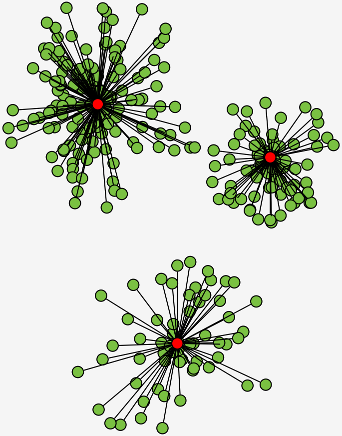
BICO: BIRCH meets Coresets for k-means

Hendrik Fichtenberger, Marc Gillé, Melanie Schmidt, Chris Schwiegelshohn, Christian Sohler



The k-means Problem

- Given a point set $P \subseteq \mathbb{R}^d$,
- compute a set $C \subseteq \mathbb{R}^d$ with $|C| = k$ centers
- which minimizes cost(P, C)

$$= \sum_{p \in P} \min_{c \in C} ||c - p||^2,$$

the sum of the squared distances.

Popular *k*-means algorithms. . .

- Lloyd's algorithm (1982)
- *k*-means++ (2007)
- several approximation algorithms (recent)

Popular k-means algorithms. . .

- Lloyd's algorithm (1982)
- k-means++ (2007)
- several approximation algorithms (recent)

. . . for Big Data

- BIRCH (1996)
- MacQueen's k-means (1967)
- several approximations using coresets (recent)

Popular *k*-means algorithms...

- Lloyd's algorithm (1982)
- *k*-means++ (2007)
- several approximation algorithms (recent)

...for Big Data

- BIRCH (1996)
- MacQueen's *k*-means (1967)
- several approximations using coresets (recent)

Implementability, Speed *and* good quality?

Popular *k*-means algorithms...

- Lloyd's algorithm (1982) — moderate speed
- *k*-means++ (2007) — moderate speed & quality
- several approximation algorithms (recent) — quality

...for Big Data

- BIRCH (1996) — fast
- MacQueen's *k*-means (1967) — fast
- several approximations using coresets (recent) — quality

Implementability, Speed *and* good quality?

Popular *k*-means algorithms. . .

- Lloyd's algorithm (1982) — moderate speed
- *k*-means++ (2007) — moderate speed & quality
- several approximation algorithms (recent) — quality

. . . for Big Data

- BIRCH (1996) — fast
- MacQueen's *k*-means (1967) — fast
- several approximations using coresets (recent) — quality

Implementability, Speed *and* good quality?

- Stream-KM++ (2010) — next slides

Popular *k*-means algorithms...

- Lloyd's algorithm (1982) — moderate speed
- *k*-means++ (2007) — moderate speed & quality
- several approximation algorithms (recent) — quality

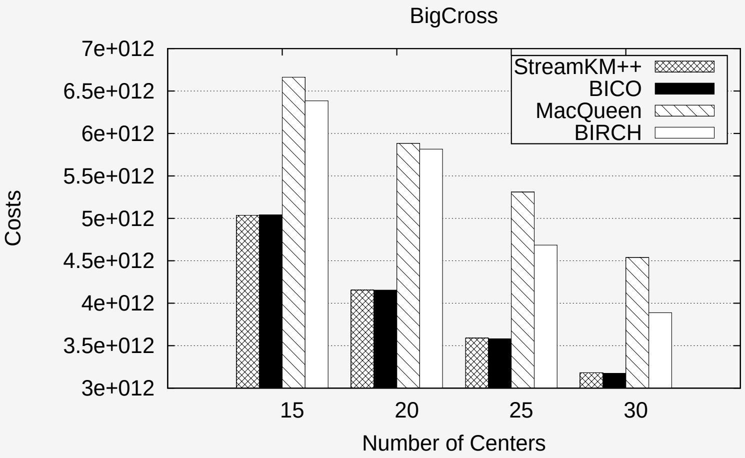
...for Big Data

- BIRCH (1996) — fast
- MacQueen's *k*-means (1967) — fast
- several approximations using coresets (recent) — quality

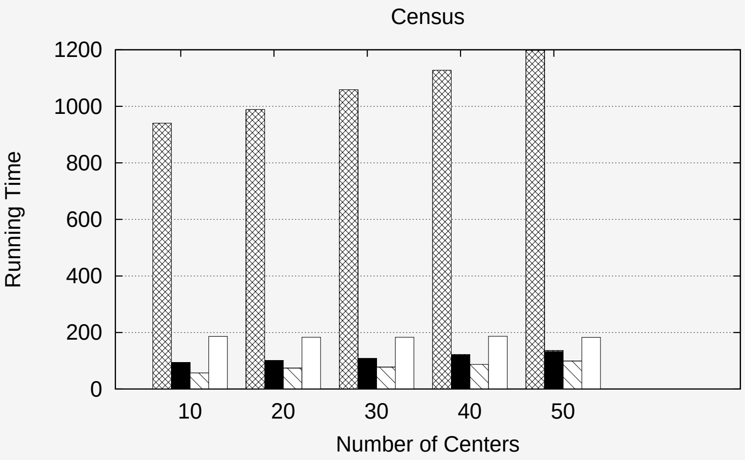
Implementability, Speed *and* good quality?

- Stream-KM++ (2010) — next slides
- BICO — next slides

Costs


BigCross
StreamKM++
BICO
MacQueen
BIRCH
7e+012
6.5e+012
6e+012
5.5e+012
5e+012
4.5e+012
4e+012
3.5e+012
3e+012
Costs
15
20
25
30
Number of Centers

Running Time



Idea

- start with BIRCH for the basic design because it is very fast
- analyze its flaws
- develop an improved algorithm based on theoretical observations

Idea

- start with BIRCH for the basic design because it is very fast
- analyze its flaws
- develop an improved algorithm based on theoretical observations

Now: Description of BIRCH

Idea

- start with BIRCH for the basic design because it is very fast
- analyze its flaws
- develop an improved algorithm based on theoretical observations

Now: Description of BIRCH

Warning

- BIRCH has several phases
- we are only interested in the main phase
- (and a little in the rebuilding phase)

BIRCH

BIRCH

- stores points in a tree

BIRCH

- stores points in a tree
- each node represents a subset of the input point set

BIRCH

- stores points in a tree
- each node represents a subset of the input point set
- subset is summarized by the number of points, the centroid of the set and the squared distances to the centroid

BIRCH

- stores points in a tree
- each node represents a subset of the input point set
- subset is summarized by the number of points, the centroid of the set and the squared distances to the centroid
- all points in the same subset get the same center

BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center

BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center

Insertion of a new point

When a new point p is added to the tree

BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center

Insertion of a new point

When a new point p is added to the tree

- BIRCH searches for the 'closest' node according to
$$\sum_{q \in (S \cup \{p\})} (q - \mu(S \cup \{p\}))^2 - \sum_{q \in S} (q - \mu(S))^2$$

BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center

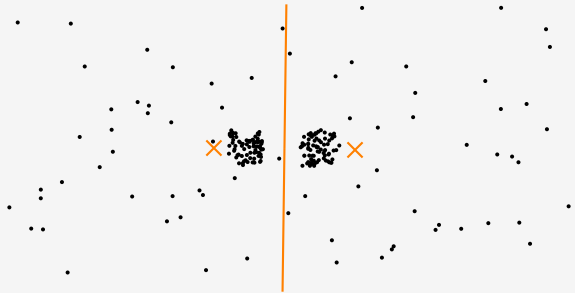
Insertion of a new point

When a new point p is added to the tree

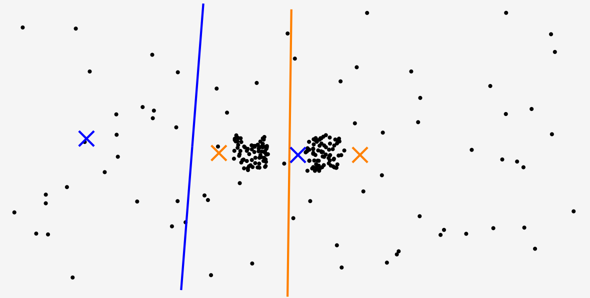
- BIRCH searches for the 'closest' node according to
$$\sum_{q\in(S\cup\{p\})}(q-\mu(S\cup\{p\}))^2-\sum_{q\in S}(q-\mu(S))^2$$
- p is added to the node representing subset S^* if
$$\sum_{q\in(S^*\cup\{p\})}(q-\mu_S)^2/(|S^*|+1)\leq T^2$$ for a given threshold T

- 150 points drawn uniformly around $(-0.5, 0)$ and $(0, 0.5)$
- 75 points drawn uniformly from $[-4, -2] \times [4, 2]$ as noise
- Centers and partitions computed by BIRCH and BICO

How BIRCH computes a summary of the data



- 150 points drawn uniformly around $(-0.5, 0)$ and $(0, 0.5)$
- 75 points drawn uniformly from $[-4, -2] \times [4, 2]$ as noise
- Centers and partitions computed by BIRCH and BICO



- 150 points drawn uniformly around $(-0.5, 0)$ and $(0, 0.5)$
- 75 points drawn uniformly from $[-4, -2] \times [4, 2]$ as noise
- Centers and partitions computed by BIRCH and BICO

Insights from BIRCH

- Fast point by point updates
- Tree structure

Insights from BIRCH

- Fast point by point updates
- Tree structure

- Insertion decision should be improved

Coresets

- small summary of given data
- typically of constant or polylogarithmic size
- can be used to approximate the cost of the original data

Coresets

Given a set of points P, a weighted subset $S \subset P$ is a (k, ϵ)-coreset if for all sets $C \subset \mathcal{C}$ of k centers it holds

$$|\mathrm{cost}_w(S, C) - \mathrm{cost}(P, C)| \leq \epsilon \, \mathrm{cost}(P, C)$$

where $\mathrm{cost}_w(S, C) = \sum_{p \in S} \min_{c \in C} w(p)(p, c)$.

Coresets

Given a set of points P, a weighted subset $S \subset P$ is a (k, ϵ)-coreset if for all sets $C \subset \mathcal{C}$ of k centers it holds

$$|\mathrm{cost}_w(S, C) - \mathrm{cost}(P, C)| \leq \epsilon \, \mathrm{cost}(P, C)$$

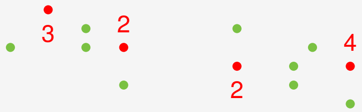
where $\mathrm{cost}_w(S, C) = \sum_{p \in S} \min_{c \in C} w(p)(p, c)$.

Coresets

Given a set of points P, a weighted subset $S \subset P$ is a (k, ϵ)-coreset if for all sets $C \subset \mathcal{C}$ of k centers it holds

$$|\mathrm{cost}_w(S, C) - \mathrm{cost}(P, C)| \leq \epsilon\, \mathrm{cost}(P, C)$$

where $\mathrm{cost}_w(S, C) = \sum_{p \in S} \min_{c \in C} w(p)(p, c)$.

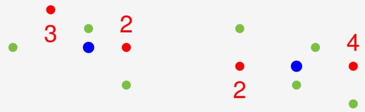


Coresets

Given a set of points P, a weighted subset $S \subset P$ is a (k, ϵ)-coreset if for all sets $C \subset \mathcal{C}$ of k centers it holds

$$|\text{cost}_w(S, C) - \text{cost}(P, C)| \leq \epsilon \, \text{cost}(P, C)$$

where $\text{cost}_w(S, C) = \sum_{p \in S} \min_{c \in C} w(p)(p, c)$.

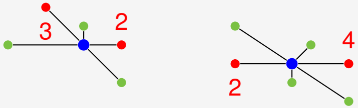


Coresets

Given a set of points P, a weighted subset $S \subset P$ is a (k, ϵ)-coreset if for all sets $C \subset \mathcal{C}$ of k centers it holds

$$|\mathrm{cost}_w(S, C) - \mathrm{cost}(P, C)| \leq \epsilon\, \mathrm{cost}(P, C)$$

where $\mathrm{cost}_w(S, C) = \sum_{p \in S} \min_{c \in C} w(p)(p, c)$.

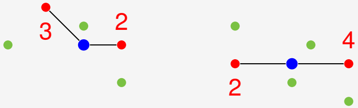


Coresets

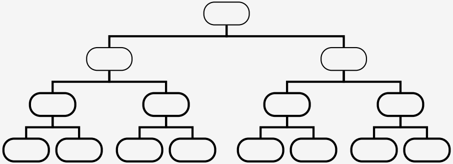
Given a set of points P, a weighted subset $S \subset P$ is a (k, ϵ)-coreset if for all sets $C \subset \mathcal{C}$ of k centers it holds

$$|\text{cost}_w(S, C) - \text{cost}(P, C)| \leq \epsilon \, \text{cost}(P, C)$$

where $\text{cost}_w(S, C) = \sum_{p \in S} \min_{c \in C} w(p)(p, c)$.

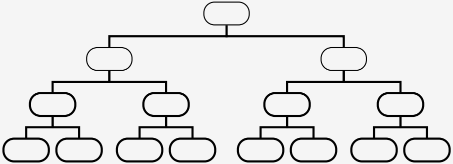


Merge & Reduce



- read data in blocks
- compute a coreset for each block $\rightarrow s$
- merge coresets in a tree fashion
- $\rightsquigarrow$ space $s \cdot \log n$

Merge & Reduce



- read data in blocks
- compute a coreset for each block $\rightarrow s$
- merge coresets in a tree fashion
- $\rightsquigarrow$ space $s \cdot \log n$

Runtime: No asymptotic increase, but overhead in practice

BIRCH uses point-wise updates :-)

Insights from Coreset Threory

- Limit the induced error

$\rightsquigarrow$ Goal: Point set P' in each node should induce at most $\varepsilon \cdot \text{cost}(P', C)$ error (for an optimal solution C)

$\rightsquigarrow$ Base insertion decision on induced error

- Replacing all points in a node by the (weighted) centroid is like moving all points to the centroid
- Induced error is connected to the 1-means cost of the set

Insights from Coreset Threory

- Limit the induced error

$\rightsquigarrow$ Goal: Point set P' in each node should induce at most $\varepsilon \cdot \mathrm{cost}(P', C)$ error (for an optimal solution C)

$\rightsquigarrow$ Base insertion decision on induced error

- Replacing all points in a node by the (weighted) centroid is like moving all points to the centroid
- Induced error is connected to the 1-means cost of the set

Side note

- Avoiding Merge & Reduce is a good idea

BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center
- improve insertion decision

BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center
- improve insertion decision

Adjustments

- Nodes additionally have a reference point and a range

BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center
- improve insertion decision

Adjustments

- Nodes additionally have a reference point and a range
- Closest is now determined by Euclidean distance

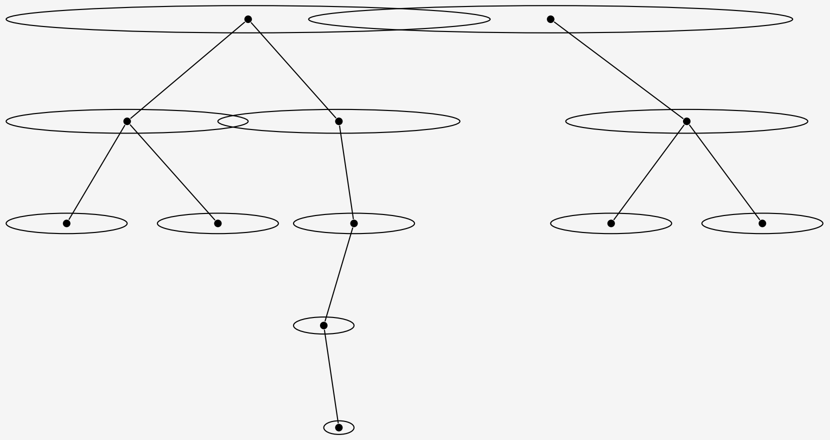
BIRCH

- nodes in the tree represent subsets of points
- points at the same node get the same center
- improve insertion decision

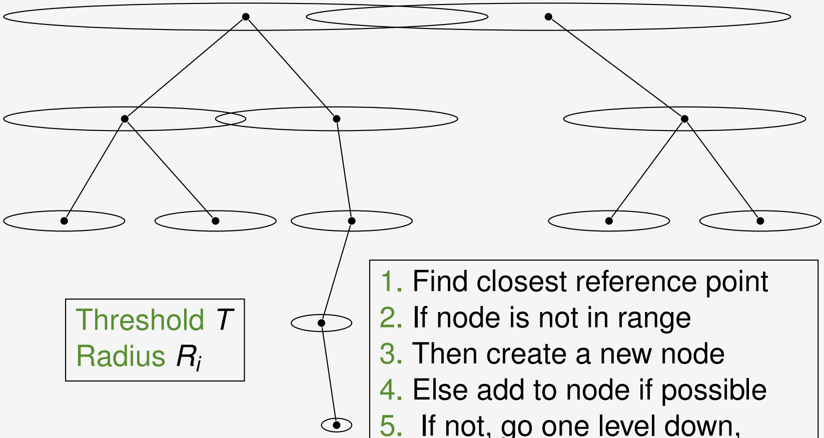
Adjustments

- Nodes additionally have a reference point and a range
- Closest is now determined by Euclidean distance
- We say a node is full with regard to a point p if adding p to the node increases its 1-means cost above a threshold T

BIRCH meets Coresets

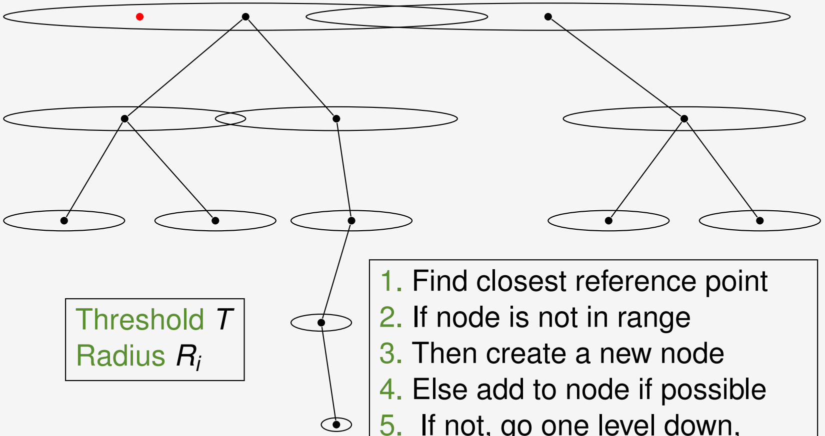


BIRCH meets Coresets



Threshold T
Radius R_i

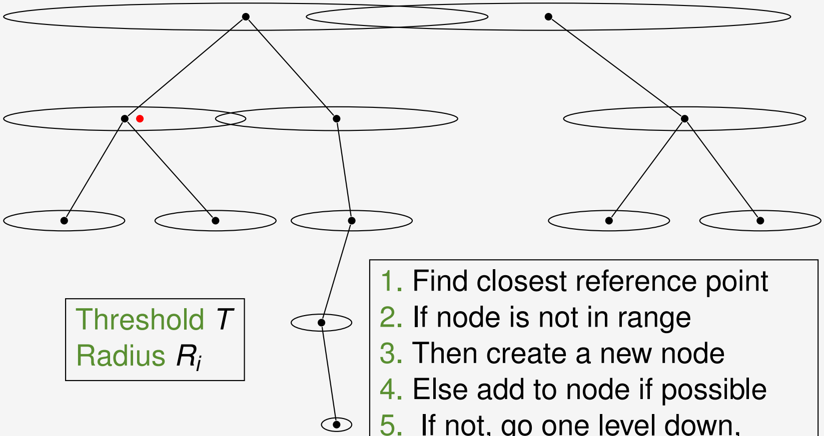
1. Find closest reference point
2. If node is not in range
3. Then create a new node
4. Else add to node if possible
5. If not, go one level down,
6. And Find closest child, goto 2.



Threshold T
Radius R_i

1. Find closest reference point
2. If node is not in range
3. Then create a new node
4. Else add to node if possible
5. If not, go one level down,
6. And Find closest child, goto 2.

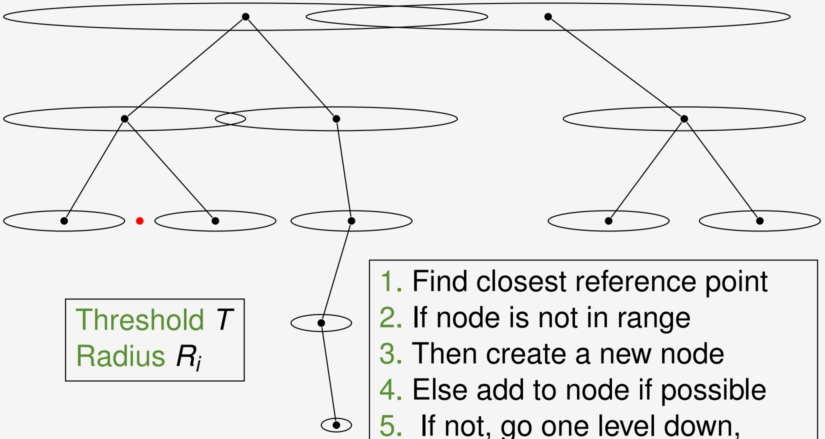
BIRCH meets Coresets



Threshold T
Radius R_i

1. Find closest reference point
2. If node is not in range
3. Then create a new node
4. Else add to node if possible
5. If not, go one level down,
6. And Find closest child, goto 2.

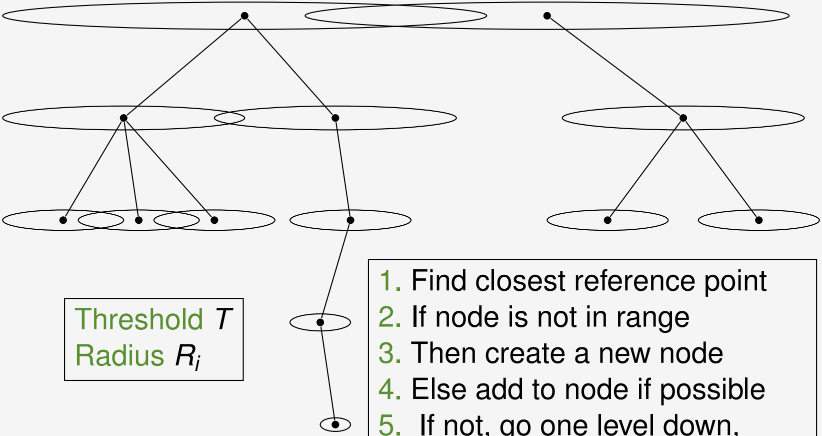
BIRCH meets Coresets



Threshold T
Radius R_i

1. Find closest reference point
2. If node is not in range
3. Then create a new node
4. Else add to node if possible
5. If not, go one level down,
6. And Find closest child, goto 2.

BIRCH meets Coresets



Threshold T
Radius R_i

1. Find closest reference point
2. If node is not in range
3. Then create a new node
4. Else add to node if possible
5. If not, go one level down,
6. And Find closest child, goto 2.

Theorem

For $T \approx OPT/(k \cdot \log n \cdot 8^d \cdot \varepsilon^{d+2})$ and $R_i := \sqrt{T/(8 \cdot 2^i)}$,

- the set of centroids weighted by the number of points in the subset is a $(1+\varepsilon)$-coreset
- for constant d, the number of nodes is $\mathcal{O}(k \cdot \log n \cdot \varepsilon^{-(d+2)})$

Theorem

For $T \approx OPT/(k \cdot \log n \cdot 8^d \cdot \varepsilon^{d+2})$ and $R_i := \sqrt{T/(8 \cdot 2^i)}$,

- the set of centroids weighted by the number of points in the subset is a $(1+\varepsilon)$-coreset
- for constant d, the number of nodes is $\mathcal{O}(k \cdot \log n \cdot \varepsilon^{-(d+2)})$

Problem

We do not know OPT and thus cannot compute T!

Rebuilding algorithm

- Double T when maximum number of nodes is reached
- ‘Rebuild’ the tree according to new T

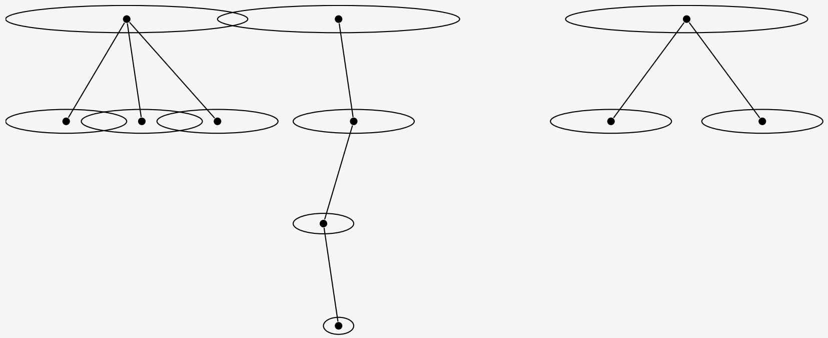
Rebuilding algorithm

- Double T when maximum number of nodes is reached
- 'Rebuild' the tree according to new T

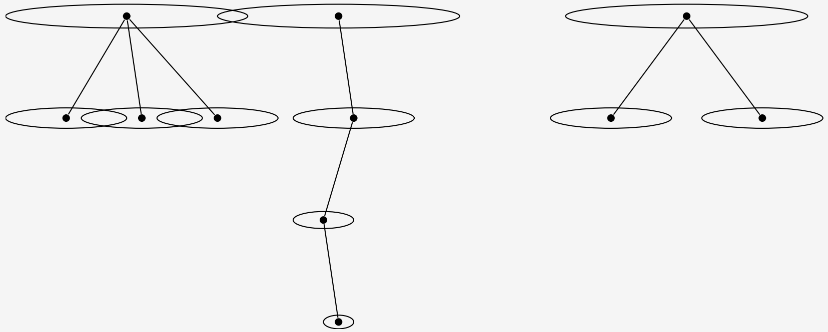
- Let T' and R'_i be before and T and R_i be after the doubling
- Move all nodes one level down and create empty first level
- Notice that $R_i = \sqrt{T/(8 \cdot 2^i)} = \sqrt{T'/8 \cdot 2^{i-1}} = R'_{i-1}$

$\Rightarrow$ Radius doesn't change!

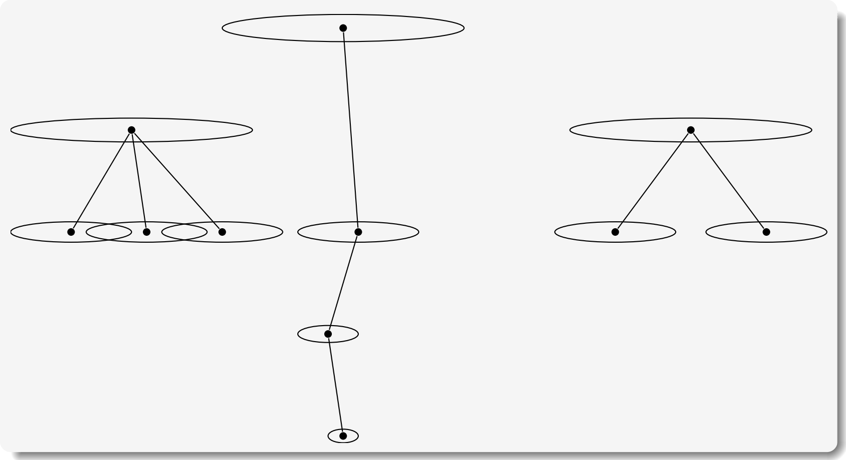
BIRCH meets Coresets



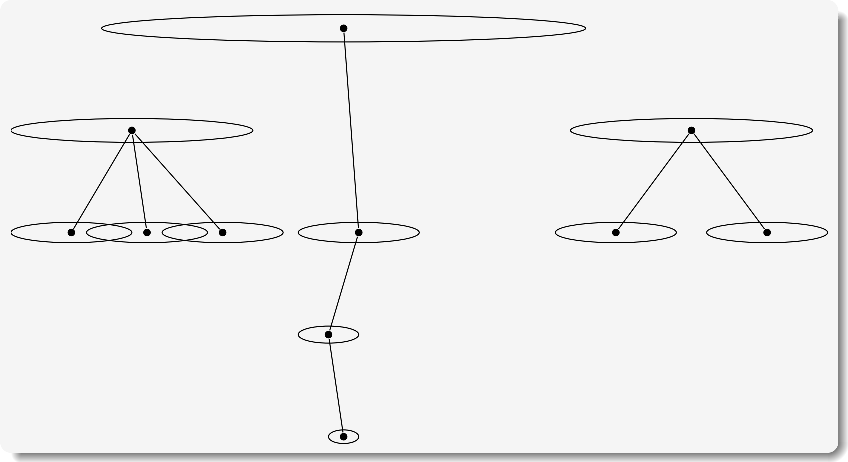
BIRCH meets Coresets



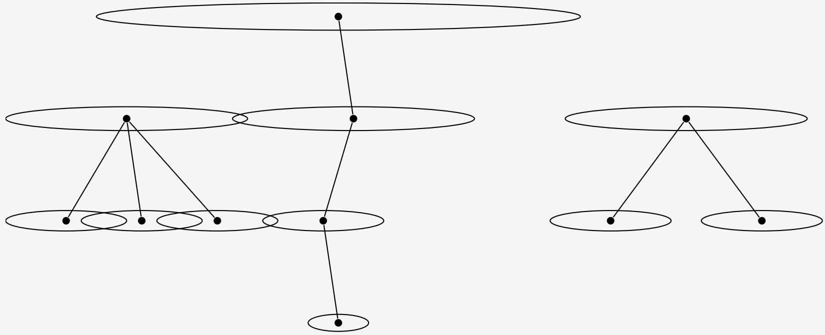
BIRCH meets Coresets



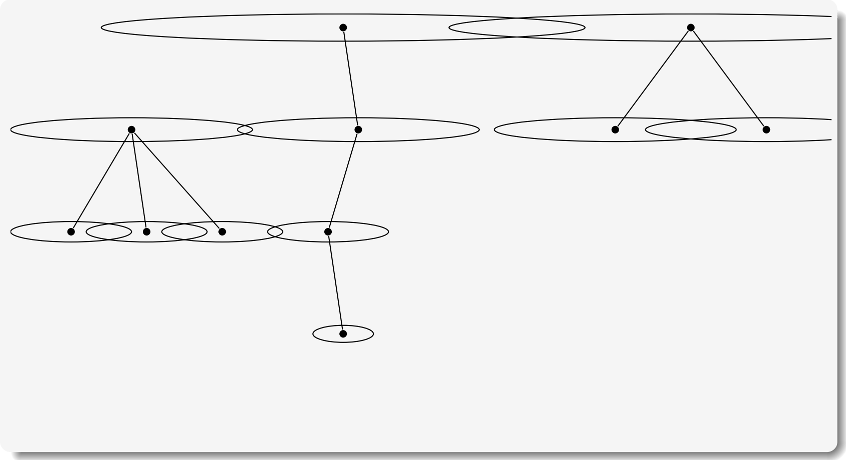
BIRCH meets Coresets

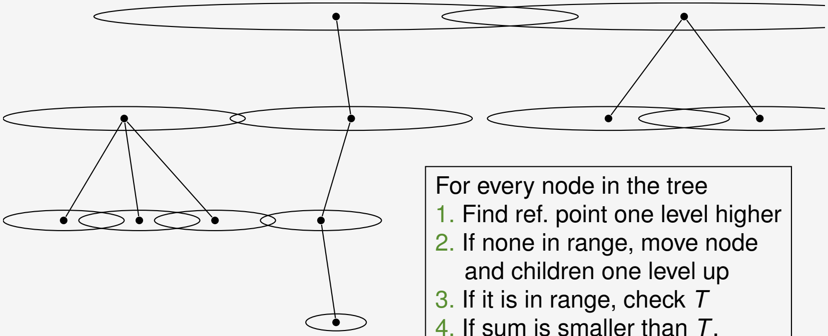


BIRCH meets Coresets



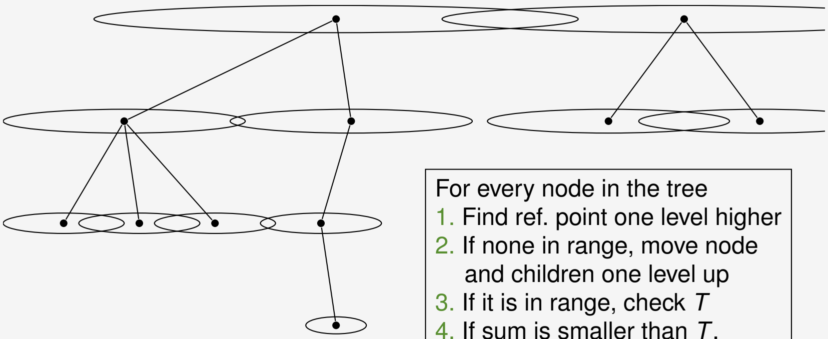
BIRCH meets Coresets





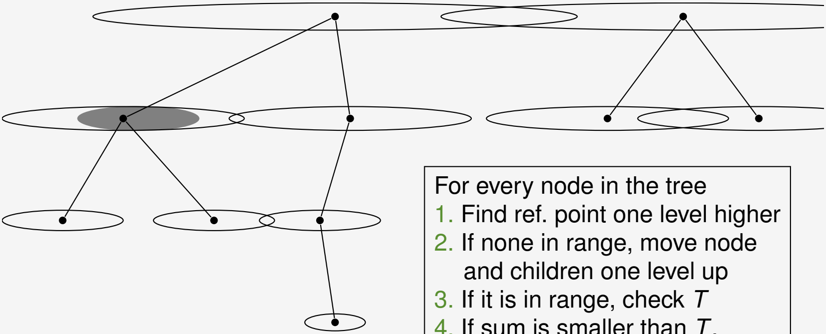
For every node in the tree

1. Find ref. point one level higher
2. If none in range, move node and children one level up
3. If it is in range, check T
4. If sum is smaller than T, merge nodes
6. Else make it a child

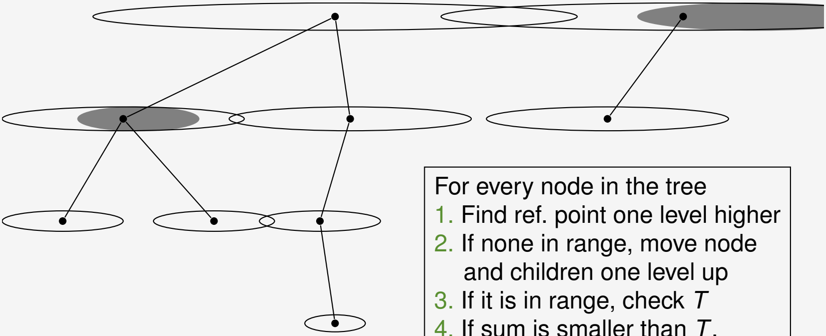


For every node in the tree

1. Find ref. point one level higher
2. If none in range, move node and children one level up
3. If it is in range, check T
4. If sum is smaller than T, merge nodes
6. Else make it a child

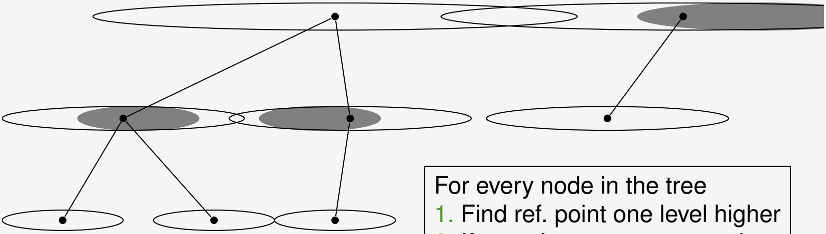


For every node in the tree
1. Find ref. point one level higher
2. If none in range, move node and children one level up
3. If it is in range, check T
4. If sum is smaller than T, merge nodes
6. Else make it a child

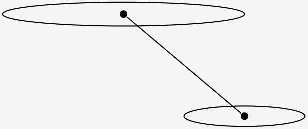


For every node in the tree
1. Find ref. point one level higher
2. If none in range, move node and children one level up
3. If it is in range, check T
4. If sum is smaller than T, merge nodes
6. Else make it a child

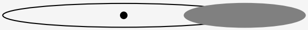
BIRCH meets Coresets



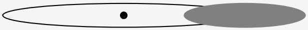
For every node in the tree
1. Find ref. point one level higher
2. If none in range, move node and children one level up
3. If it is in range, check T
4. If sum is smaller than T, merge nodes
6. Else make it a child



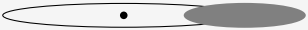
- Merging might result in violations of the range of nodes



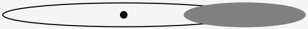
- Merging might result in violations of the range of nodes



- Merging might result in violations of the range of nodes
- Increases the coreset size by a constant factor

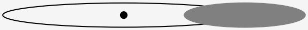


- Merging might result in violations of the range of nodes
- Increases the coreset size by a constant factor
- But does not destroy the coreset property



- Merging might result in violations of the range of nodes
- Increases the coreset size by a constant factor
- But does not destroy the coreset property

$\Rightarrow$ BICO computes a coreset in the data stream setting ☺



- Merging might result in violations of the range of nodes
- Increases the coreset size by a constant factor
- But does not destroy the coreset property

$\Rightarrow$ BICO computes a coreset in the data stream setting ☺

... if we compute lower bound on T

The actual solution is computed with k-means++.

The actual solution is computed with k-means++.

Adjustments

- Set coreset size to $200k$

The actual solution is computed with k-means++.

Adjustments

- Set coreset size to $200k$
- Add heuristic speed-up to find closest reference point

The actual solution is computed with k-means++.

Adjustments

- Set coreset size to $200k$
- Add heuristic speed-up to find closest reference point

Speed-up

- Project all ref. points to d random 1-dim. subspaces
- Project new point p to the same subspaces
- Count how many ref. points are in range of p in every subspace
- Search nearest neighbor in the shortest list

The actual solution is computed with k-means++.

Adjustments

- Set coreset size to $200k$
- Add heuristic speed-up to find closest reference point

Speed-up

- Project all ref. points to d random 1-dim. subspaces
- Project new point p to the same subspaces
- Count how many ref. points are in range of p in every subspace
- Search nearest neighbor in the shortest list

Data Sets

- Data Sets used in StreamKM++ paper from UCI repository: Tower, CoverType, Census and BigCross (cross product)
- CalTech128 by René Grzeszick, group of Prof. Fink
- consists of 128 SIFT descriptors of an object database

Data Sets

- Data Sets used in StreamKM++ paper from UCI repository: Tower, CoverType, Census and BigCross (cross product)
- CalTech128 by René Grzeszick, group of Prof. Fink
- consists of 128 SIFT descriptors of an object database

Data Set Sizes

	BigCross	CalTech128	Census	CoverType	Tower
n	11620300	3168383	2458285	581012	4915200
d	57	128	68	55	3
$n \cdot d$	662357100	405553024	167163380	31955660	14745600

Implementations

- Author's implementations for StreamKM++ and BIRCH
- implementation for MacQueen's k-means from ESMERALDA (framework by group of Prof. Fink)

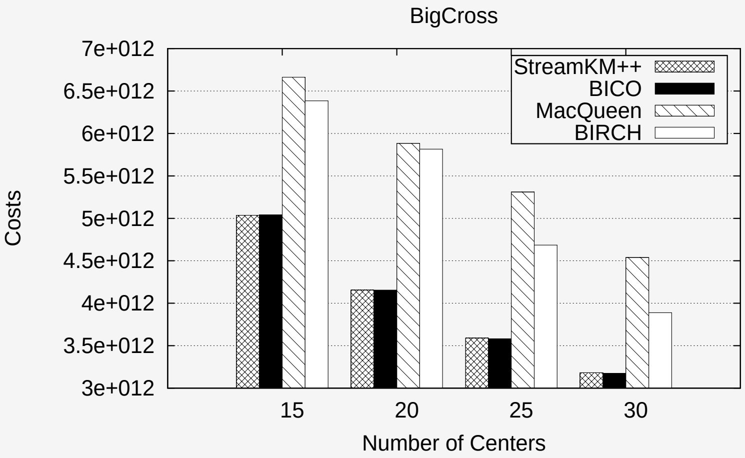
Implementations

- Author's implementations for StreamKM++ and BIRCH
- implementation for MacQueen's k-means from ESMERALDA (framework by group of Prof. Fink)

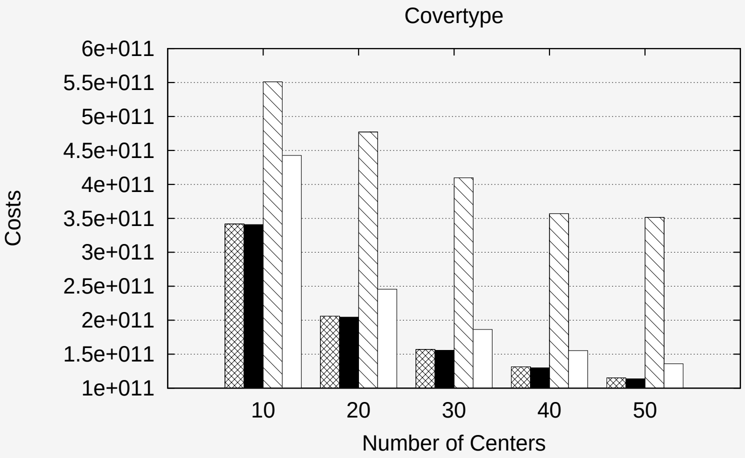
Experiments

- Experiments done on mud1-6 and mud8
- 100 runs for every test instance
- values shown in the diagrams are mean values

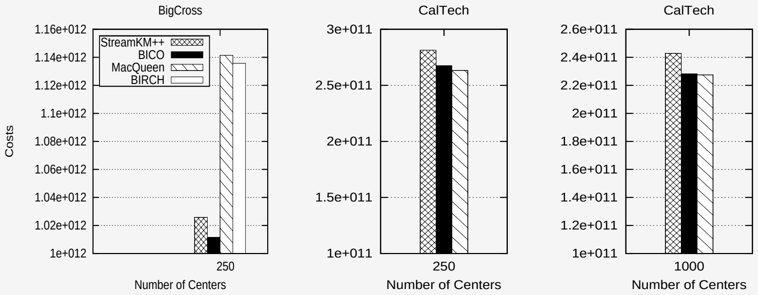
Costs


BigCross
7e+012
6.5e+012
6e+012
5.5e+012
5e+012
4.5e+012
4e+012
3.5e+012
3e+012
Costs
15
20
25
30
Number of Centers
StreamKM++
BICO
MacQueen
BIRCH

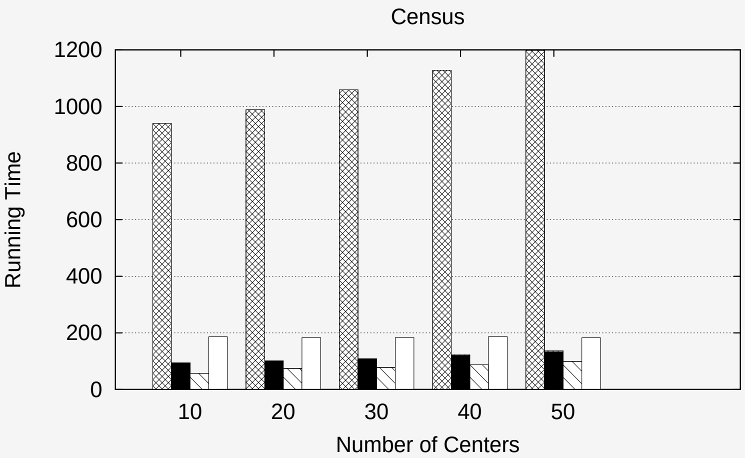
Costs


Covertype
Costs
6e+011
5.5e+011
5e+011
4.5e+011
4e+011
3.5e+011
3e+011
2.5e+011
2e+011
1.5e+011
1e+011
10
20
30
40
50
Number of Centers

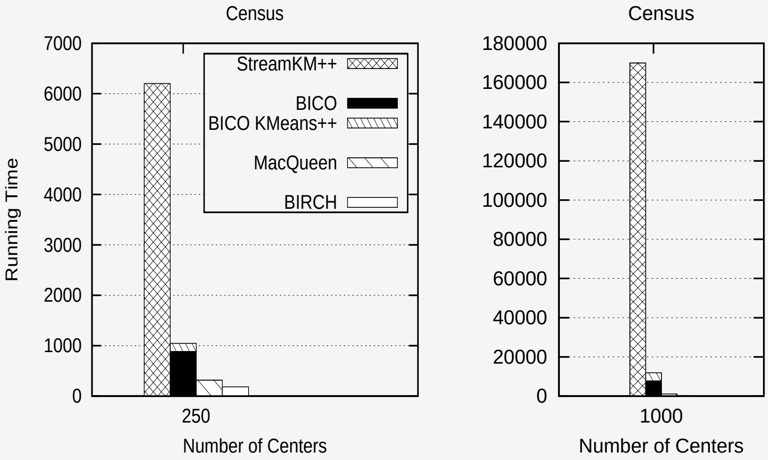
Costs


BigCross
Costs
1.16e+012
1.14e+012
1.12e+012
1.1e+012
1.08e+012
1.06e+012
1.04e+012
1.02e+012
1e+012
StreamKM++
BICO
MacQueen
BIRCH
250
Number of Centers
CalTech
3e+011
2.5e+011
2e+011
1.5e+011
1e+011
250
Number of Centers
CalTech
2.6e+011
2.4e+011
2.2e+011
2e+011
1.8e+011
1.6e+011
1.4e+011
1.2e+011
1e+011
1000
Number of Centers

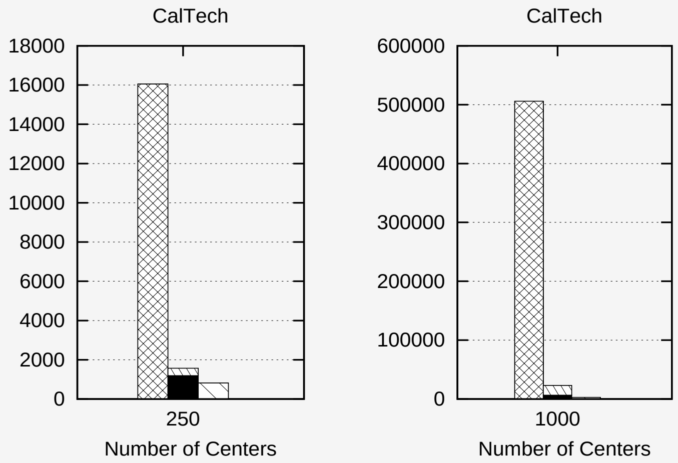
Running time



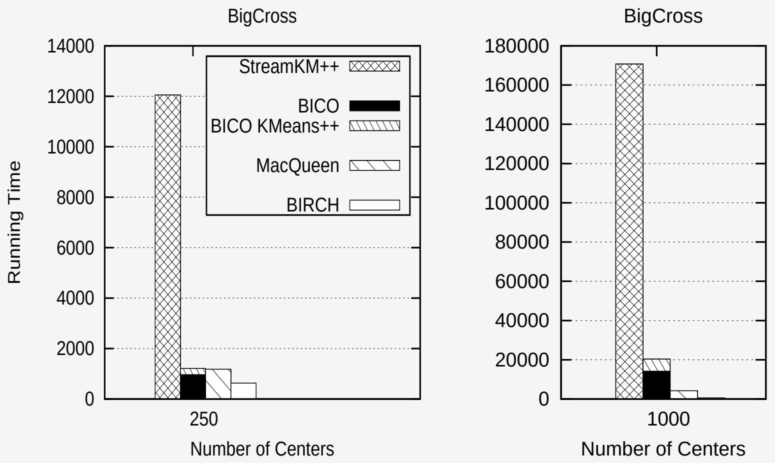
Running time



Running time



Running time



Trade-Off

	BIRCH	MQ	$m = 200k$	$100k$	$25k$
Time Costs	616 $58 \cdot 10^{10}$	4241 $72 \cdot 10^{10}$	20271 $51 \cdot 10^{10}$	5444 $52 \cdot 10^{10}$	618 $55 \cdot 10^{10}$

- Tests run on on BigCross with $k = 1000$
- BICO is less than 1.5 times slower than MacQueen with $m = 100k$ while still computing reasonable costs
- faster than BIRCH for $m = 25k$, still much better cost than BIRCH and MacQueen

Ziele

- Implementierung in bekanntem Framework / Anbindung
- $\rightsquigarrow$ Baustein für andere Algorithmen
- Vergleich verschiedener Strategien für Nearest Neighbor

	BIRCH	MQ	$m = 200k$	$100k$	$25k$
Time	616	4241	20271	5444	618
Costs	$58 \cdot 10^{10}$	$72 \cdot 10^{10}$	$51 \cdot 10^{10}$	$52 \cdot 10^{10}$	$55 \cdot 10^{10}$

Thank you for your attention!