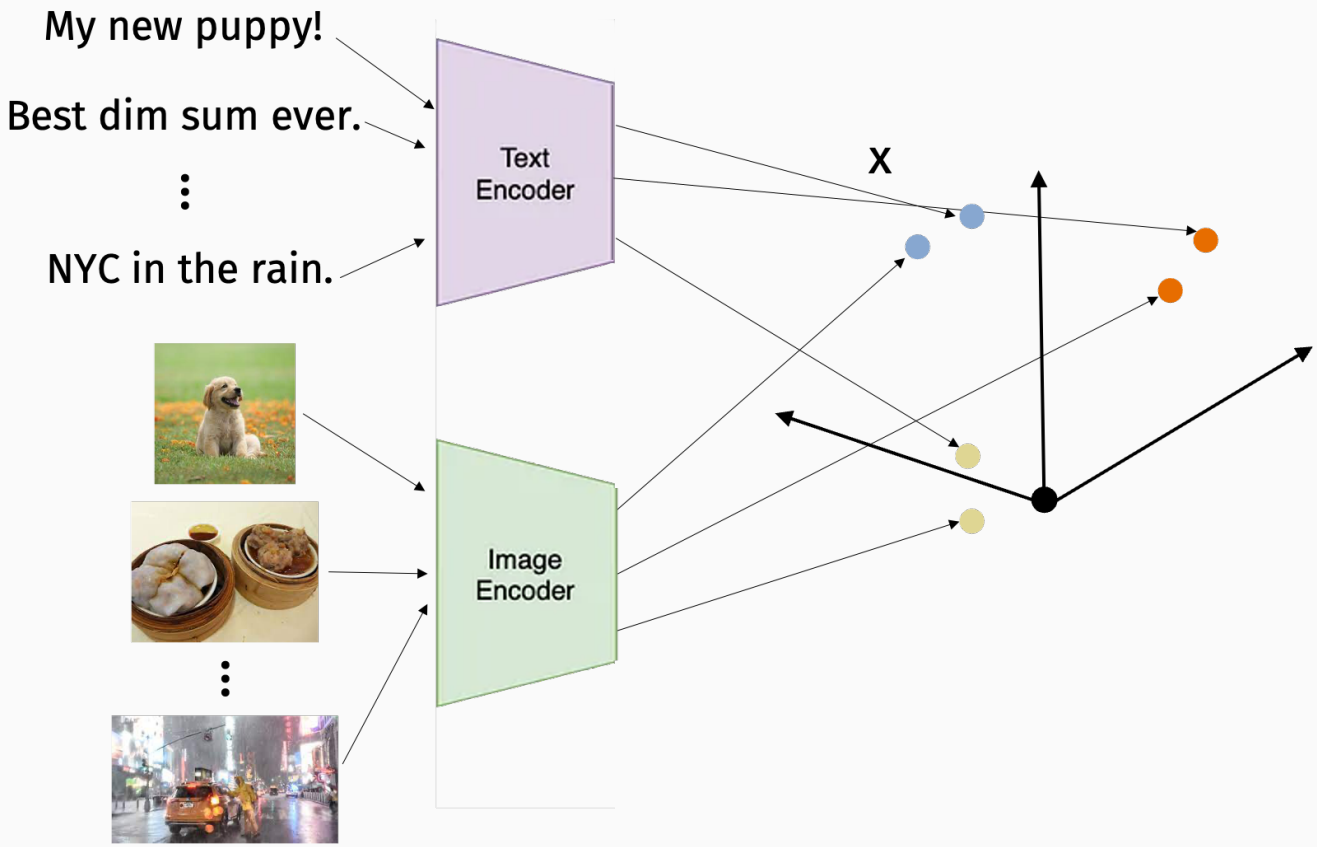
Navigability and Graph-based Vector Search

Christopher Musco, New York University

Based on collaborations with: Yousef Al-Jazzazi[1], Haya Diwan[1], Jinrui Gou[1], Cameron Musco[2], Torsten Suel[1], Alex Conway[3], Laxman Dhulipala[4], Martin Farach-Colton[1], Rob Johnson[5], Ben Landrum[3], Yarin Shechter[1], Richard Wen[4]

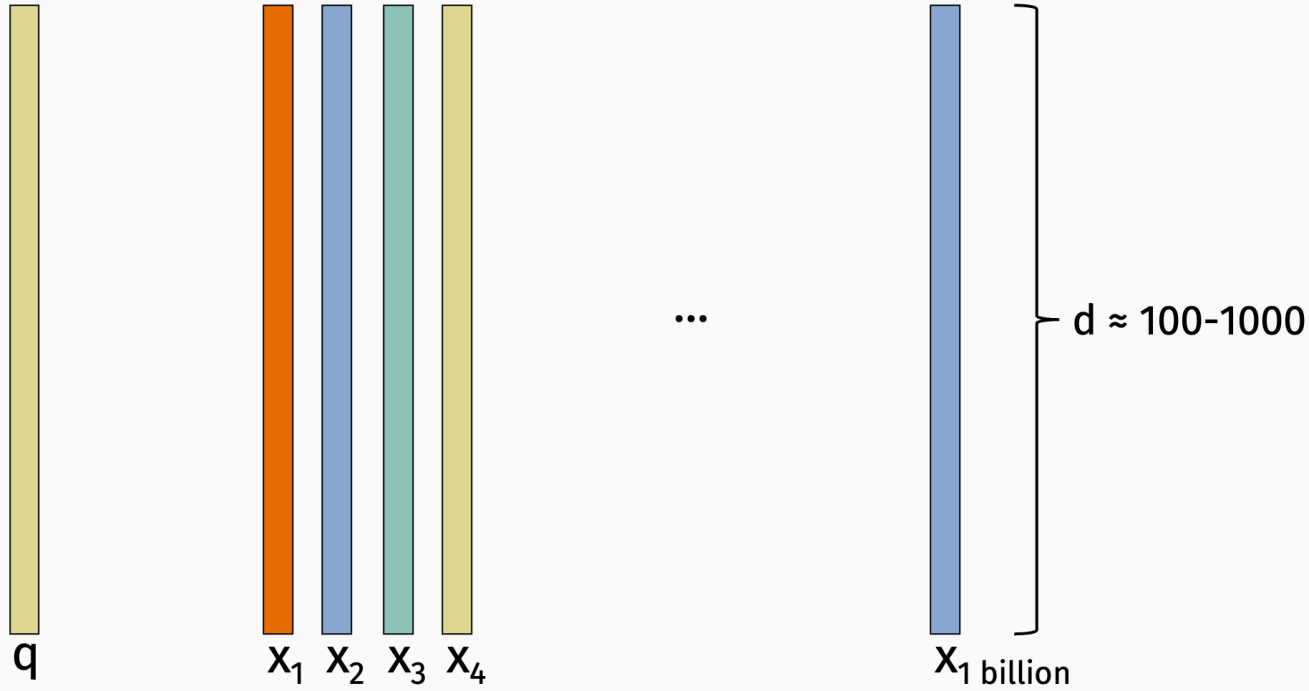
[1]NYU [2]UMass [3]Cornell [4]UMD [5]VMWare

MODERN PARADIGM FOR SEARCH



Use neural network (BERT, CLIP, etc.) to convert documents, images, and other media to high dimensional vectors. Matching results should have similar vector embeddings.

VECTOR SEARCH



Finding results for a query amounts to finding the closest k vectors in a vector database $\mathcal{X}$. E.g., for $k = 1$, return:

$$\arg\min_{\mathbf{x} \in \mathcal{X}} \|\mathbf{x} - \mathbf{q}\|.$$

Here $\mathbf{q}$ is the vector embedding of the query text.

VECTOR SEARCH IN PRACTICE

While a classic problem, there has been a recent surge of interest in nearest vector search + vector databases.

Beyond AI-based search, vector databases are a key technology behind Retrieval-Augmented Generation (RAG) systems for providing context (user emails, code snippets, documents, etc.) to language models.

What algorithms are these databases using for vector search?

Goal: Let $\mathcal{X}$ be a database of n vectors in $\mathbb{R}^d$. Find $\mathbf{x} \in \mathcal{X}$ minimizing $\|\mathbf{x} - \mathbf{q}\|$ for a query $\mathbf{q}$.

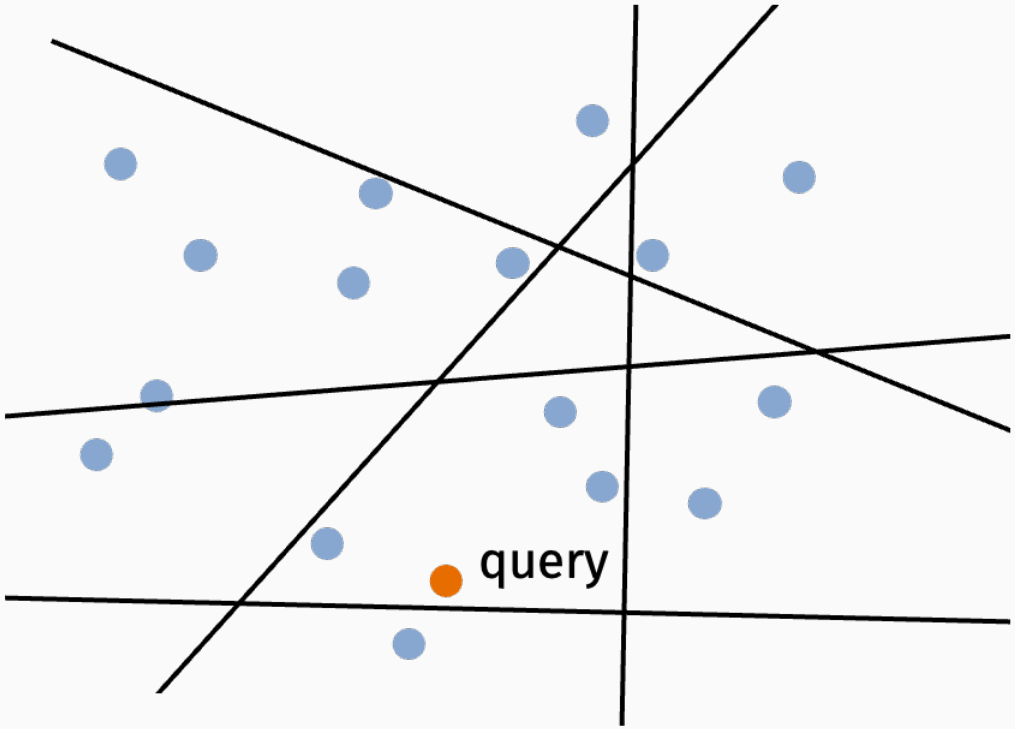
- A naive linear scan takes $O(nd)$ time.
- Ideally, we want to achieve $o(n)$ time.
- Willing to spend time preprocessing $\mathcal{X}$, possibly into a data structures that takes $\Omega(nd)$ space.

Obtaining an exact solutionis notoriously difficult outside of extremely low dimensions. Approximation algorithms have been studied since the late 80s and 90s, but their space or query complexity depends exponentially on the dimension d.

The “curse-of-dimensionality” was broken via the introduction of **Locality Sensitive Hashing [Indyk, Motwani, 1998]** .

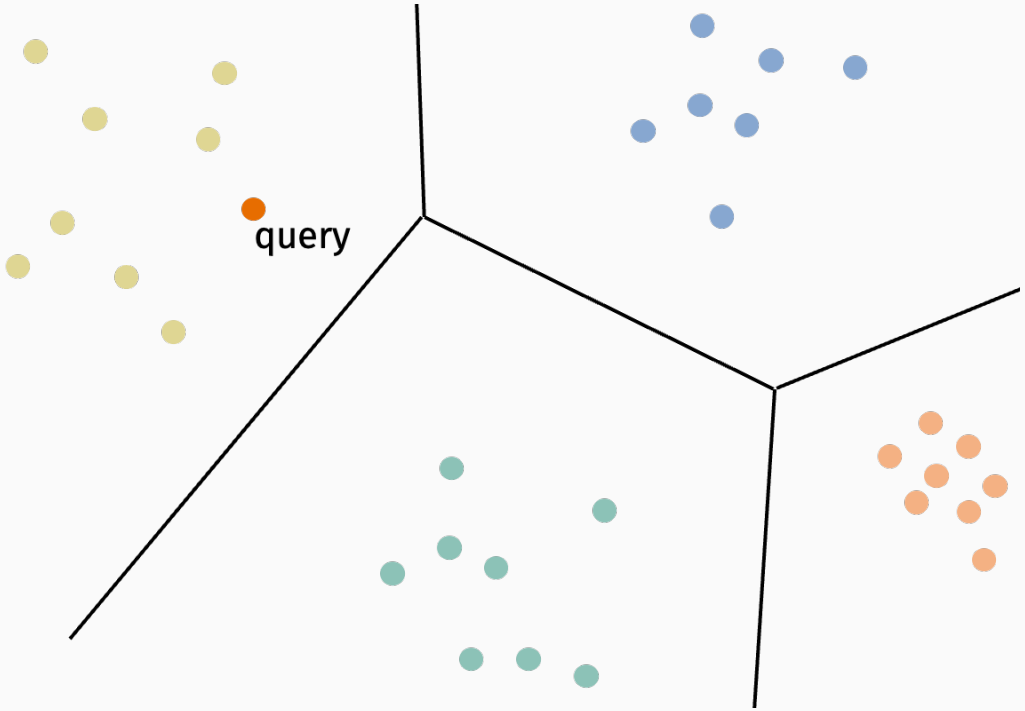
Rough idea behind LSH:

1. Pick a bunch of random hyperplanes.
2. Check which side of each hyperplane q lies on.
3. Return closest point that lies in the same region as q.
4. Repeat multiple times to avoid missing anything.



SPACE PARTITIONING METHODS

Can get better partitions by chosing hyperplanes in a data-dependent way, e.g. via clustering.



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

WORST-CASE GUARANTEES FOR LSH

Theorem (Andoni, Indyk, FOCS 2006)

For any $c \geq 1$, there is a data structure based on ***locality sensitive hashing*** *that, for any query $\mathbf{q}$, returns $\tilde{\mathbf{x}}$ satisfying:*

$$\|\tilde{\mathbf{x}} - \mathbf{q}\|_2 \leq c \cdot \min_{\mathbf{x} \in \mathcal{X}} \|\mathbf{x} - \mathbf{q}\|_2$$

and uses:

- *Query Time:* $\tilde{O}\left(dn^{1/c^2}\right)$.
- *Data Structure Space Complexity:* $\tilde{O}\left(nd + n^{1+1/c^2}\right)$.

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

LSH and related developments won Indyk, Charikar, and Broder the 2012 ACM Paris Kanellakis Theory and Practice

Few (none?) of these vector databases are using LSH or data-dependent space partitioning!

New kid on the block: Graph-based Search.

NEAREST-NEIGHBOR SEARCH IN PRACTICE

New(ish) kid on the block: Graph-based Search.

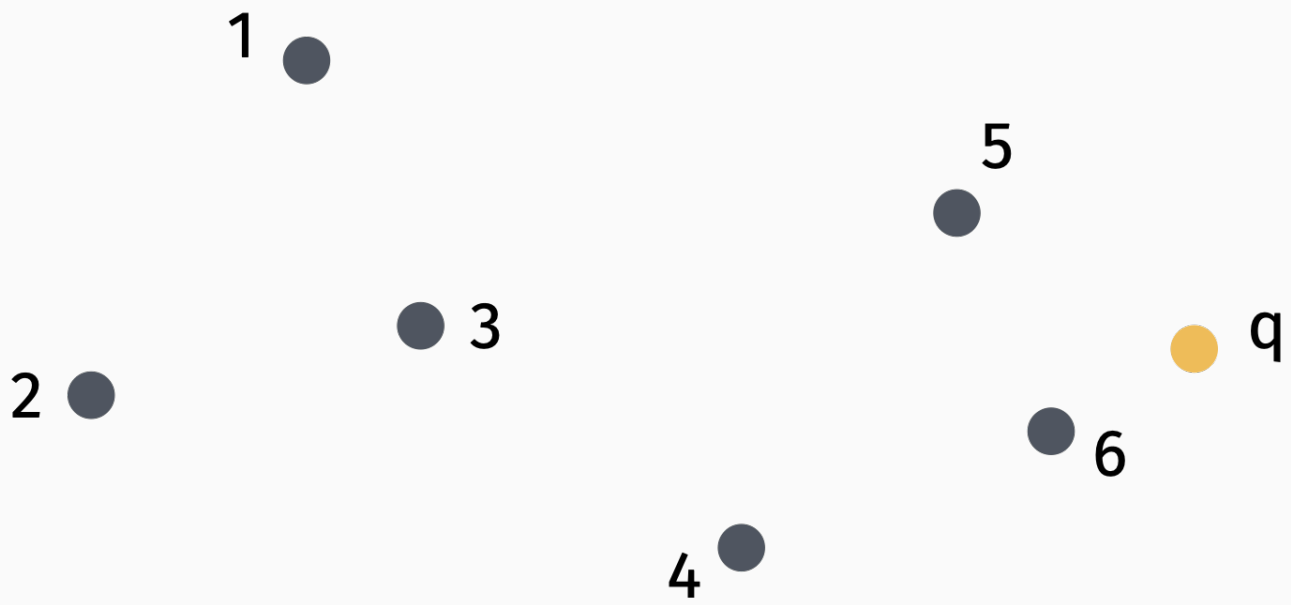
- **Navigating Spreading-out Graphs (NSG)** [Fu, Xiang, Wang, Cai, 2017]
- **Hierarchical Navigable Small World (HNSW)** [Malkov, Yashunin, 2018]
- **Microsoft's DiskANN** [Subramanya, Devvrit, Kadekodi, Krishaswamy, Simhadri 2019]

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

Connections to Milgram's famous "small world" experiments from the 1960s and later work on the small world phenomenon by Watts, Strogatz, Kleinberg, and others.

BASIC IDEA BEHIND GRAPH-BASED SEARCH

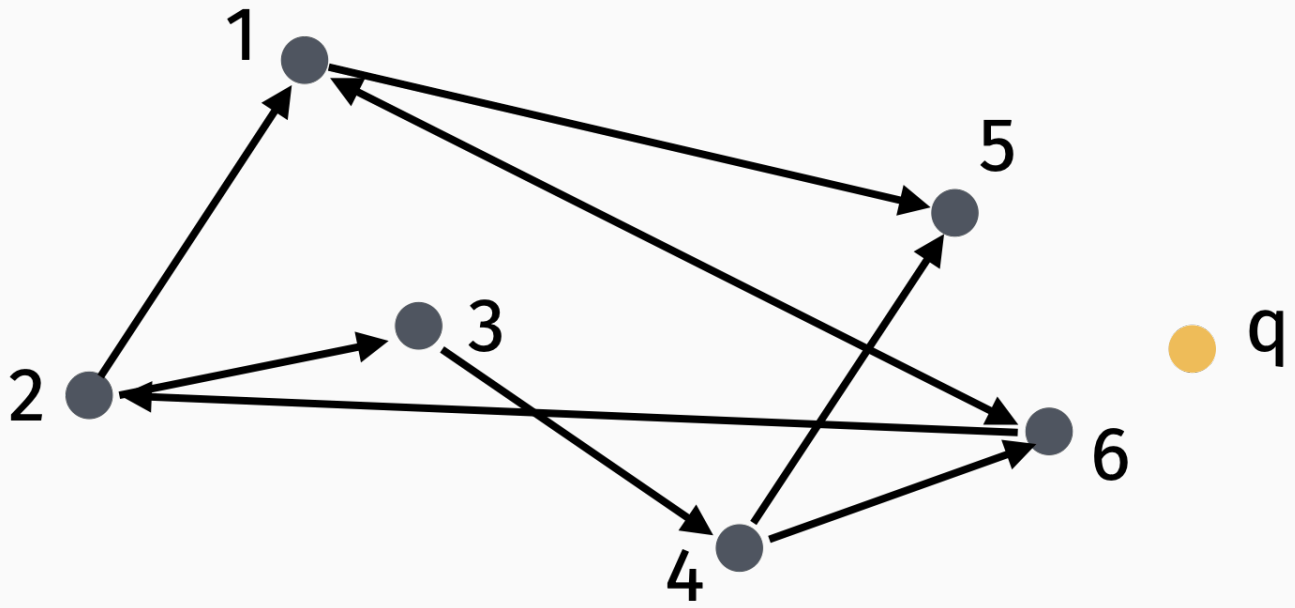
1. Construct a directed search graph over our dataset.



2. Run some variant of greedy search in the graph.

BASIC IDEA BEHIND GRAPH-BASED SEARCH

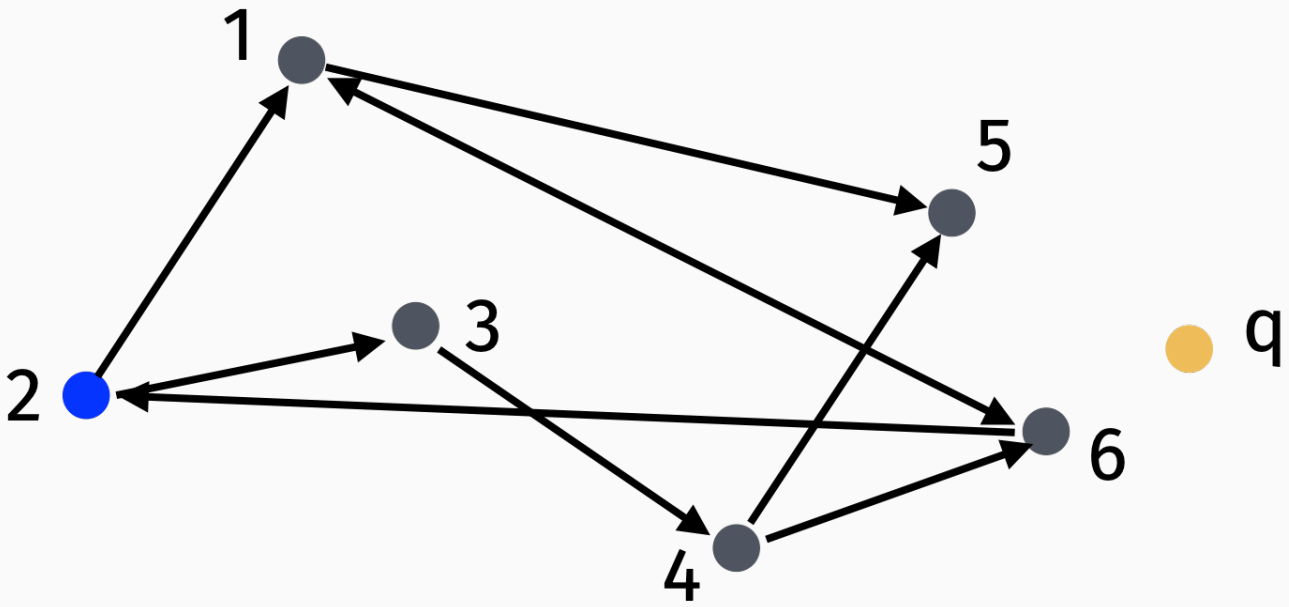
1. Construct a directed search graph over our dataset.



2. Run some variant of greedy search in the graph.

BASIC IDEA BEHIND GRAPH-BASED SEARCH

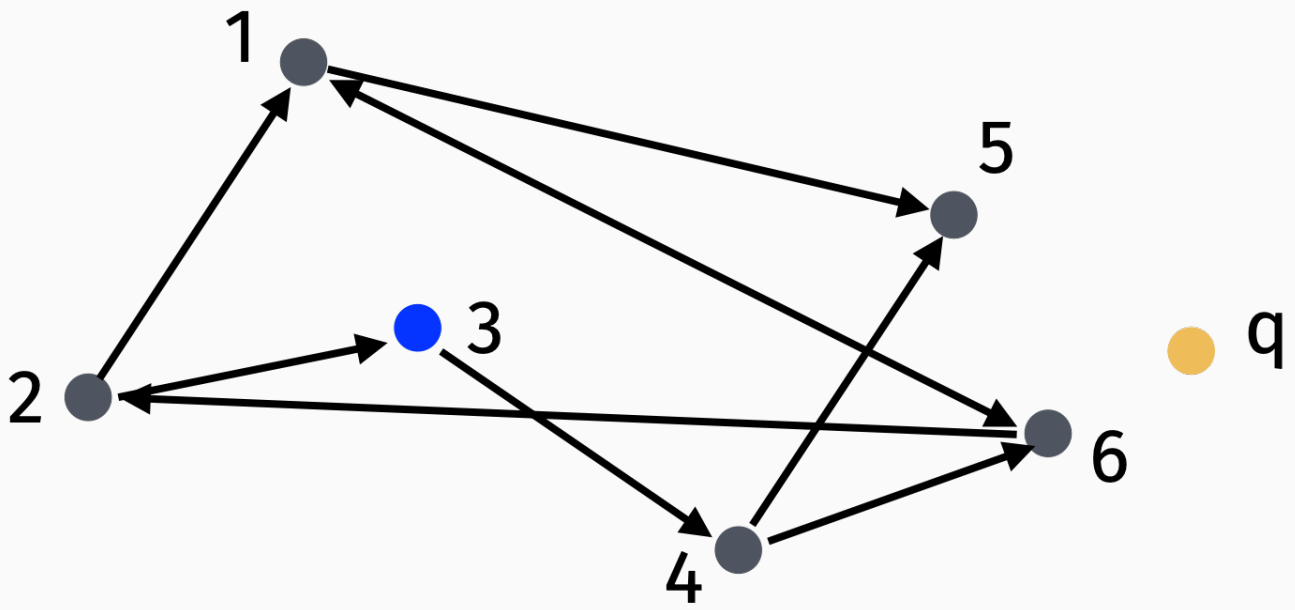
1. Construct a directed search graph over our dataset.



2. Run some variant of greedy search in the graph.

BASIC IDEA BEHIND GRAPH-BASED SEARCH

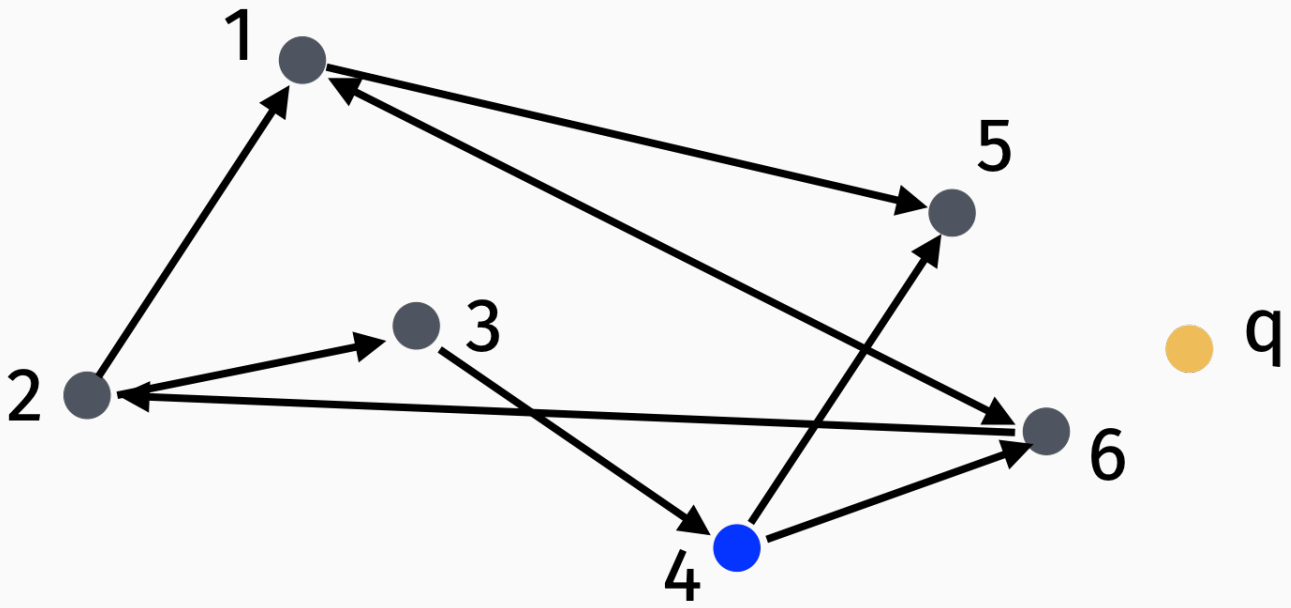
1. Construct a directed search graph over our dataset.



2. Run some variant of greedy search in the graph.

BASIC IDEA BEHIND GRAPH-BASED SEARCH

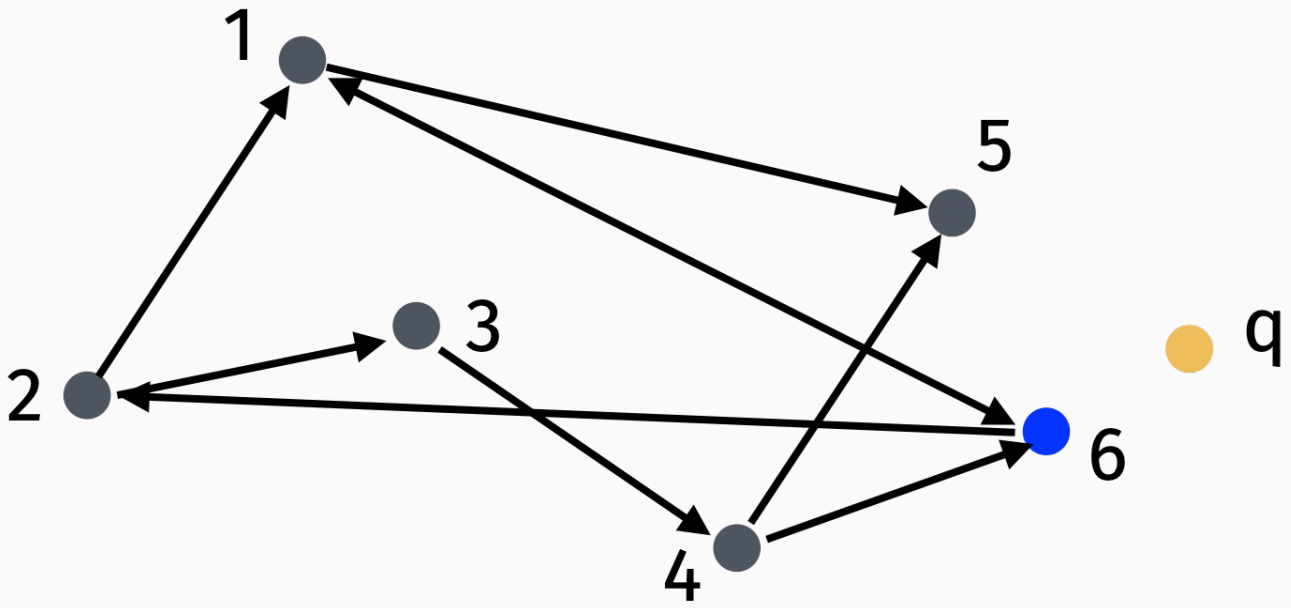
1. Construct a directed search graph over our dataset.



2. Run some variant of greedy search in the graph.

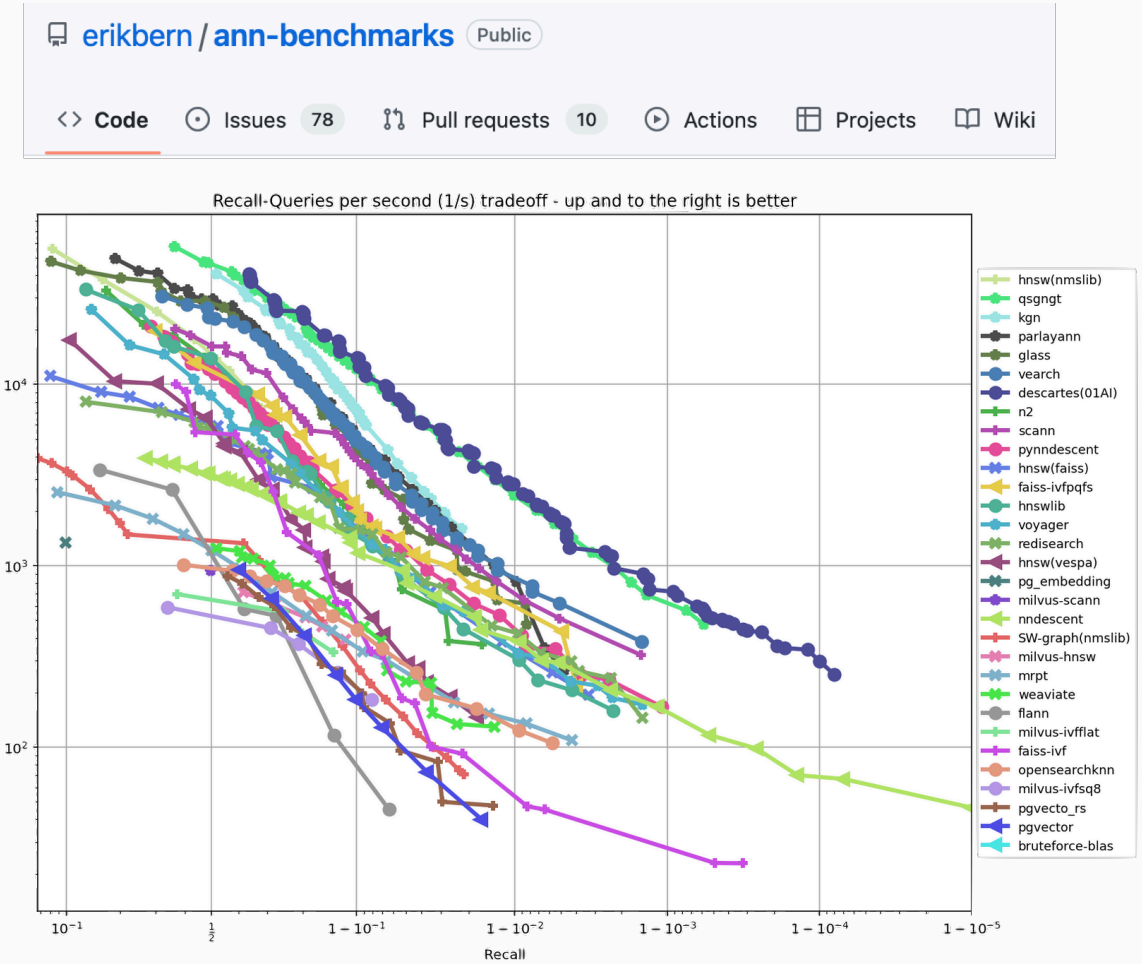
BASIC IDEA BEHIND GRAPH-BASED SEARCH

1. Construct a directed search graph over our dataset.



2. Run some variant of greedy search in the graph.

Graph-based methods are topping benchmarks and competitions!



GRAPH-BASED SEARCH IN PRACTICE

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

Open theory challenge: Can we explain the empirical success of graph-based vector search?

PLAN FOR TODAY

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 today:

- Discuss connections between the performance of graph-based search methods and the concept of graph navigability.
- Introduce recent results on the existence and construction of sparse navigable graphs (NeurIPS 2024, SODA 2026).
- 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 $\|\mathbf{x}_j - \mathbf{q}\|_2 \leq c \cdot \min_{i \in \{1,\ldots,n\}} \|\mathbf{x}_i - \mathbf{q}\|_2$ for some $c \geq 1$.

Standard guarantee for LSH methods, although people care about other metrics of success as well.

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.

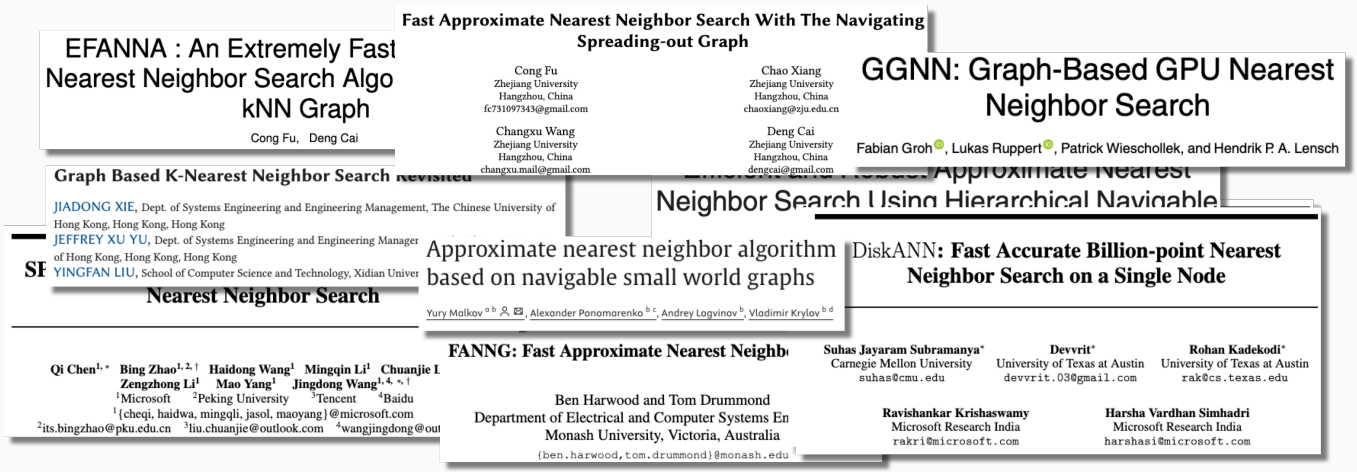
Search graph G should be chosen to at least ensure that we find q if it is in the dataset.

Ideally, G should also be sparse and require few steps to find q.

NAVIGABLE GRAPHS

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.



Among dozens of papers on search graph construction, 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 of these papers actually construct provably navigable graphs!

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

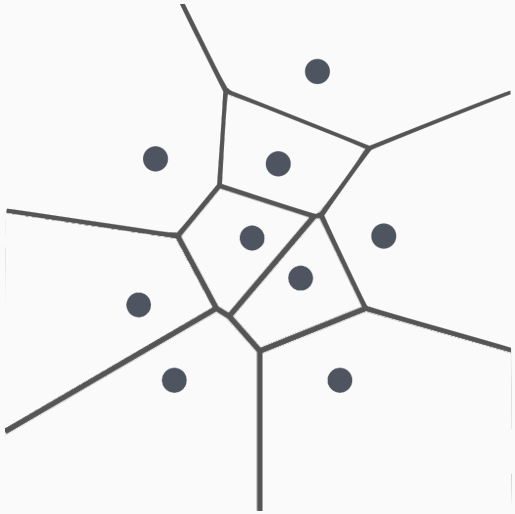
Navigability can be viewed as a minimum necessary condition for graph-based greedy search to succeed, so answering this question is critical to understanding graph-based methods.

Also a natural property in its own right. Exactly the property explored in Milgram's "small world" experiments: can greedy routing find paths between people in social networks?

SPARSE NAVIGABLE GRAPHS

Known results in low-dimensional Euclidean space:

- **2-dimensions:** The Delaunay graph can be proven to be navigable. This graph has average degree ≤ 3.

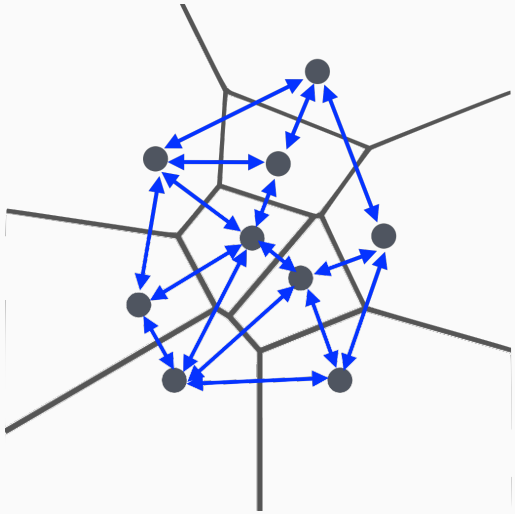


- Kleinberg studied how to further achieve low-diameter/"small world" property with a logarithmic number of edges per node.
- **d-dimensions:** The Sparse Neighborhod Graph of Arya and Mount [SODA, 1993] is navigable and has average degree $O(2^d)$.

SPARSE NAVIGABLE GRAPHS

Known results in low-dimensional Euclidean space:

- **2-dimensions:** The Delaunay graph can be proven to be navigable. This graph has average degree ≤ 3.



- Kleinberg studied how to further achieve low-diameter/"small world" property with a logarithmic number of edges per node.
- **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 for $\mathbf{x}_1, \ldots, \mathbf{x}_n$ requires average out-degree $\Omega(n^{1/2-\epsilon})$ for any fixed constant ϵ.

I will give you a proof sketch.

ALTERNATIVE DEFINITION

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 $k \in \mathcal{N}(i)$ (i's out neighborhood) satisfying:

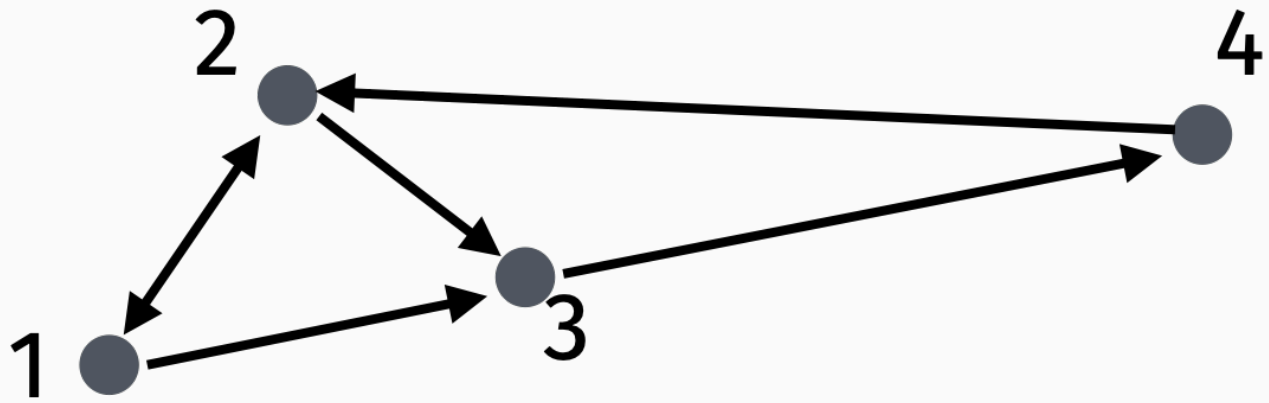
$$d(j, k) < d(j, i).$$

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 $j \neq i$, there is some $k \in \mathcal{N}(i)$ (i's out neighborhood) satisfying:

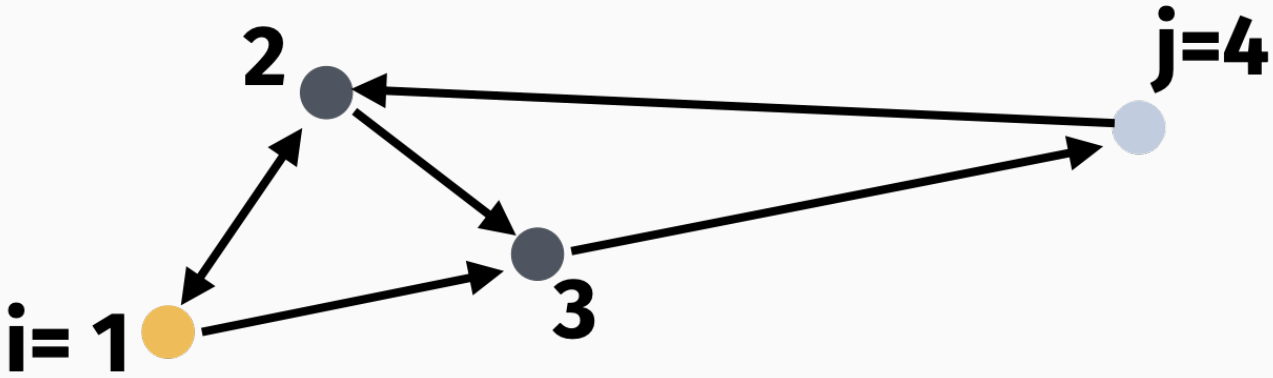
$$d(j, k) < d(j, i).$$



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 $j \neq i$, there is some $k \in \mathcal{N}(i)$ (i's out neighborhood) satisfying:

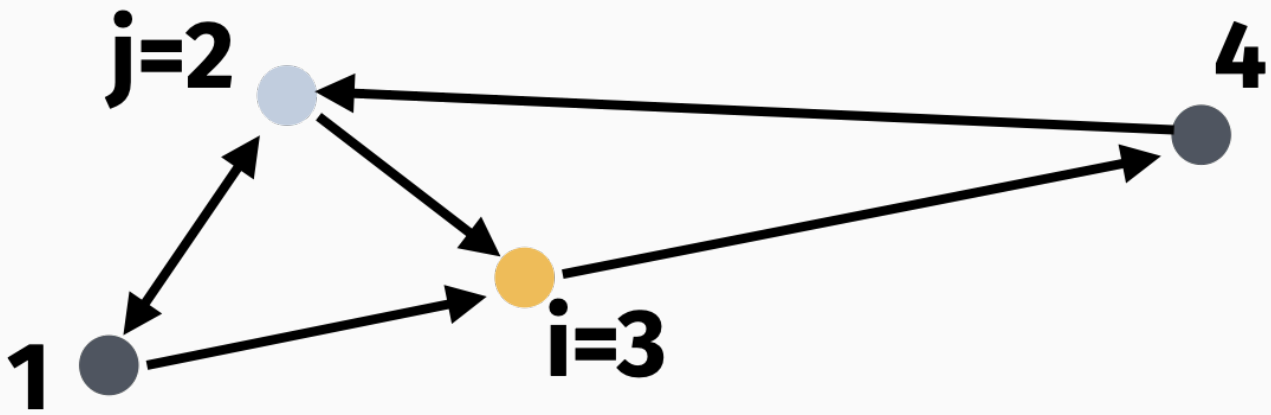
$$d(j, k) < d(j, i).$$



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 $j \neq i$, there is some $k \in \mathcal{N}(i)$ (i's out neighborhood) satisfying:

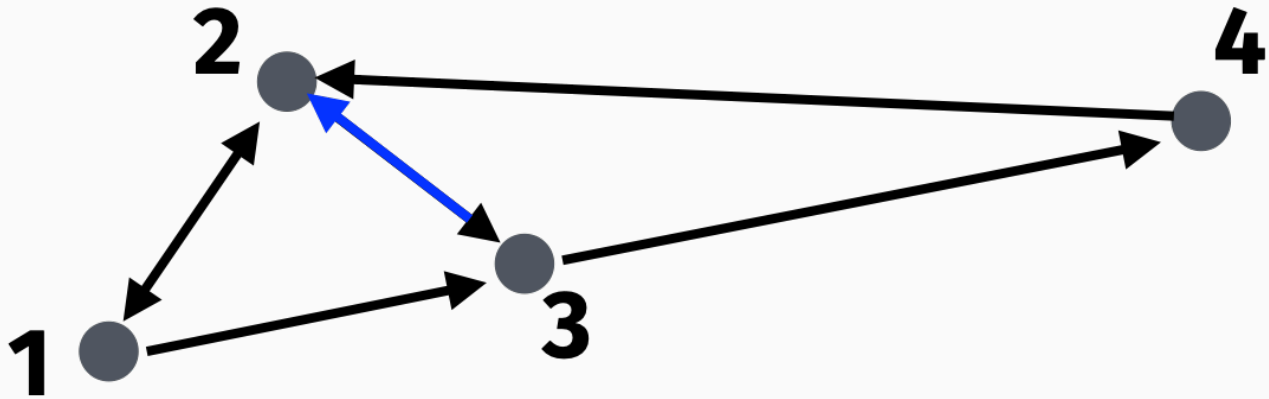
$$d(j, k) < d(j, i).$$



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 $j \neq i$, there is some $k \in \mathcal{N}(i)$ (i's out neighborhood) satisfying:

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



NAVIGABLE GRAPH CONSTRUCTION AS SET COVER

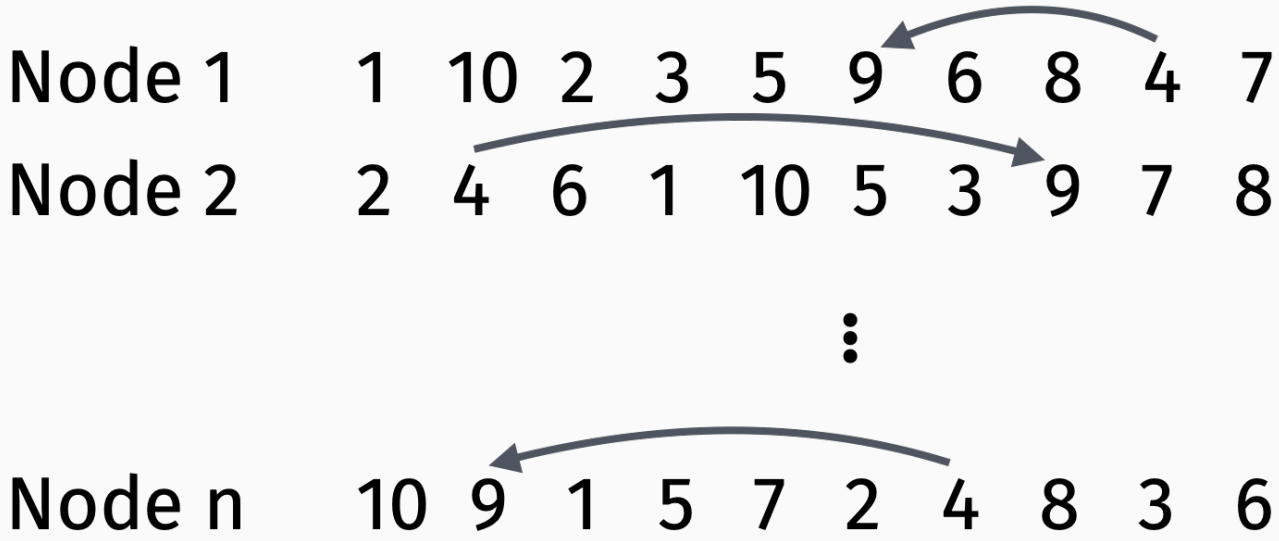
Imagine a **Distance-Based Permutation Matrix**, where the i^{th} row lists all points sorted in order of their distance to point i.

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

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

NAVIGABLE GRAPH CONSTRUCTION AS SET COVER

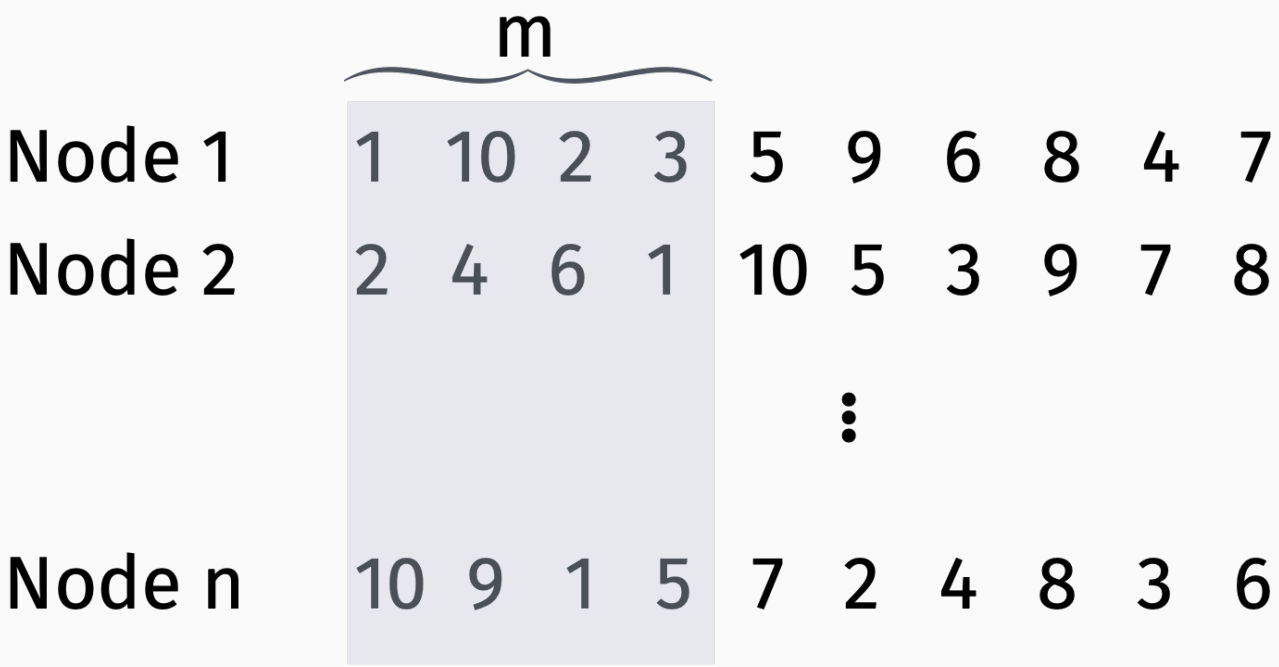
Imagine a **Distance-Based Permutation Matrix**, where the i^{th} row lists all points sorted in order of their distance to point i.



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

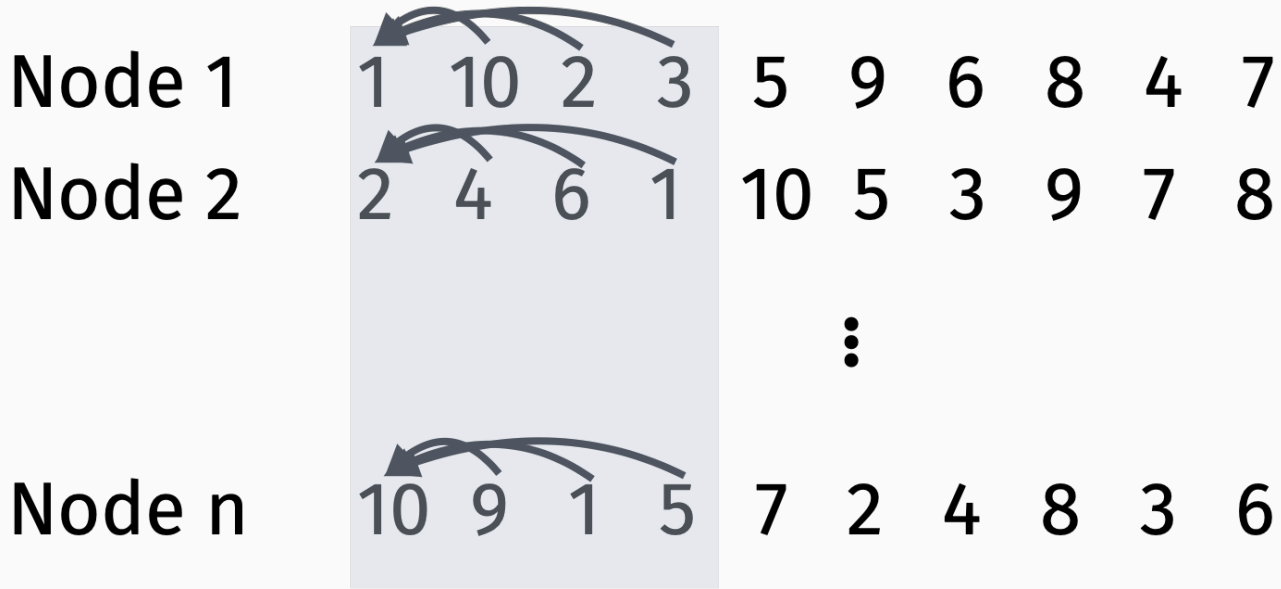
Construction: Choose $m < n$.

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 $m < n$.

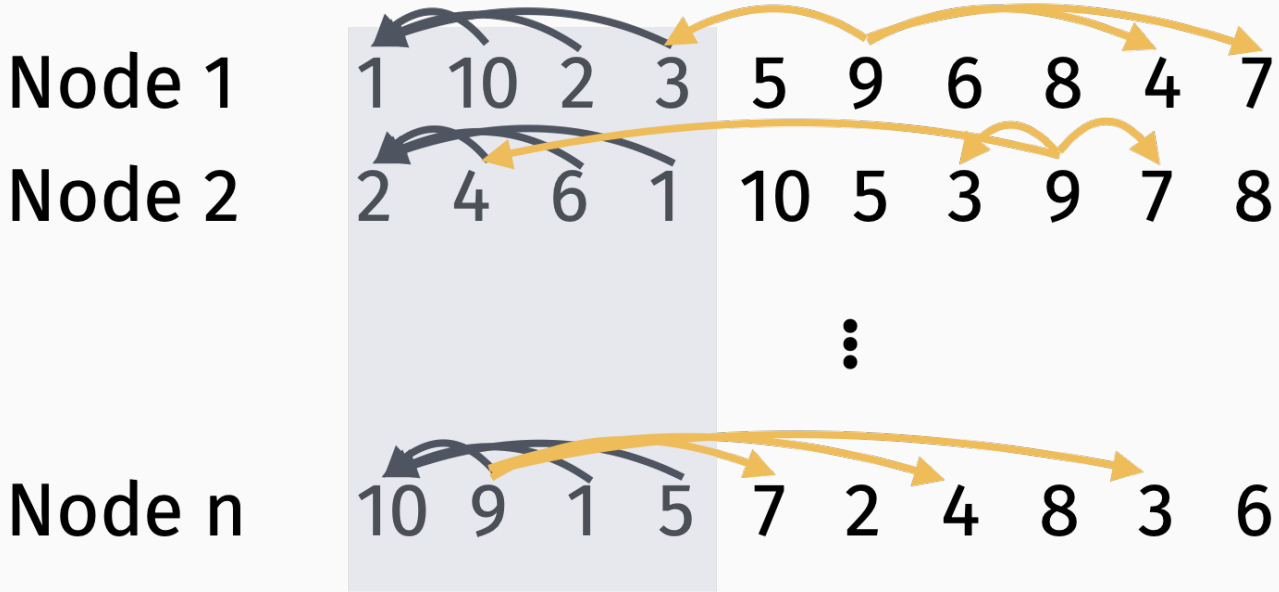
1. Add an edge from j to i if j is one of i's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



UPPER BOUND CONSTRUCTION

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.



UPPER BOUND CONSTRUCTION

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 is:

$$\leq \left(1 - \frac{m}{n}\right)^{O(\frac{n}{m} \log n)} \leq \frac{1}{n^3}.$$

By a union bound, satisfy all navigability requirements w.h.p.

UPPER BOUND CONSTRUCTION

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: $nm + n \cdot O(\frac{n}{m} \log n)$.

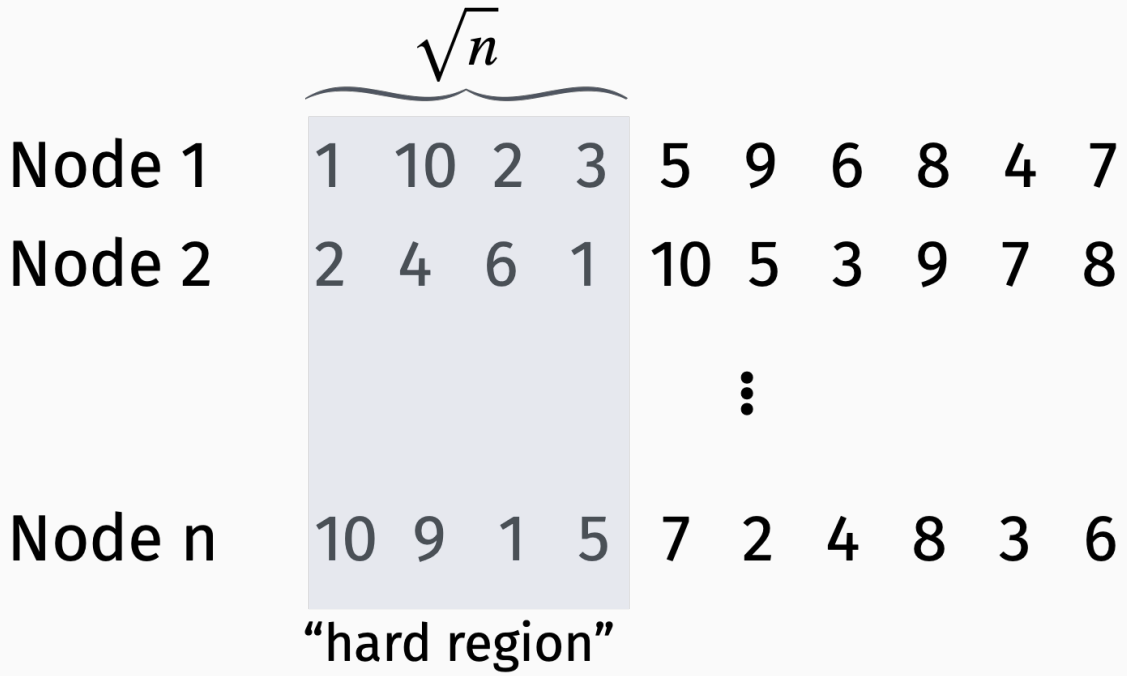
Minimize by setting $m = O(\sqrt{n \log n})$.

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

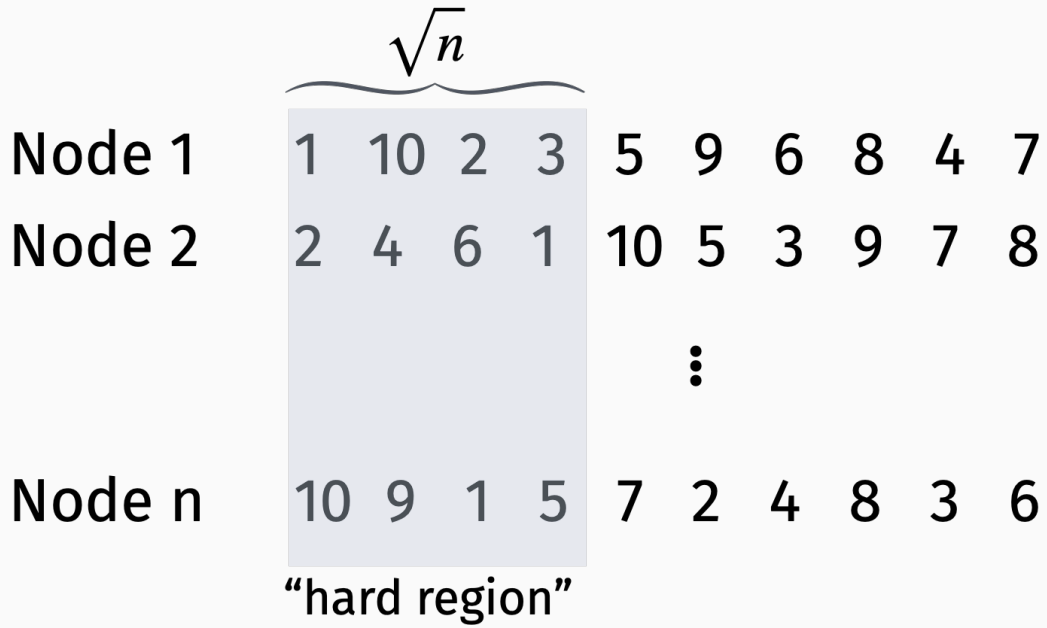
Moreover, the constructed graph has "depth" two: greedy search will converge in two steps for any query in the dataset.

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.

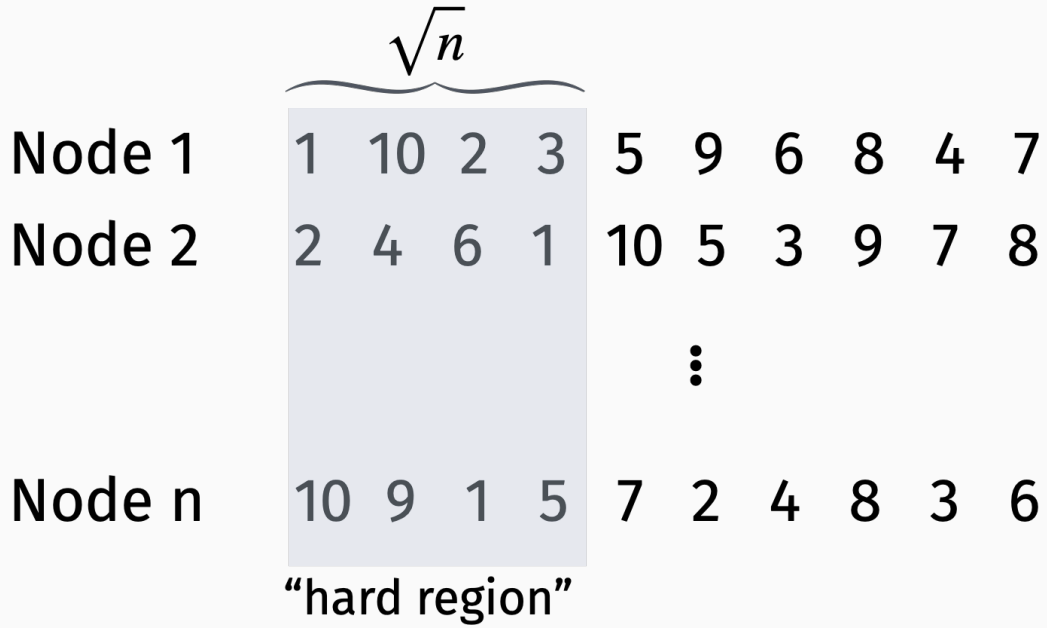


LOWER BOUND SKETCH



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.

LOWER BOUND SKETCH

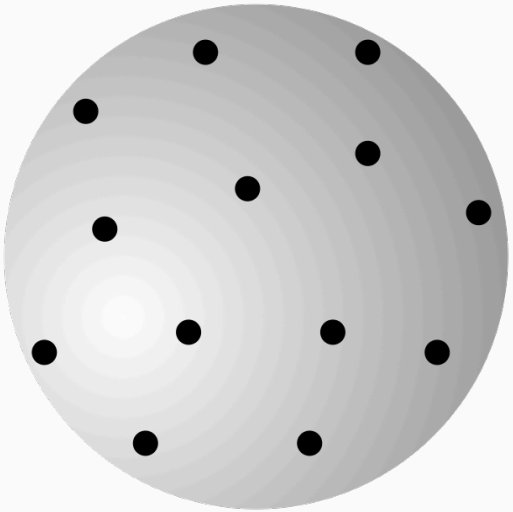


Consequence: For any i, j, k, the probability that both j and k appear in the hard region for row i is $\lesssim 1/n$.

With high probability, no pair (j, k) appears together in the hard region for more than $O(\log n)$ different rows. 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 $\tilde{O}(n^{3/2})$ edges.

$\Theta(n^{3/2})$ edges are sufficient and necessary to construct a navigable graph 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?

Instance-optimal navigable graphs

Theorem (CDFJLMSSW, SODA 2026)

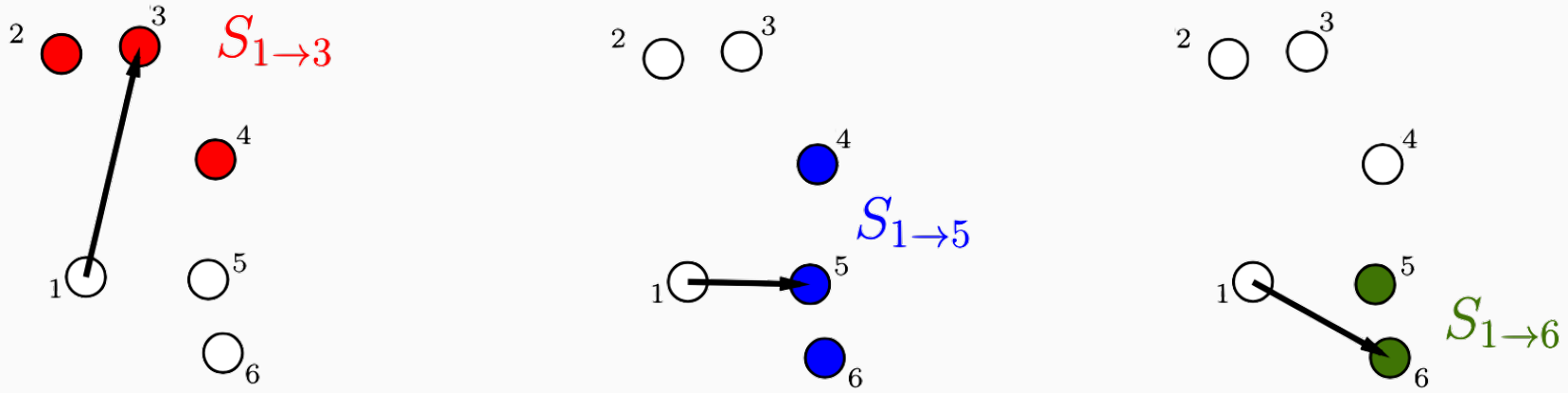
For any dataset and any distance function, we can construct a navigable graph with at most $O(\log n)$ times as many edges as the sparsest navigable graph in $\tilde{O}(n^2)$ time.

- $O(n^2)$ time is optimal even for points in $O(\log n)$ dimensional Euclidean space assuming Strong Exponential Time Hypothesis.
- $O(\log n)$ approximation factor is optimal assuming $P \neq NP$
- 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 $d(j,k) < d(j,i)$.

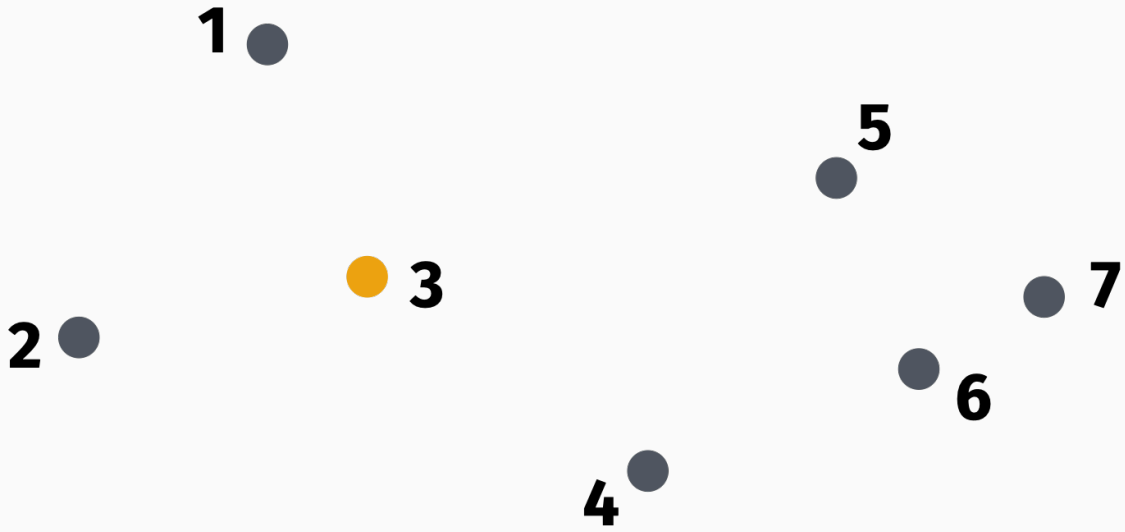


One problem for each node i. One set for all possible out neighbors. Elements to cover are all $j \neq i$.

NAVIGABILITY AS SET COVER

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

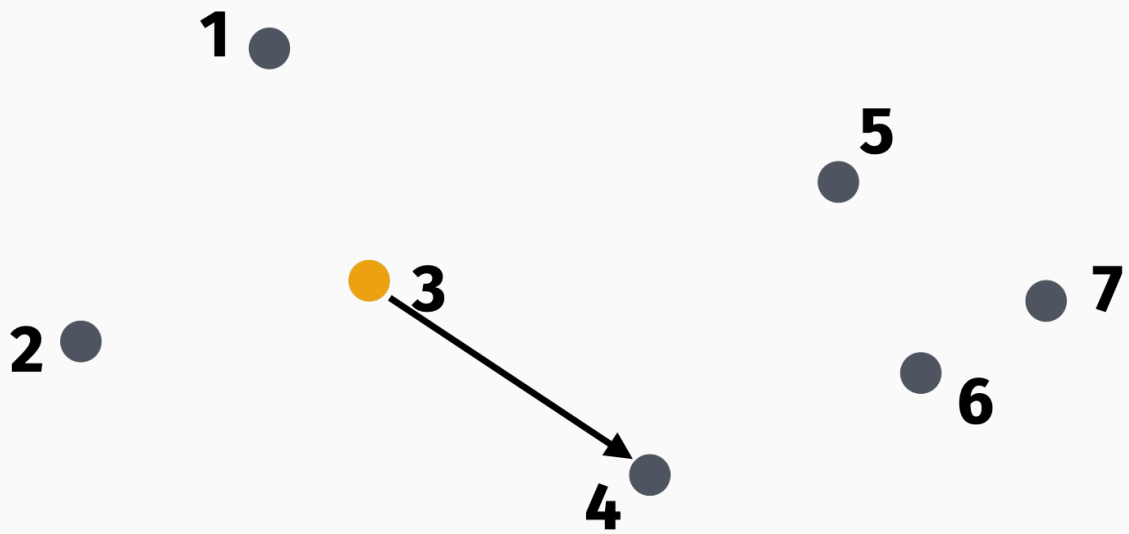
Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



NAVIGABILITY AS SET COVER

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

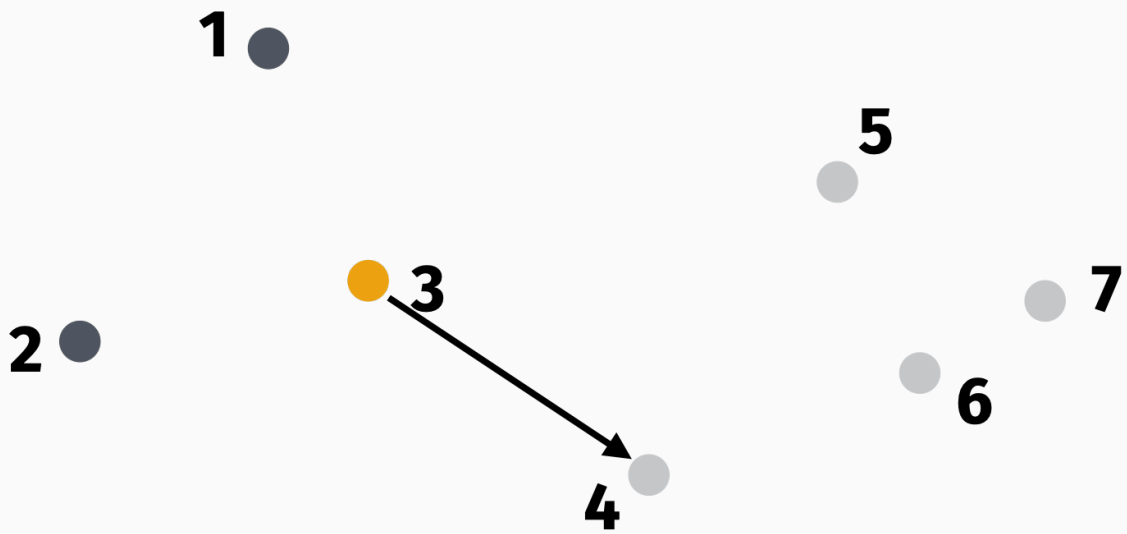
Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



NAVIGABILITY AS SET COVER

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

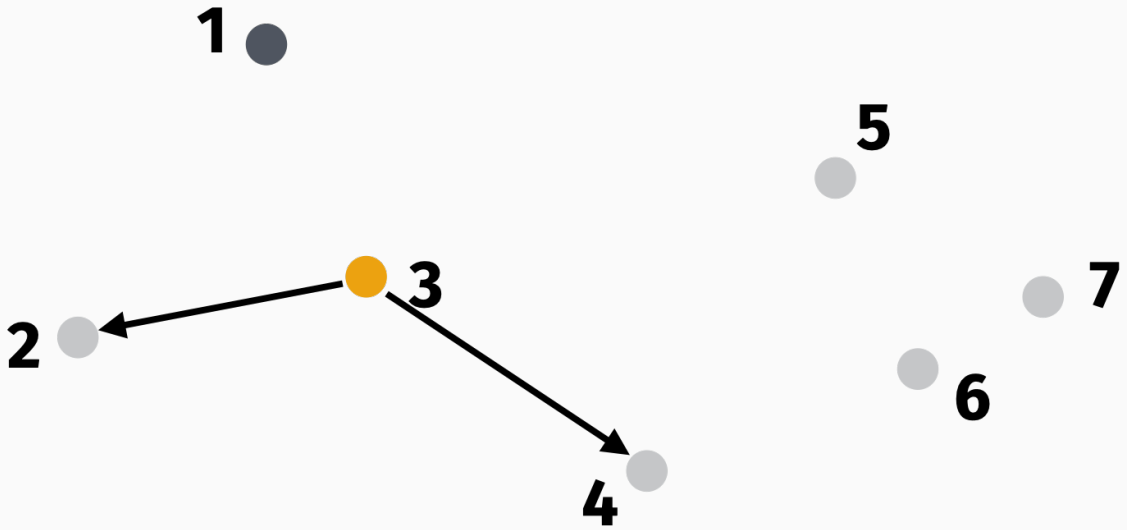
Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



NAVIGABILITY AS SET COVER

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

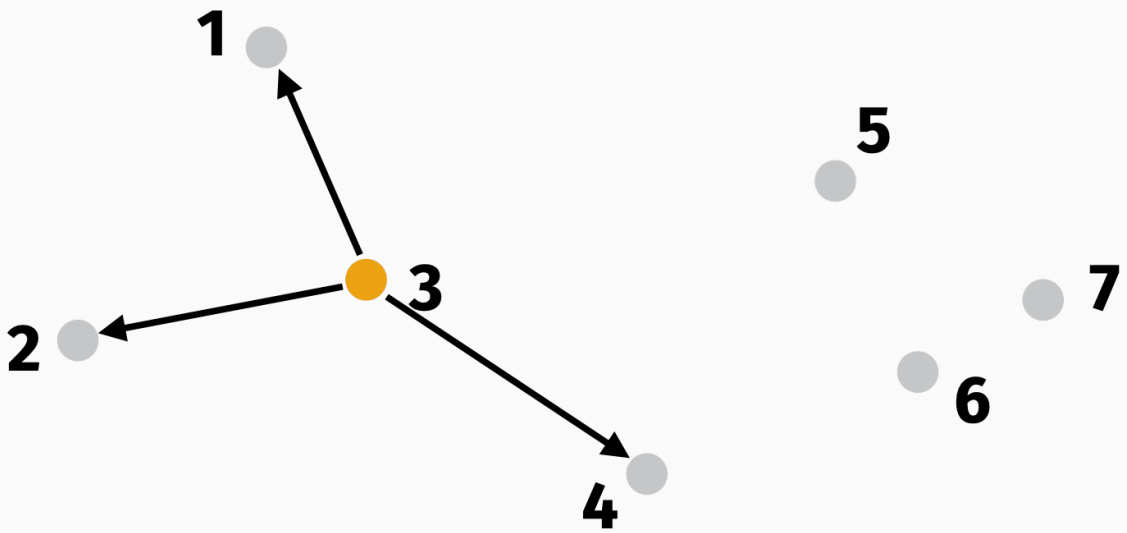
Standard greedy algorithm (pick the out-neighbor covering the most uncovered nodes) obtains an $O(\log n)$ approximation, and can be implemented $n \times O(n^2) = O(n^3)$ time.



NAVIGABILITY AS SET COVER

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.

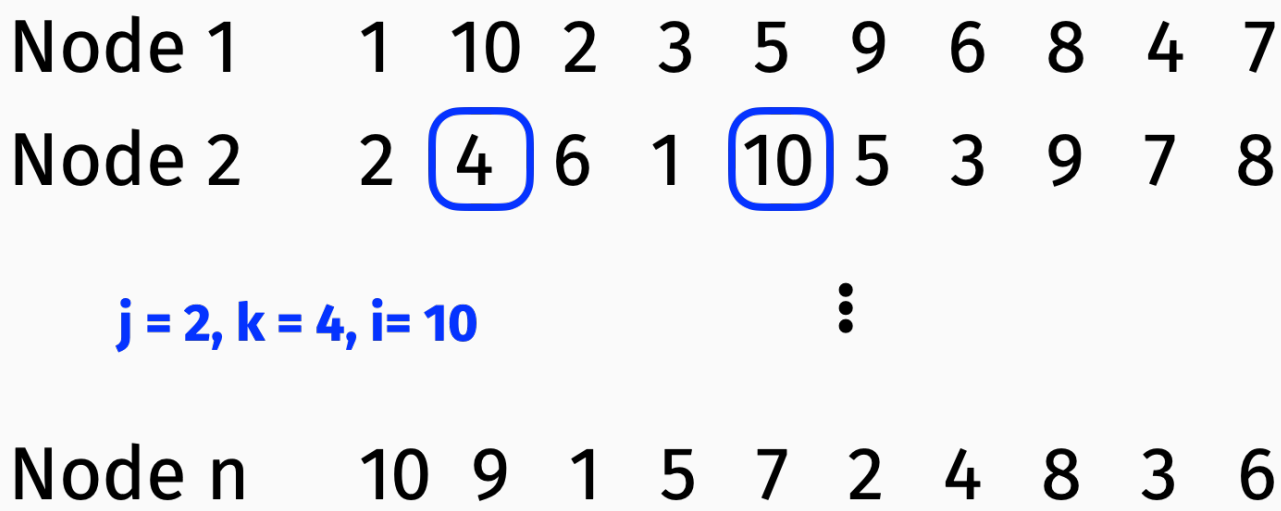
How can we beat cubic time?

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

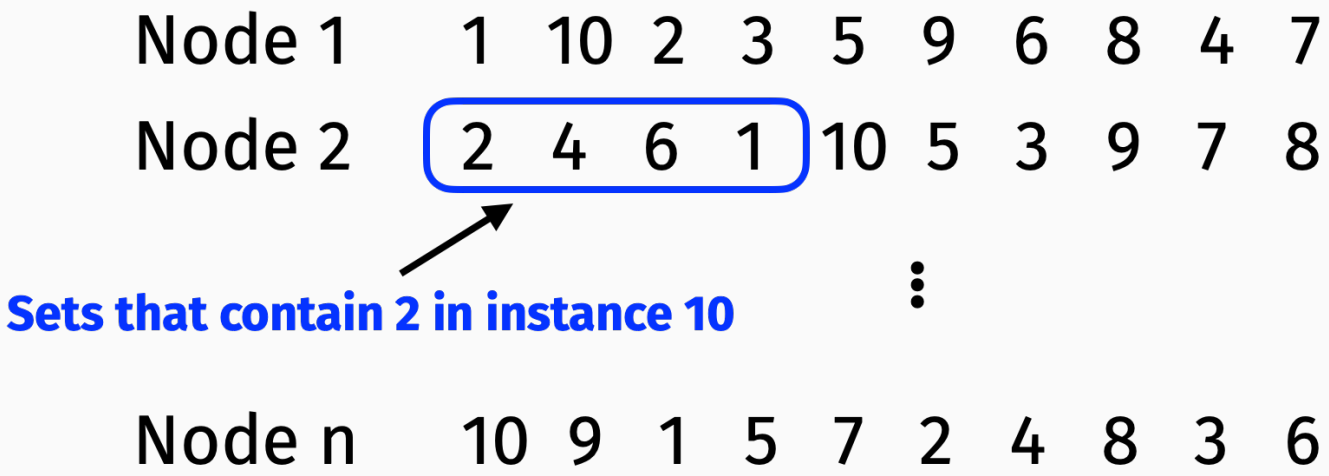
Concretely, after $O(n^2 \log n)$ time preprocessing, can implement **Membership** and **SetOf** queries in $O(1)$ time.



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

NAVIGABILITY AS SET COVER

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 k^{th} set containing j in instance i?

NAVIGABILITY AS SET COVER

Immediately let's use 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 immediately give an $O(\log n)$ approximation in time:

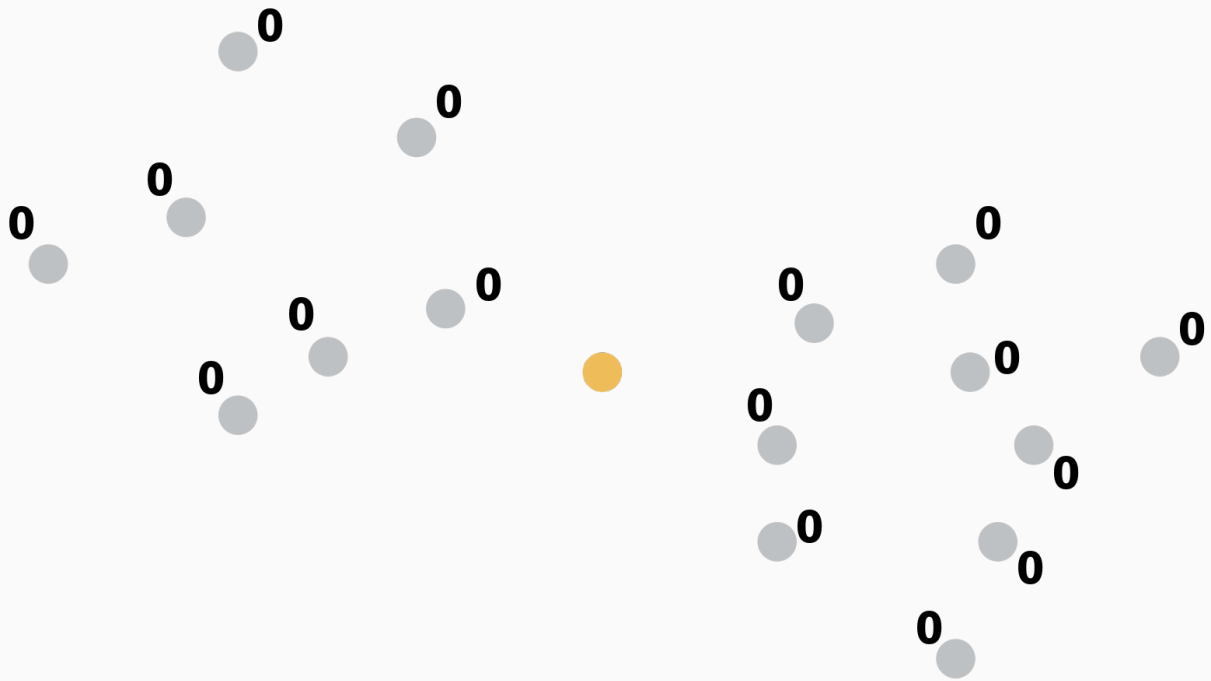
$$\tilde{O}(n \cdot OPT) \leq \tilde{O}(n^{2.5}).$$

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.

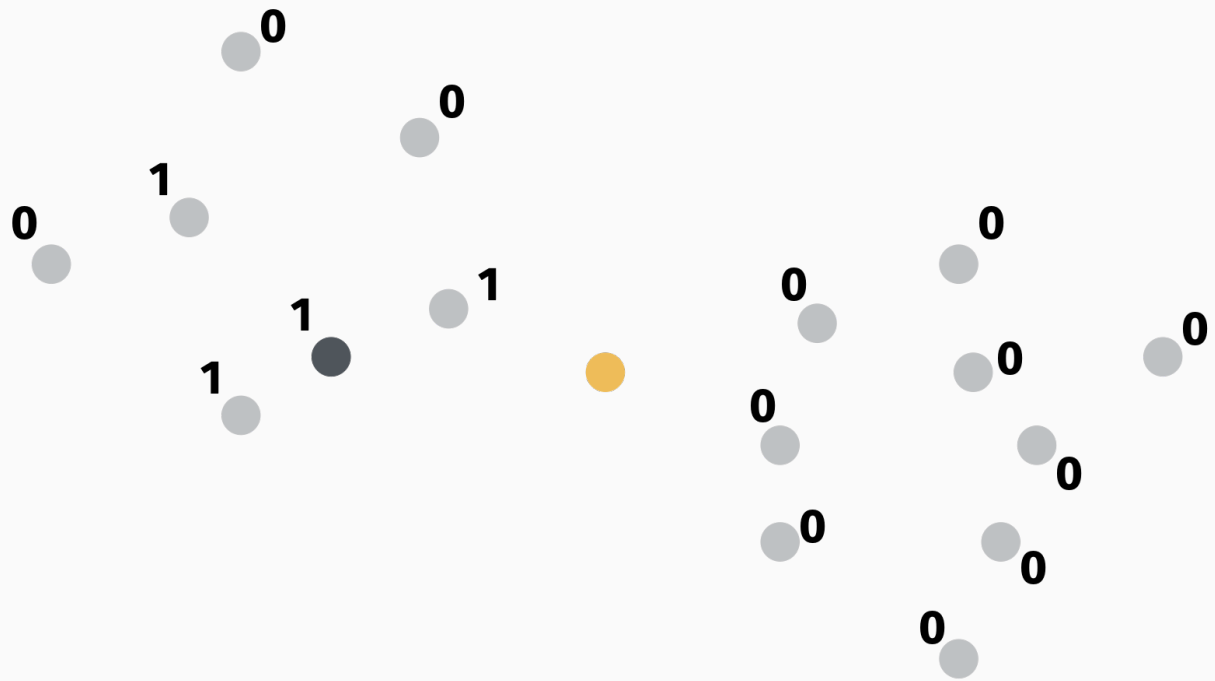
GREEDY SIMULATION

Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



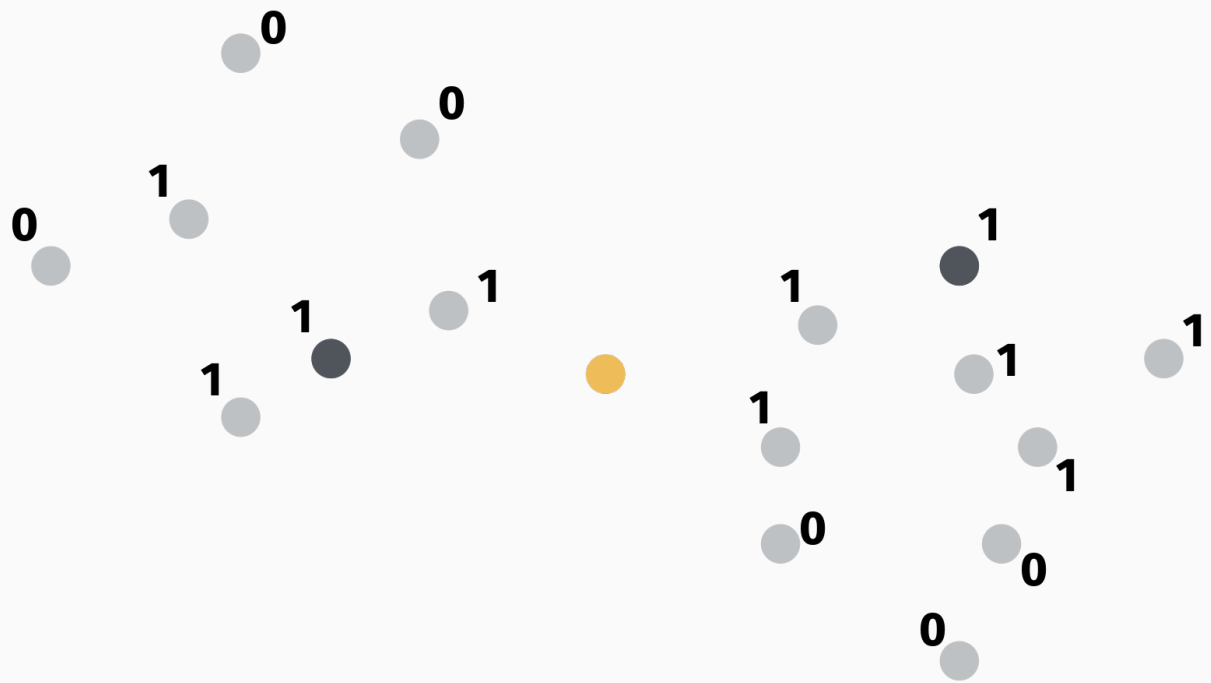
GREEDY SIMULATION

Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



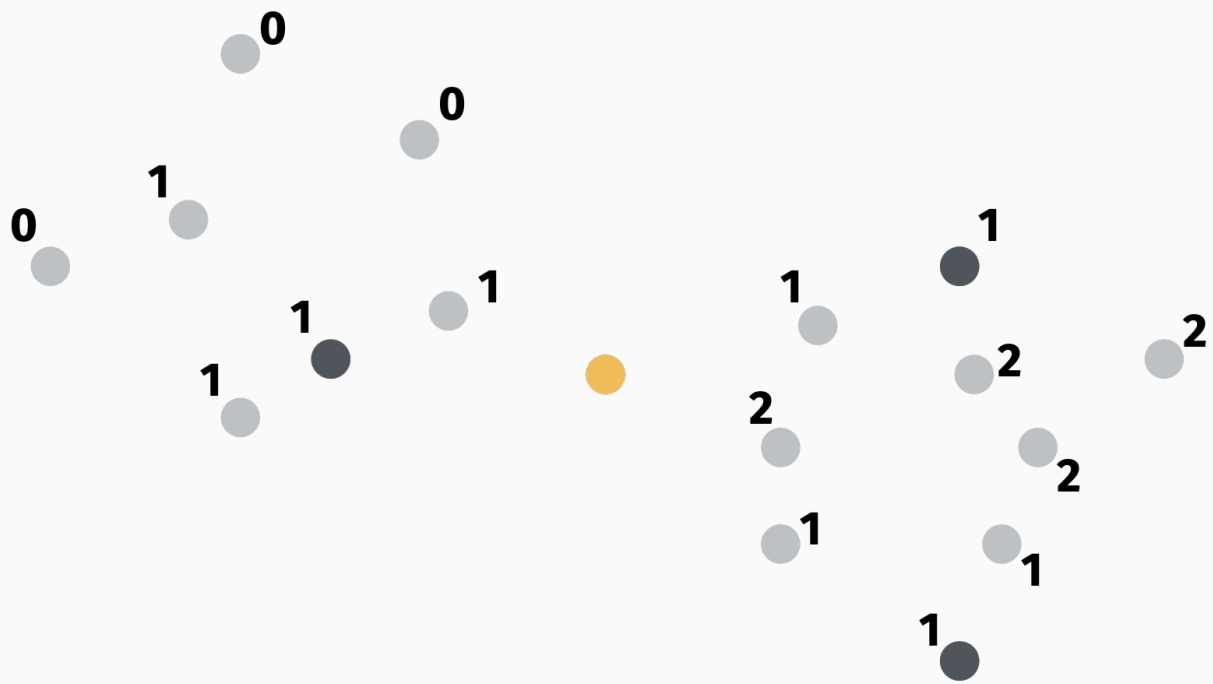
GREEDY SIMULATION

Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



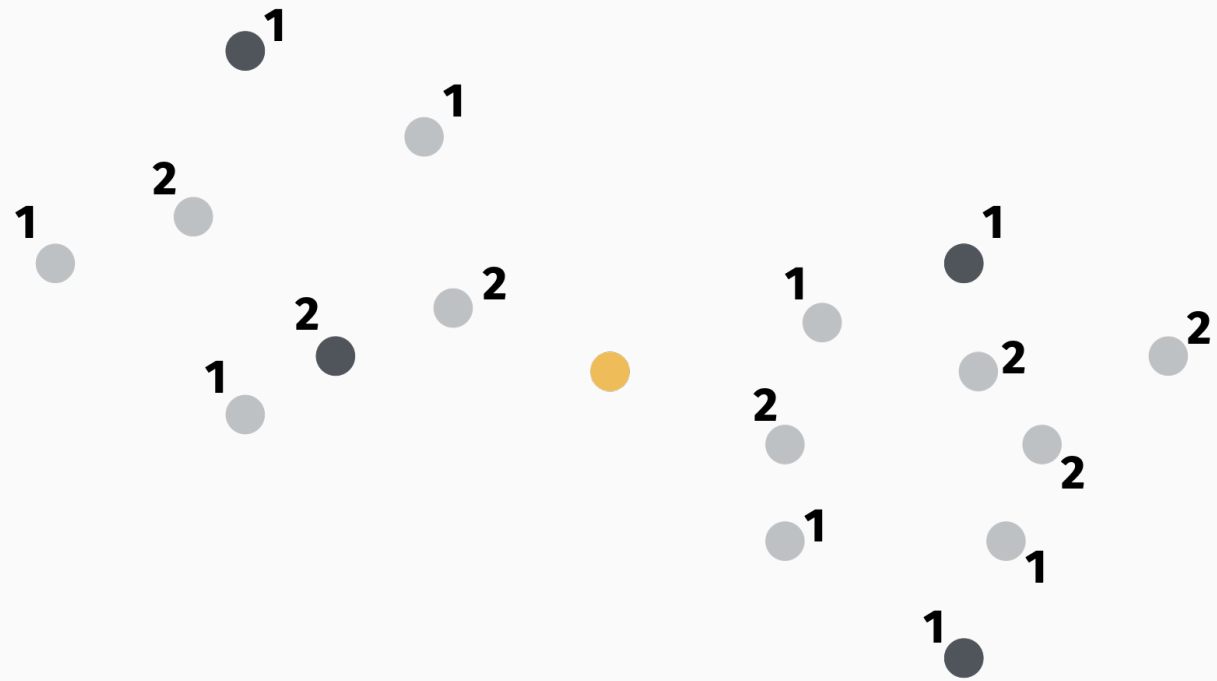
GREEDY SIMULATION

Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



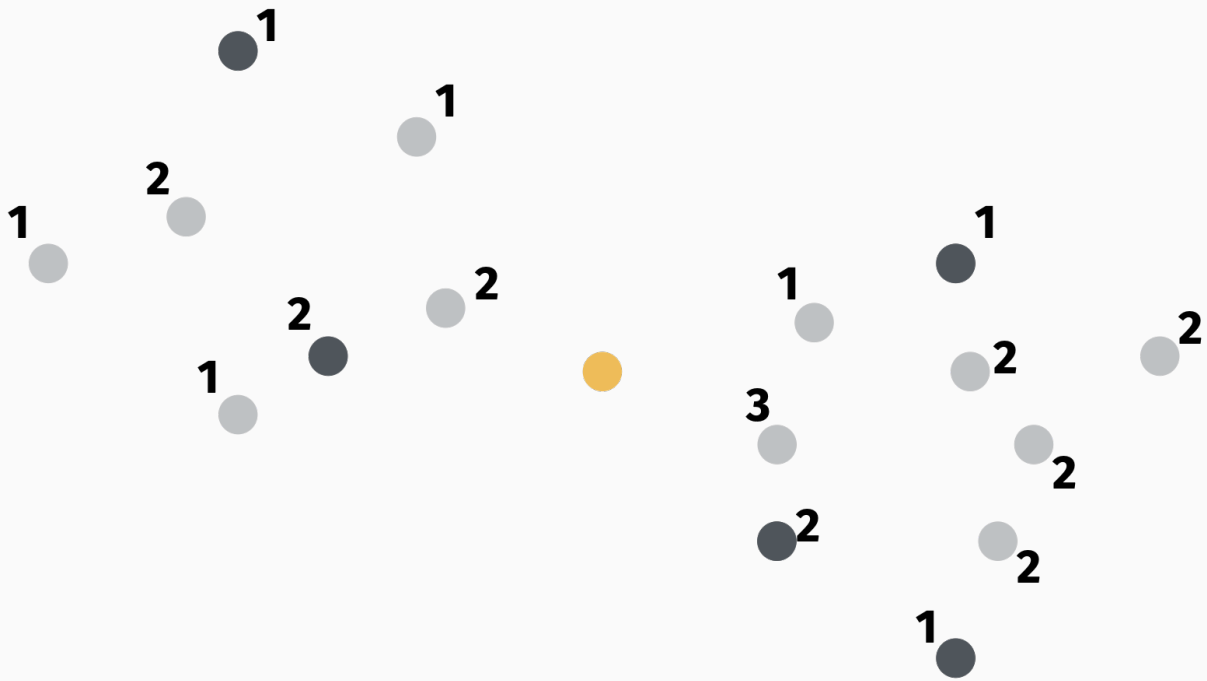
GREEDY SIMULATION

Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



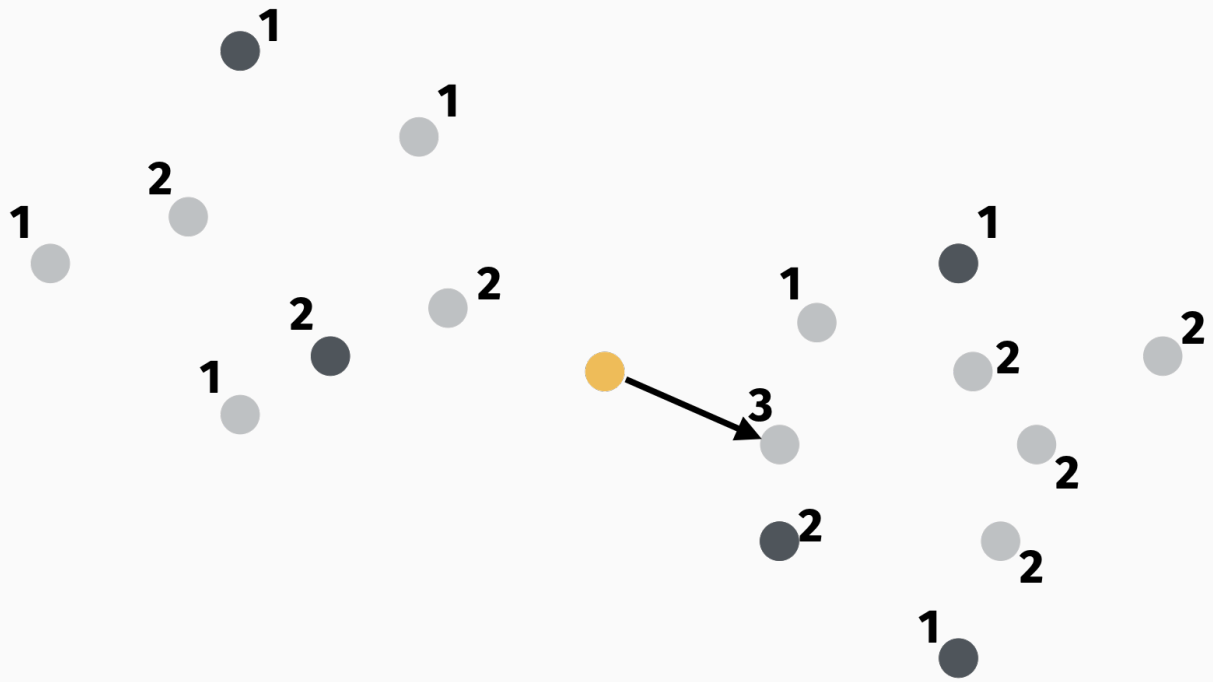
GREEDY SIMULATION

Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



GREEDY SIMULATION

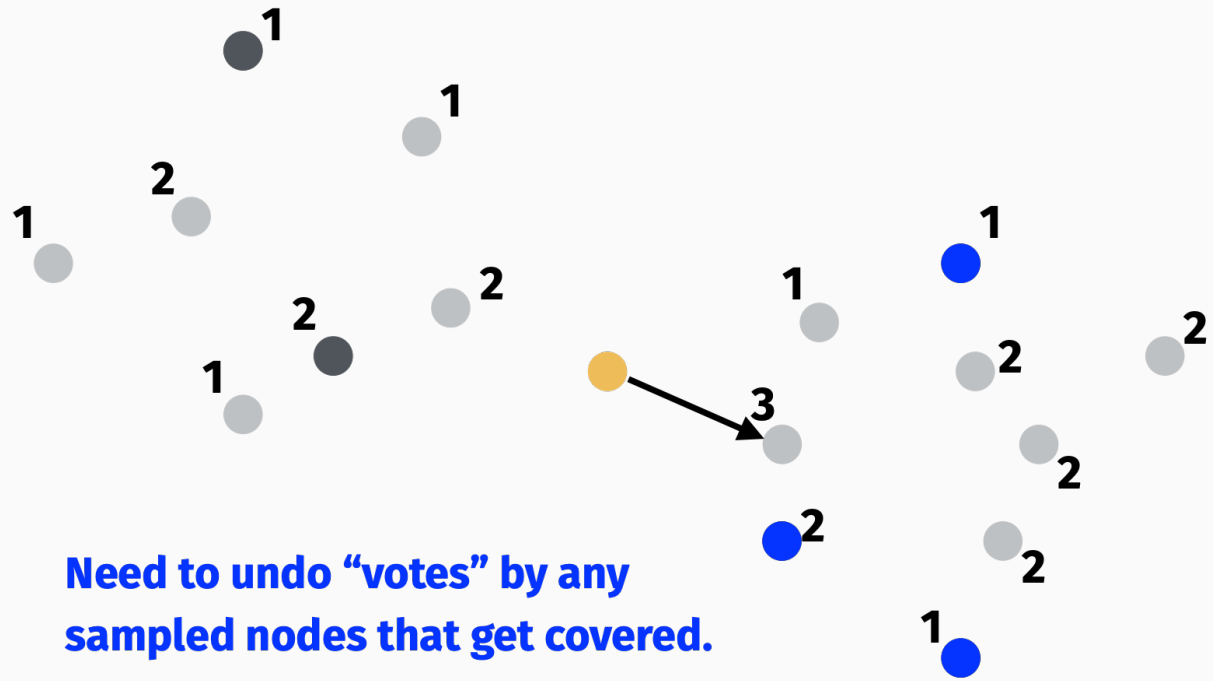
Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



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

GREEDY SIMULATION

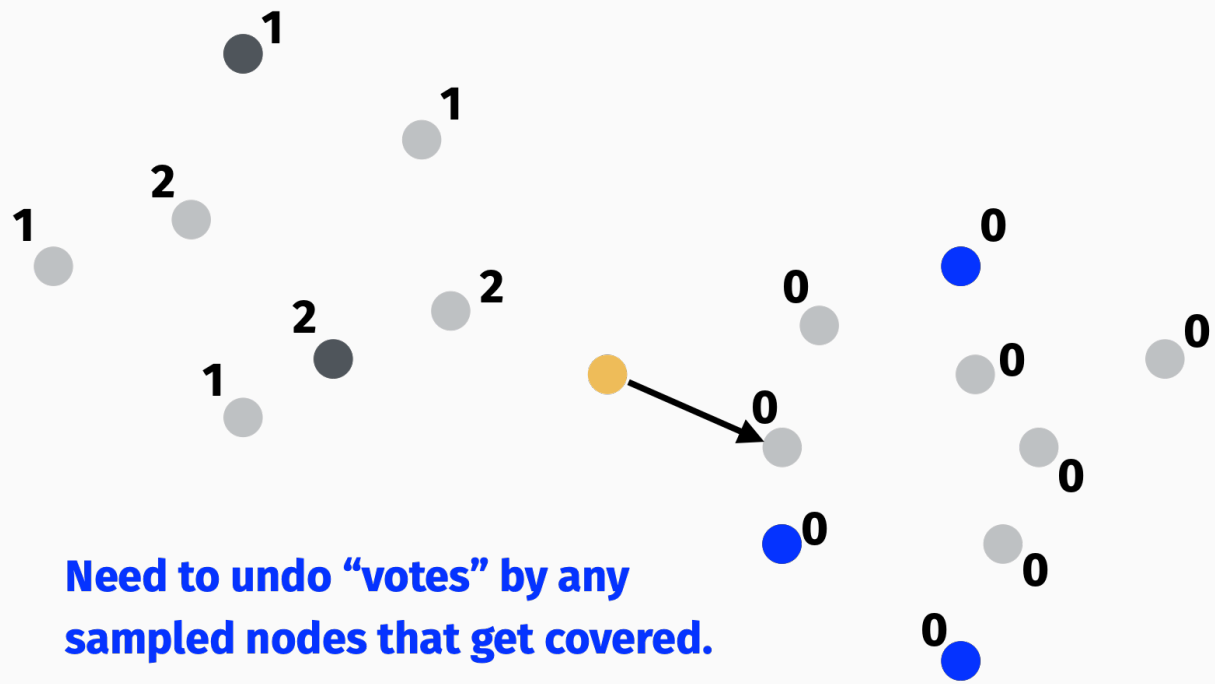
Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes it covers.



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

GREEDY SIMULATION

Sample uncovered nodes. For each set (potential out-neighbor), maintain a counter for how many sampled nodes 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.

GREEDY SIMULATION

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.

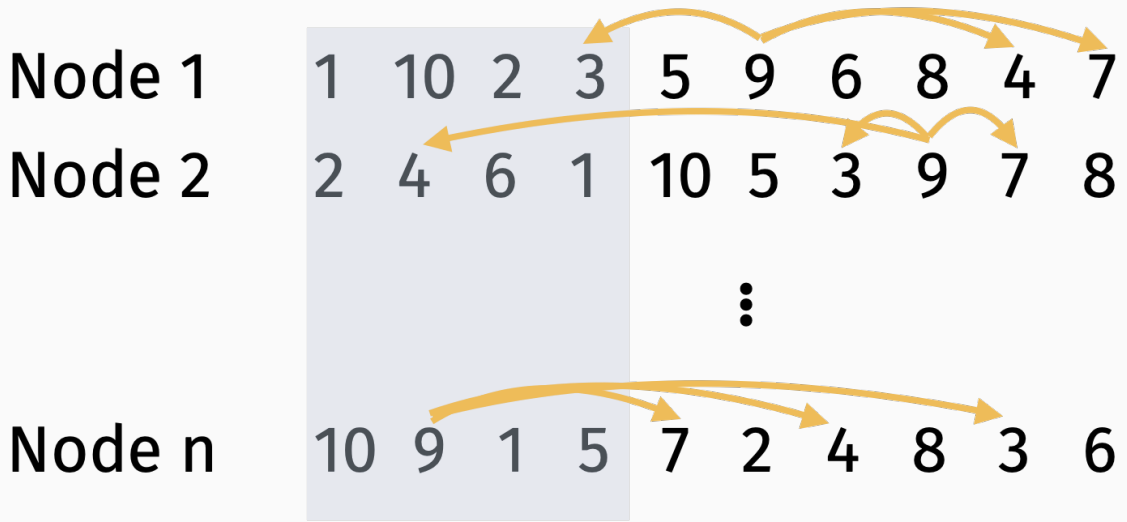
Main contributions to runtime:

1. When we sample node j, need to increase counter for all sets it is contained in.
2. Every time a node is sampled, need to check that it was not previously covered in one of our selected sets.

We can reduce the cost of both by adding random edges, just as we did in the $O(\sqrt{n \log n})$ construction!

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

1. Any element contained in $> n/(OPT/n) = n^2/OPT$ sets is covered with high probability.
2. Only n^3/OPT total elements are left uncovered.



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

1. Any element contained in $> n/(OPT/n) = n^2/OPT$ sets is covered with high probability.
2. Only n^3/OPT total elements are left uncovered.

Assume $OPT_i = OPT/n$ and we have roughly $n^2/OPT = n/OPT_i$ uncovered elements per instance. (Dealing with the "uneven" case is more technical.)

Total cost to run greedy simulation for node i is:

$$O(OPT_i \cdot \log n) \cdot \left(\underbrace{n/OPT_i}_{\text{increment counters}} + \underbrace{n/OPT_i}_{\text{\# of elements sampled}} \right) = O(n \log n).$$

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?

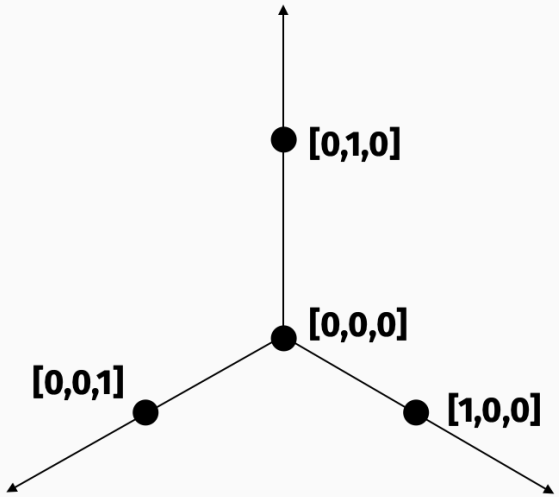
Still don't have sublinear query time bounds or strong approximation guarantees! There are **two main barriers.**

FAILED APPROXIMATION FOR GENERIC QUERIES

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 $d(j^*, q) > d(i, q)$.

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



In a navigable graph, if j is i's closest neighbor, then there must be an edge from j to i.

Some options for addressing these issues:

- Consider “backtracking” variations of greedy search, like **beam search**, that could offer better approximation guarantees.

In an upcoming NeurIPS 2025 paper, we describe an Adaptive Beam Search method that always returns a c-approximate nearest neighbor when run on any navigable graph.

- Relax the definition of navigability: only require greedy search to succeed for a particular chosen starting point. Avoids high maximum degree issue.

Other questions:

- Is there a $\tilde{O}\left(n^2\right)$ time algorithms for constructing optimal α-shortcut reachable graphs (discussed in Sepideh's talk). Best we can currently do is $\tilde{O}\left(n^{2.5}\right)$.
- 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?

QUESTIONS?