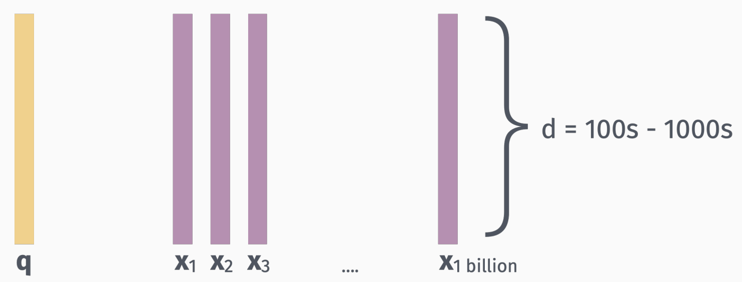
Navigability and Graph-based Vector Search

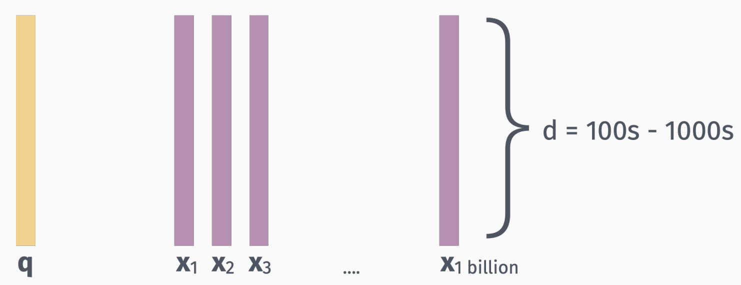
Christopher Musco, New York University

Based on work with: Haya Diwan, Jinrui Gou, Cameron Musco, Torsten Suel, Alex Conway, Laxman Dhulipala, Martin Farach-Colton, Rob Johnson, Ben Landrum, Yarin Shechter, Richard Wen



Algorithmic problem: find the closest k vectors in a vector database $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to a given query vector $\mathbf{q}$. For $k = 1$,

$$\text{return} \quad \arg\min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

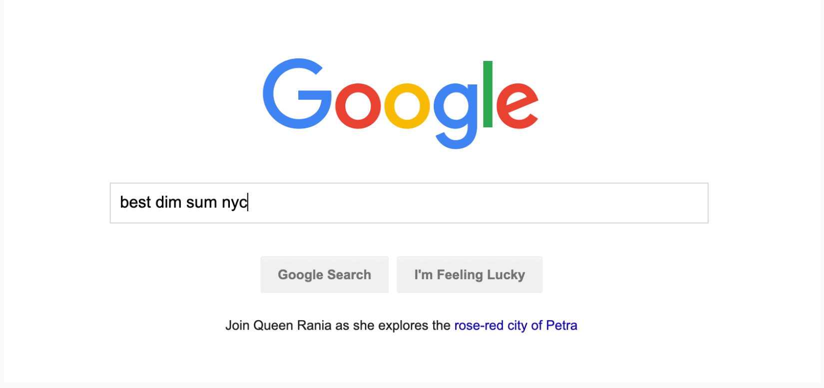


Algorithmic problem: find the closest k vectors in a vector database $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ to a given query vector $\mathbf{q}$. For $k = 1$,

$$\text{return} \quad \arg\min_{i \in \{1,\dots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

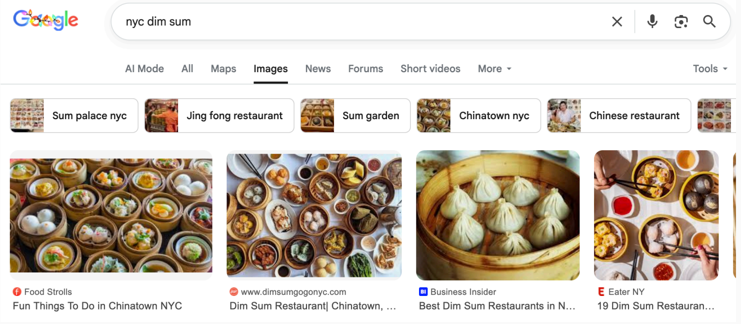
Classic problem that has received significant renewed interest in recent years. Main technique behind AI-based search.

CLASSICAL SEARCH



For decades search has been largely based on keywords.

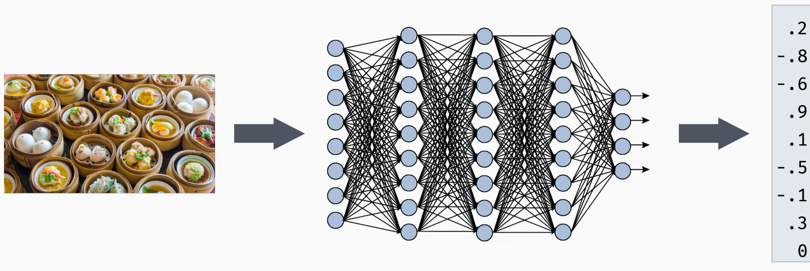
Return documents (webpages, emails, files on your computer) that contain many of the words in your query.



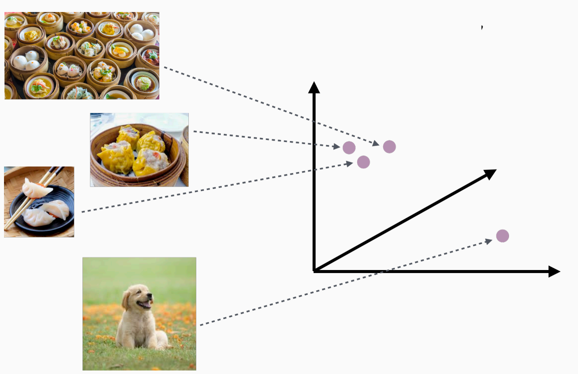
Until fairly recently, this was even true for image and multimedia search.

```
1  <div class="image-block-wrapper" data-animation-role="image">
2      <div class="sqs-image-shape-container-element">
3              <img alt="table full of colorful authentic Dim Sum
               baskets in NYC - best things to do Chinatown NYC"
               src="https://images.squarespace-cdn.com/content/v1/
               5e18d3/1624117/best+things+to+do+Chinatown+NYC+-
               +Dim+Sum" width="2500" height="1406">
4      </div>
5  </div>
```

Step 1: Deep neural network (BERT, CLIP, etc.) is used to convert documents, images, videos, and more into high-dimensional, numerical vectors.

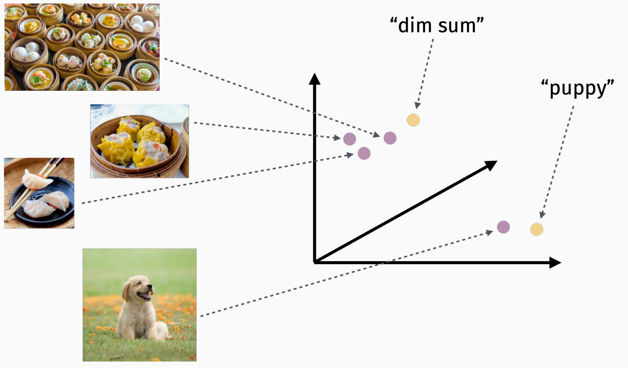


Step 1: Deep neural network (BERT, CLIP, etc.) is used to convert documents, images, videos, and more into high-dimensional, numerical vectors.



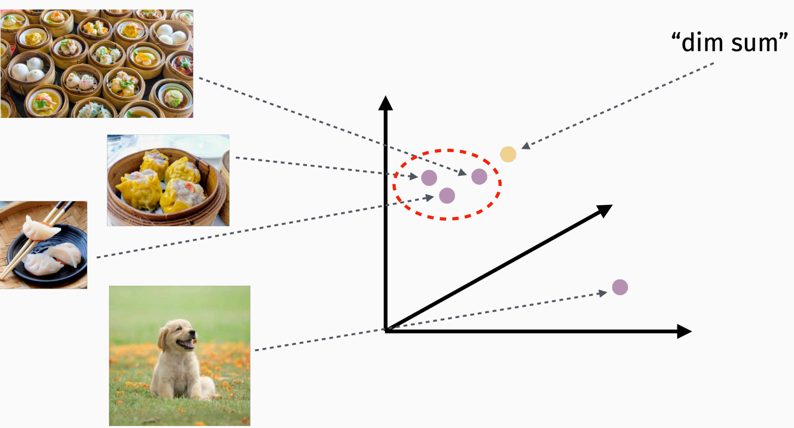
The network is trained so that semantically similar objects map to similar vectors.

Step 1: Deep neural network (BERT, CLIP, etc.) is used to convert documents, images, videos, and more into high-dimensional, numerical vectors.



The network is trained so that semantically similar objects map to similar vectors.

Step 2: Find items matching a query by embedding the query into a vector, and finding nearby vectors.



Modern embeddings have made this a state-of-the-art paradigm for a wide variety of search tasks.

$$\arg\min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

Modern embeddings have made this a state-of-the-art paradigm for a wide variety of search tasks.

$$\arg\min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

Unfortunaetly, this is a notoriously hard algorithmic problem...

For a database of n vectors in d dimensions, a naive linear scan takes $O(nd)$ time.

Modern embeddings have made this a state-of-the-art paradigm for a wide variety of search tasks.

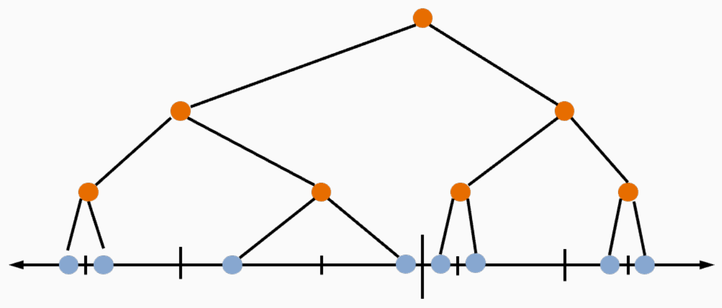
$$\arg\min_{i \in \{1,\dots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

Unfortunaetly, this is a notoriously hard algorithmic problem...

For a database of n vectors in d dimensions, a naive linear scan takes $O(nd)$ time.

Our only hope to do better is if we allow for **pre-processing.**

For 1-dimensional vectors (i.e., **q** is a number), we can solve the search problem by building a binary search tree.

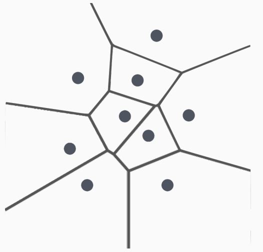


Preprocessing time: $O(n \log n)$.

Space complexity: $O(n)$.

Query time: $O(\log n)$.

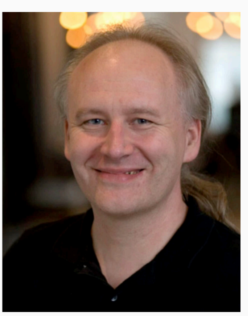
Can also be solved with $O(n)$ space and $O(\log n)$ time in $d = 2$ dimensions.



For $d \geq 3$, the space or query complexity of classical methods depends exponentially on d, even if we allow approximation.

$$\text{Find } \mathbf{x}_j \text{ such that } \|\mathbf{x}_j - \mathbf{q}\| \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

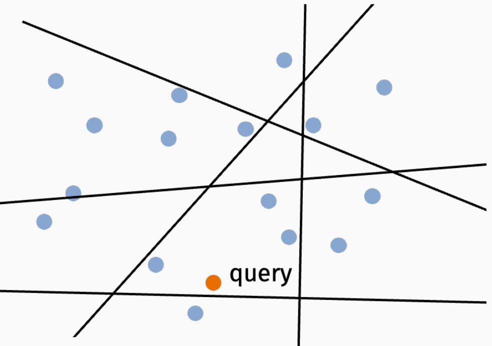
The "curse-of-dimensionality" was broken with the introduction of **Locality Sensitive Hashing (LSH).**



[Indyk, Motwani, 1998].

Rough idea behind Locality Sensitive Hashing (LSH):

1. Pick a bunch of random hyperplanes.
2. Check which side of each hyperplane $\mathbf{q}$ lies on.
    - Takes $O(d)$ time per hyperplane.
3. Return closest point that lies in the same region as $\mathbf{q}$.



Possible to prove some pretty amazing theoretical guarantees!

Theorem (Andoni, Indyk, FOCS 2006)

For any $c \geq 1$, in $\tilde{O}\left(dn^{1/c^2}\right)$ time we can find $\mathbf{x}_j$ satisfying:

$$\|\mathbf{x}_j - \mathbf{q}\| \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

Space complexity of data structure is $\tilde{O}\left(nd + n^{1+1/c^2}\right)$

Possible to prove some pretty amazing theoretical guarantees!

Theorem (Andoni, Indyk, FOCS 2006)

For any $c \geq 1$*, in* $\tilde{O}\left(dn^{1/c^2}\right)$ *time we can find* $\mathbf{x}_j$ *satisfying:*

$$\|\mathbf{x}_j - \mathbf{q}\| \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

Space complexity of data structure is $\tilde{O}\left(nd + n^{1+1/c^2}\right)$

As an example, if $c = 2$, query time scales with $n^{1/4}$, which is pretty great! $(1 \text{ billion})^{1/4} < 200$.

Possible to prove some pretty amazing theoretical guarantees!

Theorem (Andoni, Indyk, FOCS 2006)

For any $c \geq 1$, in $\tilde{O}\left(dn^{1/c^2}\right)$ time we can find $\mathbf{x}_j$ satisfying:

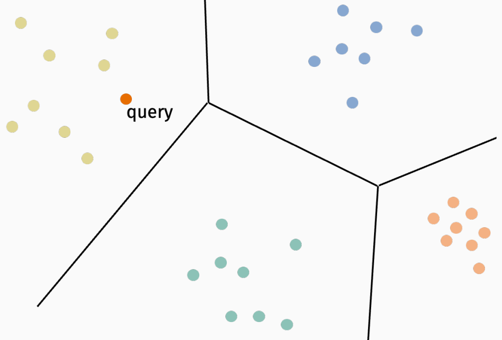
$$\|\mathbf{x}_j - \mathbf{q}\| \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|.$$

Space complexity of data structure is $\tilde{O}\left(nd + n^{1+1/c^2}\right)$

As an example, if $c = 2$, query time scales with $n^{1/4}$, which is pretty great! $(1 \text{ billion})^{1/4} < 200$.

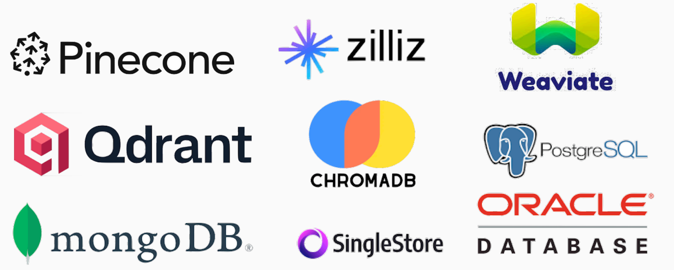
Also works well in practice. **LSH and related developments won Indyk, Charikar, and Broder the 2012 ACM Paris Kanellakis Theory and Practice Award.**

Practical systems improve accuracy by choosing hyperplanes in a data-dependent way, e.g. via k-means clustering.



Key component of state-of-the-art vector search libraries like Meta's FAISS and Google's SCANN.

A multi-billion dollar industry has developed around the vector search problem, giving a new opportunity to stress-test existing algorithms based on LSH and clustering.



Five years later, very few of these products rely primarily on space-partitioning methods.

New kid on the block: Graph-based Search.

- Navigating Spreading-out Graphs (NSG) [Fu, Xiang, Wang, Cai, 2017]
- Hierarchical Navigable Small World (HNSW) [Malkov, Yashunin, 2018]
- Microsoft's DiskANN [Subramanya, Devvrit, Kadekodi, Krishaswamy, Simhadri 2019]

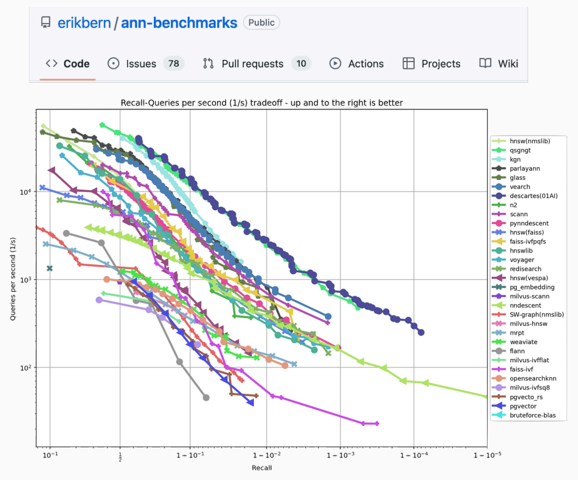
New kid on the block: Graph-based Search.

- **Navigating Spreading-out Graphs (NSG)** [Fu, Xiang, Wang, Cai, 2017]
- **Hierarchical Navigable Small World (HNSW)** [Malkov, Yashunin, 2018]
- **Microsoft's DiskANN** [Subramanya, Devvrit, Kadekodi, Krishaswamy, Simhadri 2019]

Very similar methods studied for low-dimensions in the 1990s by Arya, Mount, Clarkson, Kleinberg, and others.

The basic idea dates back even further to the "small world" experiments of Stanley Milgram from the 1960s.

Graph-based methods are topping benchmarks and competitions!



Graph-based methods are topping benchmarks and competitions!

Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search

Harsha Vardhan Simhadri[1] HARSHASI@MICROSOFT.COM
George Williams[2] GWILLIAMS@IEEE.ORG
Martin Aumüller[3] MAAU@ITU.DK
Matthijs Douze[4] MATTHIJS@FB.COM
Artem Babenko[5] ARTEM.BABENKO@PHYSTECH.EDU
Dmitry Baranchuk[5] DBARANCHUK@YANDEX-TEAM.RU
Qi Chen[1] CHEQI@MICROSOFT.COM
Lucas Hosseini[4] LUCAS.HOSSEINI@GMAIL.COM
Ravishankar Krishnaswamy[1] RAKRI@MICROSOFT.COM
Gopal Srinivasa[1] GOPALSR@MICROSOFT.COM
Suhas Jayaram Subramanya[6] SUHASJ@CS.CMU.EDU
Jingdong Wang[7] WANGJINGDONG@BAIDU.COM

[1] Microsoft Research [2] GSI Technology [3] IT University of Copenhagen
[4] Meta AI Research [5] Yandex [6] Carnegie Mellon University [7] Baidu

Results of the Big ANN: NeurIPS'23 competition

Harsha Vardhan Simhadri, Microsoft, harshasi@microsoft.com
Martin Aumüller, IT University of Copenhagen, maau@itu.dk
Amir Ingber, Pinecone, ingber@pinecone.io
Matthijs Douze, Meta AI Research, matthijs@meta.com
George Williams
Magdalen Dobson Manohar, Carnegie Mellon University
Dmitry Baranchuk, Yandex
Edo Liberty, Pinecone
Frank Liu, Zilliz
Ben Landrum, University of Maryland
Mazin Karjikar, University of Maryland
Laxman Dhulipala, University of Maryland
Meng Chen, Yue Chen, Rui Ma, Kai Zhang, Yuzheng Cai, Jiayang Shi, Yizhuo Chen, Weiguo Zheng, Fudan University
Zihao Wang, Shanghai Jiao Tong University
Jie Yin, Baidu
Ben Huang, Baidu

Graph-based methods are topping benchmarks and competitions!

Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search

Harsha Vardhan Simhadri[1] HARSHASI@MICROSOFT.COM
George Williams[2] GWILLIAMS@IEEE.ORG
Martin Aumüller[3] MAAU@ITU.DK
Matthijs Douze[4] MATTHIJS@FB.COM
Artem Babenko[5] ARTEM.BABENKO@PHYSTECH.EDU
Dmitry Baranchuk[5] DBARANCHUK@YANDEX-TEAM.RU
Qi Chen[1] CHEQI@MICROSOFT.COM
Lucas Hosseini[4] LUCAS.HOSSEINI@GMAIL.COM
Ravishankar Krishnaswamy[1] RAKRI@MICROSOFT.COM
Gopal Srinivasa[1] GOPALSR@MICROSOFT.COM
Suhas Jayaram Subramanya[6] SUHASJ@CS.CMU.EDU
Jingdong Wang[7] WANGJINGDONG@BAIDU.COM

[1] Microsoft Research [2] GSI Technology [3] IT University of Copenhagen
[4] Meta AI Research [5] Yandex [6] Carnegie Mellon University [7] Baidu

Results of the Big ANN: NeurIPS'23 competition

Harsha Vardhan Simhadri, Microsoft, harshasi@microsoft.com
Martin Aumüller, IT University of Copenhagen, maau@itu.dk
Amir Ingber, Pinecone, ingber@pinecone.io
Matthijs Douze, Meta AI Research, matthijs@meta.com
George Williams
Magdalen Dobson Manohar, Carnegie Mellon University
Dmitry Baranchuk, Yandex
Edo Liberty, Pinecone
Frank Liu, Zilliz
Ben Landrum, University of Maryland
Mazin Karjikar, University of Maryland
Laxman Dhulipala, University of Maryland
Meng Chen, Yue Chen, Rui Ma, Kai Zhang, Yuzheng Cai, Jiayang Shi, Yizhuo Chen, Weiguo Zheng, Fudan University
Zihao Wang, Shanghai Jiao Tong University
Jie Yin, Baidu
Ben Huang, Baidu

Despite their success, we have very few theoretical guarantees for graph-based methods. Notably in contrast to space-partitioning methods like LSH.

OPEN THEORY CHALLENGE: CAN WE EXPLAIN THE EMPIRICAL SUCCESS OF GRAPH-BASED VECTOR SEARCH?

There has been a lot of interesting recent work[1], but we are still far from addressing this challenge in a satisfying way.

Goal for remainder of talk:

1. Discuss connection between the performance of graph-based search and the concept of graph navigability.
2. Introduce recent results on the existence and construction of sparse navigable graphs (NeurIPS 2024, SODA 2026).
3. Discuss open questions and next steps.

[1] [Laarhoven 2018, Prokhorenkova, Shekhovtsov 2020, Indyk, Xu 2023, Gollapudi, Krishnaswamy, Shiragur, Wardhan 2025, Har-Peled, Raichel, Robson 2025, many more]

c-approximate nearest neighbor search: Return j satisfying $\|\mathbf{x}_j - \mathbf{q}\| \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|$ for some $c \geq 1$.

c**-approximate nearest neighbor search:** Return j satisfying $\|\mathbf{x}_j - \mathbf{q}\| \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|$ for some $c \geq 1$.

Observation: Assume no duplicates in $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. If query $\mathbf{q} = \mathbf{x}_j$ for some j, we must return j.

Search graph G should be chosen to at least ensure that we find q if it is in the dataset.

c**-approximate nearest neighbor search:** Return j satisfying $\|\mathbf{x}_j - \mathbf{q}\| \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|$ for some $c \geq 1$.

Observation: Assume no duplicates in $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. If query $\mathbf{q} = \mathbf{x}_j$ for some j, we must return j.

Search graph G should be chosen to at least ensure that we find q if it is in the dataset.

Ideally, G should also be sparse and $\mathbf{q}$ is found in just a few greedy steps. Runtime of search $\approx$ (out degree) $\times$ (# steps).

Definition (Navigable Graph)

A directed graph G over a point set $\mathbf{x}_1, \ldots, \mathbf{x}_n$ is navigable if, for all $i, j \in \{1, \ldots, n\}$, greedy search run on G with start node i and query $\mathbf{x}_j$ returns j.

Navigability is frequently listed as a desirably property. Lends its name to methods like “hierarchical navigable small world (HNSW) graphs” and “navigating spreading-out graphs (NSG)”

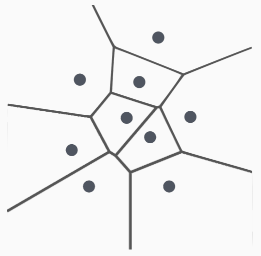
However, few papers actually construct provably navigable graphs! Employ heuristic search graph constructions.

Natural Questions: Do sparse navigable graphs even exist? Can we find them efficiently?

Navigability can be viewed as a minimum necessary condition for graph-based greedy search to succeed, so answering this question is critical to understanding graph-based methods.

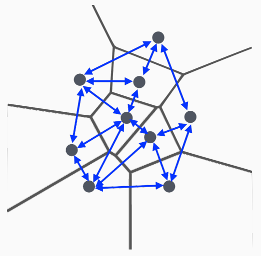
Known results in low-dimensional Euclidean space:

- **2-dimensions:** The Delaunay graph can be proven to be navigable. This graph has average degree ≤ 6.



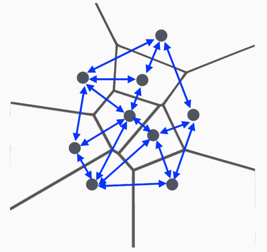
Known results in low-dimensional Euclidean space:

- **2-dimensions:** The Delaunay graph can be proven to be navigable. This graph has average degree ≤ 6.



Known results in low-dimensional Euclidean space:

- **2-dimensions:** The Delaunay graph can be proven to be navigable. This graph has average degree ≤ 6.



- **d-dimensions:** The Sparse Neighborhod Graph of Arya and Mount [SODA, 1993] is navigable and has average degree $O(2^d)$.

Claim (Upper Bound)

Any dataset $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *admits a navigable graph with average degree* $\leq 2\sqrt{n}$*, under any distance function.*

Today we will prove a slightly weaker $O(\sqrt{n \log n})$ bound.

Claim (Nearly Matching Lower Bound, NeurIPS 2024)

Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ *be random vectors in* $\{-1, 1\}^m$ *where* $m = O(\log n)$*. With high probability, any navigable graph requires average degree* $\Omega(n^{1/2-\epsilon})$ *for any fixed constant* ϵ*.*

I will give you a proof sketch.

Definition (Navigable Graph)

A directed graph G is navigable for a point set $1, \ldots, n$ and distance function $d(\cdot, \cdot)$ if, for all nodes i, for all $j \neq i$, there is some k in i's out neighborhood, $\mathcal{N}(i)$, such that:

$$d(j, k) < d(j, i).$$

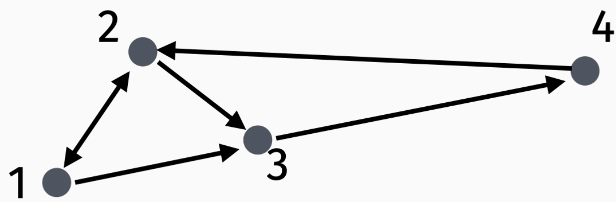
To avoid corner cases/tiebreaking, we will assume that all distances in the dataset are unique.

Definition (Navigable Graph)

A directed graph G is navigable for a point set $1, \ldots, n$ and distance function $d(\cdot, \cdot)$ if, for all nodes i, for all $j \neq i$,

$$d(j, k) < d(j, i).$$

for some k in i's out neighborhood, $\mathcal{N}(i)$

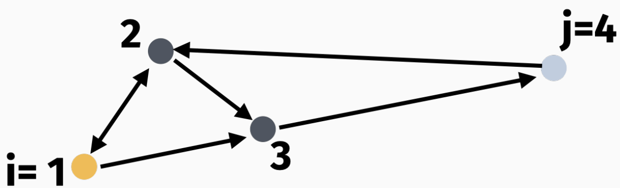


Definition (Navigable Graph)

A directed graph G is navigable for a point set $1, \ldots, n$ and distance function $d(\cdot, \cdot)$ if, for all nodes i, for all $j \neq i$,

$$d(j, k) < d(j, i).$$

for some k in i's out neighborhood, $\mathcal{N}(i)$

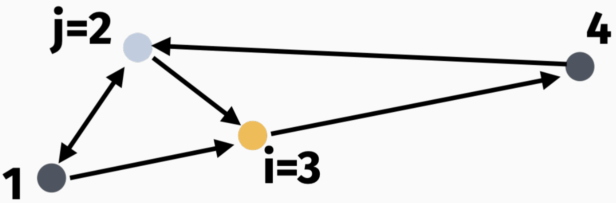


Definition (Navigable Graph)

A directed graph G is navigable for a point set $1, \ldots, n$ and distance function $d(\cdot, \cdot)$ if, for all nodes i, for all $j \neq i$,

$$d(j, k) < d(j, i).$$

for some k in i's out neighborhood, $\mathcal{N}(i)$

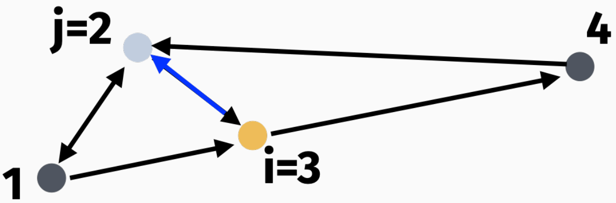


Definition (Navigable Graph)

A directed graph G is navigable for a point set $1, \ldots, n$ and distance function $d(\cdot, \cdot)$ if, for all nodes i, for all $j \neq i$,

$$d(j, k) < d(j, i).$$

for some k in i's out neighborhood, $\mathcal{N}(i)$



Imagine a Distance-Based Permutation Matrix, where the ith row lists all points sorted in order of their distance to point i.

Node 1	1	10	2	3	5	9	6	8	4	7
Node 2	2	4	6	1	10	5	3	9	7	8
					⋮					
Node n	10	9	1	5	7	2	4	8	3	6

Imagine a **Distance-Based Permutation Matrix**, where the ith row lists all points sorted in order of their distance to point i.

Node 1	1	10	2	3	5	9	6	8	4	7
Node 2	2	4	6	1	10	5	3	9	7	8
⋮										
Node n	10	9	1	5	7	2	4	8	3	6

For every edge in G, add that edge to every row of the matrix.

Imagine a **Distance-Based Permutation Matrix**, where the ith row lists all points sorted in order of their distance to point i.

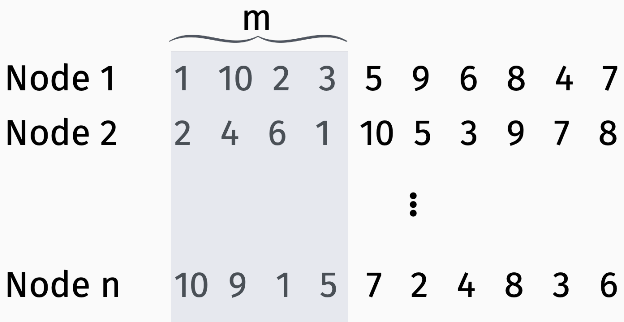
Node 1	1	10	2	3	5	9	6	8	4	7
Node 2	2	4	6	1	10	5	3	9	7	8
⋮										
Node n	10	9	1	5	7	2	4	8	3	6

For every edge in G, add that edge to every row of the matrix.

Navigability Requirement: Need at least one “left pointing” edge for every node in every list.

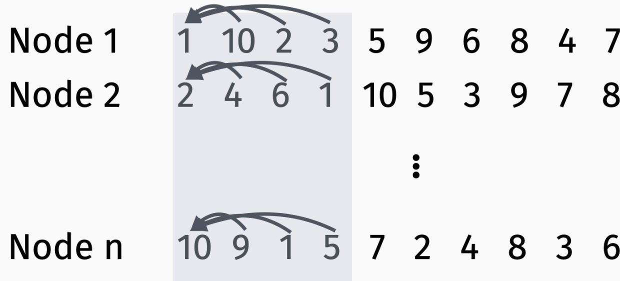
Construction: Choose $m < n$.

1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



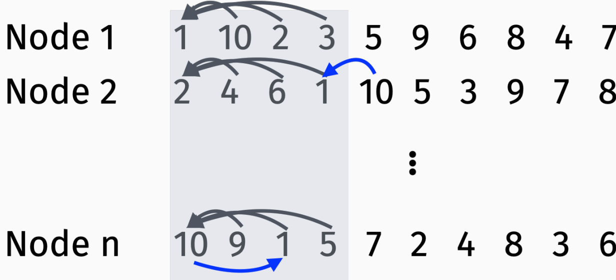
Construction: Choose $m < n$.

1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



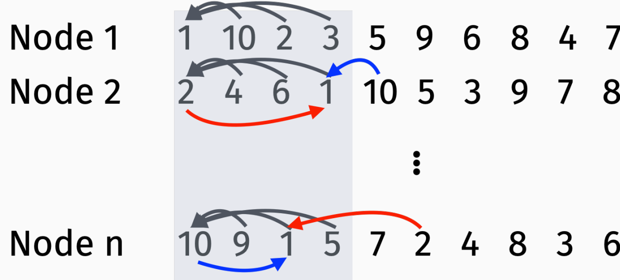
Construction: Choose $m < n$.

1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Construction: Choose $m < n$.

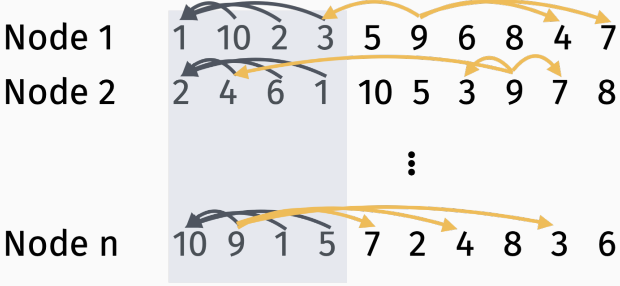
1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Construction: Choose $m < n$.

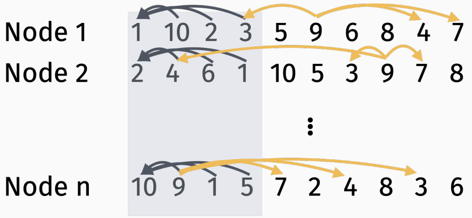
1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.

1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m}\log n)$ uniformly random out-edges to every node.



Analysis: All constraints in the left shaded region are trivially satisfied.

1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m}\log n)$ uniformly random out-edges to every node.



Analysis: Consider any node outside the left region. The probability a random edge points left is at least $\frac{m}{n}$.

RIGHT REGION

1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m}\log n)$ uniformly random out-edges to every node.

Analysis: Consider any node outside the left region. The probability a random edge points left is at least $\frac{m}{n}$. So the probability that none point left from a given i is:

$$\leq \left(1 - \frac{m}{n}\right)^{O(\frac{n}{m}\log n)} \leq \frac{1}{n^c}.$$

Construction: Choose $m < n$.

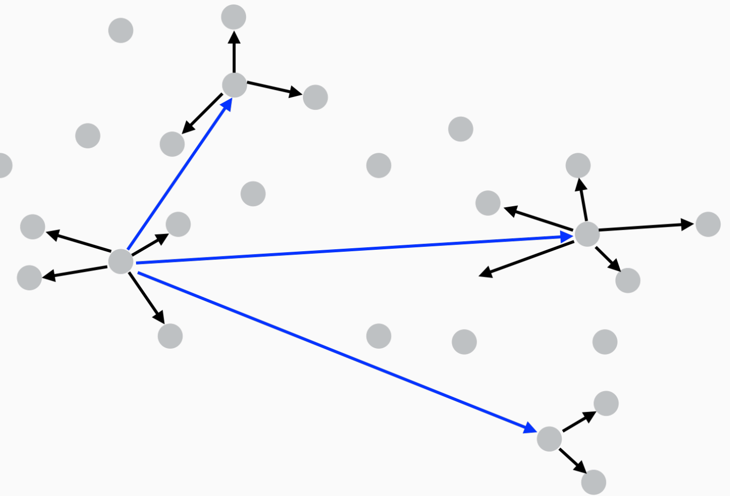
1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m}\log n)$ uniformly random out-edges to every node.

Total number of edges: $nm + n \cdot O\left(\frac{n}{m}\log n\right)$.

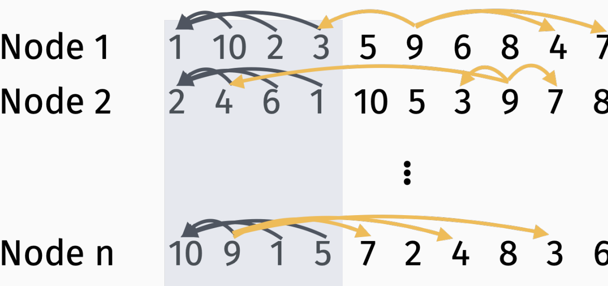
Minimize by setting $m = O\left(\sqrt{n\log n}\right)$.

Average out-degree: $O(\sqrt{n\log n})$.

Ultimately, a natural construction: combines "short distance" nearest-neighbor edges and "long distance" random edges.



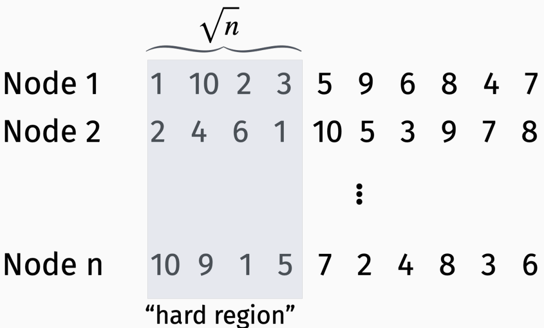
Average out-degree: $O(\sqrt{n \log n})$.

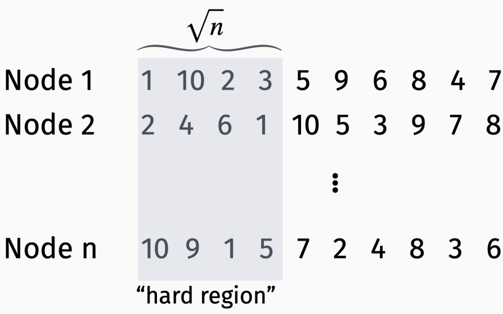


Moreover, the constructed graph has “depth” 2: greedy search will converge in 2 steps for any query in the dataset.

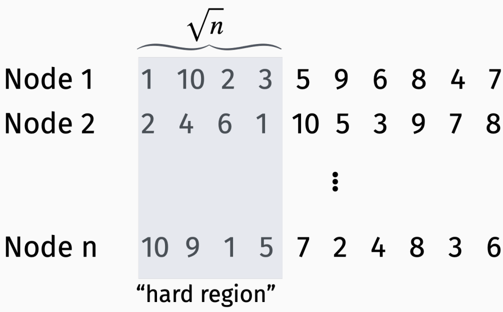
Claim (Nearly Matching Lower Bound)

Let $x_1, \ldots, x_n$ *be random vectors in* $\{-1, 1\}^m$ *where* $m = O(\log n)$*. With high probability, any navigable graph for* $x_1, \ldots, x_n$ *under Euclidean distance requires average out-degree* $\Omega(n^{1/2-\epsilon})$ *for any fixed constant* ϵ*.*



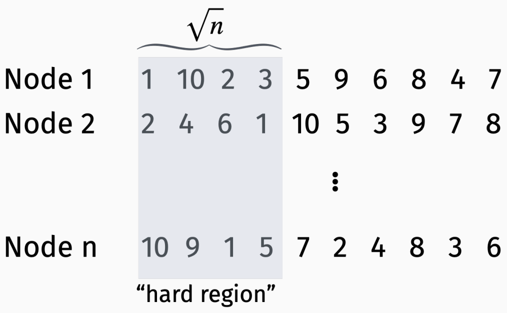


Core idea: For random points in $O(\log n)$ dimensions, the $\sqrt{n}$ closest points for each $\mathbf{x}_i$ are "close" to i.i.d. uniform random.

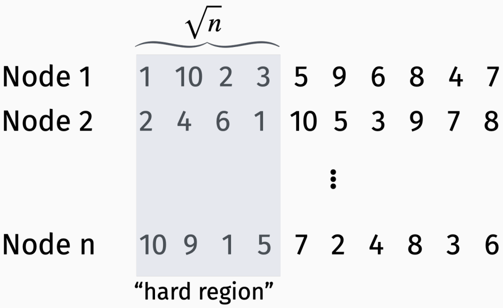


Core idea: For random points in $O(\log n)$ dimensions, the $\sqrt{n}$ closest points for each $\mathbf{x}_i$ are "close" to i.i.d. uniform random.

Consequence: For any j, k, the probability that <u>both</u> j and k appear in the hard region for row i is $\lesssim 1/n$.



With high probability, no pair (j, k) appears together in the hard region for more than $O(\log n)$ different rows.

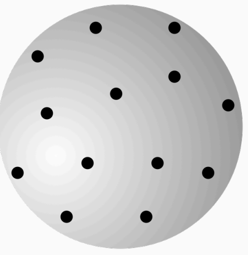


With high probability, no pair (j, k) appears together in the hard region for more than $O(\log n)$ different rows.

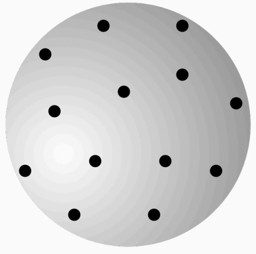
An edge from $j \to k$ can thus only cover $O(\log n)$ "hard" constraints. But there are $O(n^{3/2})$ total to cover.

So we need $\sim O(n^{3/2}/\log n)$ edges.

Takeaway: $\Theta(n^{3/2})$ edges are sufficient to construct a navigable graph, and necessary in the worst case.



Takeaway: $\Theta(n^{3/2})$ edges are sufficient to construct a navigable graph, and necessary in the worst case.



However, some datasets might admit sparser navigable graphs.

Natural algorithmic question: How quickly can we construct the sparsest navigable graph for a given dataset?

CONSTRUCTING SPARSE NAVIGABLE GRAPHS

Theorem (CDFJLMSSW, SODA 2026)

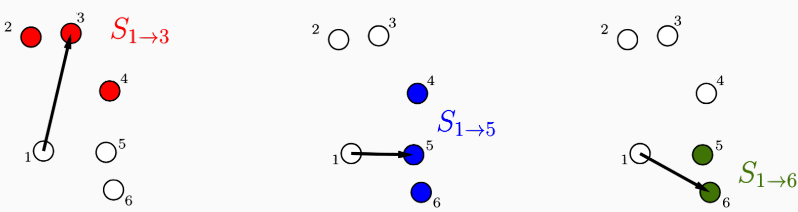
For any dataset and any distance function, we can construct a navigable graph with at most $O(\log n)$ times as many edges as the sparsest navigable graph in $\tilde{O}(n^2)$ time.

- $O(n^2)$ time is optimal even for points in $O(\log n)$ dimensional Euclidean space assuming the Strong Exponential Time Hypothesis.
- Obtaining better than $O(\log n)$ approximation is NP-hard.
- Similar results obtained by [Khanna, Padaki, Waingarten, SODA 2026].

Navigable graph construction is just n different minimum set cover problems!

Definition (Navigable Graph)

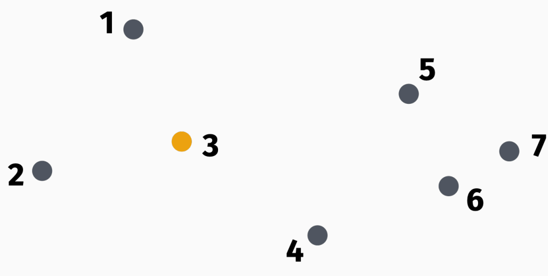
A graph G is navigable if, for all nodes i, for all $j \neq i$, there is some $k \in \mathcal{N}(i)$ (i's out neighborhood) satisfying $d(j,k) < d(j,i)$.



One problem for each node i. One set for all possible out neighbors. Elements to cover are all $j \neq i$.

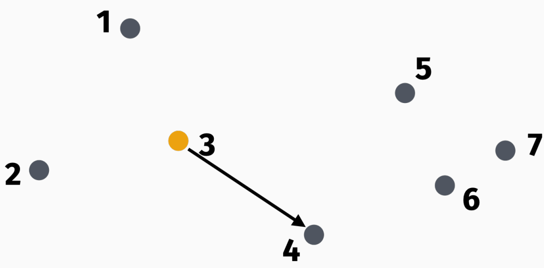
Baseline: Explicitly write down each set cover problem, which takes $n \times O(n^2)$ time.

Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



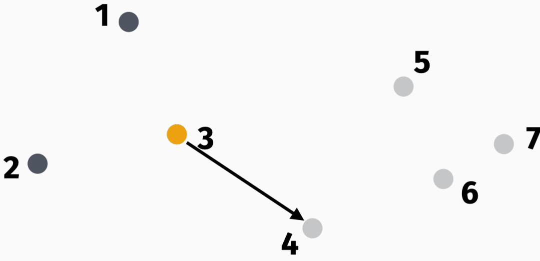
Baseline: Explicitly write down each set cover problem, which takes $n \times O(n^2)$ time.

Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



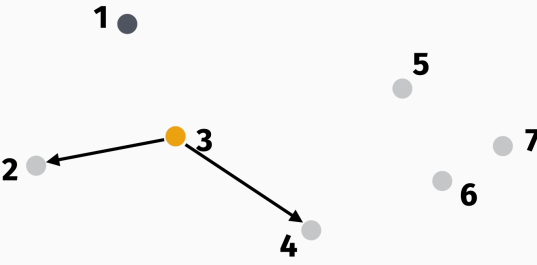
Baseline: Explicitly write down each set cover problem, which takes $n \times O(n^2)$ time.

Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



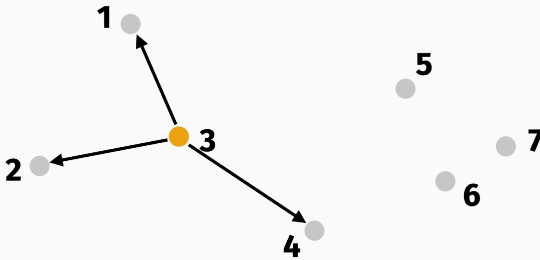
Baseline: Explicitly write down each set cover problem, which takes $n \times O(n^2)$ time.

Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



Baseline: Explicitly write down each set cover problem, which takes $n \times O(n^2)$ time.

Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



How can we beat cubic time?

Key Observation: In $O(n^2 \log n)$ time, we can implement an oracle to access information about the set cover instances, without writing them down explicitly.

How can we beat cubic time?

Key Observation: In $O(n^2 \log n)$ time, we can implement an oracle to access information about the set cover instances, without writing them down explicitly.

Simply construct the Distance-based Permutation Matrix:

Node 1	1	10	2	3	5	9	6	8	4	7
Node 2	2	4	6	1	10	5	3	9	7	8
					⋮					
Node n	10	9	1	5	7	2	4	8	3	6

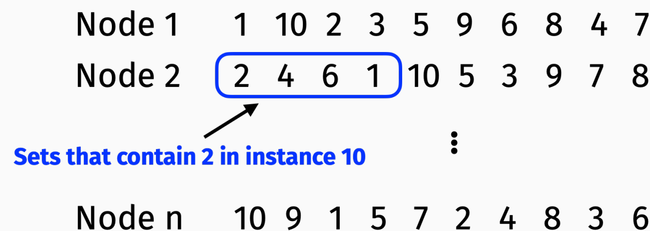
Concretely, after $O(n^2 \log n)$ time preprocessing, can implement **Membership** and **SetOf** queries in $O(1)$ time.

Node 1	1	10	2	3	5	9	6	8	4	7
Node 2	2	4	6	1	10	5	3	9	7	8
⋮										
Node n	10	9	1	5	7	2	4	8	3	6

j = 2, k = 4, i= 10

- **Membership:** Is j in set k in instance i? I.e., is $d(j,k) < d(j,i)$?
- **SetOf:** What is the kth set containing j in instance i?

Concretely, after $O(n^2 \log n)$ time preprocessing, can implement **Membership** and **SetOf** queries in $O(1)$ time.



- **Membership:** Is j in set k in instance i? I.e., is $d(j,k) < d(j,i)$?
- **SetOf:** What is the kth set containing j in instance i?

Immediately let's us leverage prior work on sublinear time set cover algorithms! [Indyk, Mahabadi, Rubinfeld, Vakilian, Yodpinyanee, SODA 2018], [Koufogiannaki, Young 2014].

Let $OPT = \sum_{i=1}^{n} OPT_i$ be the size of the sparsest navigable graph. These results give an $O(\log n)$ approximation in time:

$$\tilde{O}(n \cdot OPT) \leq \tilde{O}(n^{2.5}).$$

Immediately let's us leverage prior work on sublinear time set cover algorithms! [Indyk, Mahabadi, Rubinfeld, Vakilian, Yodpinyanee, SODA 2018], [Koufogiannaki, Young 2014].

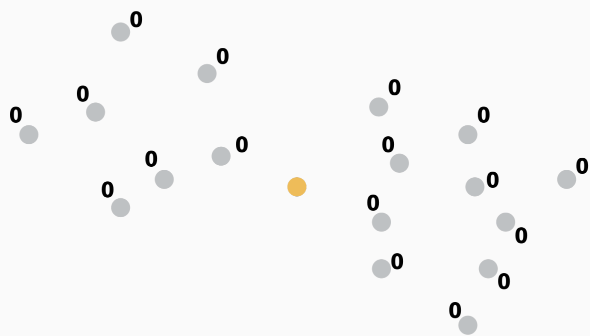
Let $OPT = \sum_{i=1}^{n} OPT_i$ be the size of the sparsest navigable graph. These results give an $O(\log n)$ approximation in time:

$$\tilde{O}(n \cdot OPT) \leq \tilde{O}(n^{2.5}).$$

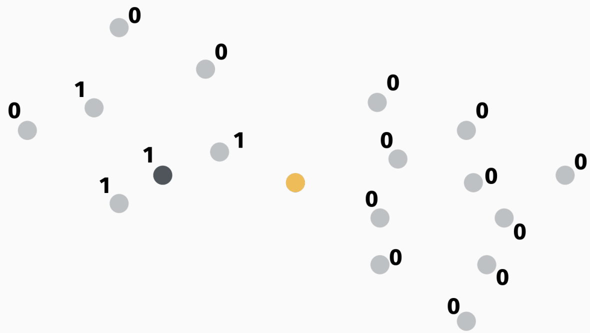
To reduce this to $\tilde{O}(n^2)$ we introduced an alternative approach to solving set cover problems in sublinear time.

Main idea: Directly simulate the greedy set cover algorithm.

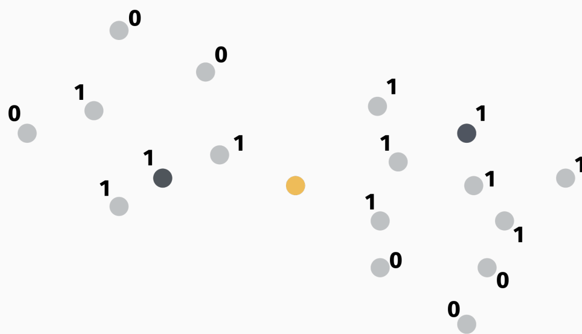
Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



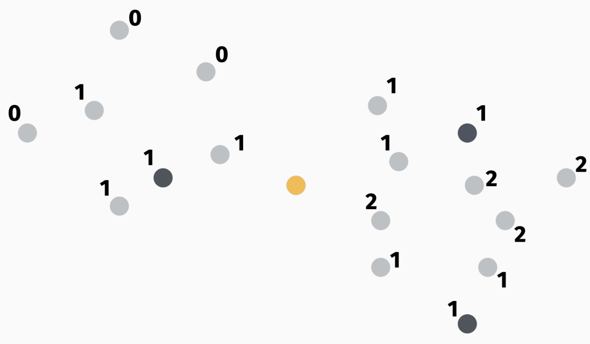
Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



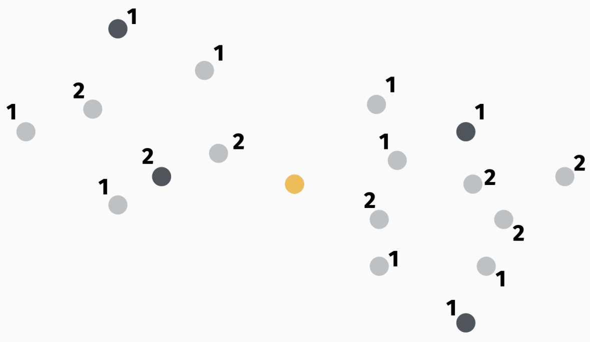
Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



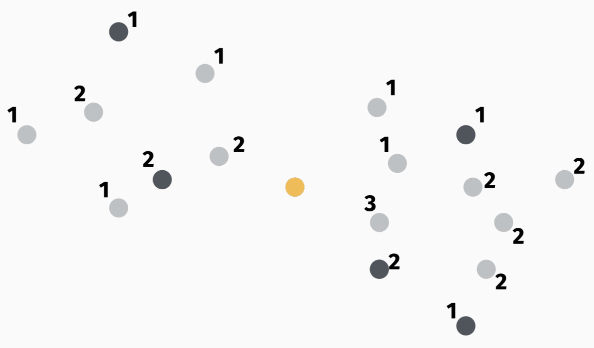
Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



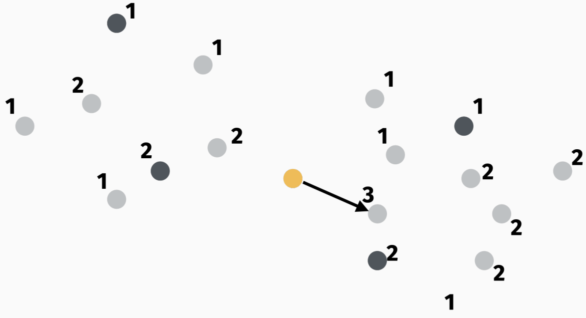
Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.

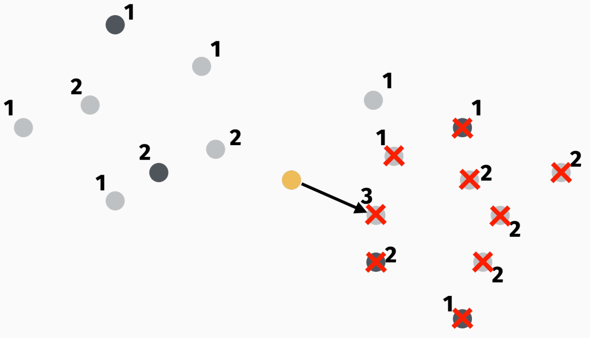


Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



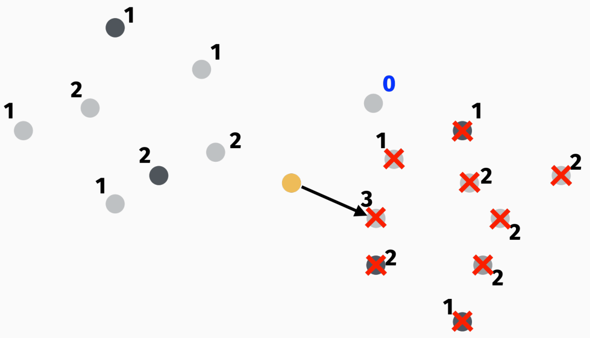
Once a node receives $O(\log n)$ votes, with high probability it covers a near maximal number of other nodes. Add to cover.

Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



Once a set is selected, need to remove all nodes covered by it and undo votes from any covered voters.

Sample uncovered nodes ("voters"). For each set (potential out-neighbor), maintain count of how many voters it covers.



Once a set is selected, need to remove all nodes covered by it and undo votes from any covered voters.

It is easy to show that, up to constant factors, greedy simulation matches the $O(\log n)$ approximation guarantees of standard greedy set cover. Bounding runtime is trickier.

Dominant costs:

1. **Voting.** When we sample j as a voter, need to increase counter for all sets it is contained in. **Naively $O(n)$ time.**
2. **Element removal.** Every time a set is selected, need to remove all covered nodes. **Naively $O(n)$ time.**

It is easy to show that, up to constant factors, greedy simulation matches the $O(\log n)$ approximation guarantees of standard greedy set cover. Bounding runtime is trickier.

Dominant costs:

1. **Voting.** When we sample j as a voter, need to increase counter for all sets it is contained in. **Naively $O(n)$ time.**
2. **Element removal.** Every time a set is selected, need to remove all covered nodes. **Naively $O(n)$ time.**

Let OPT_i be the optimal degree for node i and $OPT = \sum_{i=1}^{n} OPT_i$.

We select $O(OPT_i \cdot \log n)$ sets based on $O(OPT_i \cdot \log^2 n)$ voters.

It is easy to show that, up to constant factors, greedy simulation matches the $O(\log n)$ approximation guarantees of standard greedy set cover. Bounding runtime is trickier.

Dominant costs:

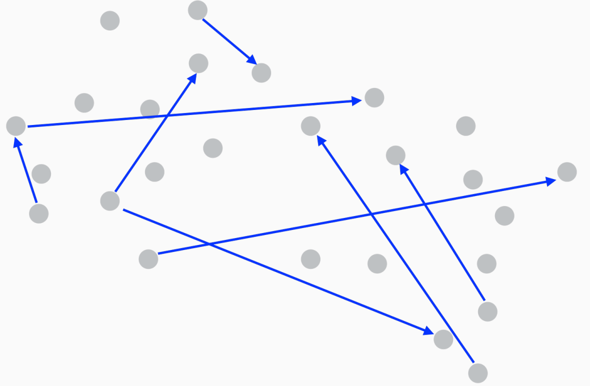
1. **Voting.** When we sample j as a voter, need to increase counter for all sets it is contained in. **Naively $O(n)$ time.**
2. **Element removal.** Every time a set is selected, need to remove all covered nodes. **Naively $O(n)$ time.**

Let OPT_i be the optimal degree for node i and $OPT = \sum_{i=1}^{n} OPT_i$.

We select $O(OPT_i \cdot \log n)$ sets based on $O(OPT_i \cdot \log^2 n)$ voters.

Runtime for node i: $\tilde{O}(OPT_i \cdot n)$. **Total runtime: $\tilde{O}(OPT \cdot n)$.**

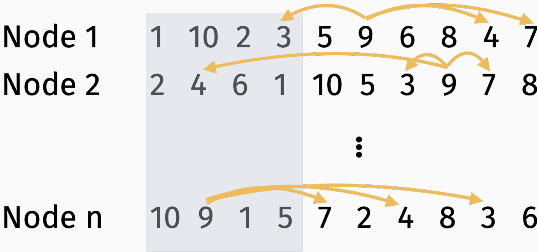
We can reduce both main costs by first adding random edges, just as we did in our $O(\sqrt{n \log n})$ construction!



Can afford to add $\tilde{O}(OPT/n)$ random out-edges to every node.

Claim: After adding $\tilde{O}(OPT/n)$ random out-edges to every node, we have two properties:

1. Only elements contained in $< n/(OPT/n) = n^2/OPT$ sets are left uncovered. **Makes voting cheaper**.
2. Only n^3/OPT total elements are left uncovered across all instances. **Makes element removal cheaper**.



VOTING COST

Assume that $OPT_i = O(OPT/n)$ for all i. For node i, we sample $\tilde{O}(OPT_i)$ voters. **Total voting cost**:

$$\underbrace{\tilde{O}(OPT_i)}_{\text{\# of voters}} \cdot \underbrace{(n^2/OPT)}_{\text{cost to increment votes}}$$

VOTING COST

Assume that $OPT_i = O(OPT/n)$ for all i. For node i, we sample $\tilde{O}(OPT_i)$ voters. **Total voting cost**:

$$\underbrace{\tilde{O}(OPT_i)}_{\text{\# of voters}} \cdot \underbrace{(n^2/OPT)}_{\text{cost to increment votes}} = \tilde{O}(OPT_i) \cdot O(n/OPT_i) = \tilde{O}(n).$$

Assume that $OPT_i = O(OPT/n)$ for all i. For node i, we sample $\tilde{O}(OPT_i)$ voters. **Total voting cost**:

$$\underbrace{\tilde{O}(OPT_i)}_{\text{\# of voters}} \cdot \underbrace{(n^2/OPT)}_{\text{cost to increment votes}} = \tilde{O}(OPT_i) \cdot O(n/OPT_i) = \tilde{O}(n).$$

Argument for bounding element removal cost is not much harder. Average cost $\tilde{O}(n)$ per node.

Final runtime: $\tilde{O}(n^2)$.

VOTING COST

Assume that $OPT_i = O(OPT/n)$ for all i. For node i, we sample $\tilde{O}(OPT_i)$ voters. **Total voting cost**:

$$\underbrace{\tilde{O}(OPT_i)}_{\text{\# of voters}} \cdot \underbrace{(n^2/OPT)}_{\text{cost to increment votes}} = \tilde{O}(OPT_i) \cdot O(n/OPT_i) = \tilde{O}(n).$$

Argument for bounding element removal cost is not much harder. Average cost $\tilde{O}(n)$ per node.

Final runtime: $\tilde{O}(n^2)$.

Dealing with case when OPT_i varies widely across nodes is tricky. Requires an alternative preprocessing step.

Recap:

1. Every dataset admits a navigable graph with average degree $O(\sqrt{n})$.
2. We can find a near-optimally sparse navigable graph for any dataset in $\tilde{O}(n^2)$ time.

Where does this leave us in terms of understanding graph-based search?

Still don't have sublinear query time bounds or strong approximation guarantees! There are **two main barriers.**

1. Greedy search on a navigable graph fails to provide any meaningful approximation for general queries, q (that are not exactly in our dataset).

Recall that, if currently at node i, greedy search moves to $j^* = \arg\min_{j \in \mathcal{N}(i)} d(j, q)$, or terminates if $d(j^*, q) > d(i, q)$.

Navigability: Requires that, for all j, i has an out-neighbor k such that:

$$d(j,k) < d(j,i).$$

α-Reachability (stronger): Requires that, for all j, i has an out-neighbor k such that:

$$d(j,k) < \frac{1}{\alpha} d(j,i) \text{ for } \alpha \geq 1.$$

[Indyk, Xu 2023] shows that α-shortcut reachability implies that greedy search returns an $\frac{\alpha+1}{\alpha-1}$-approximate nearest neighbor for any query.

α SHORTCUT REACHABILITY

Navigability: Requires that, for all j, i has an out-neighbor k such that:

$$d(j,k) < d(j,i).$$

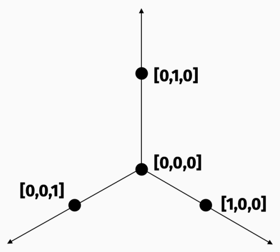
α-Reachability (stronger): Requires that, for all j, i has an out-neighbor k such that:

$$d(j,k) < \frac{1}{\alpha} d(j,i) \text{ for } \alpha \geq 1.$$

[Indyk, Xu 2023] shows that α-shortcut reachability implies that greedy search returns an $\frac{\alpha+1}{\alpha-1}$-approximate nearest neighbor for any query.

We give an $\tilde{O}(n^{2.5})$ algorithm for sparse α-shortcut reachable graph construction. [Khanna, Padaki, Waingarten, SODA 2026] give an $O(n^{\omega})$ time algorithm with a slightly weaker notation of approximation. **Can we get down to $\tilde{O}(n^2)$ time?**.

2. It is easy to construct datasets where any navigable graph must have <u>$n-1$ maximum degree</u>.



In a navigable graph, if j is i's closest neighbor, then there must be an edge from j to i. Even if graph is sparse <u>on average</u>, what if we route through a high-degree node?

Some options for addressing this issues:

- Assume the issue away :)
- Consider "backtracking" variations of greedy search, like **beam search**, that allow for better worst-case degree.
- Allow "steiner nodes" to be introduced.
- Consider hierarchical graph-based methods like HSNW.
- ??

Other questions:

- For real-world datasets, even $\tilde{O}(n^2)$ is far to slow. Can we design subquadratic time algorithms for an appropriately relaxed notion of navigability?
- What if we make assumptions about the dataset (e.g., low intrinsic dimension)?
- Can we design efficient algorithms for maintaing navigability under dynamic updates to the dataset?

QUESTIONS?