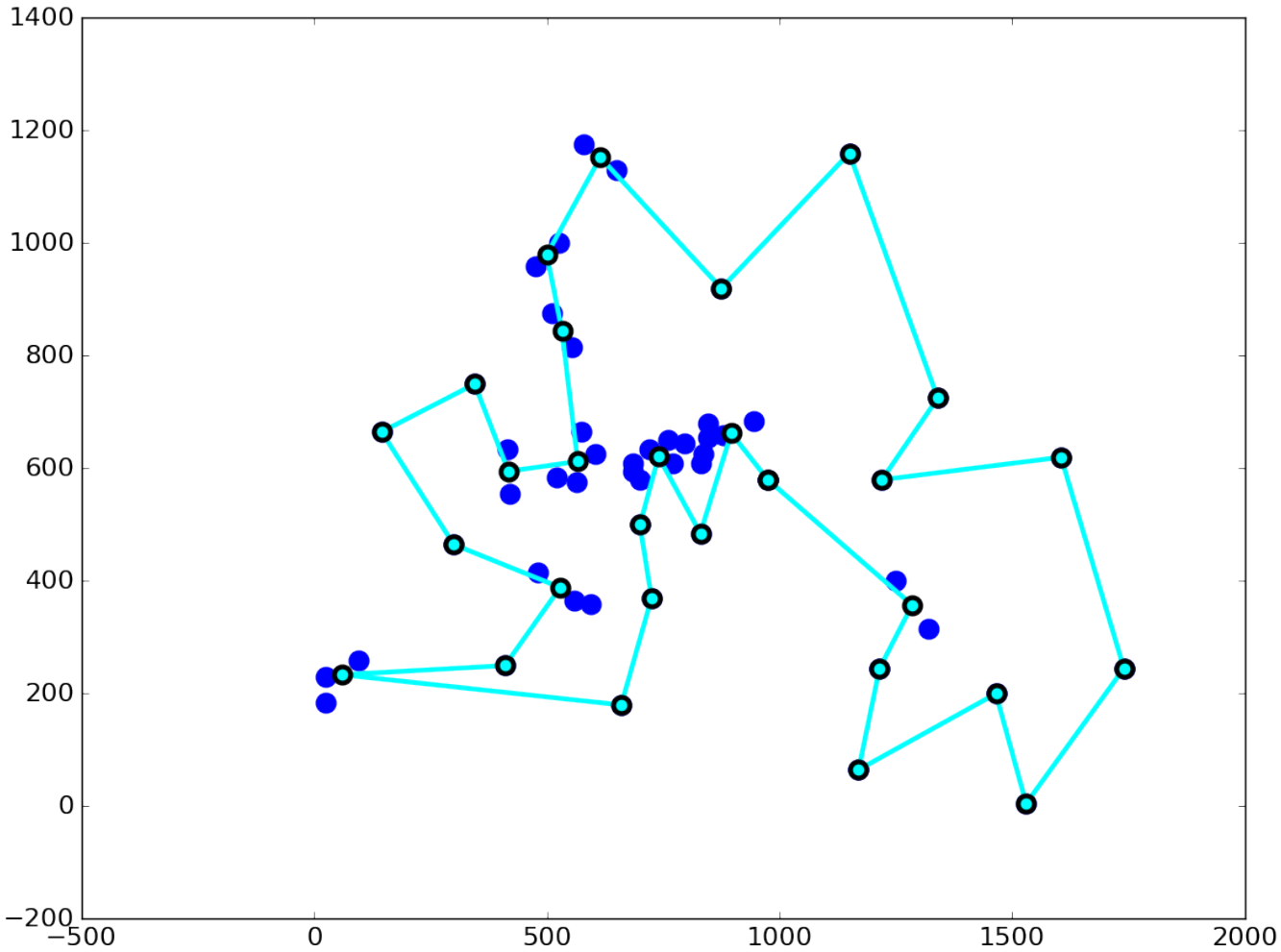
animation of the pair-center tour algorithm



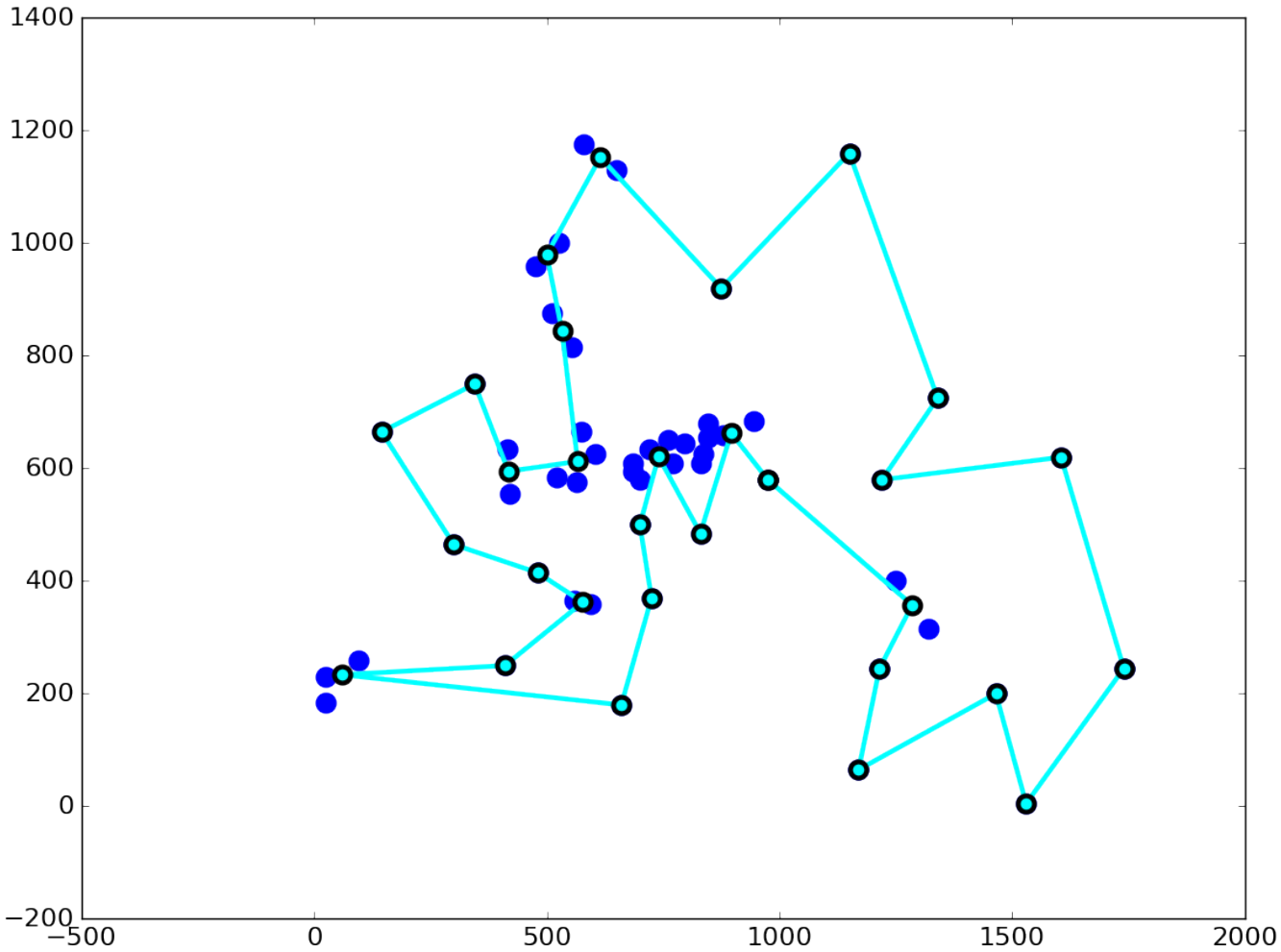
animation of the pair-center tour algorithm



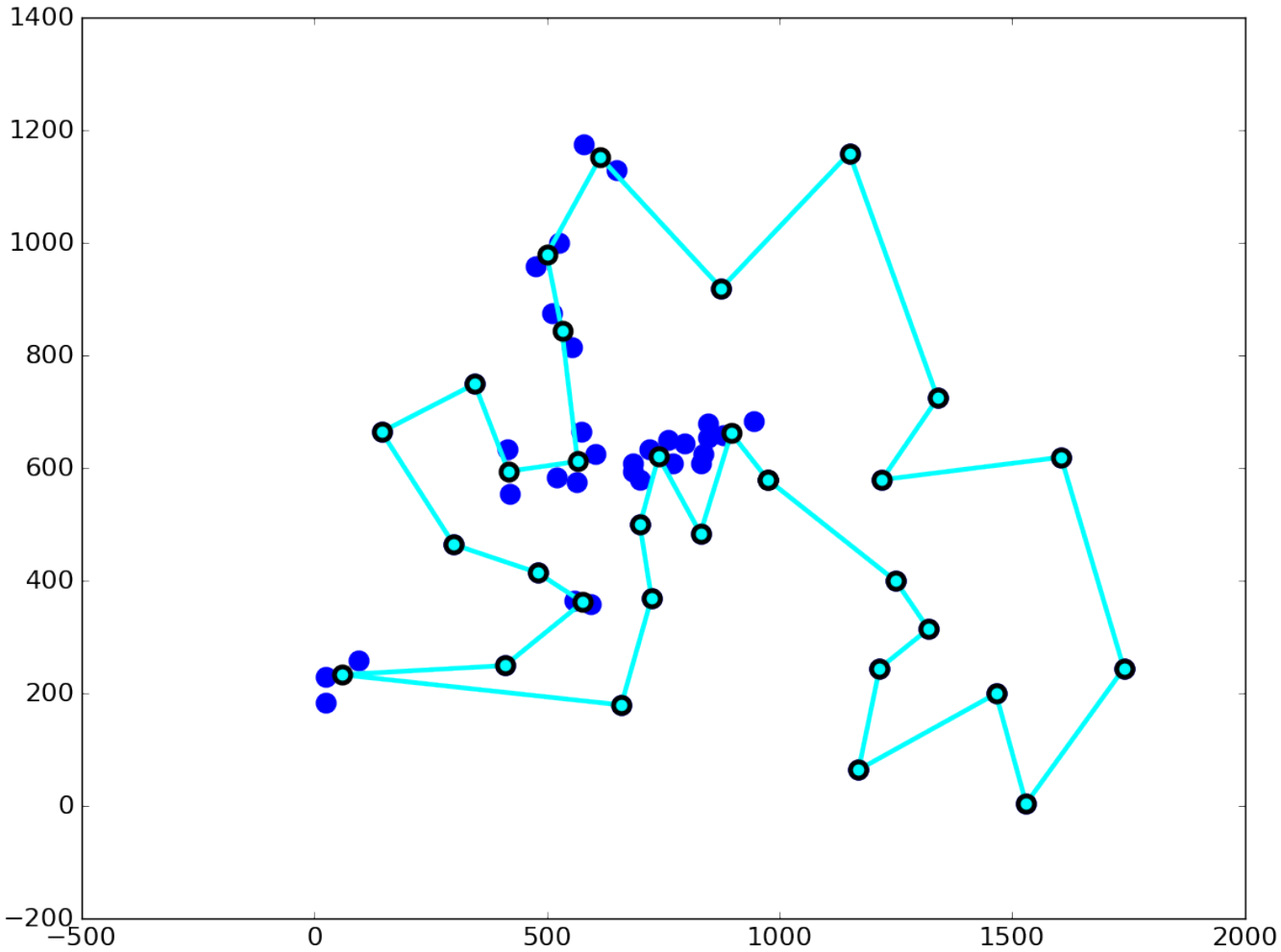
animation of the pair-center tour algorithm



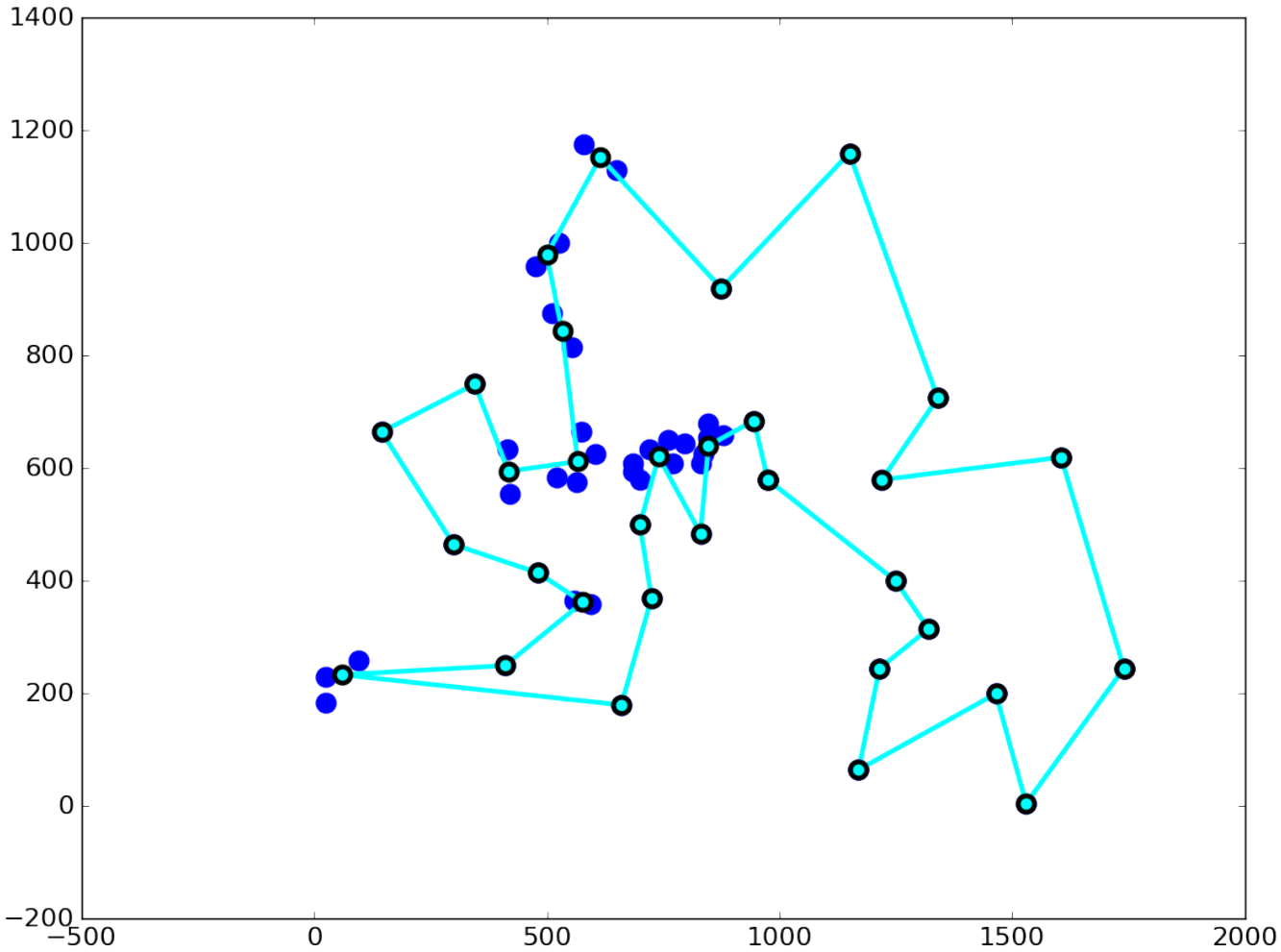
animation of the pair-center tour algorithm



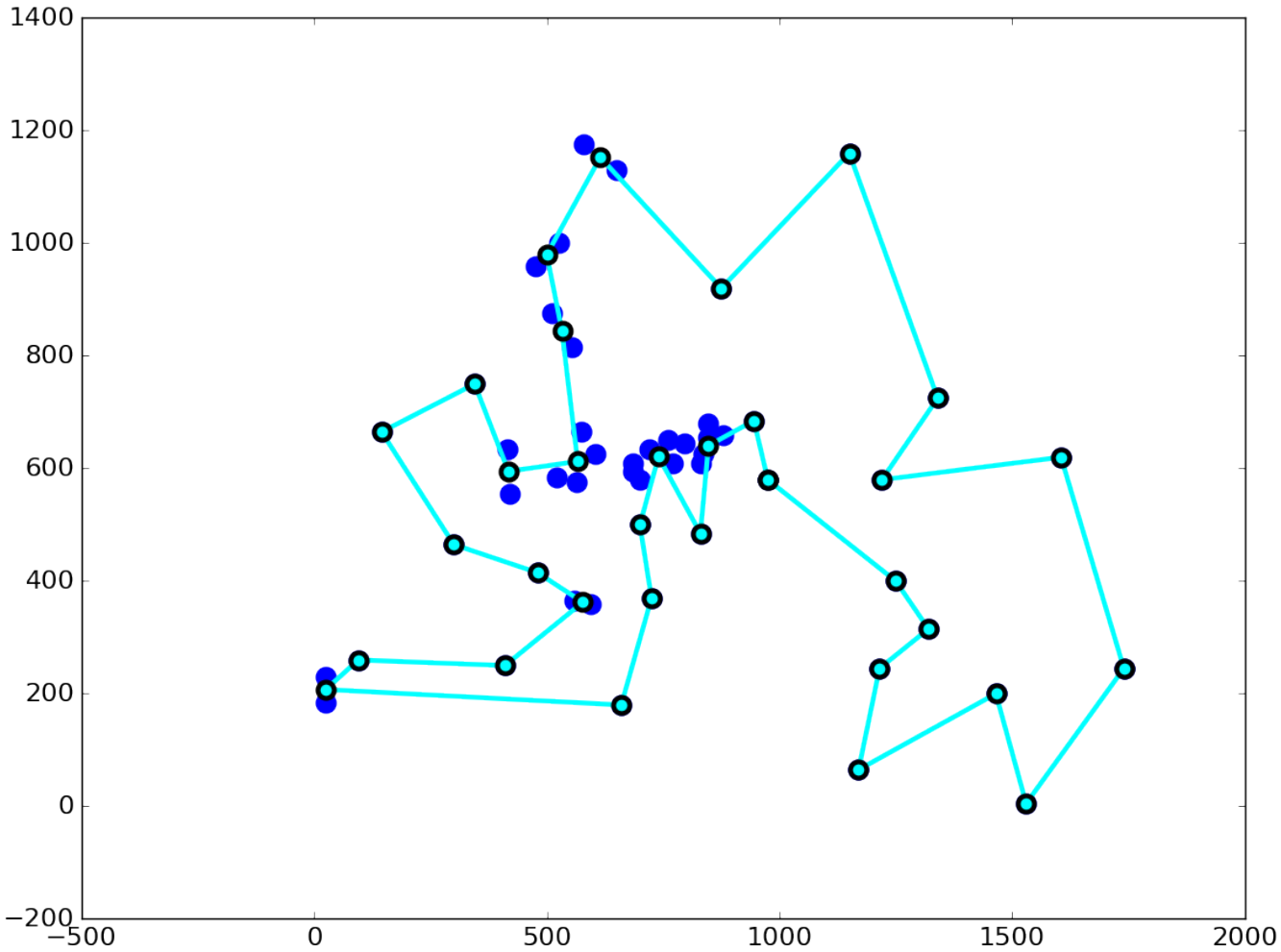
animation of the pair-center tour algorithm



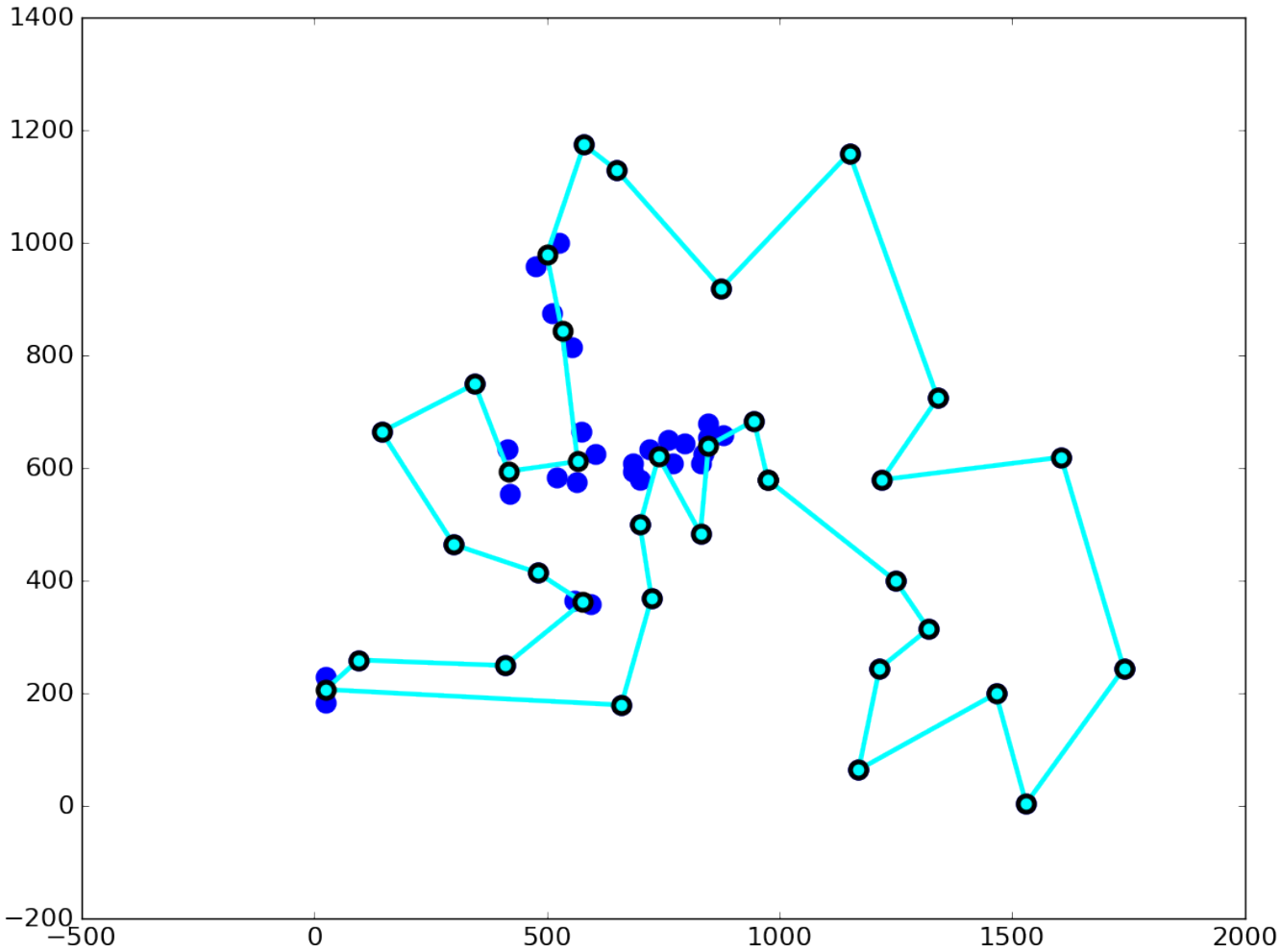
animation of the pair-center tour algorithm



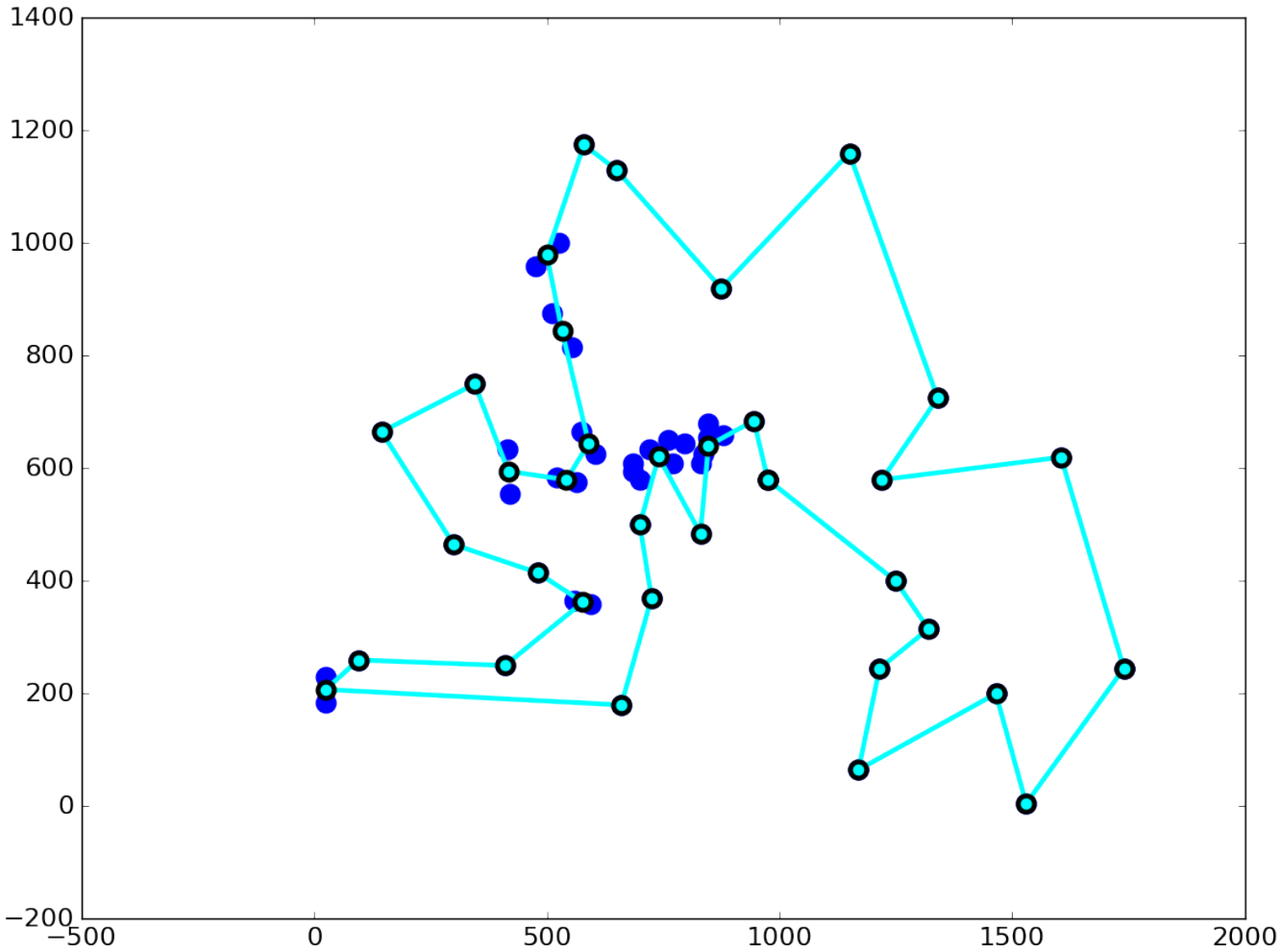
animation of the pair-center tour algorithm



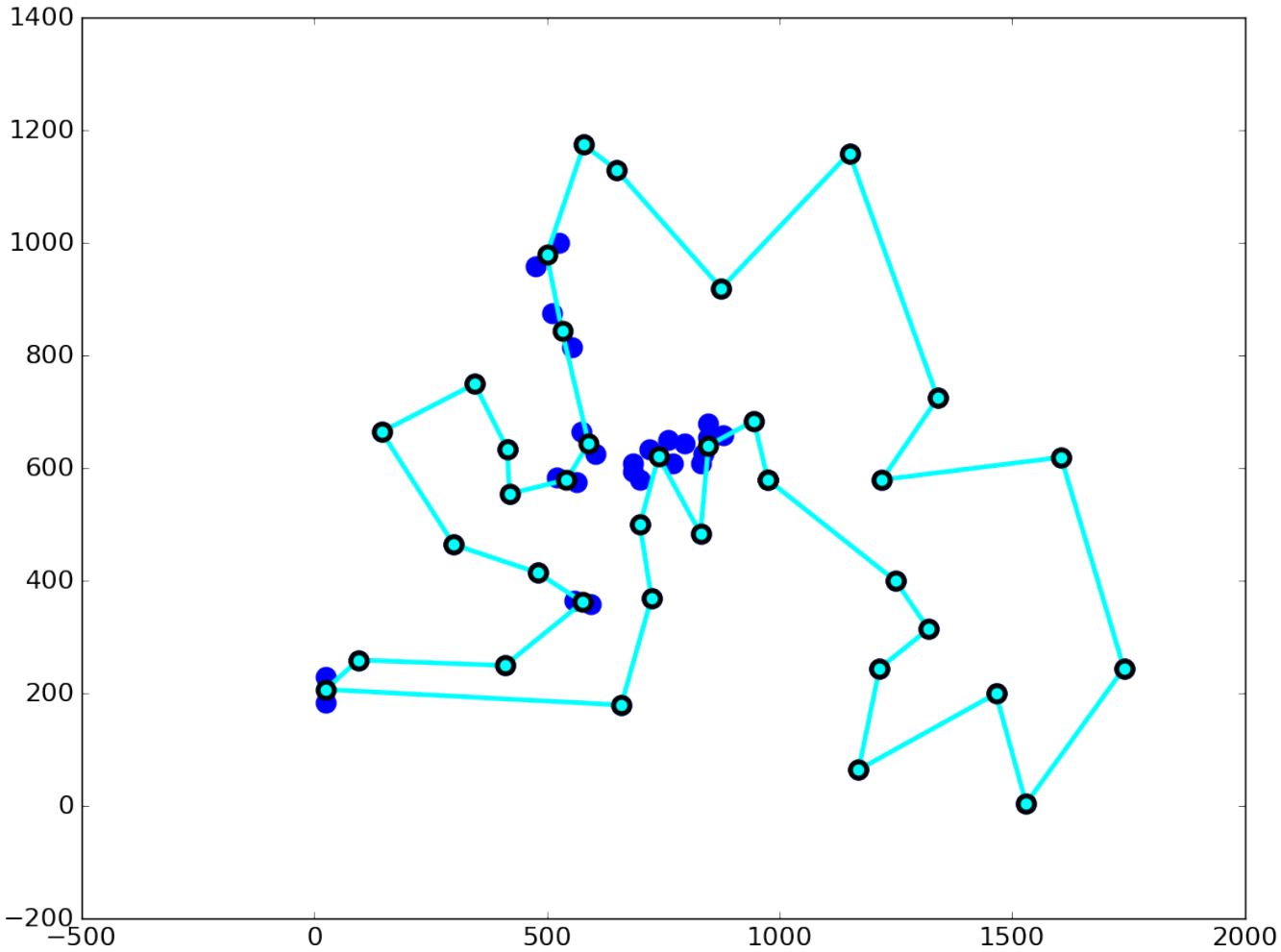
animation of the pair-center tour algorithm



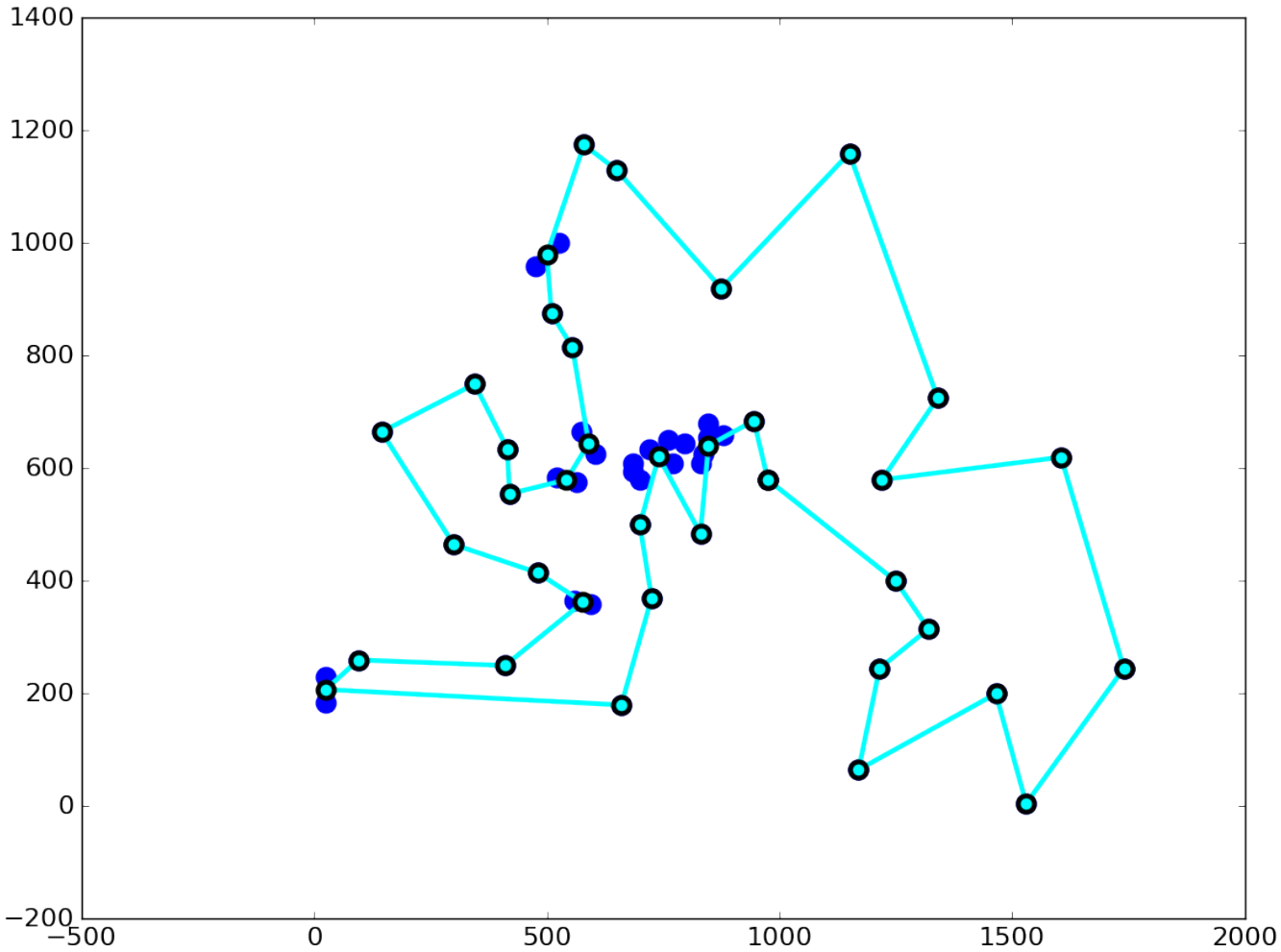
animation of the pair-center tour algorithm



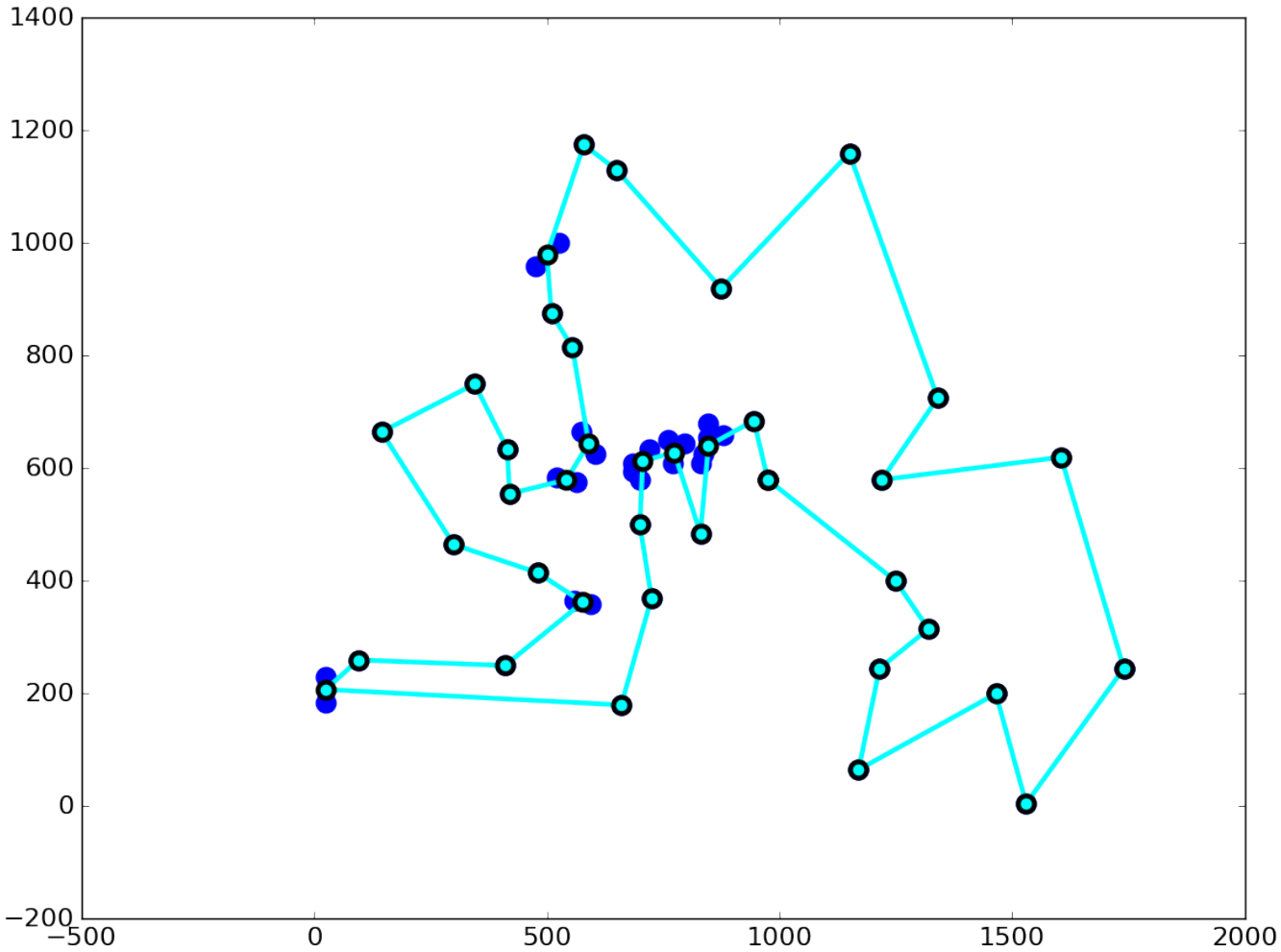
animation of the pair-center tour algorithm



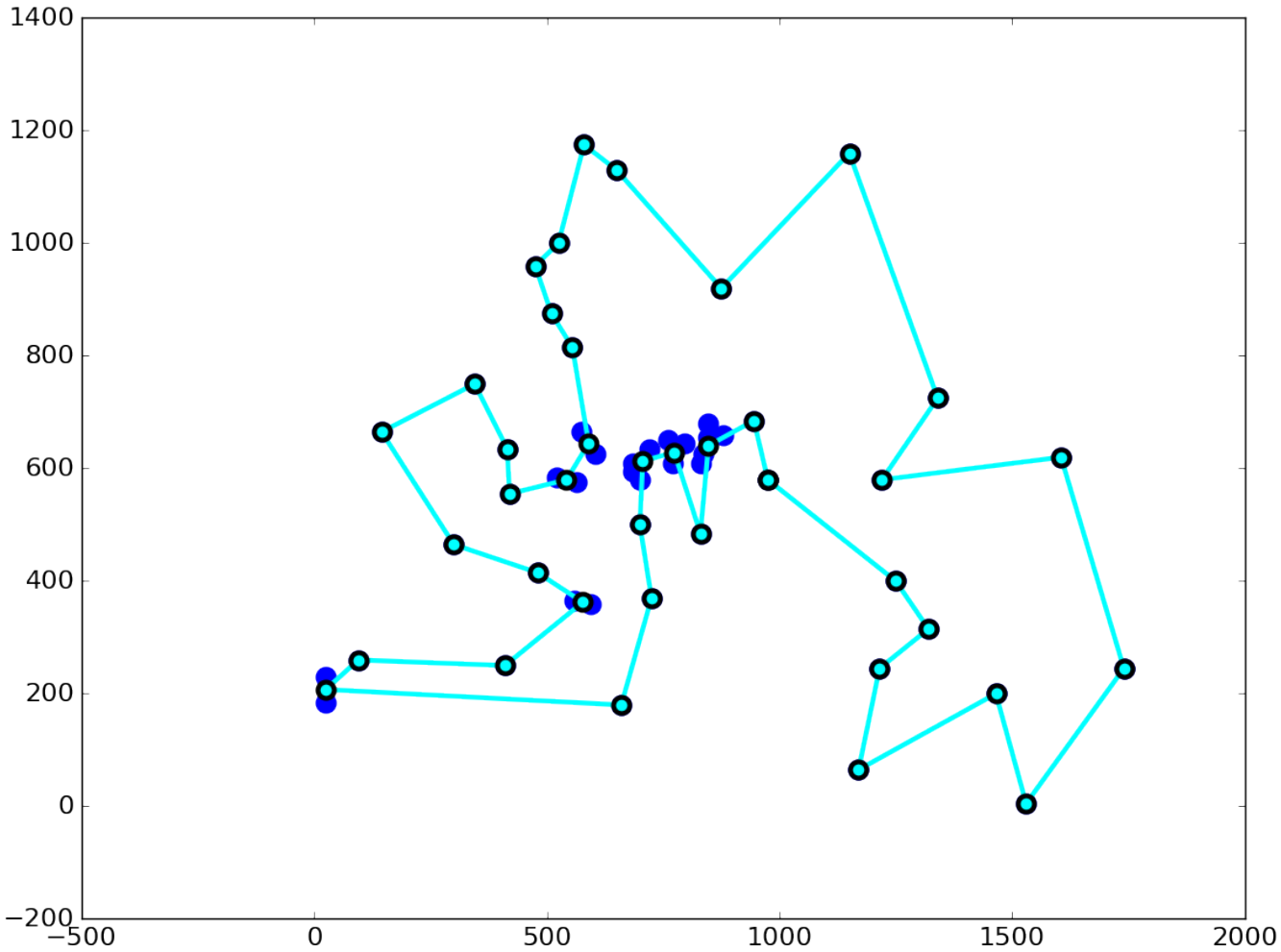
animation of the pair-center tour algorithm



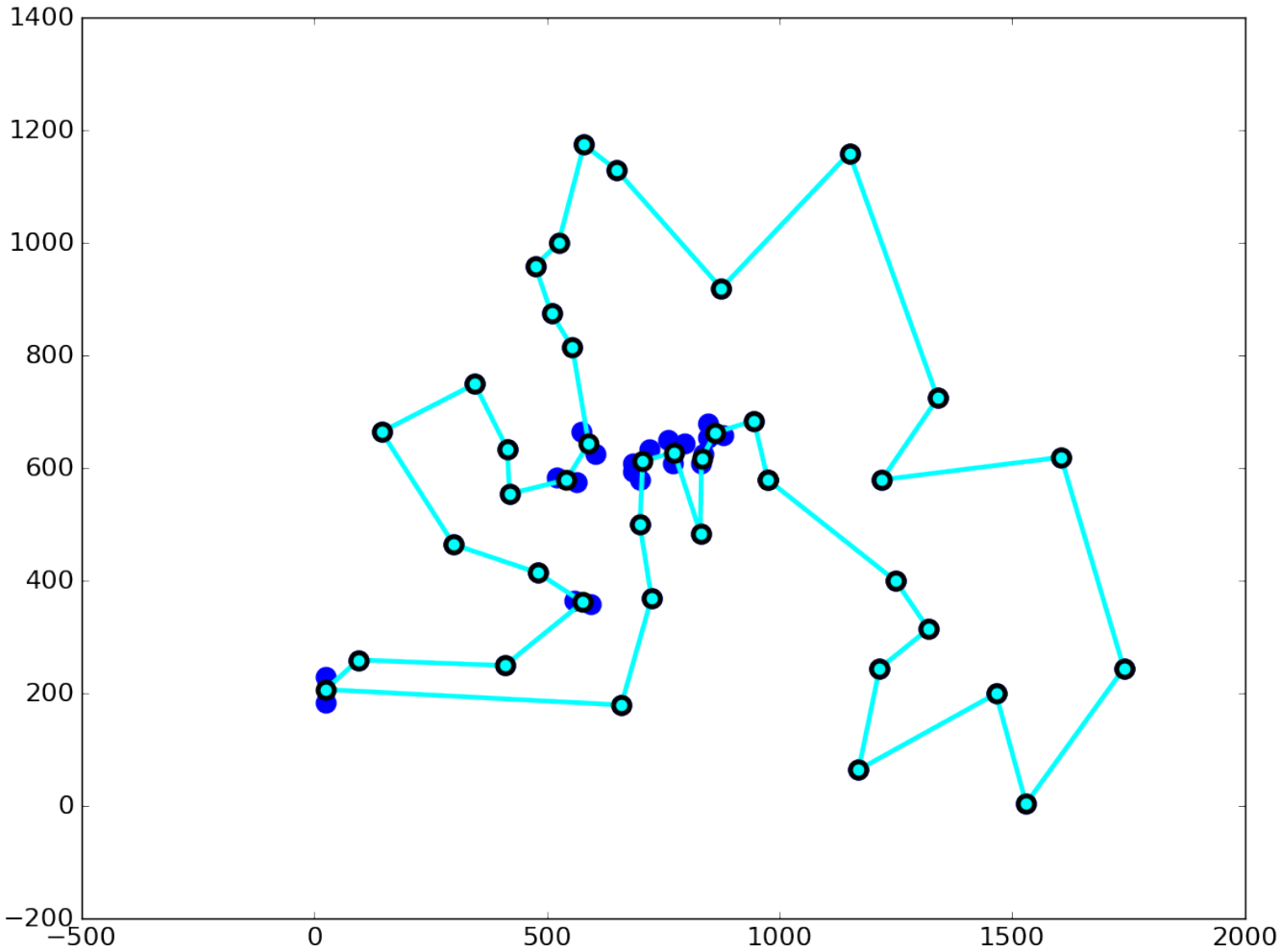
animation of the pair-center tour algorithm



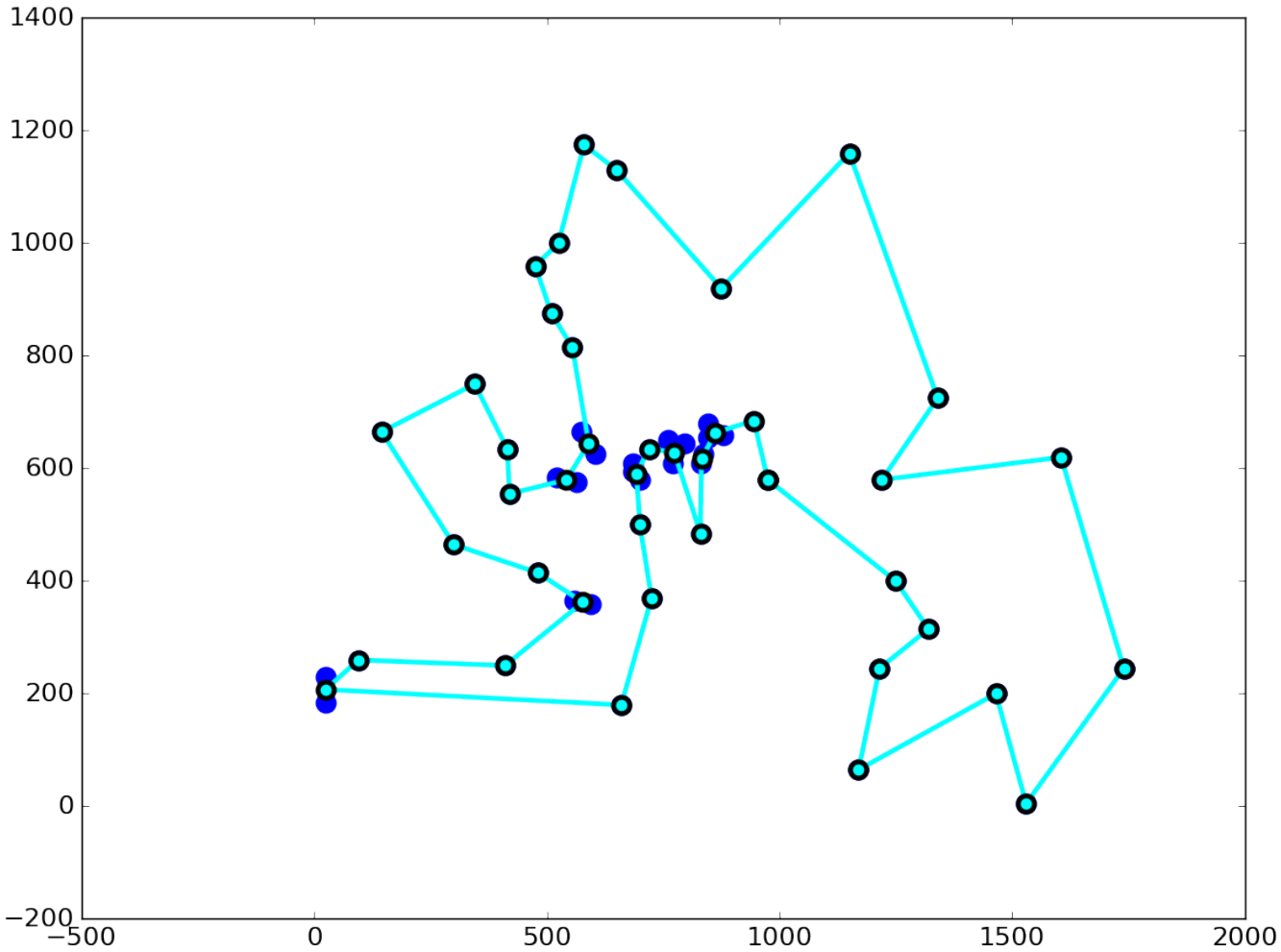
animation of the pair-center tour algorithm



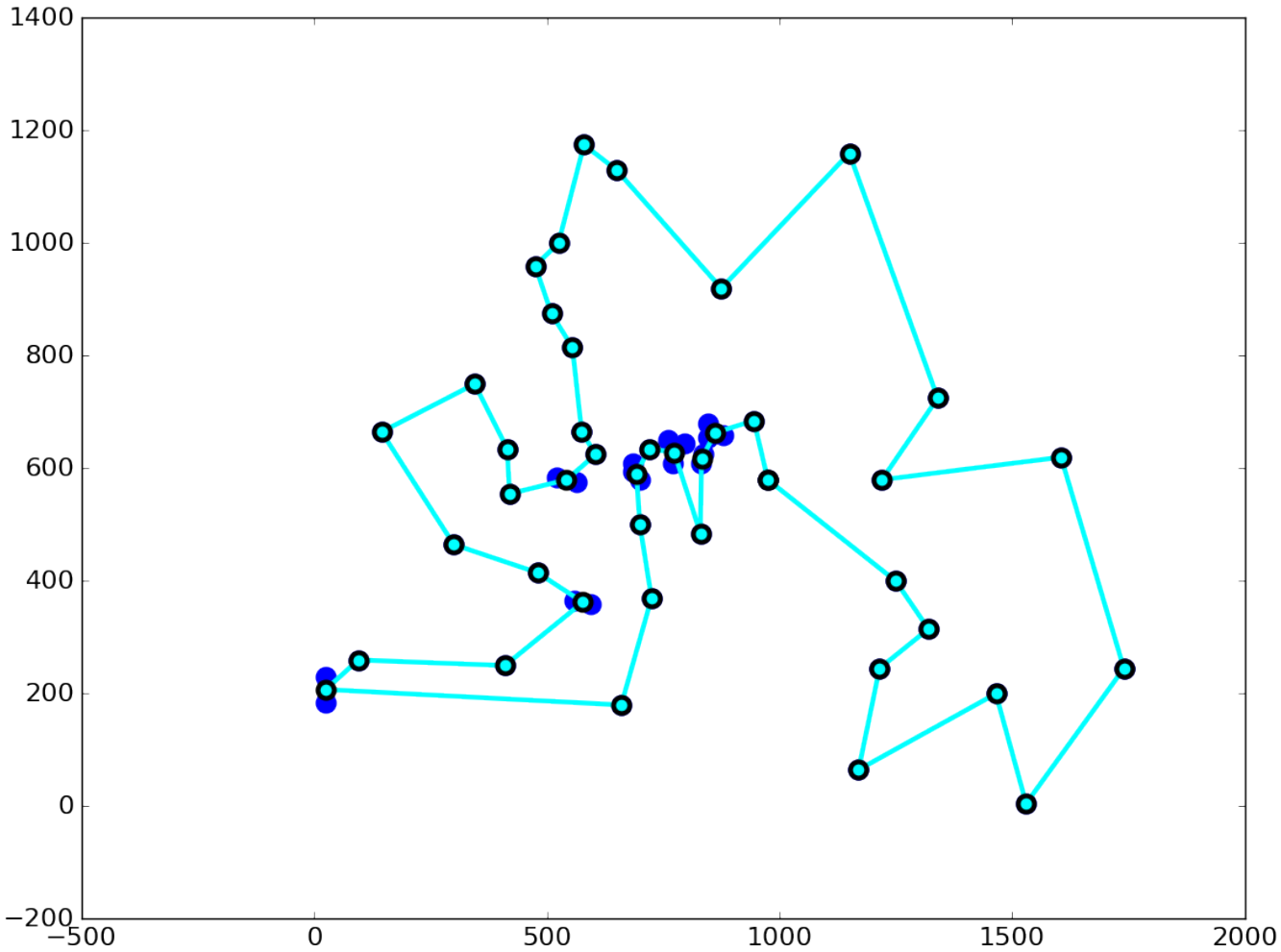
animation of the pair-center tour algorithm



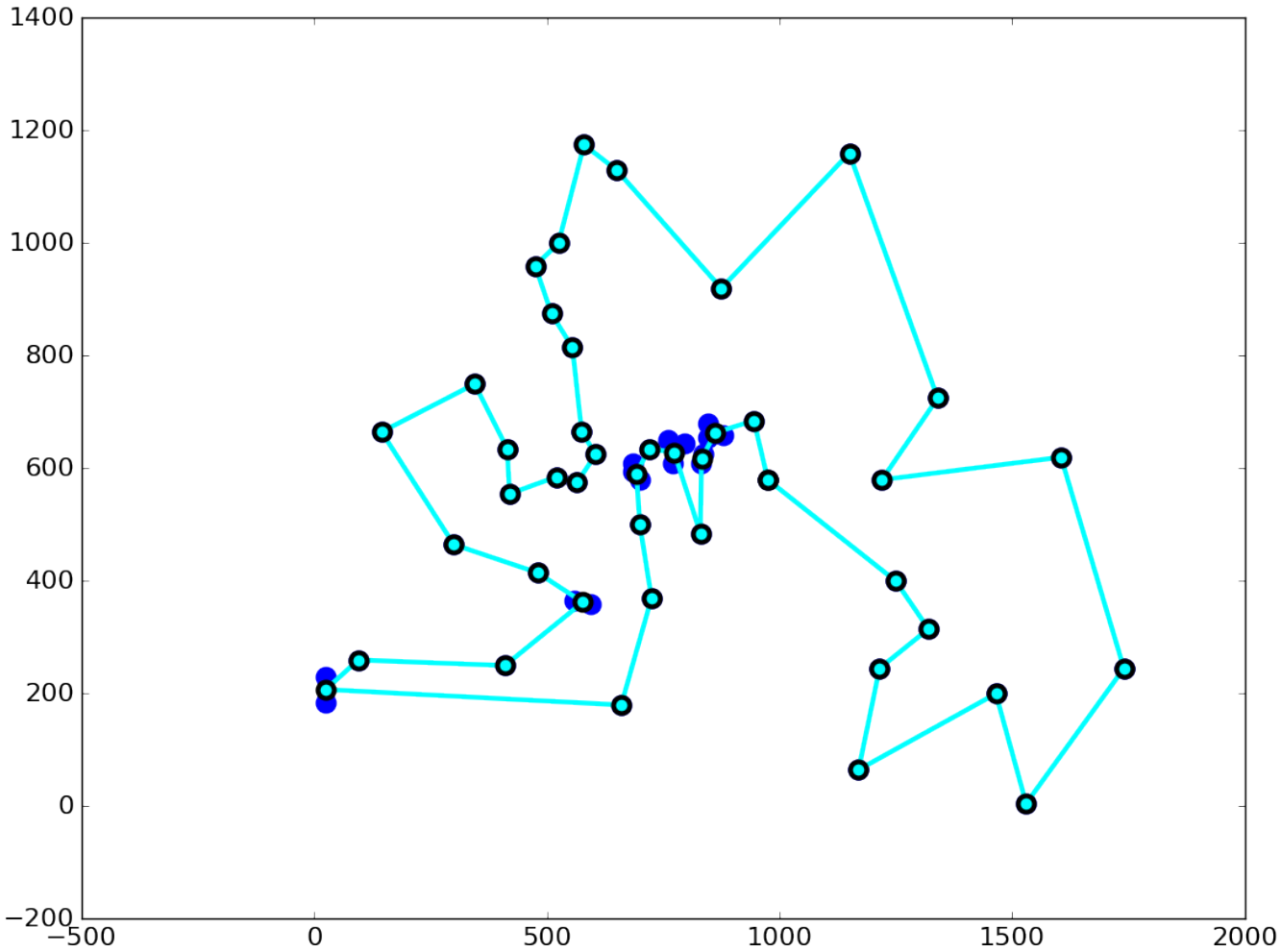
animation of the pair-center tour algorithm



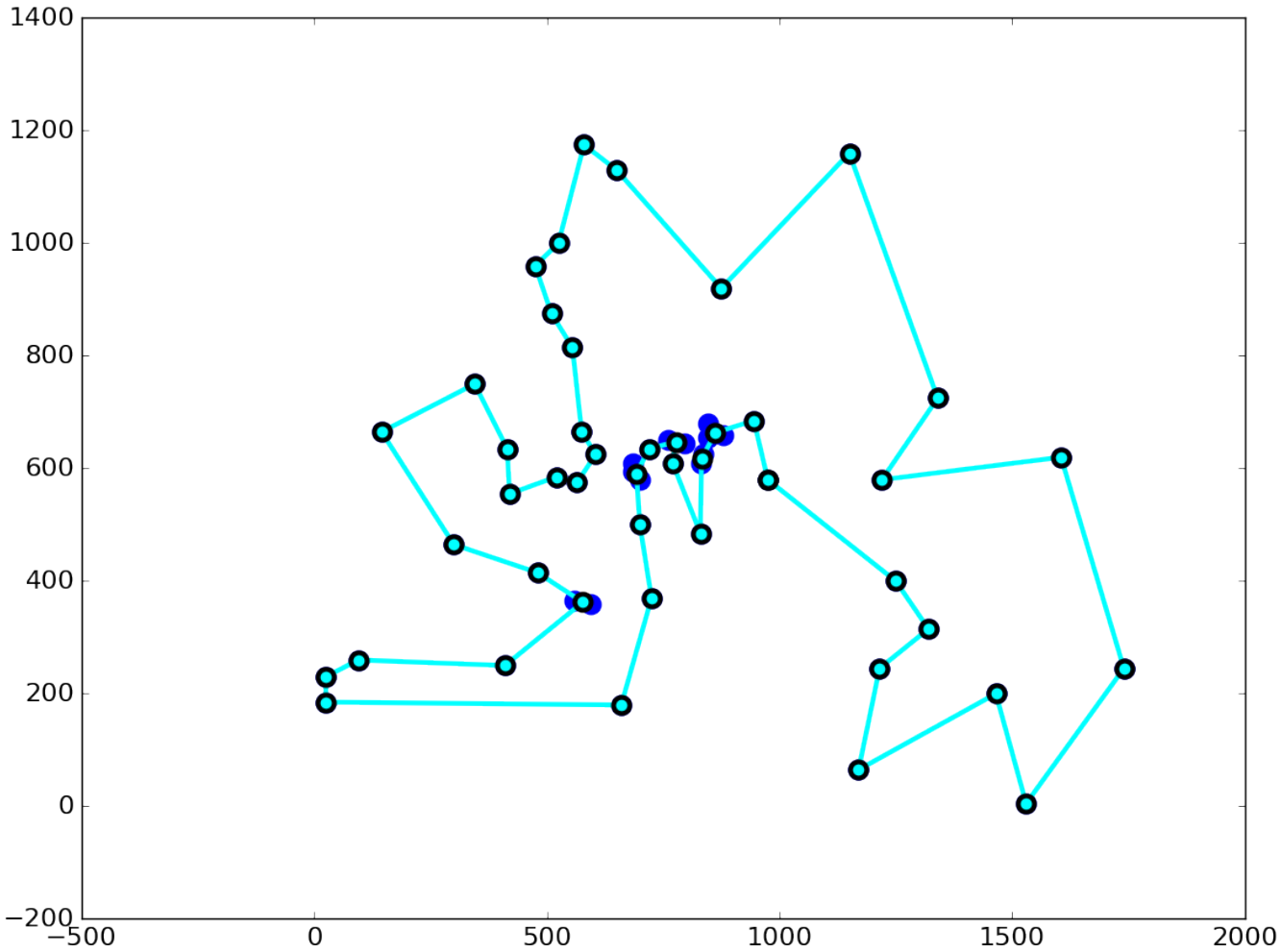
animation of the pair-center tour algorithm



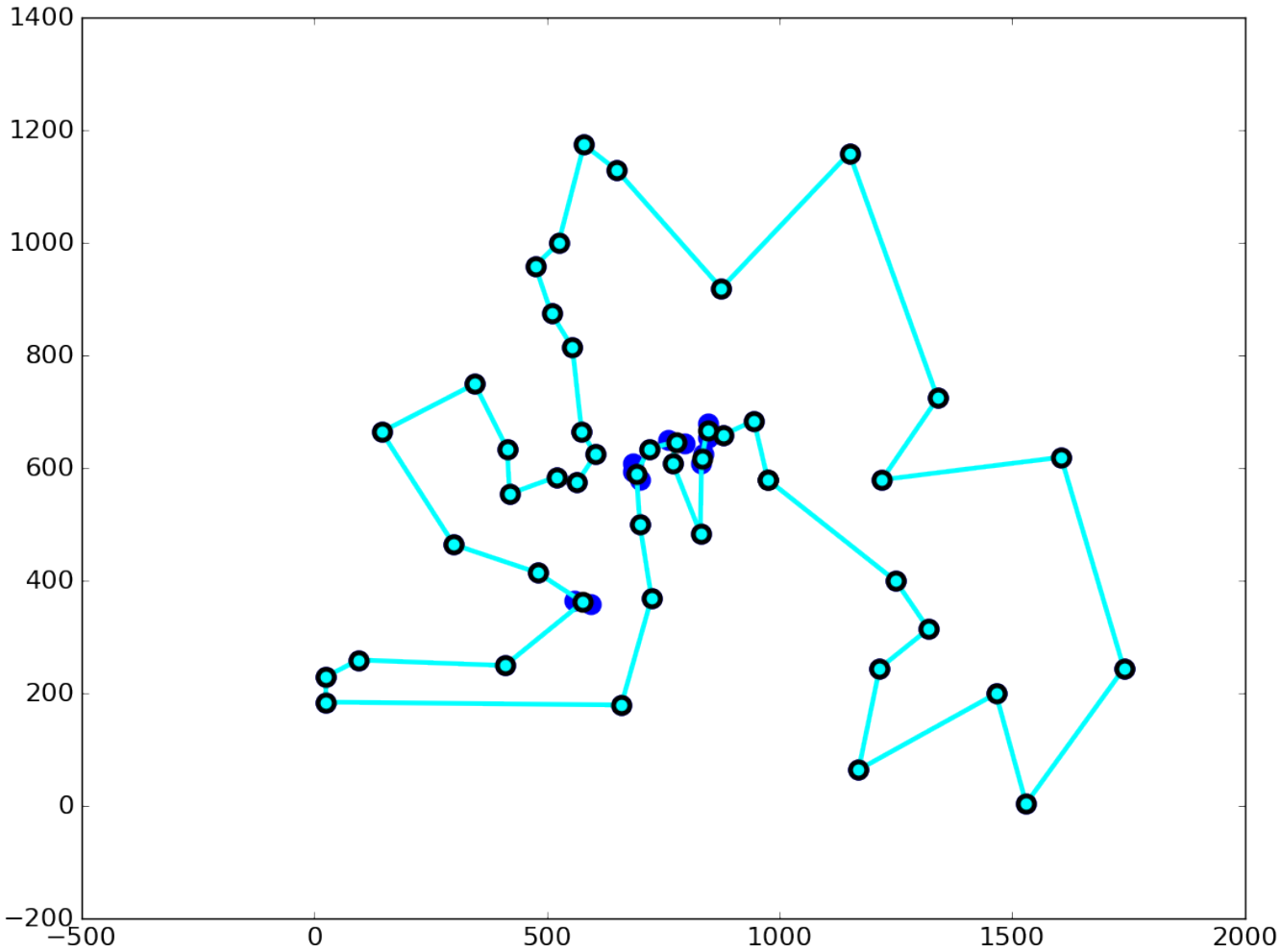
animation of the pair-center tour algorithm



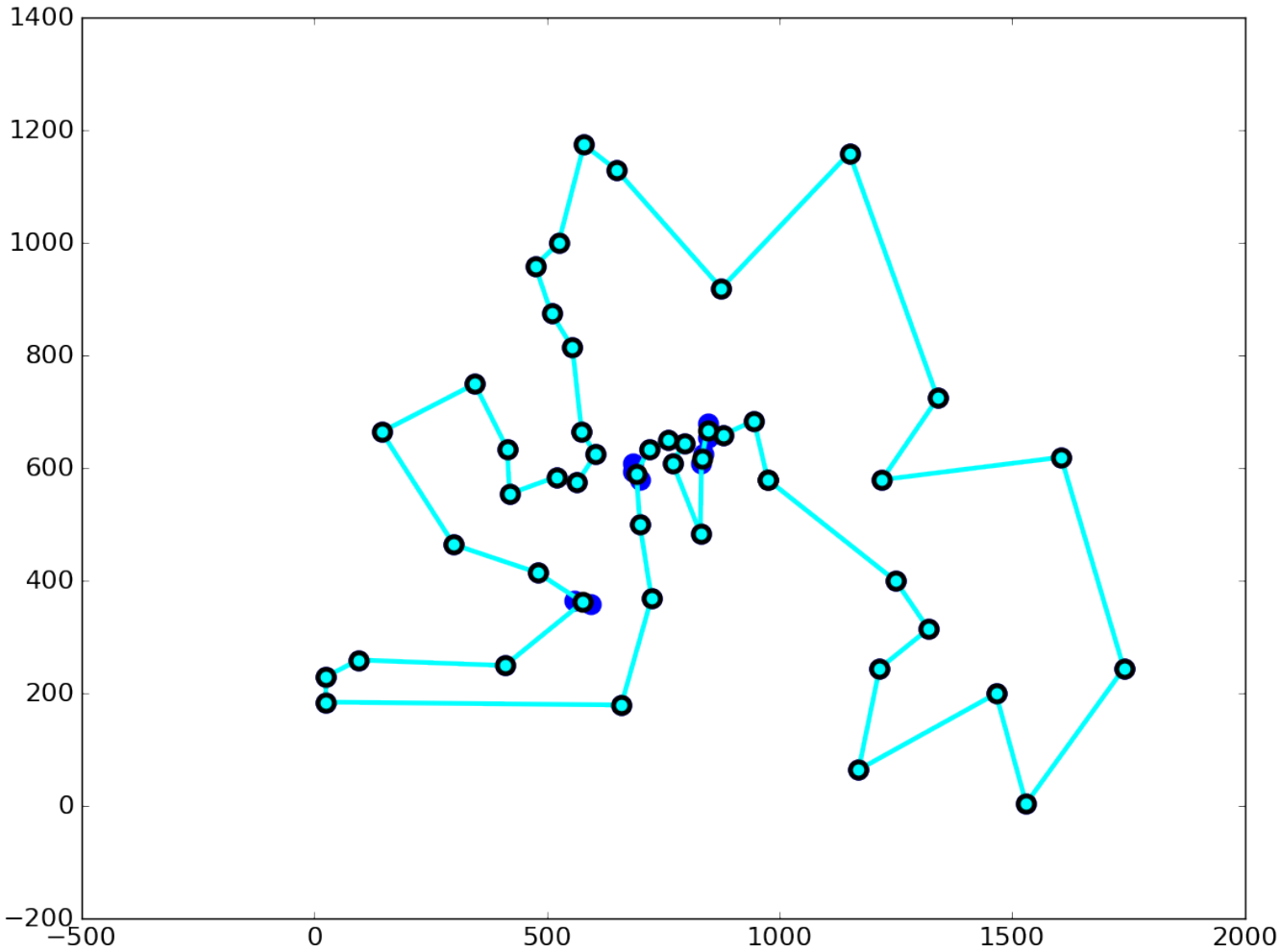
animation of the pair-center tour algorithm



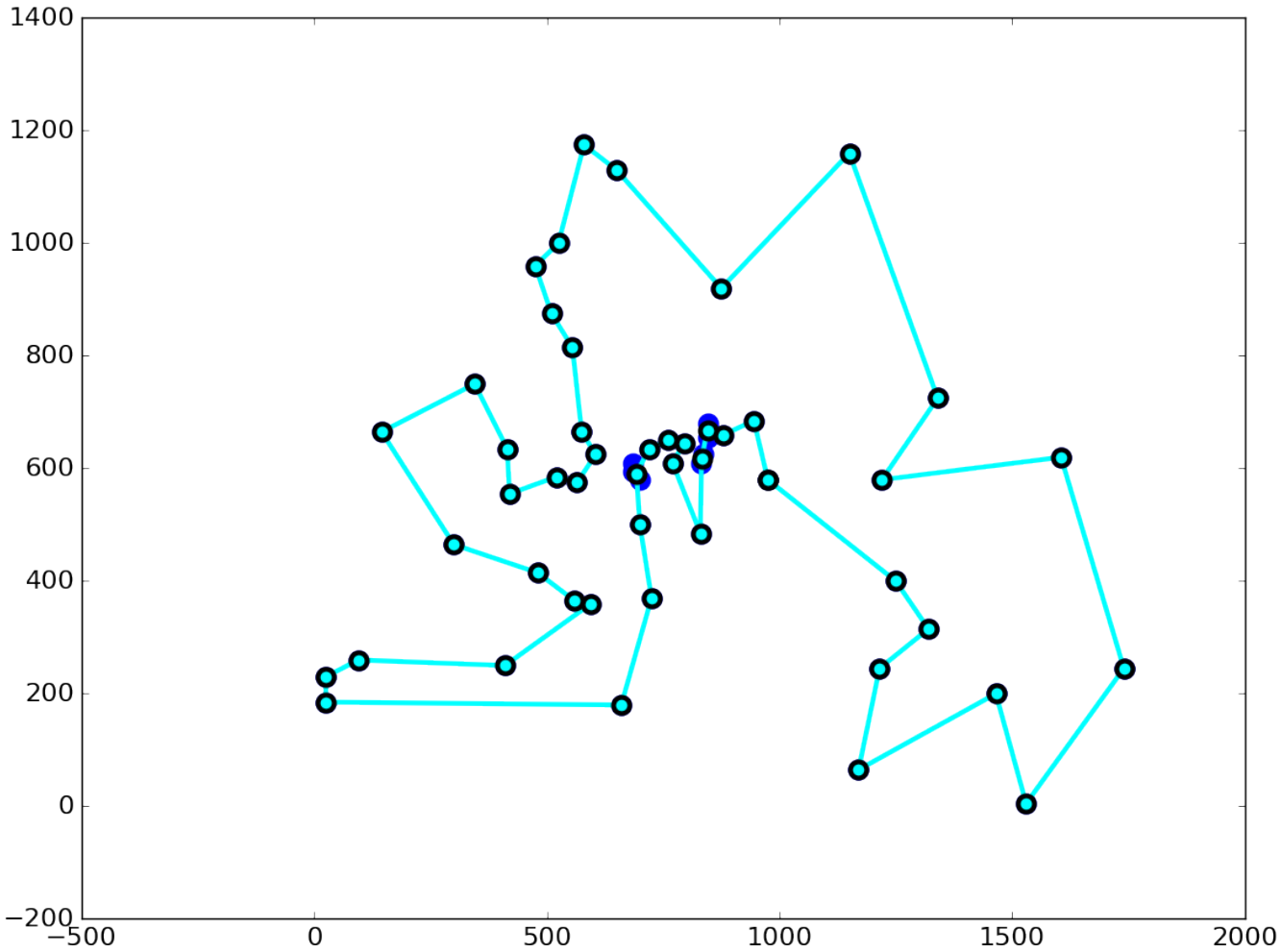
animation of the pair-center tour algorithm



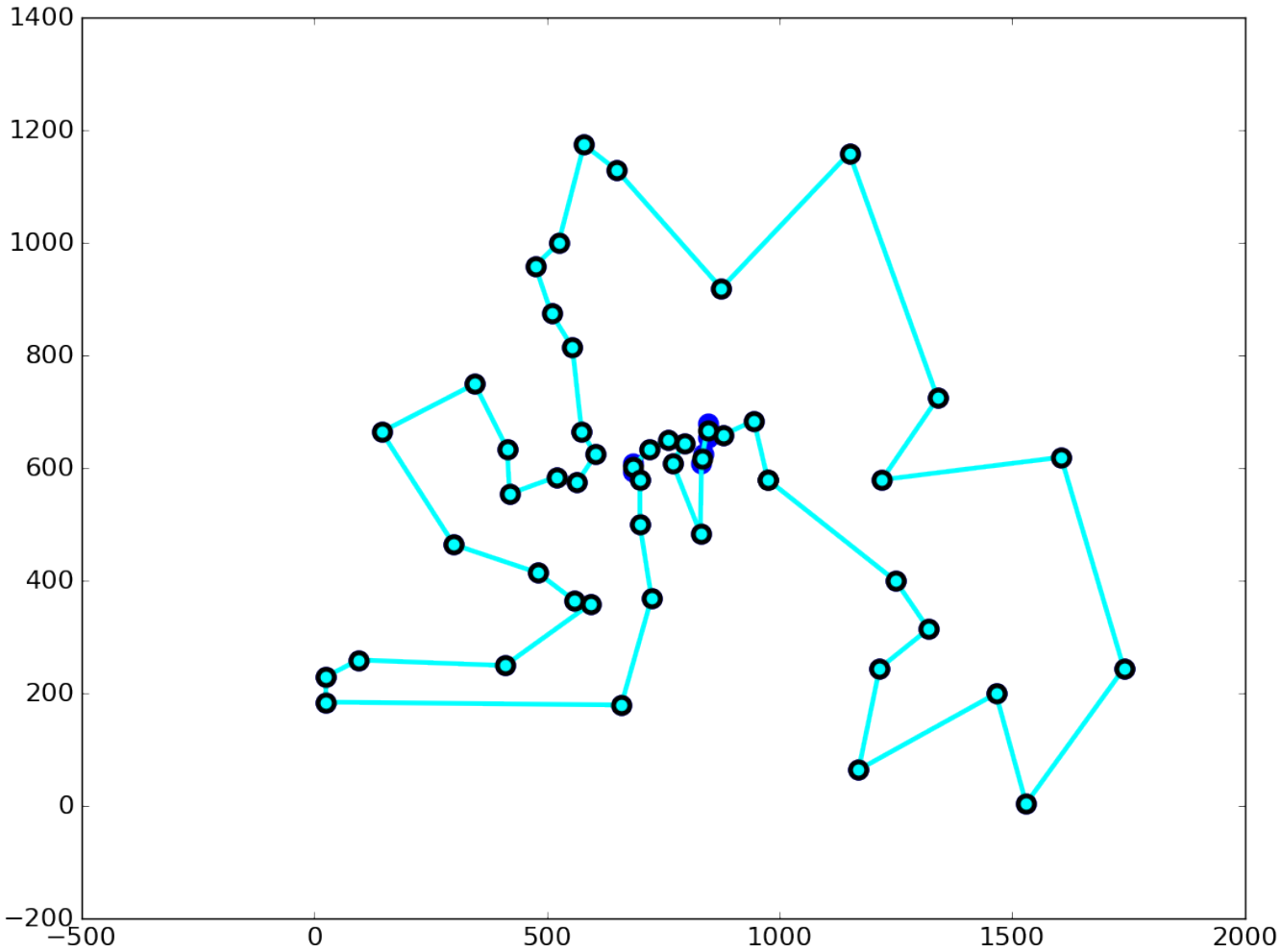
animation of the pair-center tour algorithm



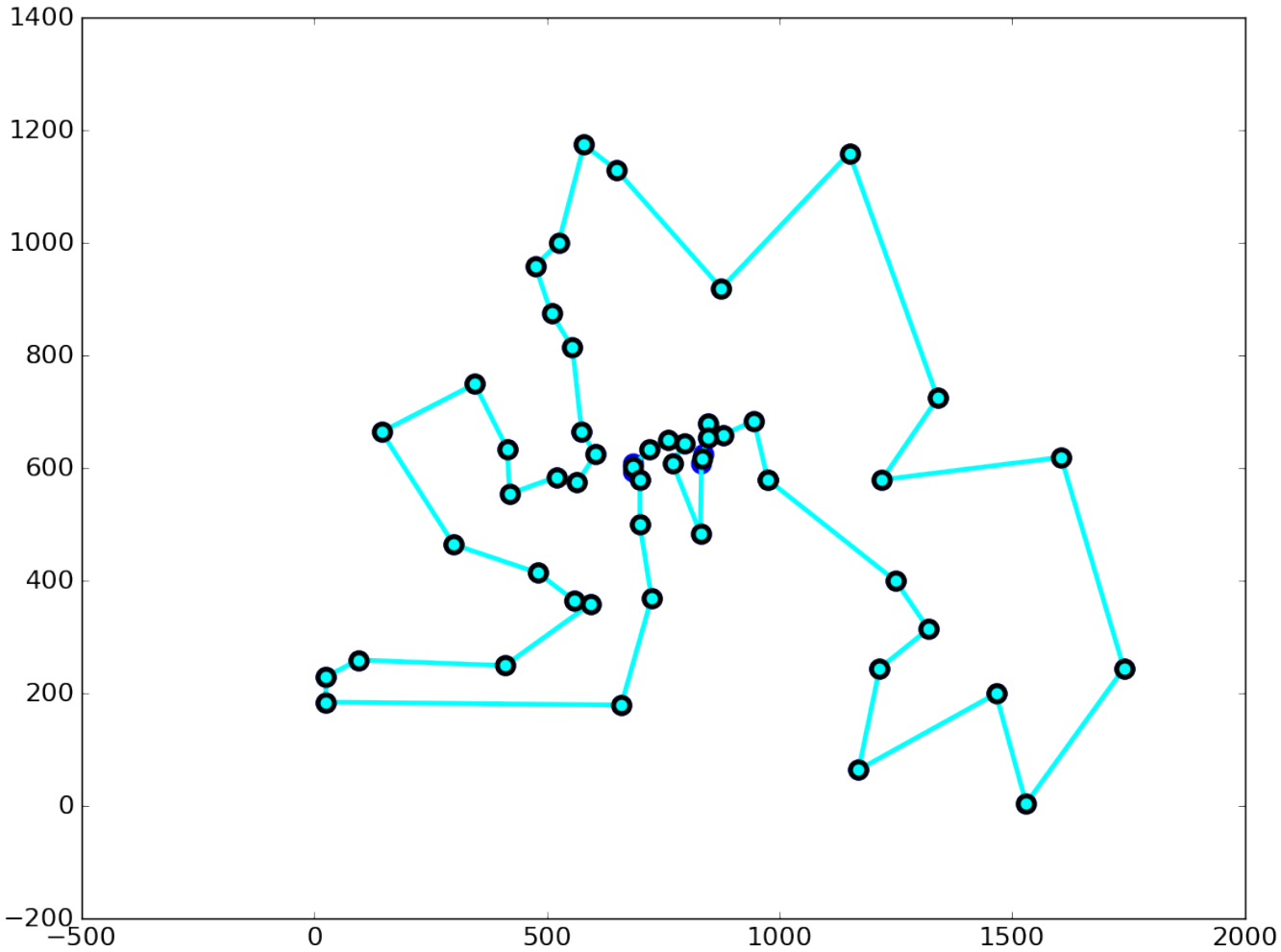
animation of the pair-center tour algorithm



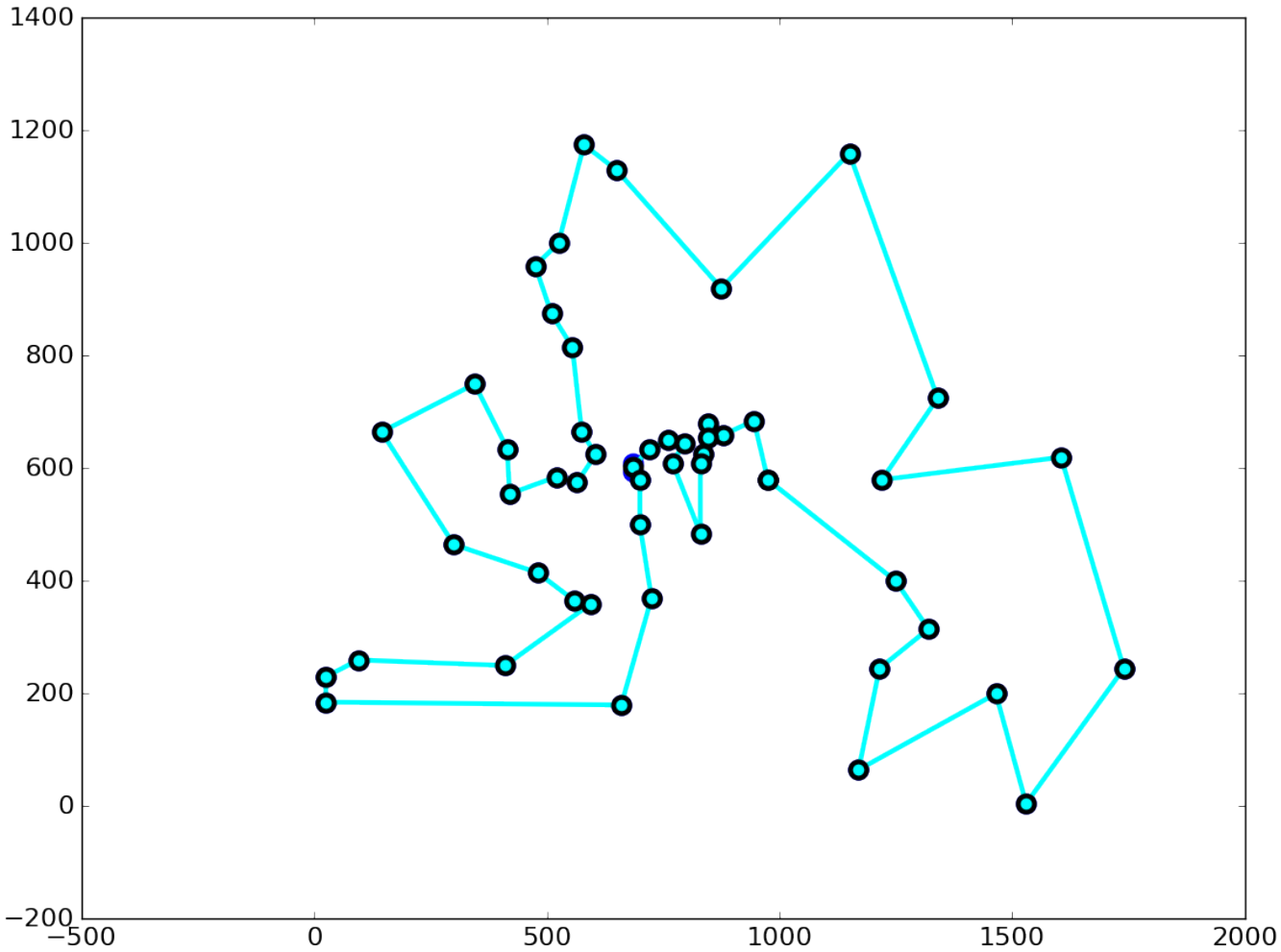
animation of the pair-center tour algorithm



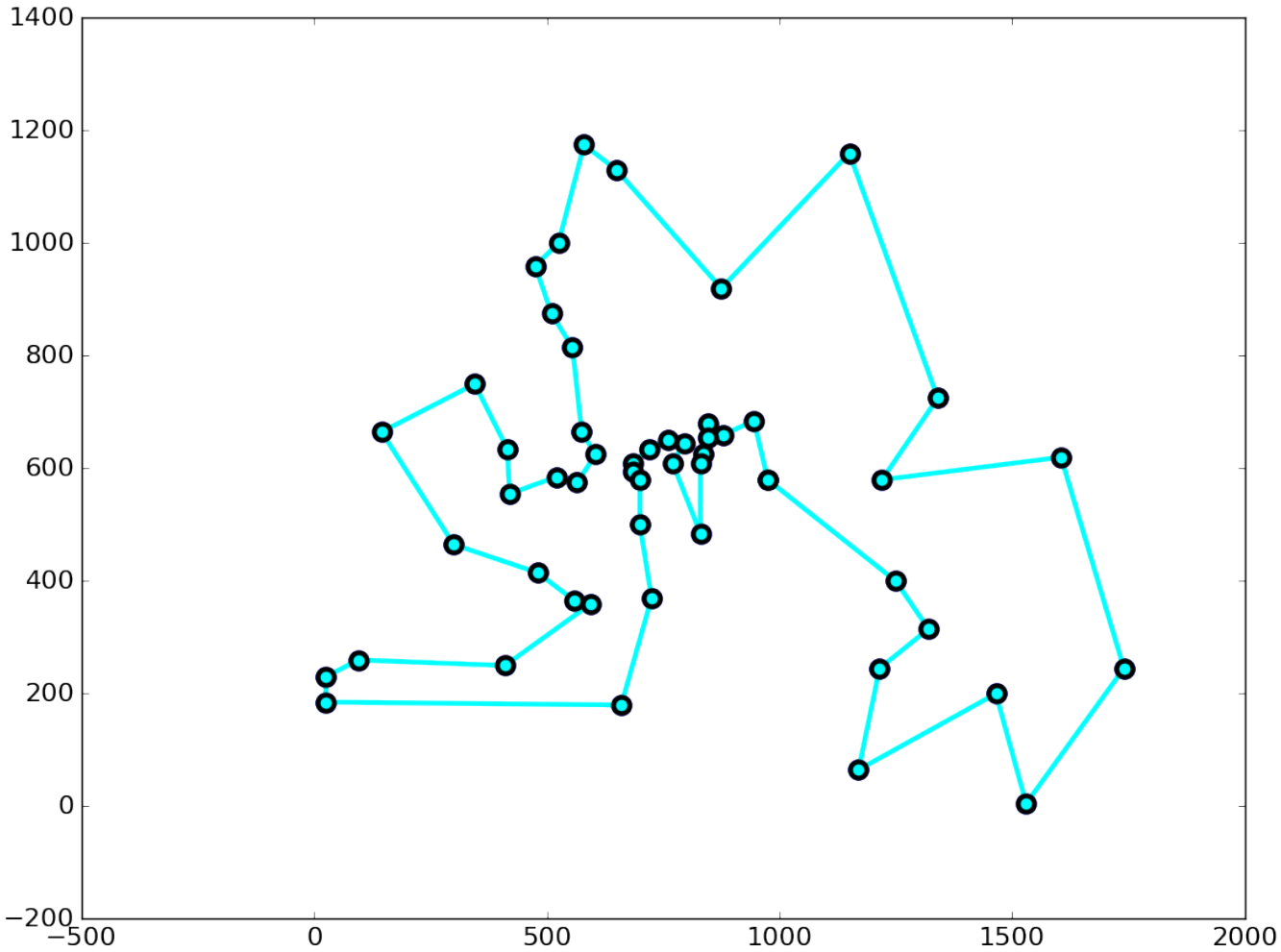
animation of the pair-center tour algorithm



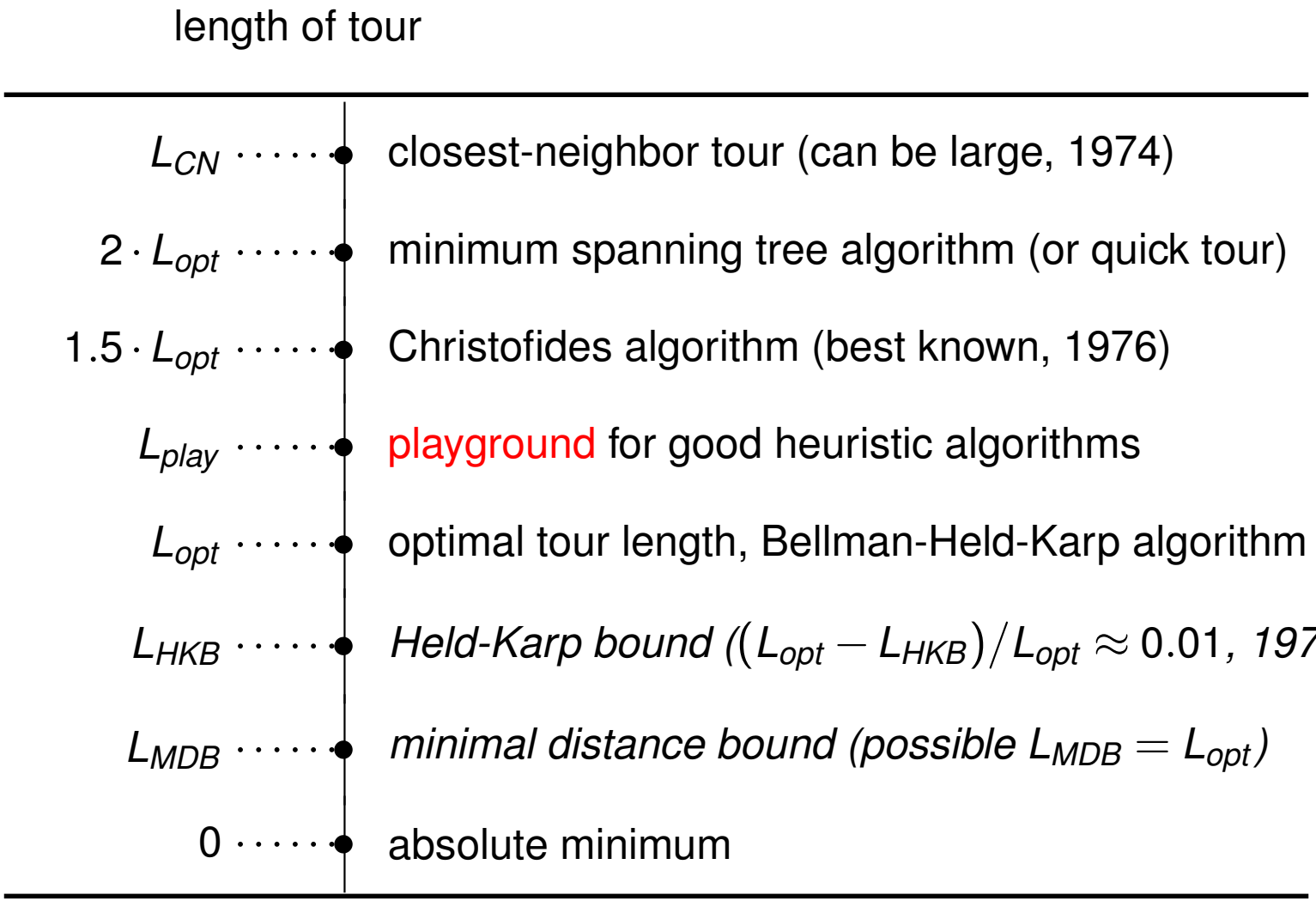
animation of the pair-center tour algorithm



animation of the pair-center tour algorithm



the TSP length-line sum-up



How does the genetic algorithm work?

A genetic algorithm is a bio-inspired probabilistic algorithm:

- initialize a set of individuals
- while *stopping criterium* not met
  - evaluate fitness of the individuals (in search space)
  - generate off-springs (mutation and crossover, in the encoding space)
  - generate a new generation, i.e., a subset of parents plus off-springs (selection)
- report best individual generated in the process

You'll work with this approach in the lab hours.

additional information

additional information and benchmark instances can be found at:

- http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ or
- https://www.math.uwaterloo.ca/tsp/data/index.html
- almost a counterexample of how to implement GA for TSP https://jaketae.github.io/study/genetic-algorithm/
- we use the work at https://github.com/guofei9987/scikit-opt

parallelization

- Monte Carlo algorithms, and hence, evolutionary algorithms are often quite easy to parallelize.
- We will not talk about parallelization in this course, however, it's an important issue in order to achieve good performance on modern systems.

the sorting problem

- Sorting is a basic, well known, and well studied problem.
- Given a sequence of n elements belonging to an orderable set, we have to compute a permutation of the input elements such that they are ordered (according to the underlying compare function).
- Simple example: given a sequence of integer numbers; sort ascending.
- Note that there are $n!$ possible permutations.
  (Funny, the same number as there are tours in TSP.)

checking a solution

- Before doing the actual sorting, let's first design an algorithm that checks the results, i.e., checks that the sequence is sorted.
- Remember: we require that we can check with an algorithm that the output/result of our initial algorithm is correct (i.e., fulfills the corresponding properties).
- Personally, I recommend that you always try to design and implement such a checker!

checker for the sorting problem

Check whether a pair of elements is sorted:

```
def PairIsSorted(v,i,j):
  return v[i]<=v[j]
```

Check whether a sequence of elements is sorted:

```
def IsSorted(v):
  for i in range(len(v)-1):
    if not PairIsSorted(v,i,i+1):
      return False
  return True
```

the bubble sort algorithm

A simple sorting algorithm (bubble sort):

```
def PairSort(v,i,j):  # sorts a pair
  if not PairIsSorted(v,i,j):
    v[i],v[j]=v[j],v[i]

def BubbleSort(v):    # LasVegas type
  while not IsSorted(v):
    for i in range(len(v)-1):
      PairSort(v,i,i+1)
```

Runs in quadratic time (worst case) and linear time (best case).

Monte Carlo sorting

Let us implement a Monte Carlo sorting algorithm:
we select a random pair of elements and interchange when necessary:

```
def MonteCarloSort(v,rounds):
  for j in range(rounds):
    i,j=RandomPair(v)
    PairSort(v,i,j)
```

Whenever the number of rounds is sufficiently large and we are lucky, the sequence will become sorted.

Las Vegas sorting

Let us implement a Las Vegas sorting algorithm: we select a random pair of elements, interchange when necessary, and stop when the sequence is sorted:

```
def LasVegasSort(v):
  while not IsSorted(v):
    i,j=RandomPair(v)
    PairSort(v,i,j)
```

Maybe we need to wait a very long time, but we always get a sorted sequence. Observe: Las Vegas algorithms are easy to design, when we have a checker!

Las Vegas Monte Carlo sorting

Whenever we have a checker, we can implement a Las Vegas algorithm on the base of a Monte Carlo algorithm, so for sorting we can do:

```
def LasVegasMonteCarloSort(v,rounds):
  while not IsSorted(v):
    MonteCarloSort(v,rounds)
```

Monte Carlo sort with Las Vegas condition

We can improve the Monte Carlo sort introducing a Las Vegas condition to stop earlier:

```
def MonteCarloLasVegasSort(v,rounds):
  while not IsSorted(v) and rounds>0:
    i,j=RandomPair(v)
    PairSort(v,i,j)
    rounds-=1
```

This idea reflects the general structure of a heuristic probabilistic algorithm: for a certain time do something maybe useful, and stop when a certain condition is met.

Efficient sorting

An efficient $O(n \log n)$ algorithm to sort is the merge-sort algorithm, here written in its iterative form (divide and conquer paradigm):

```
def Merge(v,w,left,middle,right):
  i,j=left,middle
  for k in range(left,right):
    if j>=right or (i<middle and PairIsSorted(v,i,j)):
      w[k]=v[i]; i+=1
    else:
      w[k]=v[j]; j+=1

def MergeSort(w):
  s,n=1,len(w)
  while s<n:
    v=w[:]
    for left in range(0,n,2*s):
      Merge(v,w,left,left+s,min(left+2*s,n))
    s*=2
```

sorting as an optimization problem

- In order to state the integer sorting problem as an optimization problem, we need to specify an objective function.
- Let $x = (x_1, x_2, \ldots, x_n)$ be the current sequence of integer values.
- We use $f(x) = \sum_{i=1}^{n} i \cdot x_i$ as objective function.
- Our aim is to maximize $f(x)$ at which point the sequence x is sorted; to minimize we take the negative value:

```
def SortObjective(v):
  f=0
  for i in range(len(v)):
    f+=v[i]*(i+1)
  return -f
```

or

```
def SortObjective(v):
  return -sum([v[i]*(i+1) for i in range(len(v))])
```

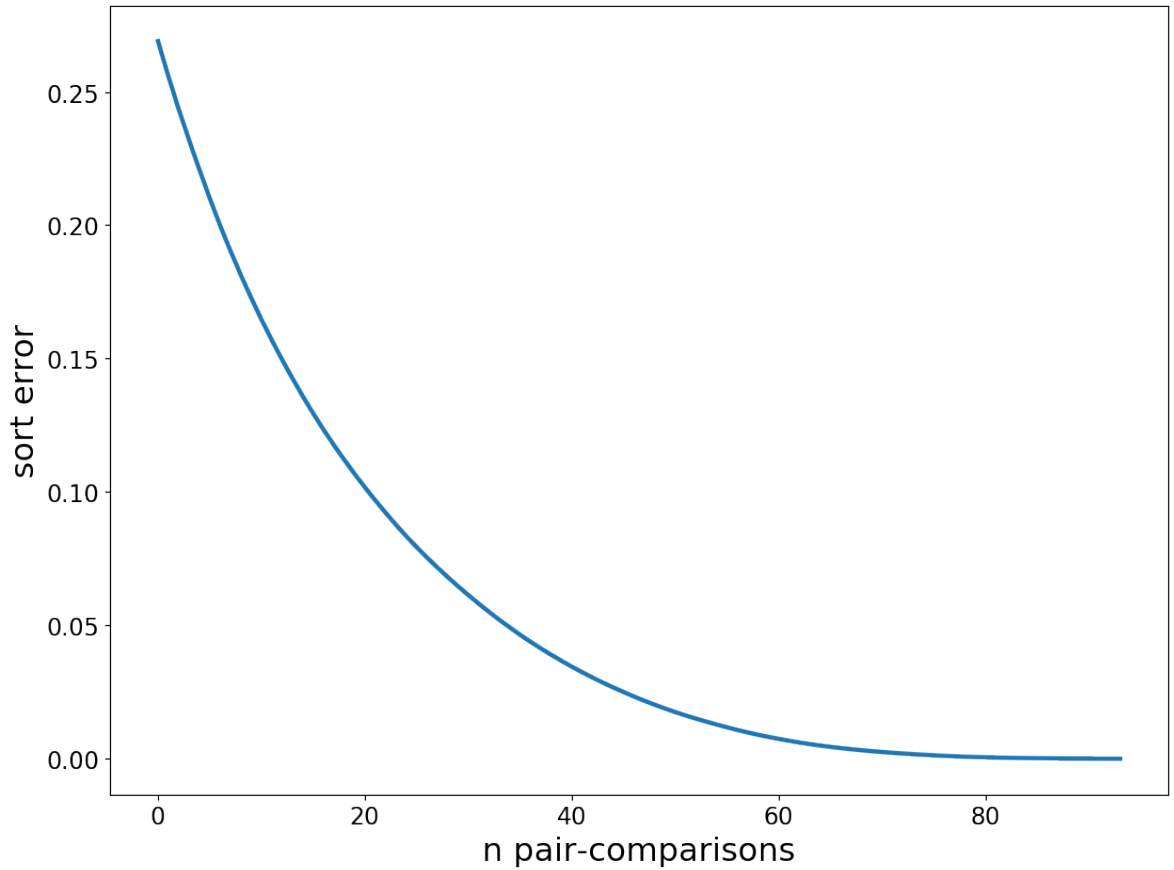
sorting as an optimization problem

Now, we can use a genetic algorithm (e.g. the one we have for the TSP problem) to sort our sequence (note, we want an order on the cities, but now with our objective function for sorting and not the one for a minimal tour length).

```
def GASort(w):
  ga=GA_TSP(
    func=fobj,n_dim=len(w),size_pop=100,
    max_iter=1000,prob_mut=1
  )
  best_points,best_val=ga.run()
  v=w.copy()
  for i in range(len(best_points)):
    w[i]=v[int(best_points[i])]
```

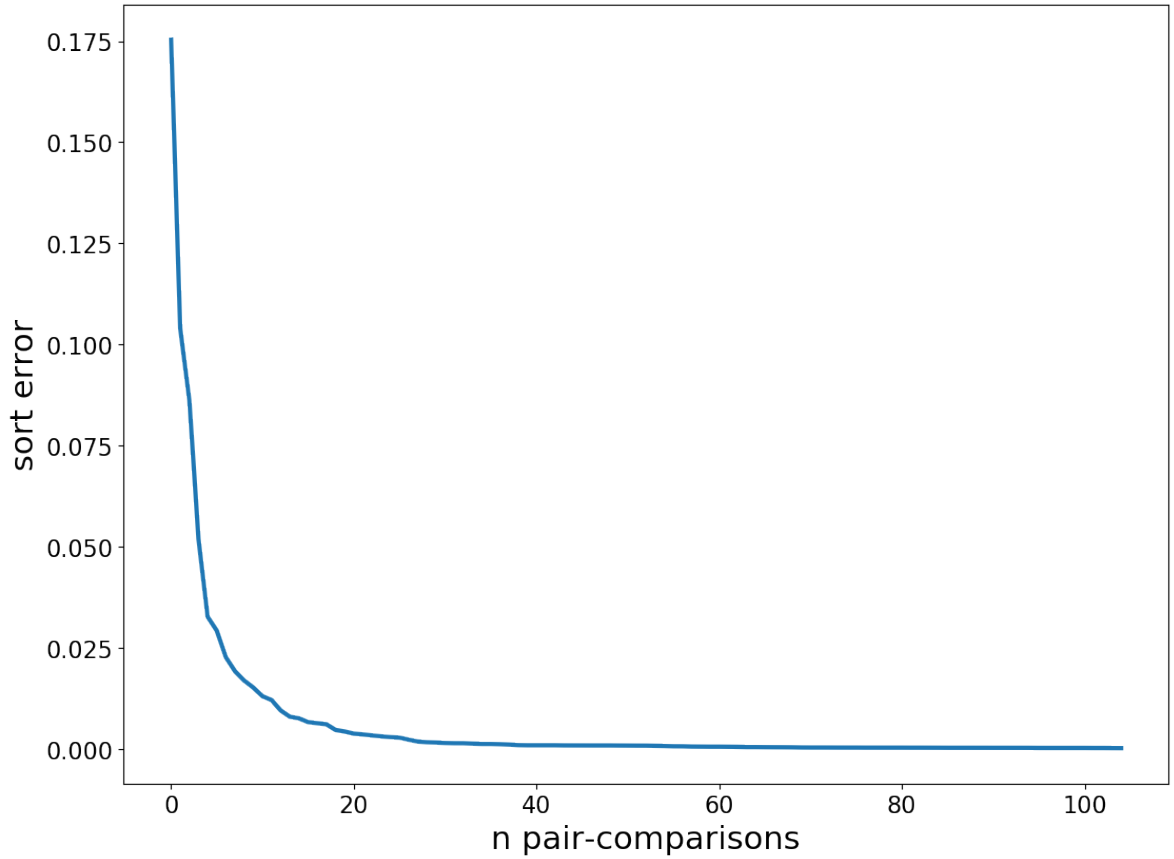
More in lab hours (`fobj` will be computed on a different data structure).

convergence of bubble sort



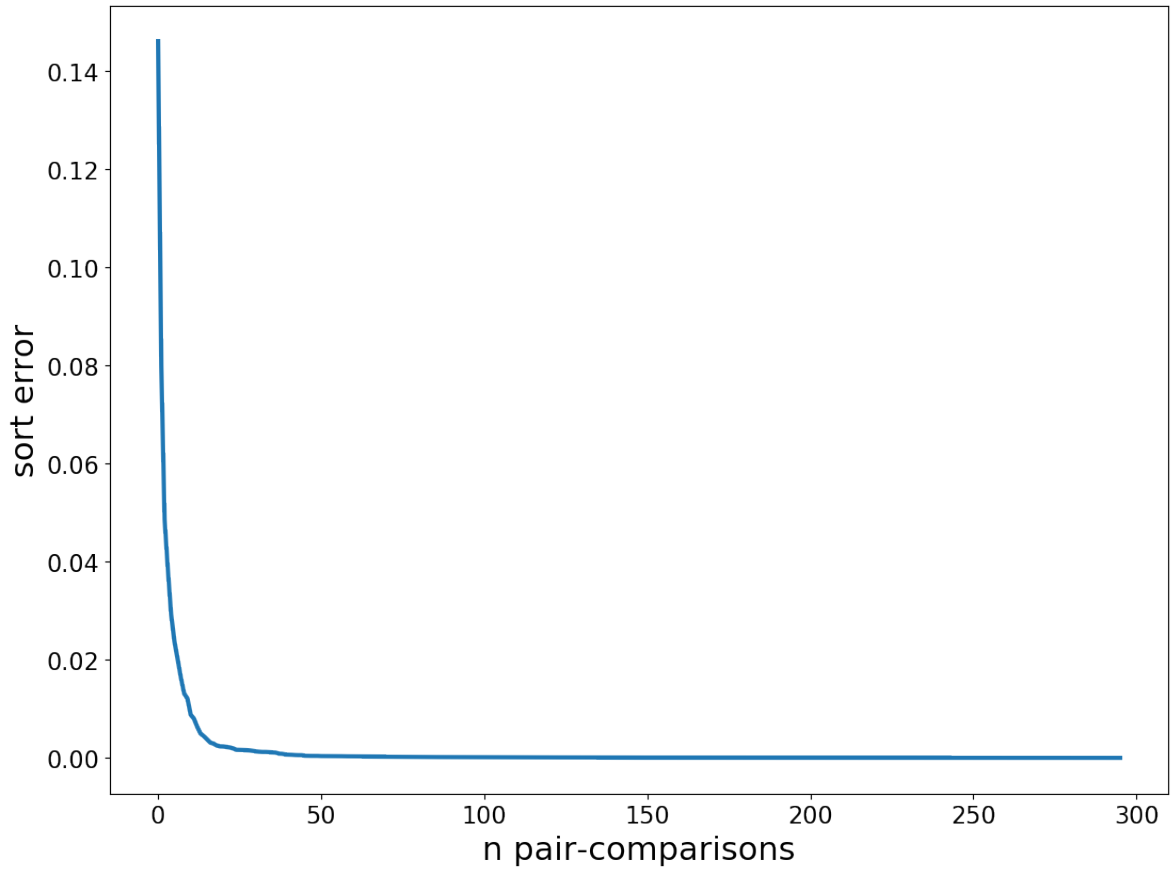
Slow improvement, finds the minimum always.

convergence of Monte Carlo sort



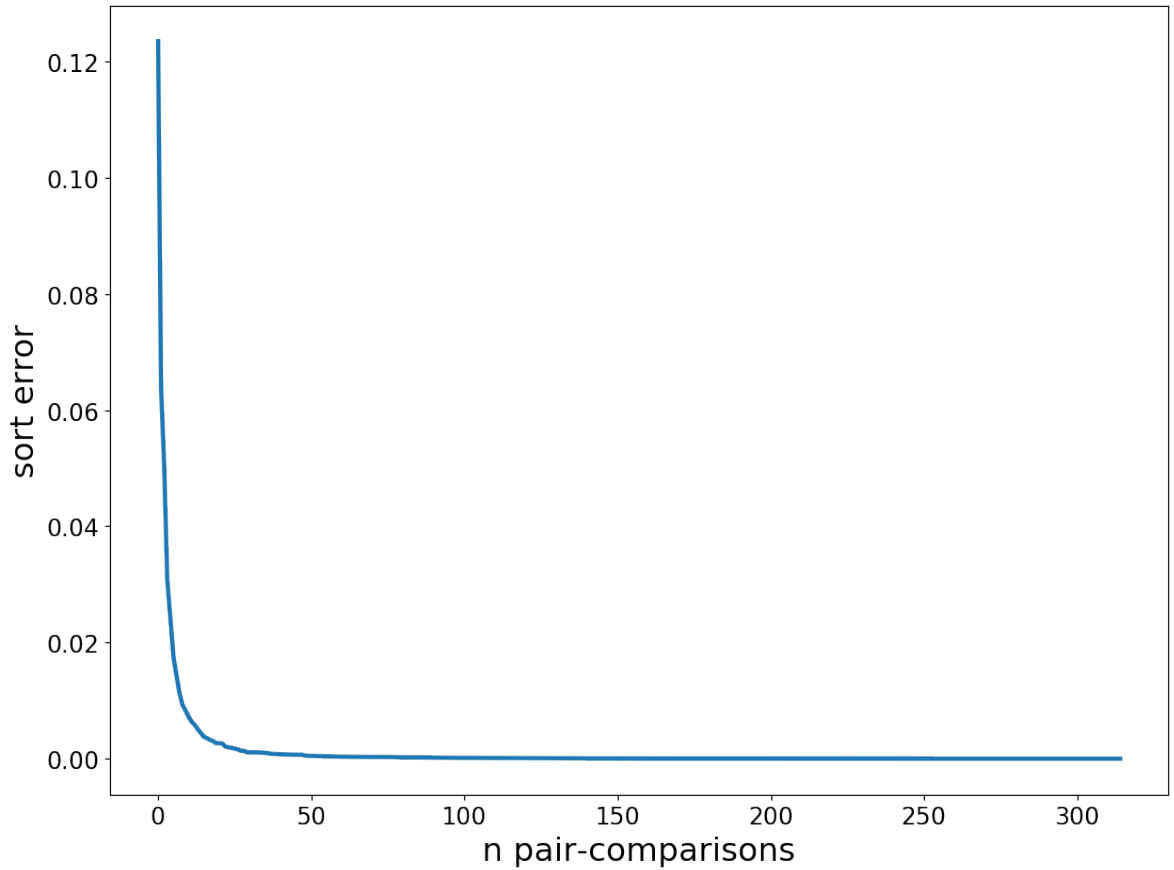
Fast improvement, fixed number of steps, might not find minimum.

convergence of Las Vegas sort

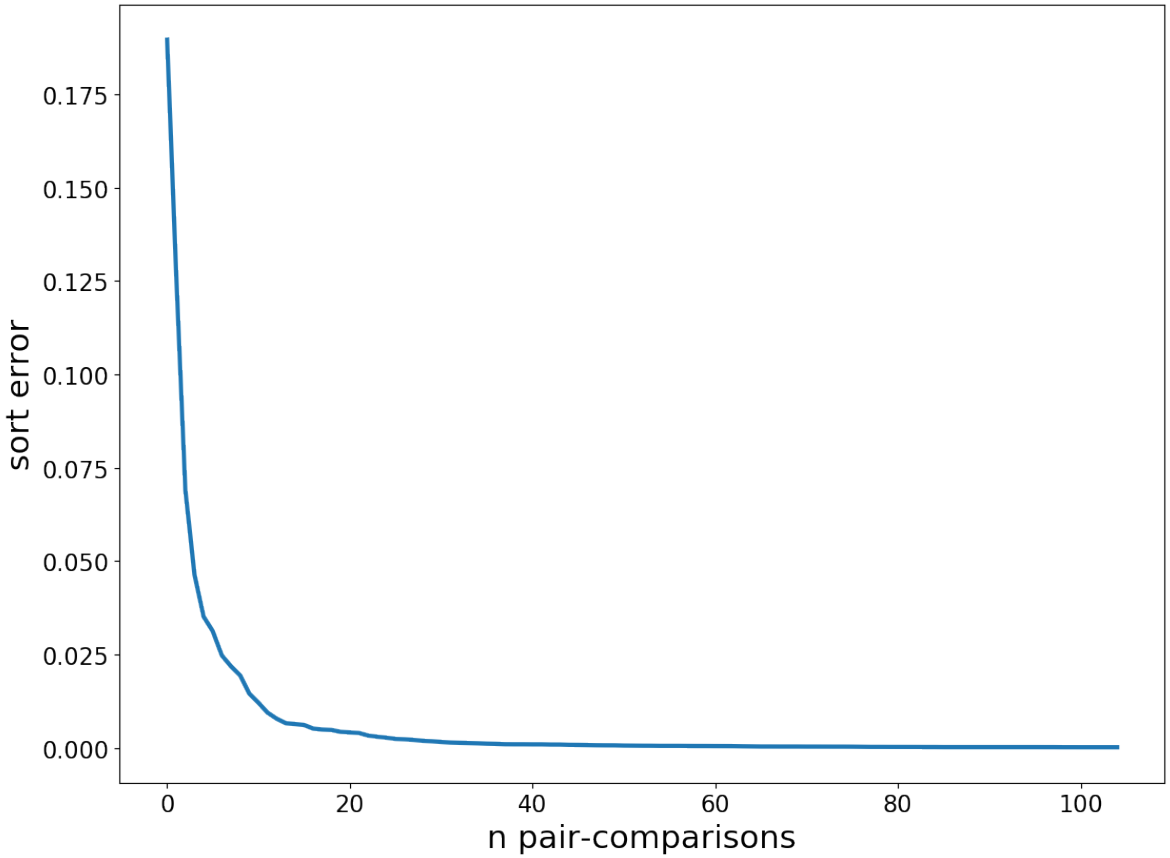


Fast improvement at the beginning, and slowly finds the minimum.

convergence of Las Vegas with Monte Carlo sort

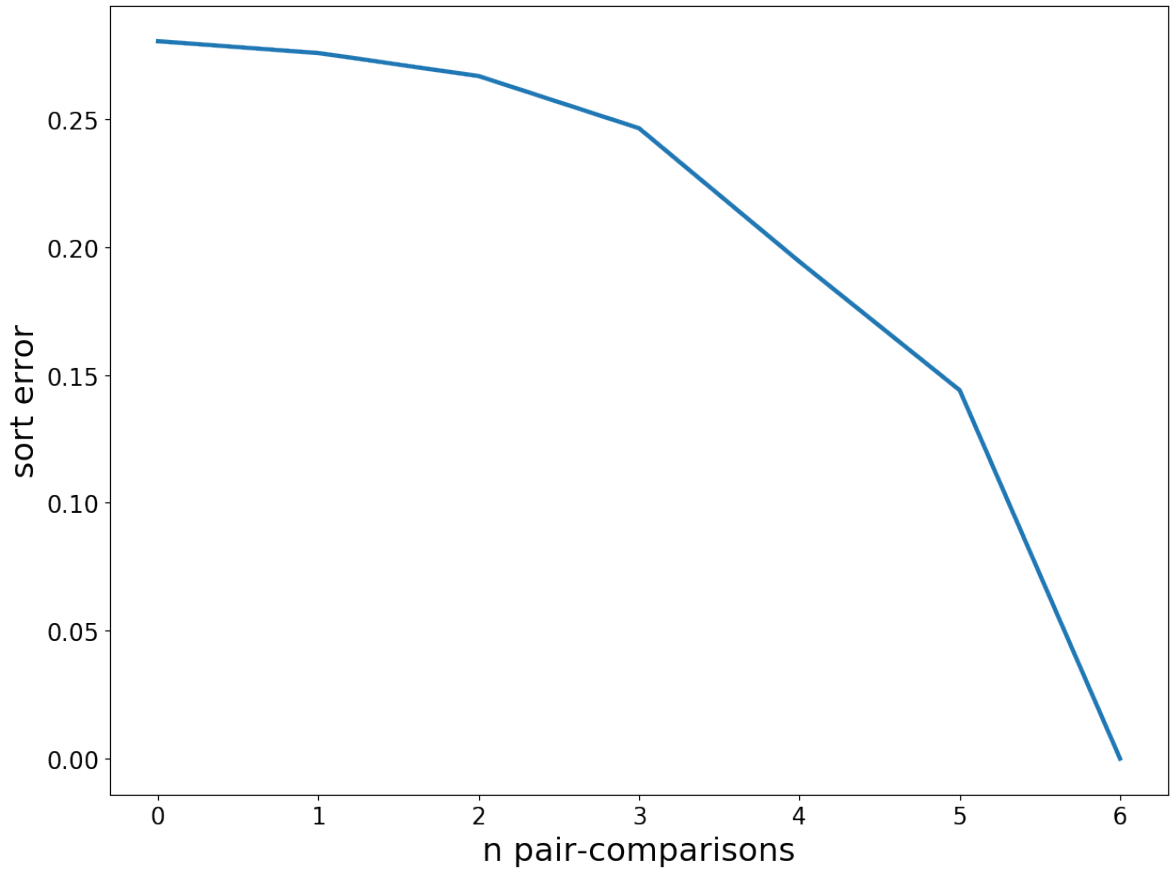


convergence of Monte Carlo sort with Las Vegas condition



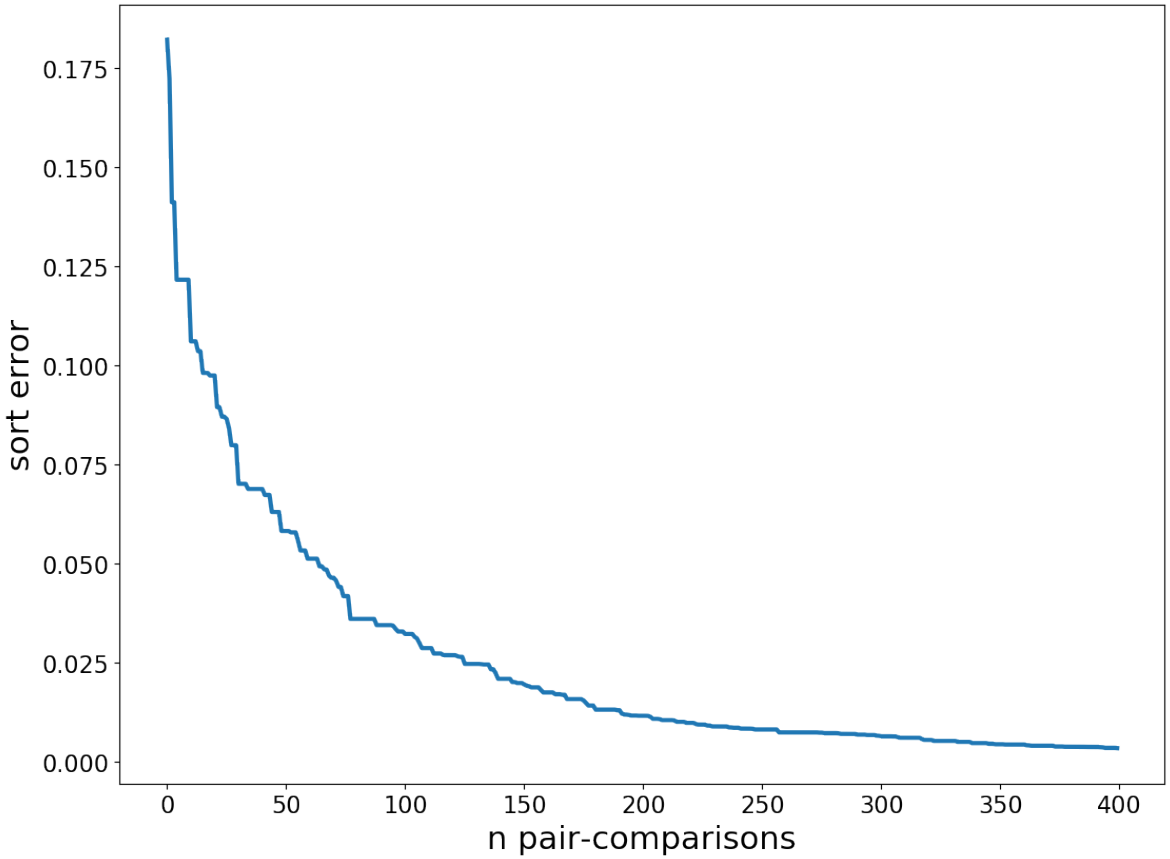
Fast improvement, but might not find minimum (however, stops if found).

convergence of merge sort



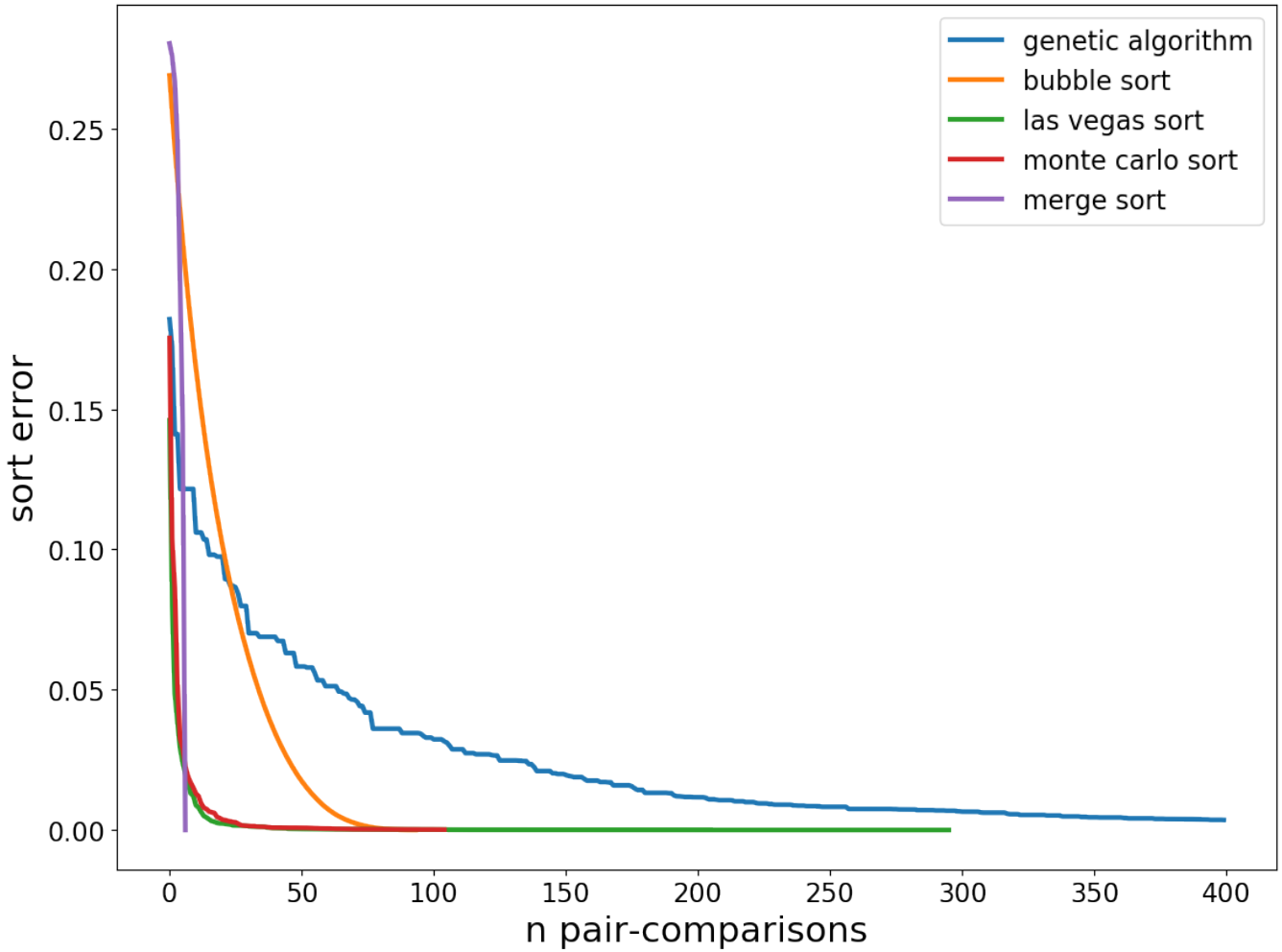
Deterministic very fast improvement, finds the minimum always!

convergence of genetic algorithm sort



Well, works, but might not find the minimum.

summary of *convergence* for sorting algorithms



Maybe the genetic algorithm is not the right choice,
better stick to the deterministic classic one.

Can we sort faster?

You can always ask: can we sort faster?

- It depends... when we have more information about the data, maybe we can sort faster!
- In the given example, we started with a random permutation of n consecutive numbers.
- So sorting them is easy: just count—starting at the minimum—up to n, hence, a linear time algorithm!
- It's always worthwhile to analyse the underlying data!

interpretation of the results

Don't get betrayed by a small number of program runs that might even *suggest* some good results (both in precision as well as in runtime).

You should always ask to see several/many runs, and to determine the variance of the results, so that you can compute the Monte Carlo standard error.

The (0,1)-knapsack problem

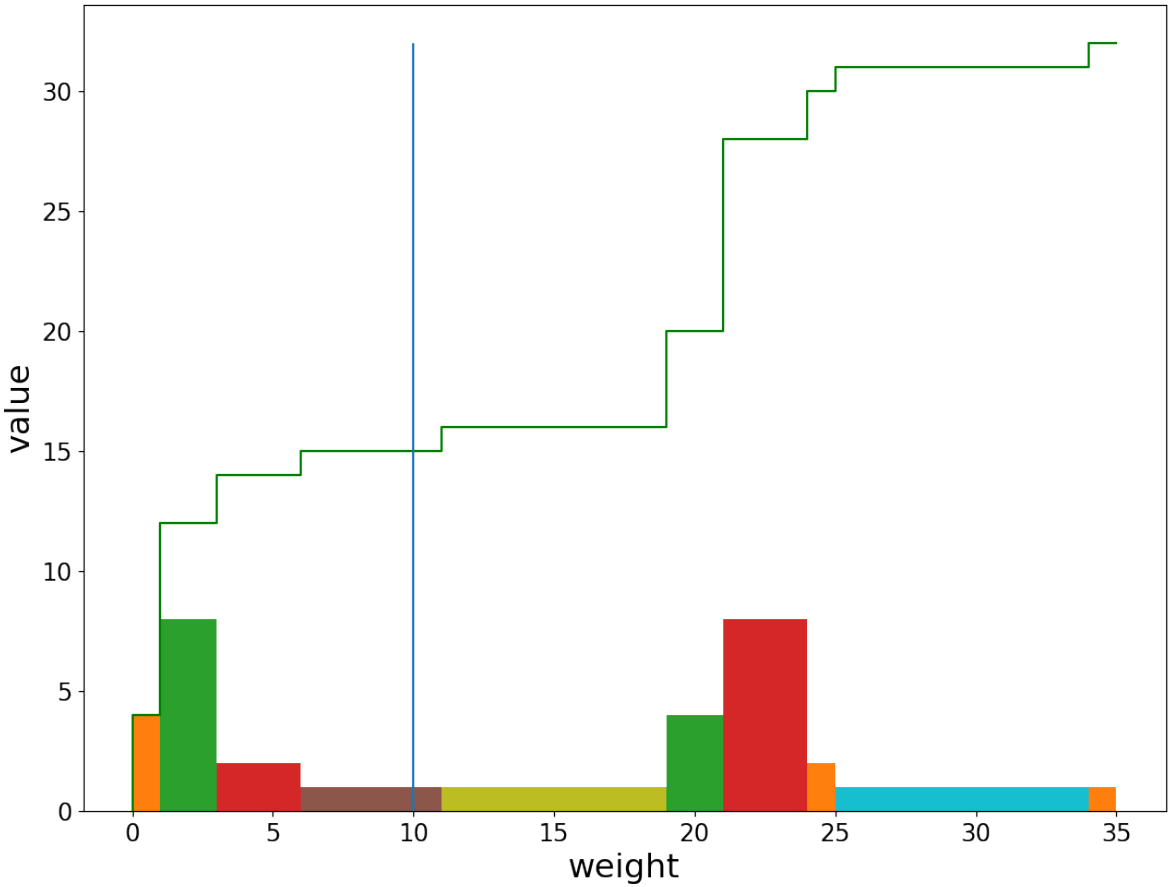
The (0,1)-knapsack problem (KSP) is another classical combinatorial optimization problem, where

- Given a set of items, each with a certain weight and value, and
- given a knapsack with a certain weight capacity,
- find the maximum total value you can carry with the knapsack.

Note that in this problem (in comparison to TSP or sorting) we have infeasible combinations (i.e., the subset might be too heavy).

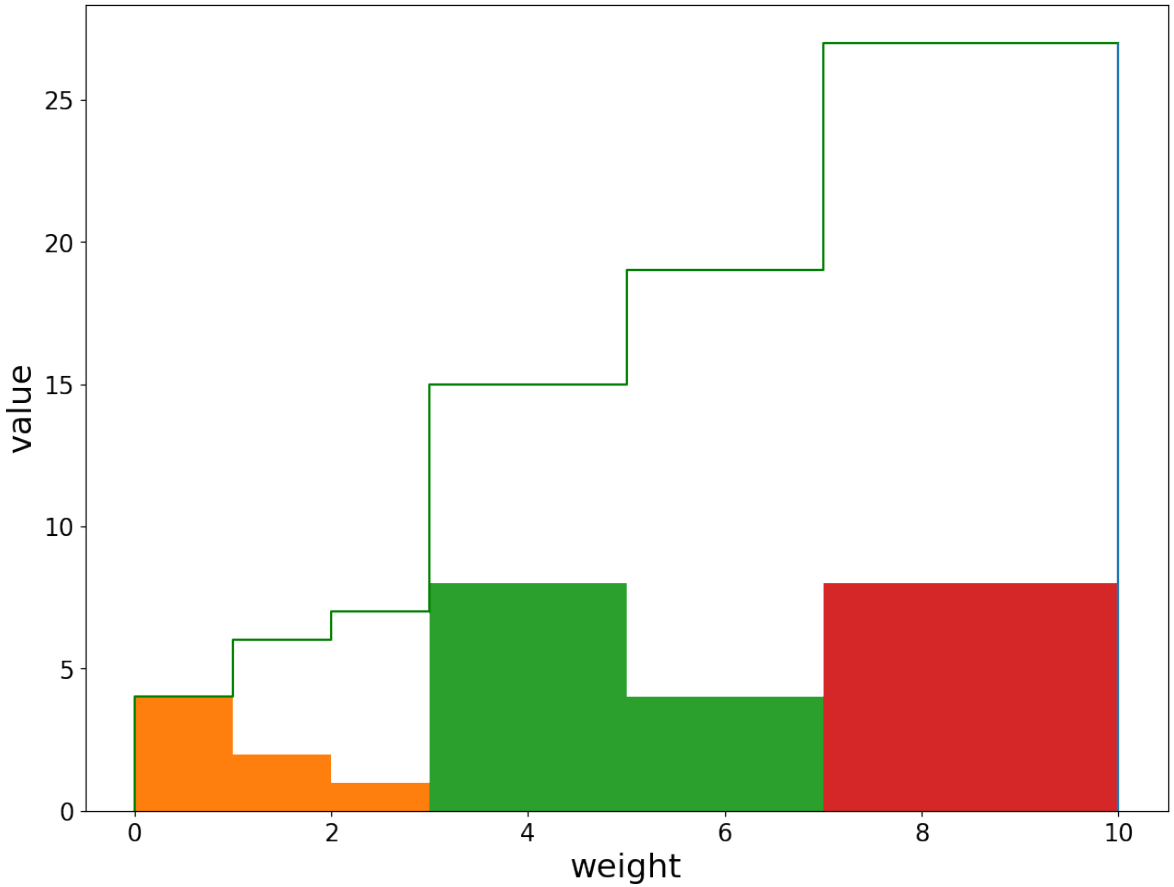
an example knapsack problem

If we take all available items:



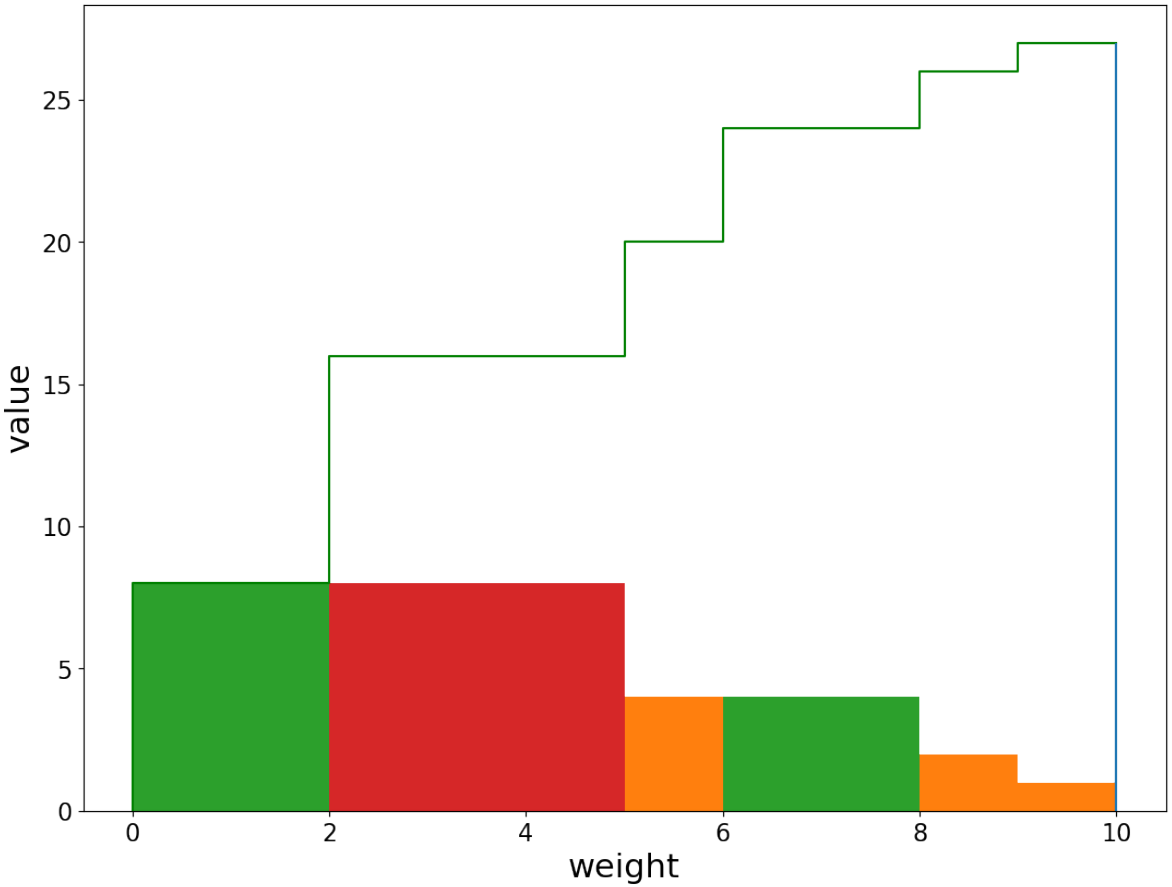
packing the knapsack with greedy weight algorithm

We take the lightest items as long as they fit:



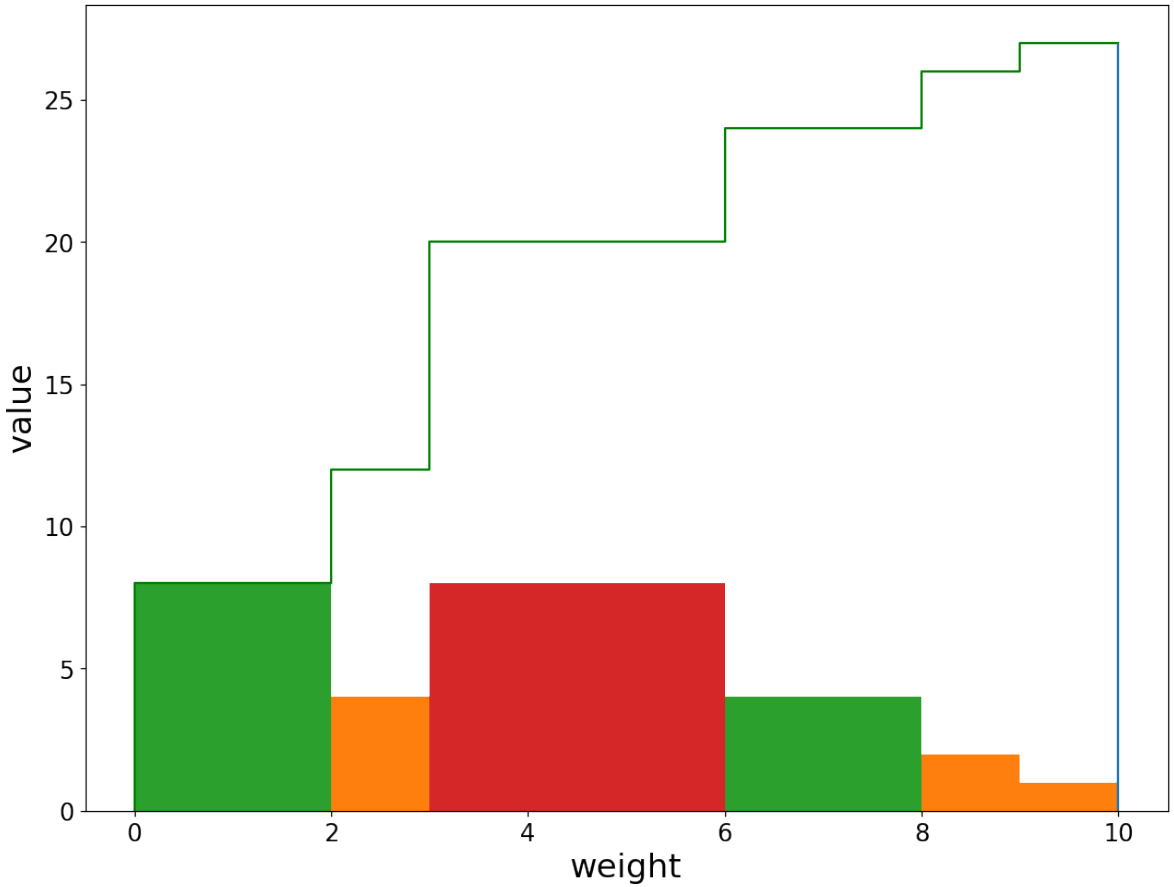
packing the knapsack with greedy value algorithm

We take the most valued items as long as they fit:



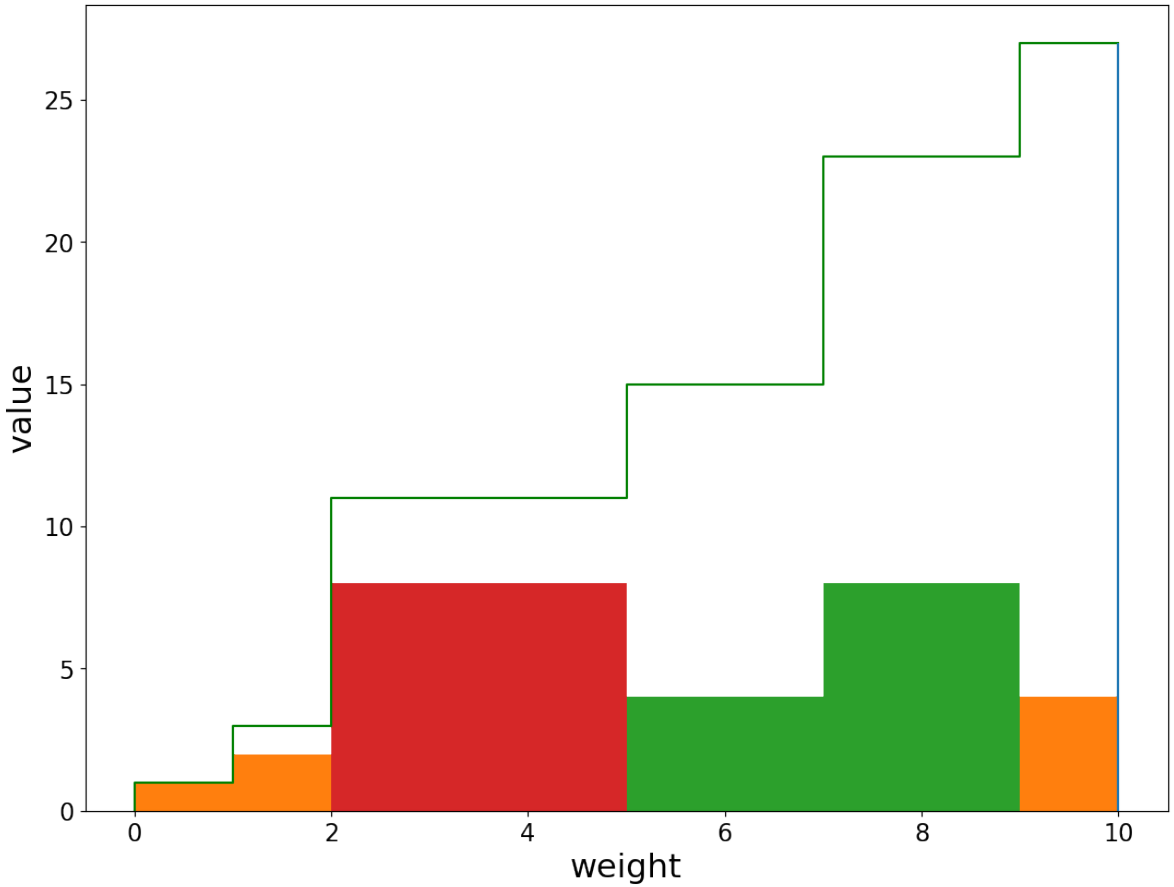
packing the knapsack with greedy ratio algorithm

We take the best rated (value per weight unit) items as long as they fit:



optimal packing the knapsack with dynamic programming

We find the optimal solution with dynamic programming:



It seems all algorithms are great?

The previous algorithms all packed a value of 27 into the knapsack...

- You noticed that I have cheated?
- All algorithms found an optimal packaging!
- You know why?
- I was lucky.

Evolutive methods

- Evolutive methods work with populations of individuals (or only one individual and a certain type of memory).
- There are probabilistic modification processes (mutation, reproduction, recombination/crossover) that change the population from one to the next generation.
- The performance of the individuals is based on a fitness which usually is the objective function (but not necessarily).
- There is a selection process to maintain a (more or less) stable state (size) of the population.
- Most of the algorithmic decisions are drawn probabilistically.

I will not give details on the history and researchers, please, take a look at the literature/bibliography.

Genetic algorithms (GA)

- We distinguish the genotype (codification of the individuals) and the phenotype (elements of the search space).
- There must exist a bijection between genotype and phenotype.
- The genotype encodes the free parameters of an individual.
- The modifications (mutation and recombination/crossover) are carried out over the genotype.
- The fitness is evaluated over the phenotype (our objective function).
- We have to explain: codification (of the genotype), initialization, mutation, recombination/crossover, selection, and stopping.

GA: principal loop

A genetic algorithm can be summarized in the following principal loop:

```
InitializePopulation()          # initialization
EvaluateIndividuals()           # evaluation
while not Stopping():           # stopping
  DetermineParents()              # selection
  GenerateChildren()              # recombination
  MutateChildren()                # mutation
  EvaluateIndividuals()           # evaluation
  ReestablishPopulation()         # selection
```

GA: encoding of the individuals

There are many possibilities how to encode the free parameters of an individual to form its genotype:

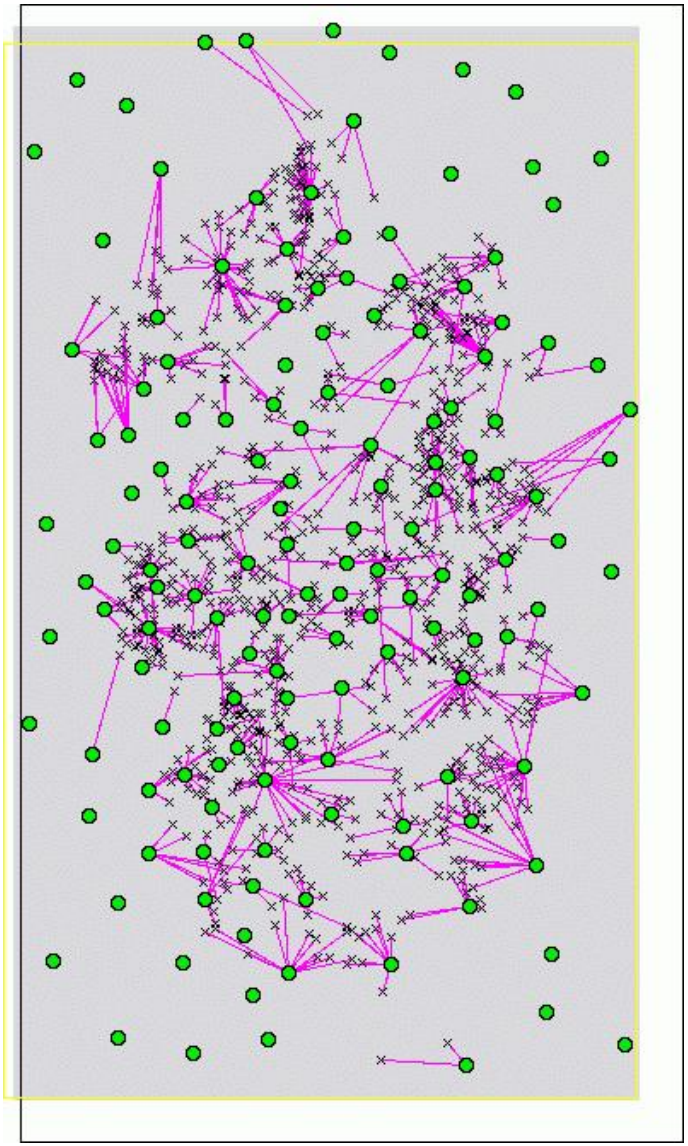
- use a binary bitstring, e.g., (101101)
- use a sequence of integer values in certain ranges, e.g., $(2, 6, 98, 3) \in [1:2] \times [1:10] \times [0:100] \times [1:5]$
- use a sequence of real values in certain ranges, e.g., $(1.23, 34.4, -2.1) \in [-50.0, 50.0]$
- use a permutation
- use a k-dimensional structure
- use a binary tree
- use a general graph
- use whatever you like (remember: do something, be happy...)

Remember: we need a bijection between genotype and phenotype and we need to implement crossovers and mutations that are able to explore the entire search space (or at least the region of interest).

GA: names taken from biology

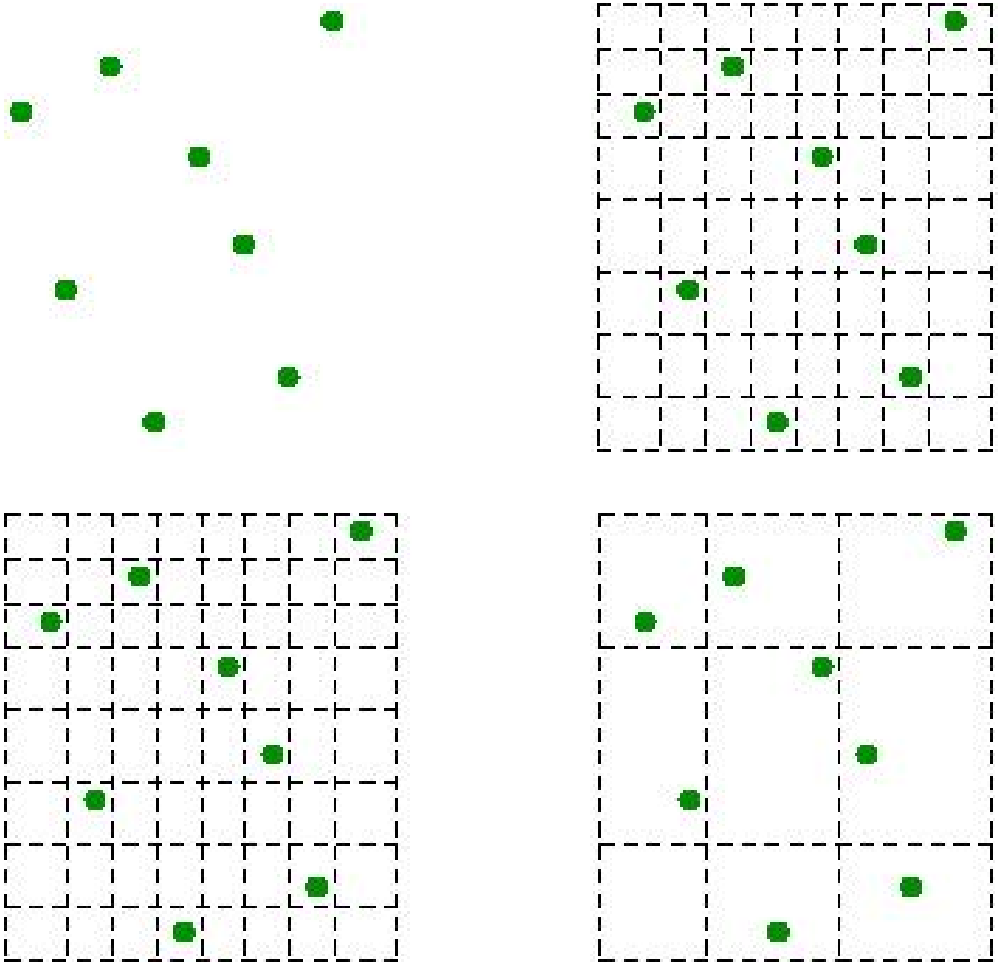
- The individual components of the sequences are called genes.
- The possible values of a gene are called allele.
- The encoding of an individual is called its genome or chromosome.

GA: genotype an example



- green: base stations
- crosses: mobile users
- magenta: assignment
- goal: find the minimal subset of base stations that guarantees an assignment of all mobiles
- Note: computation of the objective function is quite complex (and will not be detailed here).

GA: genotype an example



- initial
  8×8 grid
- reduced to
  3×3 grid
- 4 allele
  (2-bit strings)
  - unusable
  - used
  - unused
  - fixed

GA: mutation possiblities

gene mutation:
: just change one (or more) genes to another permitted allele

gene flip:
: interchange the values of two genes

gene sequence displacement:
: cut a sequence and insert at another position

gene sequence inversion:
: revert the order of a (partial) sequence

what-ever-you-like:
: do something, be happy...

GA: mutation rules of thumb

- The mutation rate should be inversely proportional to the size of the genome.
- For larger populations maybe reduce mutation rate in the on-going optimization process.

GA: crossover possiblities

simple crossover:

parents	cut	children
(101101)	$(10\|1101)$	$\longrightarrow (100111)$
(010111)	$(01\|0111)$	$\longrightarrow (011101)$
$(2,8,98,3)$	$(2,8,\|98,3)$	$\longrightarrow (2,8,40,4)$
$(1,9,40,4)$	$(1,9,\|40,4)$	$\longrightarrow (1,9,98,3)$

k-point crossover:

cut at k points and interchange the corresponding parts

(variation: take k at random)

uniform crossover:

interchange each gene with certain probability

multiple parent mating:

use more then two parents and interchange genes

(variation: merge entire parent set)

GA: crossover possiblities (continued)

arithmetic crossover: assign to children convex combination of parent genes with some random weight, $\alpha \in [0,1]$, e.g., with $\alpha = 0.7$ on second gene:

$34.5 \cdot 0.7 + 13.5 \cdot 0.3 = 28.2$

$(1.23, 34.5, -2.1) \longrightarrow (1.23, 28.2, -2.1)$

$13.5 \cdot 0.7 + 34.5 \cdot 0.3 = 19.8$

$(10.5, 13.5, 23.1) \longrightarrow (10.5, 19.8, 23.1)$

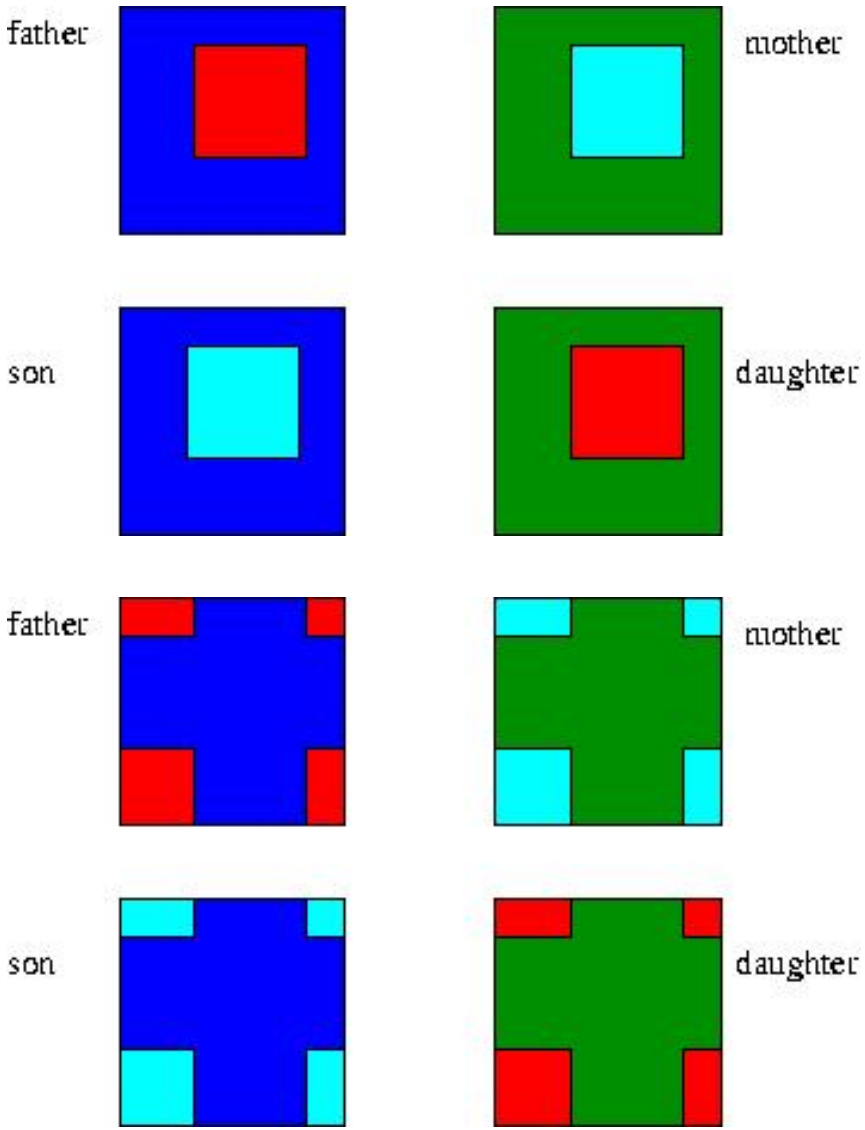
(again variations: as k-point, or with all genes, or with k at random)

blended crossover: blend two corresponding parent genes with a certain, usually fixed, value $\alpha \in [-0.5, \infty]$ according to the current gene spread

simulated binary crossover: blend two corresponding parent genes according to a suitable probability density function

what-ever-you like: remember, do something, be happy...

GA: crossover example (2-point cyclic crossover)



- select two grid points
- interchange rectangles

GA: principal loop

A genetic algorithm can be summarized in the following principal loop:

```
InitializePopulation()        # initialization
EvaluateIndividuals()         # evaluation          DONE
while not Stopping():         # stopping
  DetermineParents()            # selection
  GenerateChildren()            # recombination     DONE
  MutateChildren()              # mutation          DONE
  EvaluateIndividuals()         # evaluation        DONE
  ReestablishPopulation()       # selection
```

GA: selection

- In the principal loop, there are two selection processes:
  - How to select the parents to generate the off-springs? and
  - How to rearrange the final population or next generation?
- We use the following notation:
  - μ stands for the number of individuals in the population
  - λ stands for the number of children being generated
- We distinguish two main strategies:
  - $(\mu + \lambda)$-strategy
    - from the μ individuals of the current generation select the parents and generate λ children
    - from the $\mu + \lambda$ individuals choose the μ best ones as new generation
  - (μ, λ)-strategy
    - from the μ individuals of the current generation select the parents and generate $\lambda \geq \mu$ children
    - from the λ children choose the μ best ones as new generation

The second question from above is answered.

GA: selection options

To select the parents being allowed to have off-springs,
there exists a bunch of suggestions:

roulette wheel: assign to each individual a fraction of the wheel according to its relative fitness and spin the wheel (variation: smooth, weight, or normalize the fitness somehow, e.g., use log of objective function, or use z-score)

rank based: order the individuals according to fitness and select with a probability weighted by the rank (variation: compute selection probability with linear function of rank, so the least ranked still gets certain probability to get selected)

GA: selection options (continued)

tournament based: draw a certain number of random individuals, select the best one as parent
(variation: select directly the best two as parents)

truncation selection: only the individuals with highest fitness values will be parents

what-ever-you like: remember, do something, be happy...

The first question (from two slides earlier) is answered.

GA: initialization

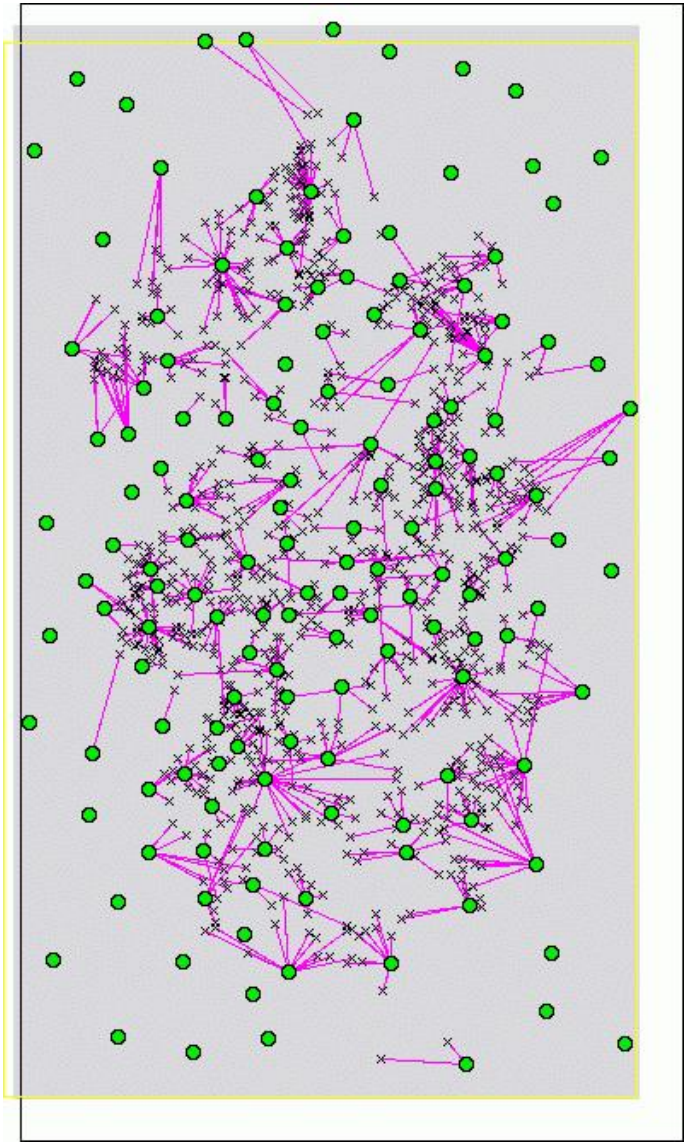
- generate the initial population with random genomes
  - take into account that a distribution in genotype not necessarily is similar to the same distribution in phenotype
  - there are maybe many individuals with very low fitness
  - the initial convergence rate might be slow
- generate the initial population with individuals from another heuristic algorithm or various such algorithms
  - the population might be biased into a certain region of the search space
  - the diversity of the population might be low
  - the convergence rate might be trapped early in a local optimum
- recommendation: use a mixture of both

GA: termination criteria

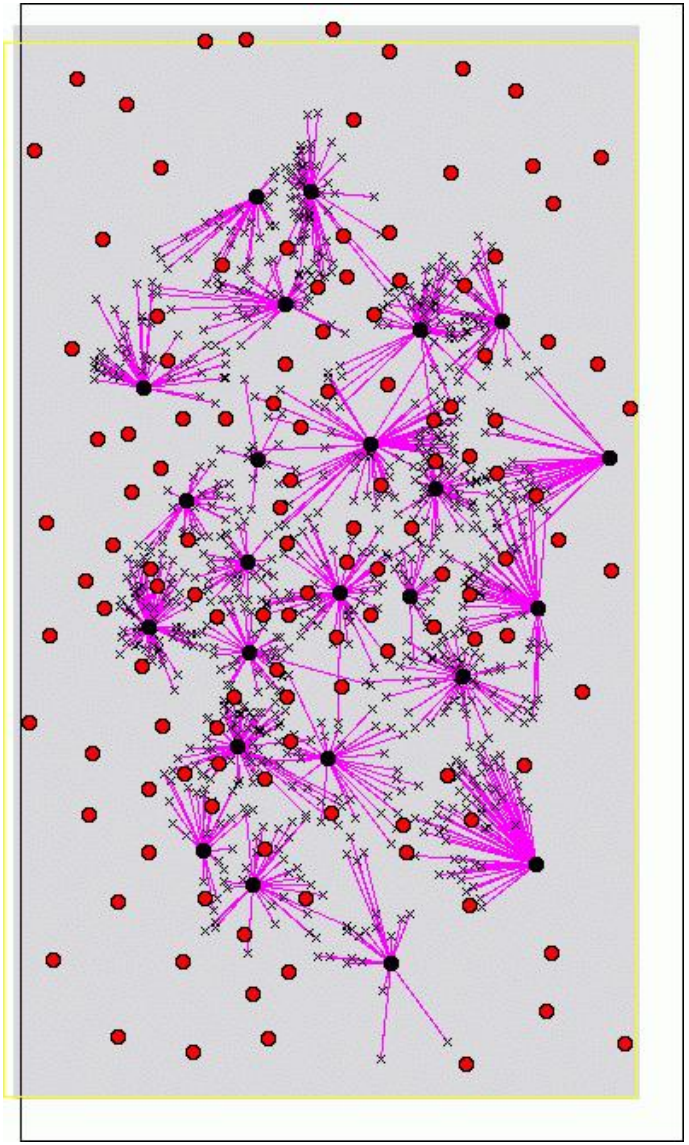
There are many possibilities when to stop the iteration of a genetic algorithm:

- once the first solution has been found
- once a sufficiently good solution has been found
- once the optimum has been found
- once a certain number of iterations has been executed
- once the diversity of the population is below a certain threshold
- once the convergence rate of the improvement is below a certain threshold
- once a certain amount of runtime has been spent
- recommendation: use an or-mixture of all

GA: results for the example (2007)



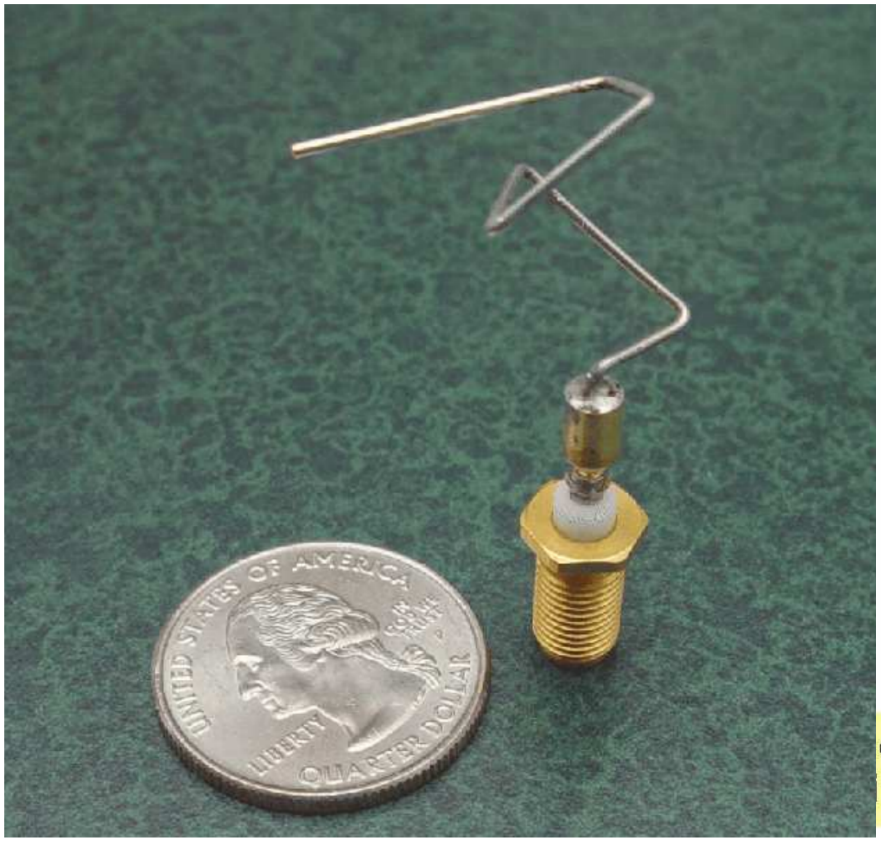
119 of 149 nodes used

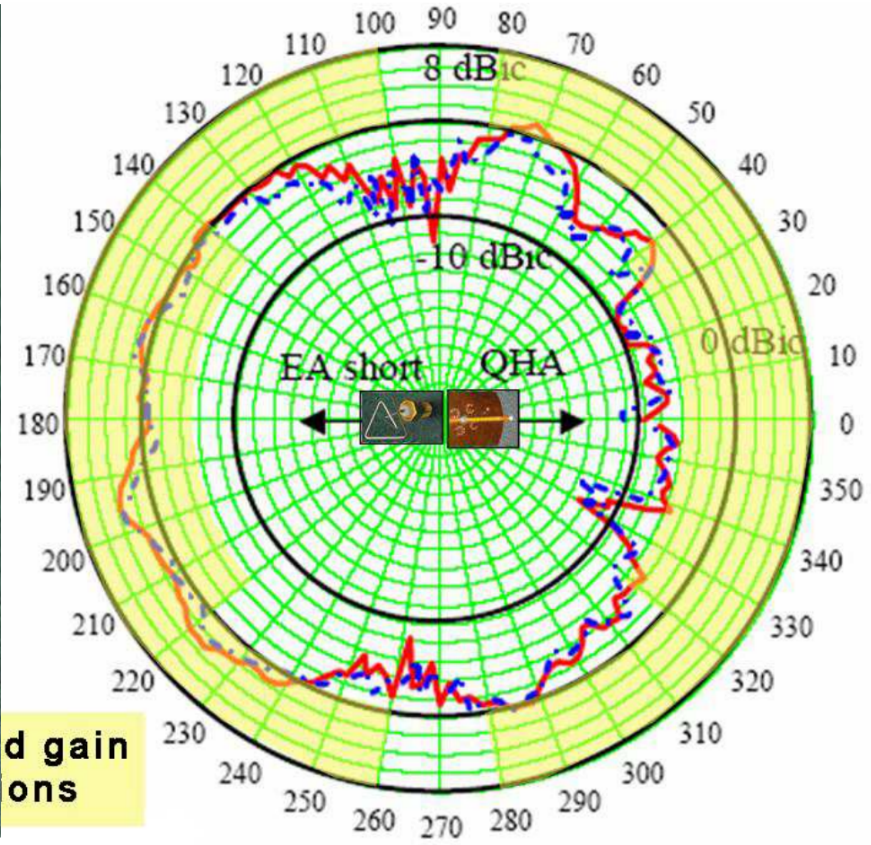


24 of 149 nodes used

GA: use for antenna design (small satellites)

Horny, AlGlobus, Linden, Lohn: Automated Antenna Design with Evolutionary Algorithms





d gain
ons

GA: diversity

- The diversity measures, in some sense, the non-similarity between the individuals of a population.
- E.g., Hamming-distance over the bitstring (using exor): $010010 \otimes 101000 = 111010 \longrightarrow 4$
- e.g., delta-distance over the integer (or real) sequence: $\sum_i |x_i - x'_i|$ (being x and x' two individuals)
- There are much more similarity measures.
- similar individuals in a population reduce the diversity and the genetic algorithm maybe gets stuck in some region of the search space (maybe, but not necessarily, a local minimum).
- To augment the diversity, we have only the mutation operation, provided the mutation becomes visible in the next generation. (Observe: whenever an allele disappears in a population, most of the crossover operations cannot regenerate it!)
- Another possibility is just to regenerate a completely or partially new population.

GA: elitism

We have to draw the decision whether the best individual(s) is (are) forced to belong unmodified to the next generation.

- Elitism might help to converge faster.
- Elitism might reduce diversity faster.
- The consequences of this trade-off are problem dependent.

GA: final remarks

- The difficulties of understanding and analizing genetic algorithms lie in the fact that they implement a combination of random search (by mutation) and biased search (by recombination).
- Genetic algorithms need unique and problem-specific mutation and recombination operators, which makes it more challenging to implement a generic version that can be easily applied to different optimization problems.
- Nature still has its somewhat better approach: DNA, RNA, gene expression, proteins, and mitochondria (mtDNA)...

evolutionary programming (EP)

Once we have seen genetic algorithms, evolutionary programming is somewhat simpler: it just uses mutation.

- there exist only the phenotypes, let's say x_i (for $i = 1, \ldots, n$), i.e., n individuals in the population
- modification (mutation) is realized over the phenotypes as:

$$x_i' = x_i + r_i \sqrt{\beta f(x_i) + \gamma}$$

  being $\beta > 0$ and $\gamma \geq 0$ tuning parameters (for instance $\beta = 1$ and $\gamma = 0$) and r_i is a random value taken from a normal distribution with mean 0 and variance 1 (i.e., $r_i \in N[0,1]^n$).
- Note that the fitness (objective function f) must be shifted, so the minimum is positive.
- Usually a $(\mu + \mu)$-selection strategy is used: all individuals are mutated and the best μ individuals are kept.

EP: principal loop

A evolutionary programming algorithm can be summarized in the following principal loop:

```
InitializePopulation()
EvaluateIndividuals()
while not Stopping():
  GenerateChildrenByMutation()
  EvaluateIndividuals()
  ReestablishPopulation()
```

differential evolution (DE)

Once we have seen genetic algorithms, differential evolution is somewhat simpler: it just uses a special type of recombination.

- there exist only the phenotypes, let's say x_i (for $i = 1, \ldots, n$), i.e., n individuals in the population
- For each individual we select three other individuals, say x_j, x_k, x_l, to compute a mutant vector v_i

$$v_i = x_j + F \cdot (x_k - x_l)$$

  being $F \in [0.4, 0.9]$ (usually) a tuning parameter.
- Then we generate an off-spring with a uniform crossover between individual x_i and mutant v_i using a certain threshold c
- Usually a $(\mu + \mu)$-like selection strategy is used: all individuals are used to generate off-springs, and the best are kept.

DE: principal loop

A differential evolution algorithm can be summarized in the following principal loop:

```
InitializePopulation()
EvaluateIndividuals()
while not Stopping():
  GenerateChildrenByDiffusion()
  EvaluateIndividuals()
  ReestablishPopulation()
```

DE: some variations

- One might consider to use always the best individual found so far as individual x_j.
- The tuning parameter F might vary, i.e., taking the value from a uniform or a normal distribution.
- One might use DE on discrete sets as well by just rounding the mutants appropriately (or search in the close integer neighborhood according to the dimension of the underlying problem).
- (My opinion) Differential evolution is not just a genetic algorithm, as there is no genotype, rather the other way round: a genetic algorithm using the phenotype as genotype, no mutation, and a random recombination, becomes a differential evolution algorithm.

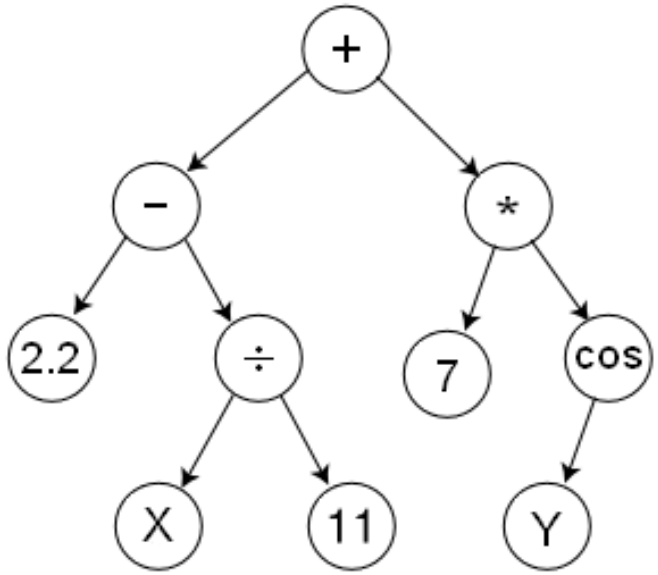
genetic programming (GP)

Once we have seen genetic algorithms, genetic programming is a genetic algorithm with some special phenotypes and genotypes.

- the genotype is a (simple) program described as a syntax tree that can be written as well with Polish notation (prefix notation), see next slide...
- the parenthesis can be eliminated, interpretation of the corresponding expression is easy to perform with a stack automaton.
- some properties of the execution of the resulting program (as phenotype) are used as fitness (see example, later)

GP: syntax tree

syntax tree and Polish notation



- `(2.2-(x/11))+(7*cos(y))`
- `(+ (- (2.2 (/ X 11))) (* (7 cos(Y))))`
- `+ - 2.2 / X 11 * 7 cos Y`

image taken from wikipedia

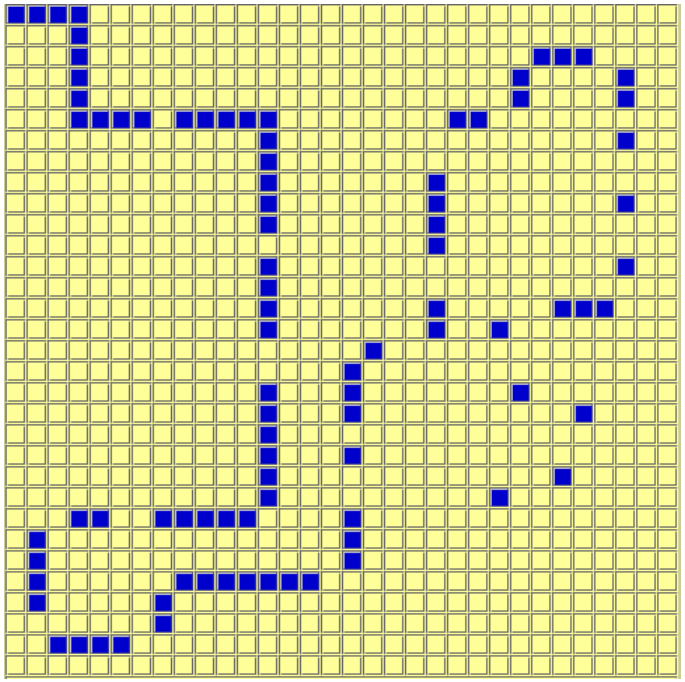
GP: mutation and crossover operations

- the programs are modified with adecuate mutation and crossover operations
- mutation:
  - change a node, but take care to keep a valid syntax tree (maybe subtrees must be removed or added)
  - rotate nodes
  - interchange nodes
- crossover: interchange a subtree of one parent with a subtree of the other parent

GP: example

Program a robot (ant) that starts at some cell (usually a corner) and tries to find as many objects (food) with as few steps as possible.

Santa Fe Trail



nodes: turn-left, turn-right, move, if-food-ahead

particle swarm optimization (PSO)

- The inspiration comes from social behavior of individuals within an environment including other individuals.
- We work with n individuals that move in a continuous d-dimensional search space.
- The individuals move (in steps) through the search space and adjust their velocities according to information gathered from others (and their own *histories*).
- The individuals are grouped into neighborhoods.

PSO: velocity actualization

- x_i vector of current positions
- v_i vector of current directional velocities
- b_i best local position vector
- h_i best neighbor position vector
- $\varphi_1 = 2.05, \varphi_2 = 2.05$ influence values (just some *magic*)
- $\xi \in [0.4, 1]$, e.g. $\xi = 0.729$ inertia reduction value
- velocity actualization

$$\begin{aligned} v_i &= \xi v_i + U[0, \varphi_1] \circ (b_i - x_i) + U[0, \varphi_2] \circ (h_i - x_i) \\ x_i &= x_i + v_i \end{aligned}$$

- The $\circ$ operator is either a Hadamard-operation (i.e., component-wise), or a linear operation (i.e., scalar multiplication)

PSO: principal loop

A particle swarm optimization can be summarized in the following principal loop:

```
InitializePopulation() # i.e. x_i, v_i
EvaluateIndividuals()  # i.e. b_i
DefineNeighborhoodSize()
while not Stopping():
  DetermineNeighborhoodValues()   # h_i
  UpdateIndividuals()             # i.e., x_i, v_i, b_i
```

PSO: some more details

- The velocity can be confined not to pass a certain maximum velocity, which helps to avoid explosion, i.e., that the area of the search space being explored becomes exponentially larger.
- Initial velocities can be zero or some random values.
- Small neighborhoods tend to provide a better global search, while large neighborhoods tend to produce a faster convergence (but maybe premature).
- Neighborhoods can be defined as nearest neighbors, as fixed and overlapping, or entail the entire population, or what-ever-you-like.
- The inertia reduction can be increased with the simulation time.
- The best global individual g can be included in the equation: add $+U[0,\varphi_3]\circ(g-x_i)$
- The worst (local and global) positions can be *avoided*: add $-U[0,\varphi_4]\circ(\overline{b}_i-x_i)$ and/or $-U[0,\varphi_5]\circ(\overline{h}_i-x_i)$ and/or $-U[0,\varphi_6]\circ(\overline{g}-x_i)$

PSO: different versions

binary version: the variables are interpreted as binary values according to a distribution or threshold

discret version: the variables are interpreted as integer values (for instance with simple rounding)

dynamic version: the search space is reinitialized and/or the local variables are reset (type of outer Monte Carlo loop)

PSO: convergence

- the individuals should exhibit certain diversity
  (recall the similarity measures)
- diversity can be forced dynamically by adapting the parameters alongside the simulation time
- or one might use the lack of diversity as a stopping condition

ant colony optimization (ACO)

The idea stems from stigmergy: exercise indirect communication and coordination through the environment
(leave a trace and act on findings).

- The inspiration stems from ants, bees, termites, wasps, etc.
- The individuals of a population leave information (pheromones) in the search space.
- The decisions are based on individual information or behavior and on the pheromones encountered.
- The information (pheromones) is volatile and can evaporate.
- The pheromones or a statistical evaluation of the individuals define the solution.
- Initially invented to deal with combinatorial problems (like TSP).

ACO: principal loop

An ant colony optimization can be summarized in the following principal loop:

```
InitializePheromoneValues()
while not Stopping():
  for individuals in range(n):
    ConstructSolution(individual)
    UpdatePhermoneValues()
    UpdateIndividuals()
```

ACO: how TSP can be approached

- The ant colony optimization takes place on the graph of the underlying problem (e.g., the complete graph among all cities).
- The ants are placed at the cities.
- The initial pheromones are placed on the edges (either constant value or inversely proportional to the distance).
- The ants (in an appropriate iteration) run along a path in the graph (excluding already visited cities) and draw at each city a decision in which direction to continue.
- The decision is based on: pheromones on each possible edge, maybe on some own information stored at the individual, and on a random value.
- Once the tour is completed for all ants, all of them deposit their pheromone on their tracks.
- The general evaporation process is applied to all/changed edges.
- The currently best tour is memorized.
- The iteration is repeated until a certain stopping condition is met.

ACO: when to use?

ACO approaches are especially possible when the underlying problem allows for a constructive solution (as seen with the nearest-neighbor heuristic for the TSP).

Simon gives the example that an ACO approach found a tour with 3% deficit on the Berlin52 problem.

The no-free-lunch theorem

The no-free-lunch theorem states that the performance of all optimization (search) algorithms, amortized over the set of all possible functions, is equivalent. The implications of this theorem are far reaching, since it implies that no general algorithm can be designed so that it will be superior to a linear enumeration of the search space (exhaustive search).

What are practical implications of the no-free-lunch theorem?

- Each problem (or each type/class of problem) might need its own and proper optimization method.
- Maybe for interesting problems we find good optimization algorithms (we are not interested in *all* problems).
- Benchmarking optimization algorithms is a challenge, as general benchmarks might just provide *average* data, but our algorithm might be special for a niche of problems.
- There is a need to categorize problems and algorithms to obtain some insight on which type of problem a certain type of algorithm performs well.

How to compare different approaches?

In order to compare different algorithms one might take into account:

- wall clock runtime on comparable systems
- (average) number of objective functions evaluations
  (but the rest of the inverted time must not be neglected)
  difficult to be used when comparing constructing algorithms
- the result as distance to optimium or to some known lower bound
- mean best fitness
- properties of the solution histogram (fitness of all solutions found)
- scaling properties with problem size (applied to any measure above)

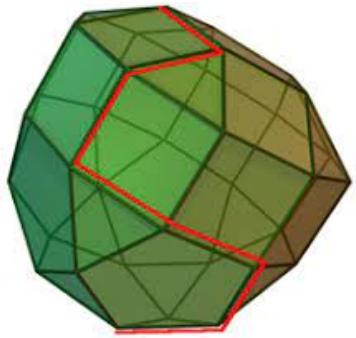
Practical aspects to be considered

One has to decide what is really needed:

- need a good (or best) solution independent of runtime
  (e.g. controler for space telescope or the evolved antenna)
- need a moderate solution fast
  (e.g., daily TSP with time windows, where finding a feasible solution is already NP-hard)

Local search methods

- local search methods explore the search space by inspecting (close or far) neighbor solutions
- they stop at a local minimum, i.e., all neighbors are greater (remind: we searching for a minimum)
- a classical example is the simplex method for linear programming



- or the Newton method (or Newton-Raphson method) applied to optimization (here formulated as maximization)

```
while GradientFobj(xi) > tolerance:
  xi=xi-GradientFobj(xi)/SecondDerivativeFobj(xi)
```

(Take care: should check if eventually really maximum, and not minimum.)

Local search methods

Further classic iterative optimization methods for the minimization of real-valued functions that need gradient information, are:

- Gauss-Newton method (variation of the Newton-method by using the Jacobean-matrix instead of the Hessian-matrix)
  second derivatives are not required here
- gradient descent (or steepest decent)
- Levenberg-Marquardt methods (interpolation between Gauss-Newton methods and gradiant descent)

Local search methods

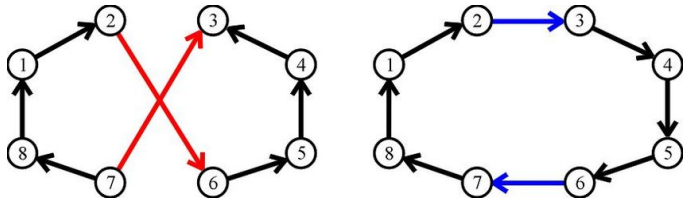
Further classic iterative optimization methods for minimization of real-valued functions that don't need gradient information, are:

- Nelder-Mead method (heuristic), converges to a stationary point (minimum, maximum, or saddle, i.e., gradient is zero)
- idea: shrink, reflect, and expand a simplex (triangle in 2D), by evaluating the objective function on corners and faces (edges in 2D)
- García-Palomares method, converges to a local minimum
- idea: explore the neighorhood according a random local spanning coordinate system and proceed at a point that has been found with a sufficiently steep descent (otherwise iterate with smaller tolerance)

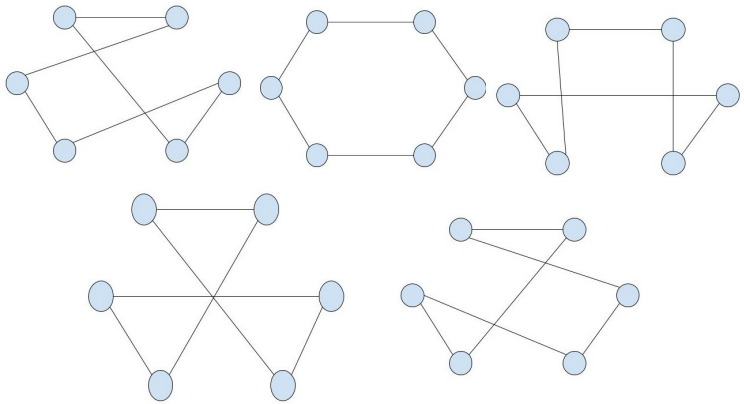
Local search for traveling salesperson problem

Idea: define a local operation that changes a given tour into another tour:

- 2-opt move:



- 3-opt move:



- k-opt move
- Lin-Kernighan-heuristics (LKH) is a combination of 2-opt, 3-opt, and rare k-opt moves (recall, still state-of-the-art to solve TSP)

Use of local search

Observe: local search methods can be used in any other optimization algorithm in order to (try to) converge to a local minimum. That is exactly what LKH (Lin-Kernighan-Helsgaun, the very good implementation) does. However:

- Can all tours be reached with 2-opt moves? (when starting with a certain initial tour) ...*still an open question*
- There are worst case scenarios where the 2-opt heuristics has exponential runtime until convergence.
- What about 3-opt, or k-opt, moves? ...*still an open question*

Reactive Tabu Search

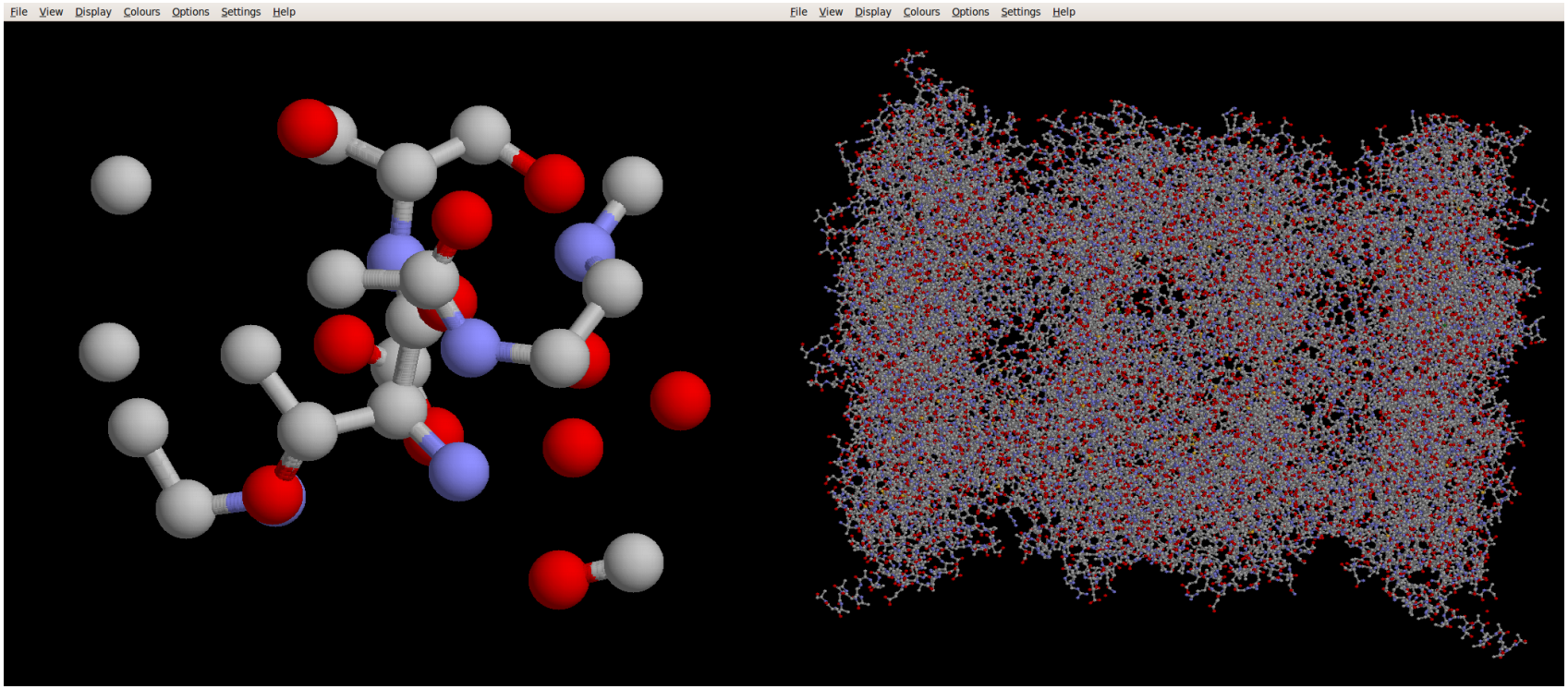
- start with a feasible solution (e.g., with some heuristics)
- search for possibilities to improve the current solution (e.g., search in the neighborhood)
- if we can improve: choose one, the best or a random one.
- if we cannot improve (i.e., trapped at a minimum):
  - search for possibilities to worsen the current solution
  - if we can escape: try again improvements
  - if we cannot escape: jump to another feasible solution

The Tabu criterion

- avoid repetitive movements taking advantage of a memory that stores forbidden intermediate solutions
  (or forbidden specific features of the current neighborhood search)
- i.e., mark certain movements as tabu for a certain number of iterations, i.e.,the memory is volatile!
- reactive means that the tabu period is dynamically adapted,

psm: point set match for proteins

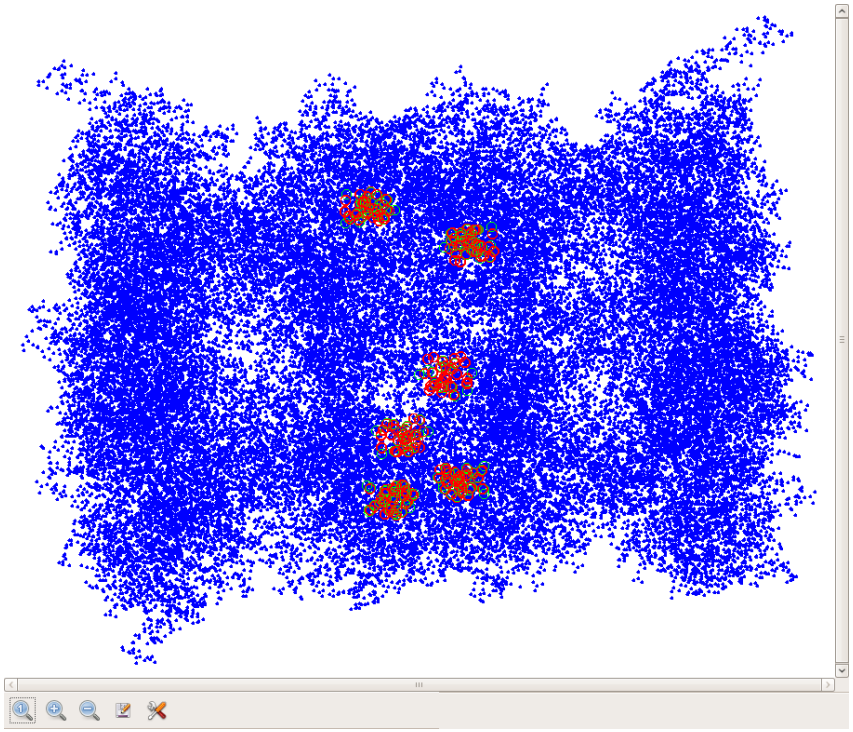
A template and graph based method with local search and use of domain knowledge for approximate match.



searching a 3D-structure (34 atoms) in a protein (50000 atoms)

psm: point set match for proteins

psm finds, for instance, six locations:



things to take into account

the search space and/or the objective function can be:

discrete	continous
total	partial
simple	complex
explicite	implicite
modelado	experimental
linear	non-linear
convex	non-convex
differentiable	non–differentiable
single-objective	multi-objective
constrained	unconstrained
static	dynamic

We have seen already a lot of examples of all kind.

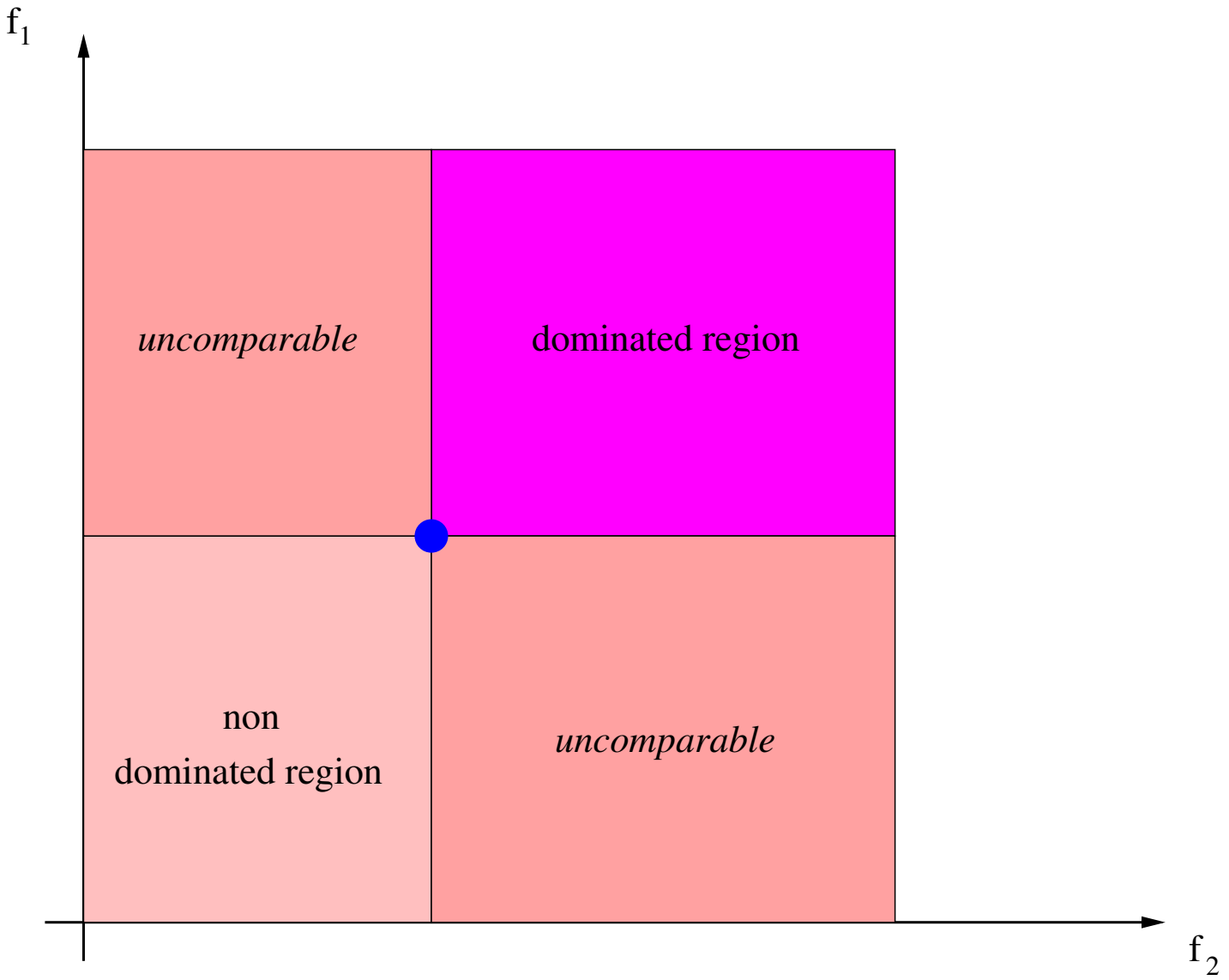
multi-objective optimization

- Given a search space $\mathbb{X}$ (called as well *search domain* or *problem space*) and
- a set of k functions f_i (bounded from below) from the search space to the real numbers (or at least a totally ordered set), e.g. $f_i : \mathbb{X} \longrightarrow \mathbb{R}$,
- find an element $x^\star \in \mathbb{X}$ such that $f_i(x^\star) \leq f_i(x)$ for all $x \in \mathbb{X}$ and all k functions f_i.
- Maybe there is no point in $\mathbb{X}$ that minimizes all the k functions simultaneously, then we look for Pareto-optimal solutions, i.e., solutions that cannot be improved without worsening at least one of the other objective functions.

Pareto front

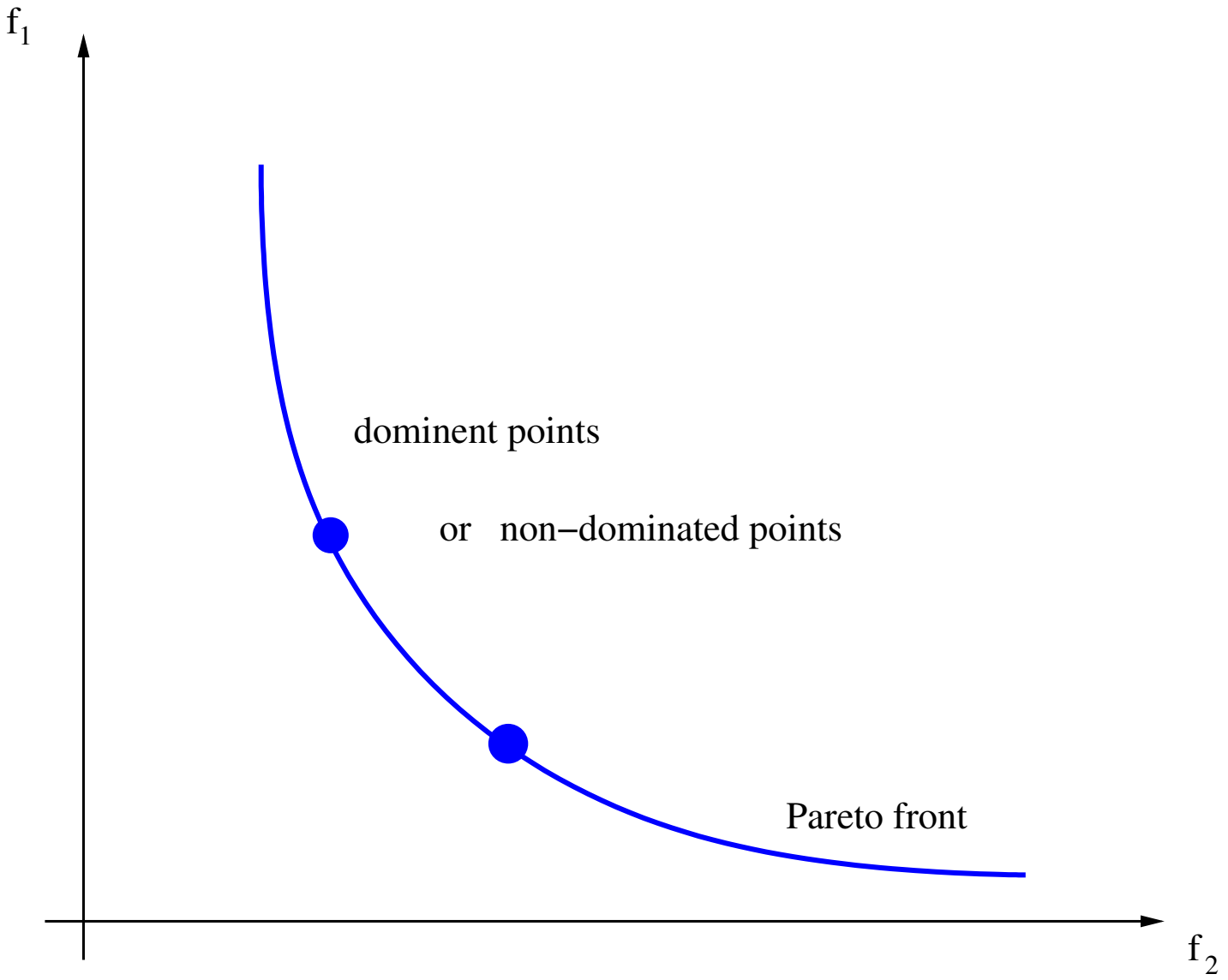
- Remind we have more than one independent objective function
- Pareto optimal (global): every other component for all other solutions is worse (or equal)
  (other names are: efficient points, dominant points, non-interior points)
- Pareto optimal (local): every other component for all other solutions in a local neighborhood is worse (or equal)
- Hence, the Pareto frontier describes the trade-off between the different objectives.

Dominant points and regions



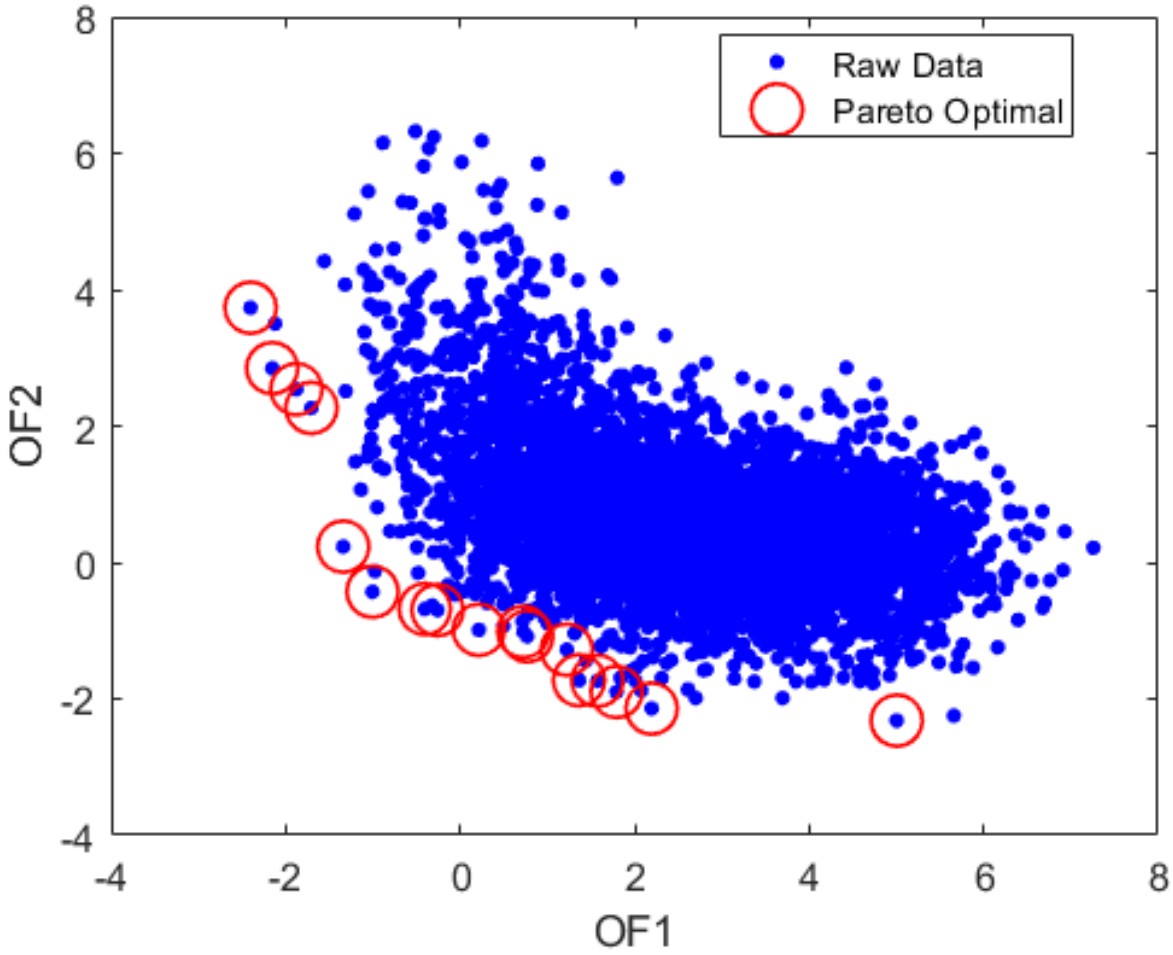
Pareto front

Example of a Pareto front with two marked non-dominated points:



Pareto front

Example of a Pareto front for two objectives (on measured data):



Multi–objective optimization

How can we find a *good* solution to the optimization problem with more than one objective function?

- convex combination of the objectives
- homotopic techniques, i.e., compute the entire Pareto frontier, for instance with a population based algorithm, and select later... (to obtain the Pareto frontier one might explore the coefficient space of the convex combination)
- goal programming, i.e., fixed values for all objectives and minimize the distance of all objectives to the predefined goals (according to some convenient distance metric)

Multi–objective optimization

- priority optimization, i.e., fix thresholds for all but one objective function beforehand and optimize above the threshold according to the most important one
- priorization (multi-level) programming, i.e., optimize according to a predefined ordering of the objective functions.
- fixed trade-off, i.e., find the point in the Pareto front that is tangent to a certain hyperplane (especially usefull when Pareto front is convex and low dimensional).

Optimization with restrictions (or constraints)

In many application there are restrictions (or constraints) that limit the optimization process:

- find an element $x^\star \in \mathbb{X}$ such that $f_i(x^\star) \leq f_i(x)$ for all $x \in \mathbb{X}$ and all k functions f_i and
- a certain number L of inequality constraints $g_j(x) \geq 0$ (for all $j \in [1 : L]$) are fulfilled, and
- a certain number E of equality constraints $h_n(x) = 0$ (for all $n \in [1 : N]$) are fulfilled.

Simple constraints are for instance so-called box-constraints, i.e., the search space is confined in each dimension by an interval.

Such box constraints are often handled separately in the optimization packages.

Optimization with restrictions (or constraints)

- The inequality and equality constraints again might be linear or non-linear functions.
- Due to the restrictions there might arise points (elements of the search space) during the optimization process which are unfeasible, i.e., no valid objective function values can be computed (or even the objective function cannot be computed at all).
- Sometimes even trying to find some feasible solution is already a very complex task (for instance: for TSP with time windows the problem is already NP-hard).

Optimization with restrictions (or constraints)

To tackle constraints there are two main classical approaches:

- use of penalties, i.e., assign a *sufficiently large* values to the objective function(s)
- use of interior methods, i.e., make sure not to leave the feasible region
  (main idea: use the fulfillment of the constraints as additional objective function building a so-called *barrier function* with an additional parameter μ that is continuously shrinked to reach zero.)

Multi–objective optimization with evolutionary methods

- Evolutionary methods can approximate the Pareto frontier in parallel (with the help of the diversity among the individuals).
- For instance particle swarm systems varying the weights of a convex combination periodically during the iterations.
- For instance a genetic algorithm can hold a population that tries to converge (in some sense uniformly) towards the Pareto front.

multi-objective evolutionary algorithms (MOEA)

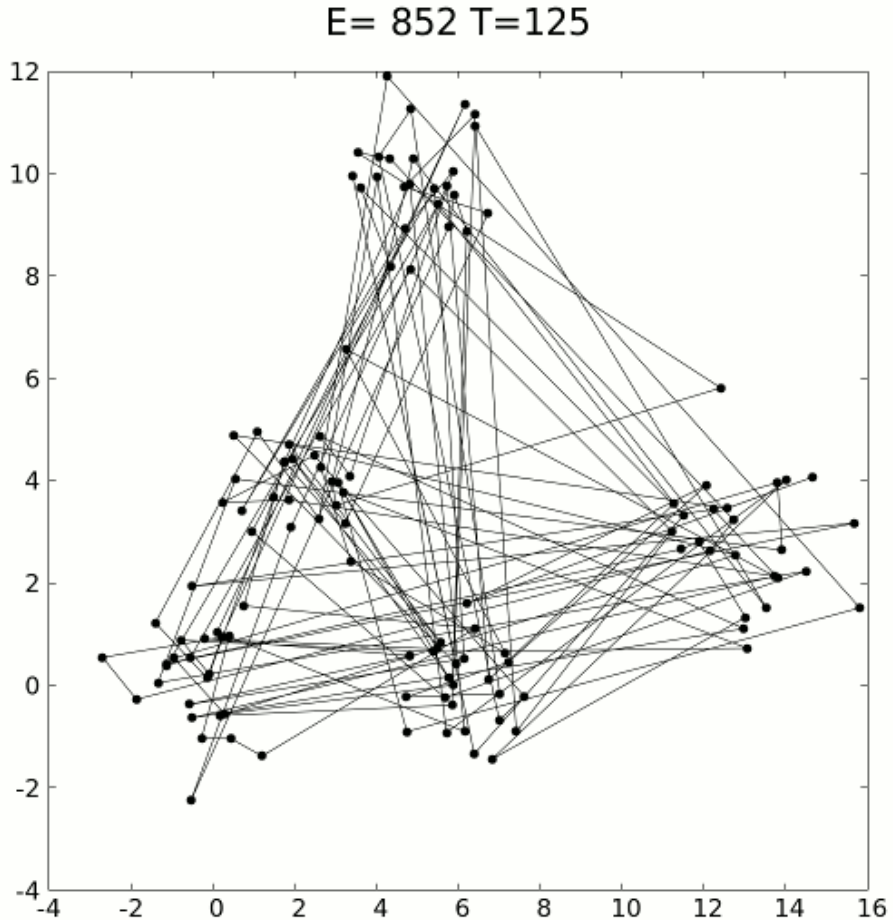
- VEGA, vector evaluated genetic algorithm
  Idea: in the selection process parts of the mating parents are selected according to each objective function
- MSGA, multi-sexual genetic algorithm
  Idea: individuals are marked as belonging to a certain objective function, ranking is used to select parents, only differently marked individuals are used to generate children
- NSGA, non-dominant sorting genetic algorithm
  Idea: sort individuals according to their dominance, and design the selection according the dominance classes (i.e., work with several frontiers, intending to converge eventually to the Pareto front.

multi-objective evolutionary algorithms (MOEA)

- NSGA-II, including elitism, dominant individuals are preserved in population, clustering is avoided
- SPEA, strength Pareto Evolutionary algorithm
  Idea: maintain a fixed set of best individuals while guaranteeing that they are spread over the Pareto front without to much clustering

Simulated Annealing

The name and idea stems from the slow and repeated heating and cooling process of certain materials (usually alloys of different metals with addings, e.g., iron+carbon) until certain properties are achieved.



https://en.wikipedia.org/wiki/Simulated_annealing

Simulated Annealing

- Explores a neighbor in the search space, whenever there is an improvement in the objective function at the neighbor or it is not worse than a temperature dependent threshold with some probability.
- Let Δf be the difference between current solution f and neighbor solution f_n.
- Let T be a temperature.
- Then the neighbor is accepted whenever either $\Delta f < 0$
  (we are going downhill)
  or if $e^{-\Delta f/T} > r$, for r being a random value in $[0:1]$
  (we are going uphill).
- With increasing iteration rounds, i.e., during a certain number of iterations the temperature is held constant, the temperature is reduced, hence, the threshold converges towards zero.

Simulated Annealing cooling schemes

- linear cooling: $T = T_0 - \eta k$
  with T_0 an initial temperature, k the iteration number and η some free parameter, being constant during optimization.
  (To avoid negative temperature: $T = \max(T_0 - \eta k, T_{\min})$)
- exponential cooling: $T = aT$
  with $a \in [0.8, 1)$ typically; slower cooling the a close to 1.
- inverse cooling: $T = T/(1 + \beta T)$
  with β being a small constant (e.g. $\beta = 0.001$).
- logarithmic cooling: $T = T_0/\log k$
  with c a suitable constant.
  (Used as well a generalization: $T = T_0/\log(k + d)$).
  Not really practical in applications but was used to prove convergence to global minimum under certain conditions.
- inverse linear cooling: $T = T_0/k$.
  Not really practical in applications but was used to prove convergence to global minimum under certain conditions.

Simulated Annealing cooling schemes

- It seems that any monotonely decreasing function, that is neither too steep at the beginning and neither to slow decreasing in the end, might serve.
- Cooling can be implemented differently in each dimension in multi-dimensional problems.
- The entire process can be restarted from another location in search space (Monte Carlo approach around inner simulated annealing process).

Simulated Annealing vs. Tabu Search

- Both methods belong to the random path methods:
  - there is only one individual
  - the moves in the search space explore the neighborhood in a random manner
  - a move is accepted whenever
    - TS: a random, but not tabu, move among the improving once in first place and non-improving onces in second place.
    - SA: a random move that fulfills the annealing condition
- Tabu search is, as improvements are prefered, a local search method that finds local minima;
  which might not be the case for simulated annealing (you need to add some final local search approach).

memetic algorithms

The population based algorithms usually do not perform local optimization of the individuals. Including such an approach is called a memetic algorithm:

- use an adequate local search strategy (e.g. steepest decent, iterated local search etc.) to improve the fitness of the individuals
- this improvement of an individual is also called *individual learning*
- the result of the local optimization:
  - might change the genotype of the individual (Lamarckian learning), i.e., the changed individual participates in the genetic algorithm, or
  - might not change the genotype of the individual (Baldwinian learning), i.e., the population is not changed
- the local search can be restricted to a certain part of the population, for instance, the best ranked ones
- the local search can be restricted to be performed only after a certain amount of iterations in the overall genetic algorithm

biogeography-based optimization

- The biogeography-based optimization uses the idea of populations evolving at islands that once in a while exchange individuals via migration.
- Implemented as further meta-heuristic in the framework of genetic algorithms (especially for separable objective function, where optimization of their convex combination is an option).
- Can be applied to particle swarm optimization as well (I haved found implementation examples, e.g. from 2017).

much more nature-inspired work done

Without going into details, there is a huge bunch of other nature-inpired optimization heuristics (see introduction as well):

- ant lion optimization
- artificial bee colonies
- bat algorithm
- dragonfly algorithm
- firefly optimization
- grasshopper optimization
- grey wolf optimizer
- whale optimization
- invasive weed optimization
- ...
- cristalization of materials
- great deluge algorithm
- gravitational search algorithm
- etc. etc. etc.

issues to be considered

The following issues should be considered when implementing and using a certain optimization approach for a given problem:

- implementation difficulty
- number of parameters to be adjusted
- population size
- number of objective function evaluation
- convergence velocity (convergence profile)
- handling of local optima (stucking)
- handling of restrictions
- statistically correct evaluation on benchmark problems and MonteCarlo runs

efficiency issues to be considered

The following efficiency aspect might be useful in a certain implementation of an optimization approach on a given platform:

- use of parallelization
- use of caching
- use of efficient data structures
- use of adequate precision in calculation
- use of approximation algorithms (especially in early phases for the objective function computation)
- use good random number generators

hyper-parameter tuning

Many of the different heuristics seen so far have a certain number of free parameters that must be fixed when starting the optimization, but

- we might use another optimization algorithm that tries to find a good parameter set for the original algorithm
- for instance, we run the algorithm on a suitable benchmark suite to obtain good free parameters, and then run the optimization with these settings on the real problems we are interested to solve
- this process can be iterated, i.e., each use of the optimizer might adapt to a certain degree its free parameters
- eventually we arrive at an online-algorithm which adapts its parameters with each problem it is faced to

extension: an optimization problem

Given for a undirected graph certain properties:

- number of nodes
- number of edges
- distribution of node degrees
- assortativity of the interconnexions

$$a = \frac{\sum_{(i,j)}(d_i - \overline{d}_i) \cdot (d_j - \overline{d}_j)}{\sqrt{\sum_i (d_i - \overline{d}_i)^2} \cdot \sqrt{\sum_j (d_j - \overline{d}_j)^2}} \quad (1)$$

- distribution of length of edges
  (according some coordinates of the nodes in the plane)
- must be connected

Find a (large) random graph that fulfills (more or less) all properties.

example

- number of nodes: 1.000.000
- number of edges: 149.000.000
- distribution of node degrees: facebook distribution
- assortativity of the interconnexions: -0.189 (facebook)
- distribution of length of edges: population in Germany
- must be connected

current state-of-solution

Input:

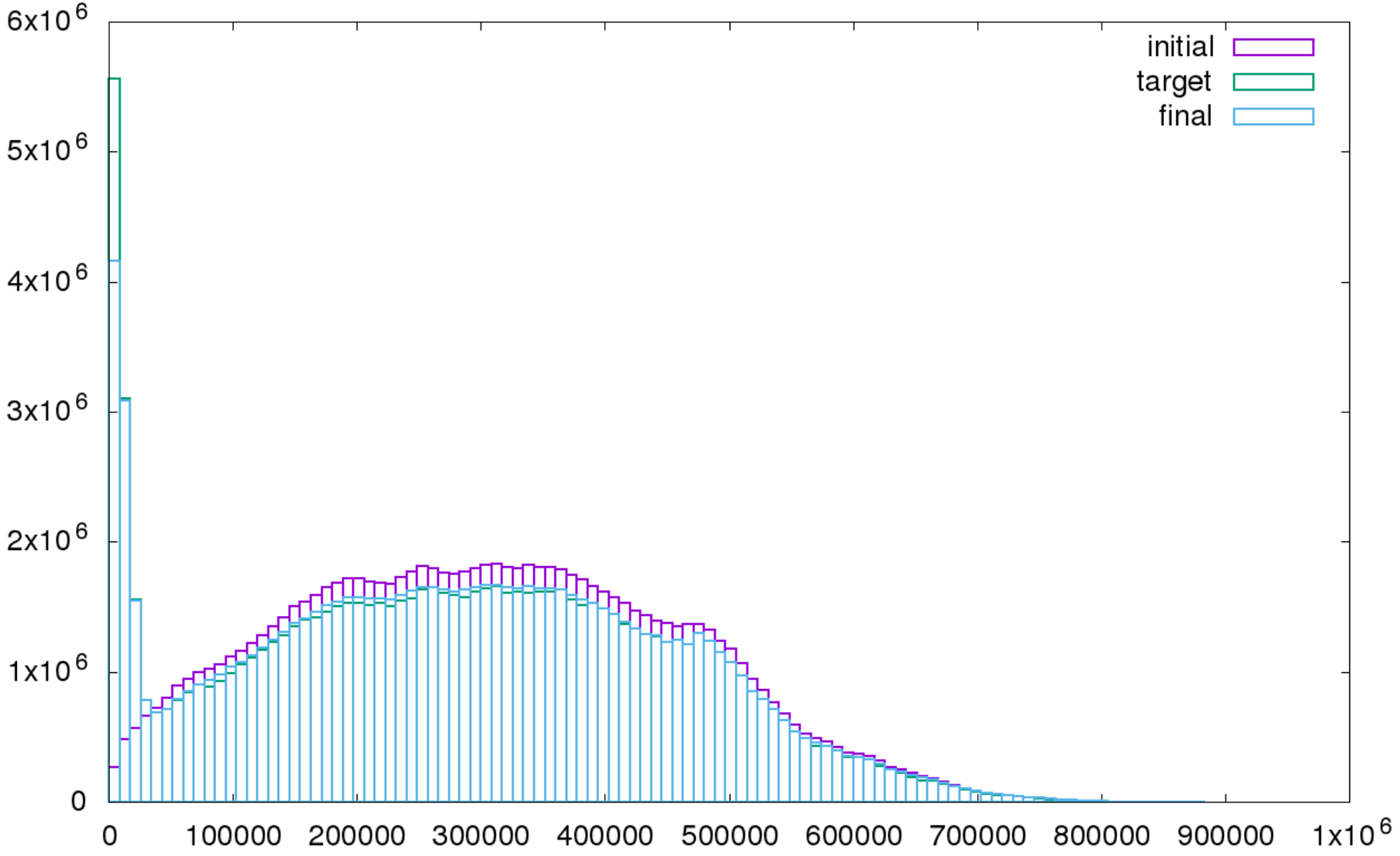
- fix number of nodes
- fix degree distribution
- scale degree distribution, hence, get number of edges
- fix target assortativity
- fix target length distribution
- must be connected

current state-of-solution

Algorithm:

- randomized Havel-Hakimi-Horvat-algorithm generates graph with exact degree distribution
- alternating annealing algorithm using some type of 2-opt operation on graph
- connectivity tested and corrected once in a while

result



length distribution

result

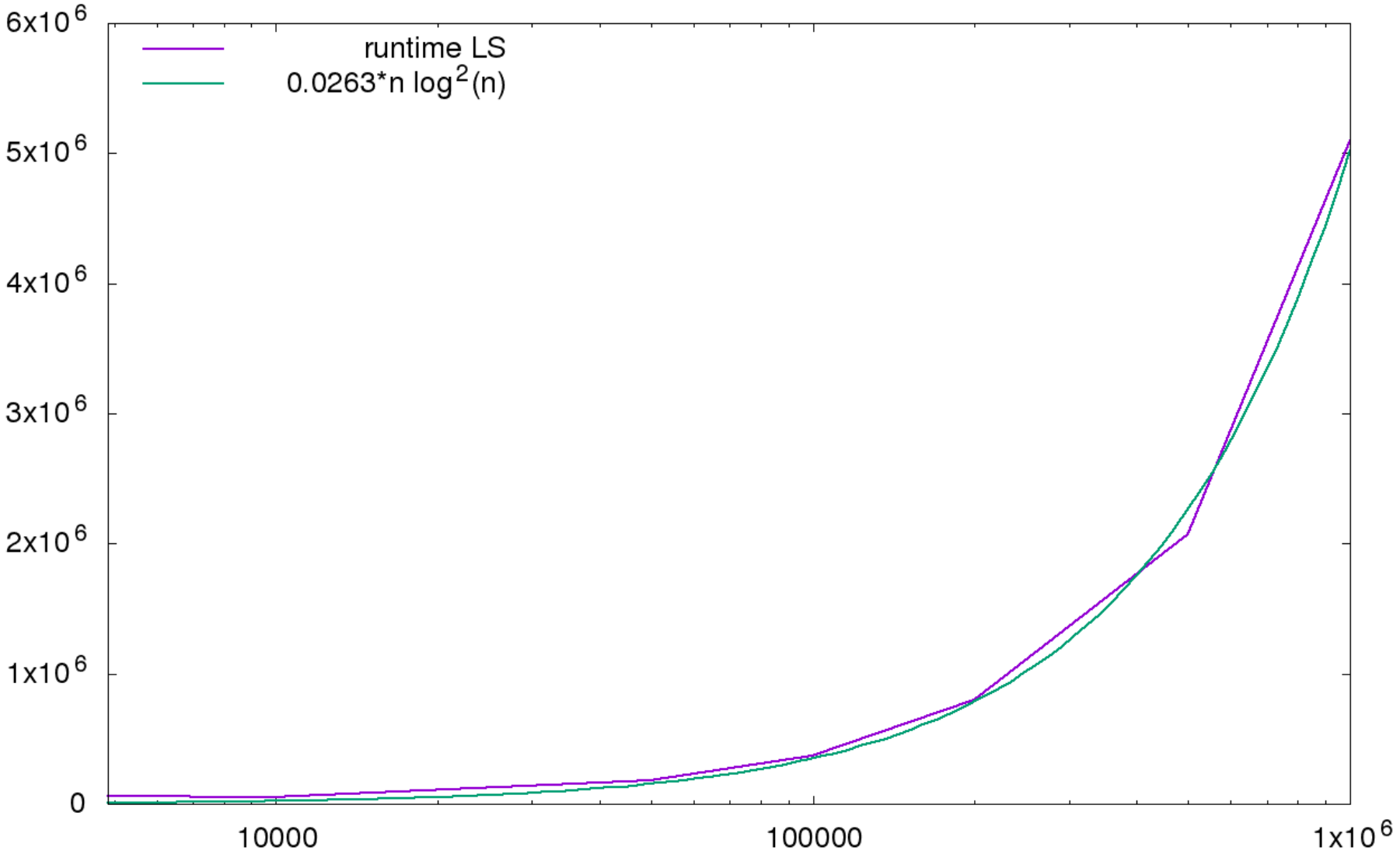
The previous slide shows:

- magenta: the initial length distribution
  (after having run the randomized HHH-algorithm),
- green: the target length distribution,
- blue: the generated length distribution
  (after having run the alternating annealing algorithm)

where the assortativity is below 0.1% of the target assortativity.

Hence, we found a connected graph with 1.000.000 nodes and 149.000.000 edges that has exactly the given degree distribution and very close assortativity and length distribution regarding the given target values.

runtime of graph generating tool



runtime versus number of nodes in [ms]

last but not least

that's it, folks ... thanks. questions?